Towards Modular Development of Reusable Language Components for Domain-Specific Modeling Languages in the MagicDraw and MontiCore Ecosystems

Arvid Butting[*], **Rohit Gupta**[†], **Nico Jansen**[*], **Nikolaus Regnat**[†], **and Bernhard Rumpe**[*]
[*]Software Engineering, RWTH Aachen University
[†]Siemens AG, Munich, Germany

ABSTRACT The modularization of domain-specific modeling languages (DSMLs) during language development is important for languages or language parts to be reusable. As languages and their components are created using different language workbenches or modeling tools, the definitions and concepts for such language components are often limited to their individual technological spaces. In reality, language development requires significant effort, and DSMLs are often built from scratch within a single technological space, with little consideration for the generalization of the development concepts to other technological spaces. In this article, we discuss common notions of language components that are valid across the textual and graphical technological spaces and provide definitions, concepts, and realization techniques to foster the reusability of such language parts during DSML development. To this end, we first individually define language components and the various forms of language composition in MontiCore, a textual language workbench, and MagicDraw, a graphical modeling tool, and then describe unified cross-cutting concepts of language components that are crucial in developing variants or families of similar languages valid in both the textual and the graphical technological spaces. These language components that are described regardless of their individual technological spaces ensure the reusability of common language parts, ultimately supporting language engineers in developing complex modular DSMLs in the large.

KEYWORDS Software Language Engineering, Language Components, Domain-Specific Modeling Languages, MontiCore, MagicDraw

1. Introduction

Domain-specific modeling languages (DSMLs) can be modularized into composable units that we refer to as *language components*. The modularization of languages serves several purposes, a central one being the reusability of languages or language parts. The definition and use of language components can be achieved with different language workbenches and modeling tools, while some explicitly support it. However, the technological spaces for DSMLs are heterogeneous. For example, some language workbenches, such as MontiCore (Hölldobler et al. 2021), enable defining languages with a textual syntax, while others, such as MagicDraw (MagicDraw Enterprise 2022), allow the creation of graphical languages (Gupta et al. 2021). Another difference may be how the language workbenches assign meaning to models (Harel & Rumpe 2004) of the language, which can be achieved, e.g., by interpreters or code generators. Some language workbenches check the well-formedness with context conditions specified in OCL-like notations or implementations of programming languages, while others support modeling of well-formed models by means of sophisticated model editors. Although the literature (Combemale et al. 2018; Butting & Wortmann 2021; Butting et al. 2018; Clark et al. 2015) proposed several definitions and notions of language components or synonyms of it, they are often tied to individual technological spaces. Various forms of language composition have been

JOT reference format:
Arvid Butting, Rohit Gupta, Nico Jansen, Nikolaus Regnat, and Bernhard Rumpe. *Towards Modular Development of Reusable Language Components for Domain-Specific Modeling Languages in the MagicDraw and MontiCore Ecosystems*. Journal of Object Technology. Vol. 22, No. 1, 2023. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2023.22.1.a4

discussed by (Erdweg et al. 2012) in the textual technological space. Notions of language extension, language restriction, self-extension, language unification, and extension composition have been detailed in their work. The concepts presented in our article build on this related work by describing language composition mechanisms in MontiCore and extending these notions to the graphical technological space of MagicDraw. Further, we also present a unified approach to language composition in the technological spaces of textual and graphical languages.

Today, it is still challenging to reuse languages properly. Often, reusing a language requires more effort than implementing a new language or reusing languages via clone-and-own (Méndez-Acuña et al. 2016; Şutîi et al. 2018). Furthermore, language reuse can mostly be achieved in the same technological space (CKM+18 2018). Thus, a common notion for language components can serve as the foundation for language reuse across different technological spaces. The contributions of this paper are:

- Individual definitions for language components in MontiCore and MagicDraw, which are generalized to other tools capable of defining textual and graphical languages.
- An investigation of similarities and differences in the two notions of language components and language composition mechanisms in MagicDraw and MontiCore.
- A joint definition of language components in textual and graphical-based technological spaces.

The remainder of this paper is structured as follows: section 2 introduces the technological spaces that this paper covers before section 3 describes a running example that we use to demonstrate language components in MontiCore and MagicDraw. The general notion of language components is explained in section 4, while section 5 and section 6 explain its application in the frameworks MontiCore and MagicDraw. The comparison of the two approaches to form mutual definitions is elaborated in section 7, with section 8 discussing the mutual notions of language composition. In section 9, we discuss the related work and section 10 concludes.

2. Technological Spaces for Engineering Software Languages

Our work involves the analysis of different technological spaces and their concepts of modular language components. In this context, modularity denotes self-contained units that can be distributed (e.g., as archive artifacts). Such modules are developed and used independently of each other and have an interface for the features they provide and require. We investigate to what extent MagicDraw and MontiCore support the notion of modular language components to derive a generalized definition.

2.1. Engineering Software Languages in MontiCore

MontiCore (Hölldobler et al. 2021) is an open-source[1] language workbench for engineering textual DSMLs. The combined concrete and abstract syntax of a MontiCore DSML is described via

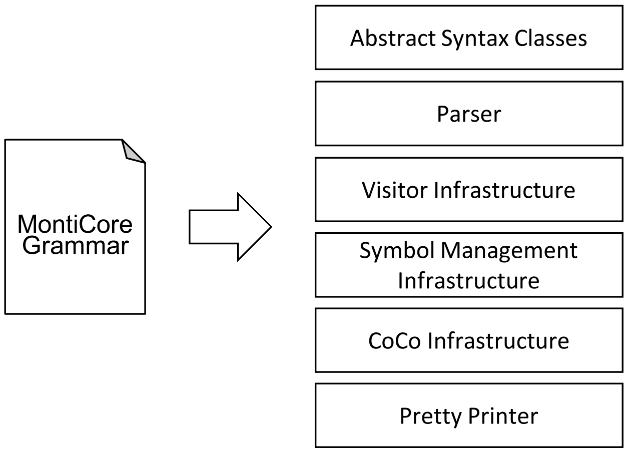


Figure 1 Conceptual overview of the language infrastructure MontiCore provides for an input grammar.

a MontiCore grammar, which comprises a custom, EBNF-based (Wirth 1996) notation for context-free grammars. From these grammars, MontiCore generates language infrastructure, such as a parser, an abstract syntax data structure, a visitor infrastructure for traversal of the abstract syntax, and a context condition (CoCo) infrastructure. The latter enables language engineers to define and check well-formedness rules in the form of context conditions realized as Java classes. Figure 1 depicts an overview of MontiCore and the language infrastructure it generates from a grammar. The abstract syntax data structure comprises classes for instantiating the abstract syntax tree (AST) when parsing a textual model. Additionally, MontiCore generates a symbol management infrastructure (SMI), providing a symbol table that interrelates with the AST and enables cross-referencing and quick navigation. The instances of the AST directly result from parsing, whereas the symbol table is instantiated by traversing the AST with the language's visitor.

The SMI is the foundation to realize type checks and different forms of language composition, as explained later in section 5. It further abstracts from the AST by representing only the essence of the language's abstract syntax. Despite that, it may introduce additional relationships between abstract syntax elements. While the AST is a tree data structure, a symbol table is a graph structure containing scopes and symbols. Symbols correspond to names defined by model elements, and scopes impact the visibility of the symbols. Conceptually, MontiCore demands that each symbol is contained in an (enclosing) scope and that the scopes are arranged in a tree shape. A scope may import and export symbols with regard to other scopes. There are two special kinds of scopes for each language: artifact scopes, describing the visibility of symbols for an entire model artifact, and the (singleton) global scope, describing the visibility of symbols among different model artifacts.

For performance reasons, the symbol table of a model can be persisted in a file. This speeds up type-checking for other models. Furthermore, it enables the decoupling of language infrastructures and, hence, supports language composition, as described later in this paper.

[1] MontiCore is available via: www.monticore.de

To give meaning to a model, MontiCore supports the realization of pretty printers that traverse an AST to translate it into source code conforming to another language. By default, MontiCore already provides a printer that translates an AST instance back into its textual representation. Alternatively, MontiCore has a built-in infrastructure to realize template-based code generation that is built around FreeMarker (Forsythe, Charles 2013). Based on this engine, MontiCore provides for facile adaptation of the generation process, including adaptive template exchange for language engineers as well as for advanced modelers.

2.2. Engineering Software Languages in MagicDraw

The engineering of graphical DSMLs is often tied to specific departments in a large organization. This introduces various challenges in the development of such DSMLs. The combination of using a modeling language with a methodology and using an appropriate graphical modeling tool is important in developing efficient graphical DSMLs. To this end, we have worked with a handful of commercial graphical modeling tools such as Enterprise Architect (Enterprise Architect 2022), Rational Rhapsody (IBM Rhapsody 2022), and MagicDraw (MagicDraw Enterprise 2022). We have focused on MagicDraw as a choice of graphical modeling tool as it supports the definition of popular modeling languages such as Unified Modeling Language (UML) and Systems Modeling Language (SysML). In addition, MagicDraw provides an extensive range of customization capabilities, including support for Java, that is subsequently used to capture all, if not most, aspects of domain-specific problems and project requirements. The customization capabilities of MagicDraw can be leveraged to create a language profile consisting of language component artifacts. One such example of an artifact is a language element, referred to as a *stereotype* in MagicDraw. *Customization* elements for each stereotype allow the definition of custom rules for the MagicDraw DSML. These custom rules define how model elements or diagrams can be created, which elements can be configured from a MagicDraw shortcut menu, and which specific properties of the elements are displayed to the user, among other customization. MagicDraw provides Open API Java-based plugin mechanisms that support the integration of the automation and creation of custom functionalities that are not supported by default. Further, model templates for individual languages allow for a predefined model structure to be automatically instantiated during modeling, reducing manual efforts in modeling and providing a good starting point for modelers. *Perspectives* in MagicDraw help language engineers configure the visible number of tool or DSML functionalities that different kinds of users require. Novice users may require fewer functionalities for more focused and guided modeling, while advanced users may need extra functionalities of MagicDraw or the DSML to fully realize their modeling with the combination of the tool and the modeling language.

A graphical DSML in MagicDraw is defined using: (1) an abstract syntax that defines the structure of its models, e.g., in the form of class diagrams; (2) a graphical concrete syntax that defines how the models are presented; (3) semantics, in the sense of meaning (Harel & Rumpe 2004); and (4) context conditions in Java, to check the well-formedness of the language. Naturally, as with any other software, graphical DSMLs are also subject to the usual challenges faced in software and systems engineering. Further, every DSML should, in all completeness, include the functional and non-functional aspects of the project requirements. As domains become more complex and heterogeneous, the complexity of the language syntax and semantics grows, thus requiring the need to integrate a suitable methodology for providing a good modeling experience. Especially in industrial DSMLs, novice users who are introduced to graphical modeling tools require special training and guidance for efficient modeling.

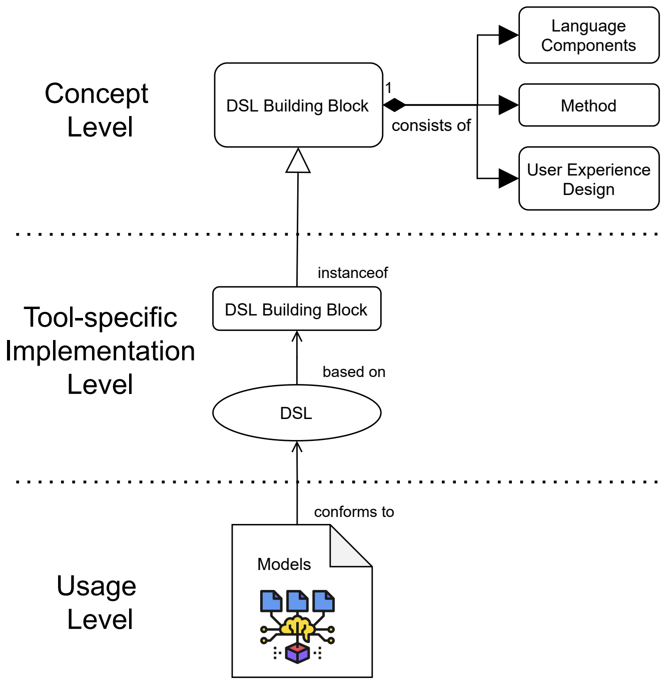


Figure 2 A conceptual model for the development and usage of a graphical DSML describing the different levels in the engineering process, including defining the language components, suitable methods for the language, and user experience design decisions to improve the overall modeling experience for DSML users (Gupta et al. 2022b).

A systematic process of engineering industrial DSMLs using modular and reusable DSL Building Blocks in MagicDraw is described in (Gupta et al. 2021). Figure 2 describes parts of the systematic process by separating the concerns of industrial engineering and deployment of the DSMLs on the following levels: (1) Concept level, where language engineers define: (i) the reusable language components that, completely or in part, defines the language (Rumpe 2016); (ii) the method, describing a methodology for the DSML ultimately intended to guide users in their modeling; and (iii) the user experience design (UXD) decisions, where standards and usability heuristics are defined to improve the overall user experience (Gupta et al. 2022a); (2) Tool-specific implementation level: where language engineers realize the aspects described in the concept level using a modeling tool such as MagicDraw; and (3) Usage level: where modelers design their models using the DSML and their mod-

```
01 grammar UseCaseDSL extends MCBasics {
02   symbol scope UCDiagram =
03     "usecasediagram" Name "{" UseCase* "}" ;
04
05   symbol UseCase =
06     "case" Name ("extends" ext:Name@UseCase)? ";";
07 }
```

Figure 3 MontiCore grammar for the use case diagram modeling language.

eling tool of choice. Ultimately, these reusable DSL Building Blocks are composed together to create the resulting DSML that consists of various heterogeneous domain constructs. In the scope of this paper, we discuss language composition on the level of reusable language components for such DSL Building Blocks. The method and UXD decisions are considered independent of composition aspects, meaning they either compose naturally or must be configured later during DSML engineering.

MagicDraw provides capabilities to define plugins that are ultimately bundled together into an archive *.mdzip* file and installed as a DSML. These plugins consist of a MagicDraw project containing the language definition, predefined templates for creating new models, perspectives to limit the functionalities of the DSML or the tool, and custom Java extensions that enhance the existing capabilities of the resulting DSML. Standard modeling language constructs for UML and SysML are bundled together as artifacts within the DSML that users can simply install and use. These capabilities of MagicDraw as a graphical tooling environment make it a good fit for realizing the different forms of language composition and make it applicable to comparable frameworks, with the underlying principle of graphical modeling remaining the same: language development.

3. Running Example

To compare the language composition mechanisms of the different technological spaces, we create two basic languages in MagicDraw and MontiCore, which can be composed together in different ways. First, we create a language similar to UML use case diagrams for modeling simple use cases and their relationships, UseCaseDSL. Second, we provide an ActorDSL in which actuators and their tasks can be represented. These two DSLs are standalone modeling languages, thus representing entirely separate language components. Moreover, their composition offers additional distinct modeling techniques. In the following, we describe these languages in detail and elaborate on their implementation in MagicDraw and MontiCore.

3.1. Use Case Language

Models of UseCaseDSL should be able to represent individual use cases within a corresponding diagram. Figure 3 shows the context-free grammar in EBNF notation (Wirth 1996), representing the realization within MontiCore. It contains production rules, defining nonterminals on the left side, with terminals or references to other nonterminals on the right side, separated by an equals sign. Furthermore, nonterminals can be augmented with stereotypes, such as `symbol` or `scope`, indicating unique access via their name, enabling structuring and cross-referencing in the symbol table. The grammar defines the overall diagram (ll. 2-3), starting with a respective keyword, identified via a name, and containing an arbitrary number of use cases. These use cases also have a unique name, as they are defined as `symbols`, and may extend other use cases by referencing these via their name (ll. 5-6).

To realize the UseCaseDSL in MagicDraw, we define a corresponding metamodel (cf. Figure 4). Again, a language's diagram contains an arbitrary number of use cases that, in turn, have a name (here represented by the String attribute `attribute1`). However, the MagicDraw implementation adds two additional versions of use cases, `MyFirstUseCase` and `MySecondUseCase`, with the second one explicitly prohibiting any relationships between these diagram elements. An example of restricting such relations is in the healthcare domain, where it would be advisable to impart training to novice modelers for modeling different parts of an X-ray system with a rather limited set of DSML constructs and not burdening such modelers with a large number of interlinked relations to other constructs. Thus, we achieve similar languages in MontiCore and MagicDraw, primarily distinguished in their textual and graphical representation, respectively.

3.2. Actor Task Language

The ActorDSL enables modeling diagrams with different actors, tasks, and their corresponding relations. Figure 5 presents the textual grammar, defining the language within MontiCore. A model (ll. 2-4) starts with a set of import statements, followed by the actual diagram definition containing the respective keyword, a name, and an arbitrary number of diagram elements (cf. `RDElement`) in curly brackets. The import statements allow referencing and thus importing other models and their features, facilitating the composition of different artifacts. `RDElement` is defined as an interface nonterminal, enabling the grouping of several nonterminals. Thus, each nonterminal that `implements` this interface (cf. `Actor` l. 8) can be employed where the general `RDElement` is referenced. This mechanism grants greater flexibility adopted from object-oriented programming. It also offers an explicit hook point intended for extension in inher-

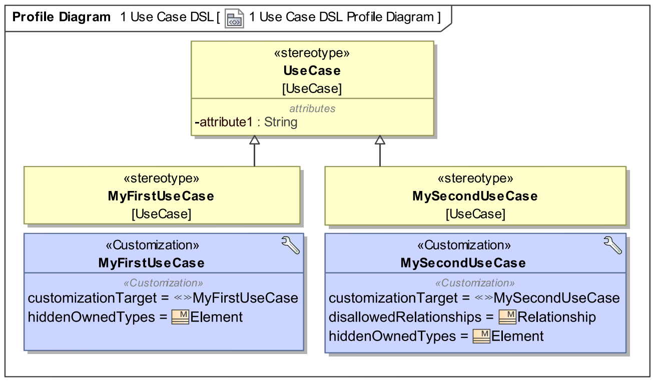

Profile Diagram 1 Use Case DSL [1 Use Case DSL Profile Diagram]
«stereotype»
UseCase
[UseCase]
attributes
-attribute1 : String
«stereotype»
MyFirstUseCase
[UseCase]
«stereotype»
MySecondUseCase
[UseCase]
«Customization»
MyFirstUseCase
«Customization»
customizationTarget = «»MyFirstUseCase
hiddenOwnedTypes = Element
«Customization»
MySecondUseCase
«Customization»
customizationTarget = «»MySecondUseCase
disallowedRelationships = Relationship
hiddenOwnedTypes = Element

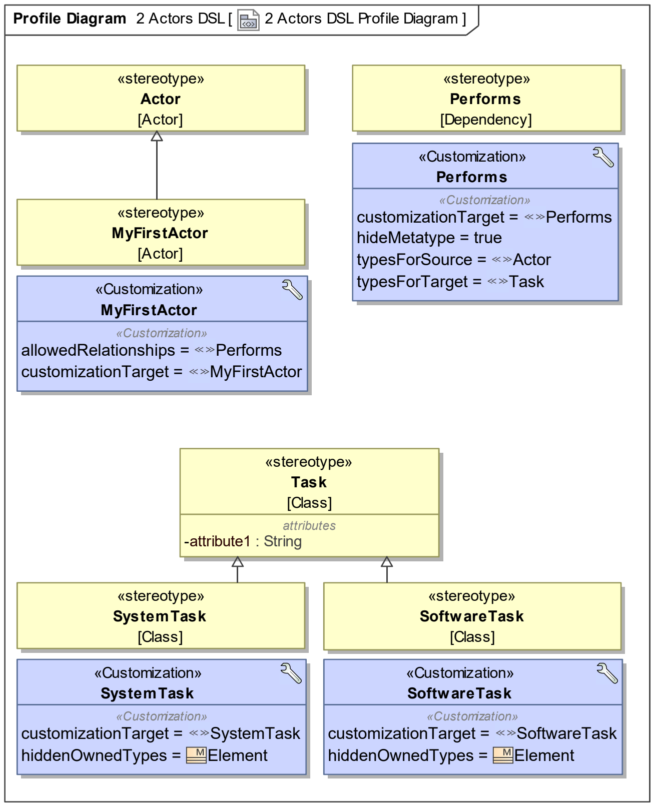
Figure 4 MagicDraw metamodel for the use case diagram modeling language.

```
01 grammar ActorDSL extends MCBasicTypes {                          MG
02   symbol scope RoleDiagram =
03     MCImportStatement* "rolediagram" Name
04     "{" RDElement* "}" ;
05
06   interface RDElement;
07
08   symbol Actor implements RDElement = "actor" Name
09    ("extends" sup:Name@Actor)? ";" ;
10
11   symbol Task implements RDElement = "task" Name ";";
12
13   Relation implements RDElement =
14     actor:Name@Actor "->" task:Name@Task ";" ;
15 }
```

Figure 5 MontiCore grammar for the actor task modeling language.

iting languages. The diagram itself can contain actors, tasks, and relations. An actor (ll. 8f.) has a name and can extend other actors by referencing them via their name. Tasks (l. 11) start with a corresponding keyword and are also identified via their unique name. Finally, relations (ll. 13f.) associate actors with their tasks, utilizing their property as a symbol for unique cross-referencing access via their names.

For implementing the ActorDSL in MagicDraw, we create a metamodel, as shown in Figure 6. Similar to the production rules in MontiCore, there are corresponding stereotype definitions for the main diagram elements, i.e., `Actor`, `Task`, and `Performs`. These either have direct customizations specifying their contents or are extended by more specialized variants. For example, the `MyFirstActor` stereotype extends the general actor and specifies that it can be connected via the `Performs` relationship. While we only have one specialized actor, we have established a hook point via the general stereotype for easy extension. A task has a name represented via its String attribute `attribute1`. Furthermore, it is decomposed into two specialized tasks, allowing further distinction. Finally, the metamodel contains the `Performs` relation, indicating the connection between tasks and actors. As the definition of the relation refers to the general stereotypes of the linked elements, the relationship also holds for each specialized actor or task.



Profile Diagram 2 Actors DSL [2 Actors DSL Profile Diagram]
«stereotype» Actor [Actor]
«stereotype» Performs [Dependency]
«Customization» Performs
«Customization»
customizationTarget = «»Performs
hideMetatype = true
typesForSource = «»Actor
typesForTarget = «»Task
«stereotype» MyFirstActor [Actor]
«Customization» MyFirstActor
«Customization»
allowedRelationships = «»Performs
customizationTarget = «»MyFirstActor
«stereotype» Task [Class]
attributes
-attribute1 : String
«stereotype» SystemTask [Class]
«stereotype» SoftwareTask [Class]
«Customization» SystemTask
«Customization»
customizationTarget = «»SystemTask
hiddenOwnedTypes = Element
«Customization» SoftwareTask
«Customization»
customizationTarget = «»SoftwareTask
hiddenOwnedTypes = Element

Figure 6 MagicDraw metamodel for the actor task modeling language.

4. Language Components and General Composition Techniques

4.1. Requirements for Language Components

Language components, in general, should support the flexible reuse of (potentially incomplete) languages for their composition (Butting 2023). Their purpose is bundling modeling languages to form self-contained, logical units that can be included using various language composition techniques. An important feature is that language components are developed independently of each other, although potentially designed with extension in mind. They foster building a library of languages (and their constituents) that are eventually beneficial by eliminating the need for defining languages completely from scratch. Thus, a language component must comprise all artifacts relevant for describing the concrete and abstract syntax as well as additional validation and tooling, such as well-formedness rules, symbol management, and syntheses via pretty-printing, code generation, or interpretation.

4.2. Properties

In essence, the composition of languages is not a property of the languages: this implies any two languages can be composed with new syntax and semantics for the composed language (Erdweg et al. 2012). Instead, the composition of languages is a property of language definitions and, by implication, of their language components, meaning two language components can work together without alteration towards a common goal. A language component may itself be dedicated to composition such that it may be used to create a variant of a standalone language. Language components must be defined precisely but still be flexible to permit extensions and adaptations in their definition for easily composing new languages or variants of languages. A language component should be capable of providing interfaces that provide hook points for other language components. This could be via a syntactic interface connecting syntax, variables, and names via inter-language cross-referencing or a purely technical interface that connects the behavior of two language components. Finally, the actual language composition is deferred to a later binding point, which means all the language component artifacts can be generated independently, thus separating the concerns of DSML engineering.

4.3. Language Composition

Integrating language components into sophisticated DSMLs requires composing their corresponding constituents. Over the years, various composition techniques have emerged to tackle this challenge. While their terminology slightly varies in different technological spaces, modern language workbenches (Erdweg et al. 2013) support (at least some of) the following forms of language composition.

4.3.1. Language Inheritance A language inherits from one or more other languages if its definition (e.g., a grammar or metamodel) inherits from their respective specifications. If a definition inherits from another definition, it can reuse, extend, and override all its constituents. Overriding means redefining an existing nonterminal from the inherited language and, thus, specifying a new concrete or abstract syntax for it. Furthermore, the language can either reuse the start rule from the inherited languages or specify a new one. The other parts of the language infrastructure realize language inheritance individually. Most parts generated should be compositional and follow the inheritance of the languages. For instance, generated abstract syntax types of a language inherit from the abstract syntax classes of inherited languages. Providing the artifacts of an inherited language must also incorporate handwritten tooling against the generated infrastructure, such as well-formedness rules. As language inheritance enables the reuse of all inherited nonterminals, the validation rules implemented against inherited nonterminals must be reusable in an inheriting language. These conditions of inherited languages can then be checked against the AST of a language.

4.3.2. Language Extension In language extension, a language adds novel parts to a reused language. Such language extensions are considered *conservative*, i.e., each model of the original language is still a valid model of a language that extends the original language. Thus, extension is just a specific (more restrictive) form of inheritance, adding further constructs to a language while respecting existing ones.

4.3.3. Language Embedding Language embedding describes that a language E is embedded into a host language H without the need for the languages to be aware of one another. It can be compared with multiple language inheritance controlled via a novel language definition that inherits from the definitions of E and H (and potentially more). Usually, the novel language reuses the start rule of the language H. At some point, a nonterminal of the language H is either overridden, extended, or implemented (in case the nonterminal is an interface nonterminal) to add novel language syntax obtained from the language E. This integrates the syntax of the two languages at a single point. On the level of well-formedness, the sets of rules of the two languages, H and E, can be unified.

4.3.4. Language Aggregation Language aggregation differs from the other forms of language composition in the fact that aggregation does not require integrating the models of the languages in a single artifact. Instead, the models remain individual, and there is only a loose coupling between the involved languages. The coupling enables modelers to refer to elements of distinct models of an aggregated language. This requires a cross-referencing mechanism on the model level via unique names or other object identifiers. In summary, inheritance, extension, and embedding produce integrated models, whereas, in aggregation, models remain separated.

5. Realization of Language Components in MontiCore

5.1. Language Component Models in MontiCore

A language component in MontiCore is a (potentially incomplete) language definition particularly designed for being composed with other languages. They comprise a context-free grammar for defining the partial sentences of the corresponding component (Hölldobler et al. 2021). Language components are designed to define hook points and fill these of other languages. A hook point serves as an extension point, which can be enriched with syntax constituents of the overall language in a composed scenario. Here, productions marked as "external" serve as hook points to be filled, while default nonterminals are provided features. Interface nonterminals can serve as both, as they provide information but can additionally be externally extended. While each extending grammar can add further productions and new nonterminals, predefined hook points have the additional advantage that they allow for late binding. Thus, language components can provide productions referencing nonterminals via their interfaces, which inheriting grammars can seamlessly integrate. Besides the grammar, a language component can comprise additional infrastructures, such as context conditions for setting up well-formedness rules, the symbol table constituents for the specified symbols, and code generation functionalities.

A language component in MontiCore (Butting & Wortmann 2021) is defined as follows:

Definition 1 (Language component in MontiCore) *A language component in MontiCore is a reusable unit encapsulating a potentially incomplete language definition. A language definition comprises the realization of (concrete and abstract) syntax, well-formedness rules, and semantics of a software language.*

In other words, a language component in MontiCore contains all artifacts that are part of the realization of the language. By default, the language component definition does not distinguish different kinds of artifacts that contribute specific parts to the language, such as a grammar file or a context condition class. Related language component definitions (Butting, Eikermann, et al. 2020) often rely on such concrete artifact kinds (e.g., a grammar file) or describe a language in terms of the conceptual contributions that these artifacts make (e.g., abstract syntax). However, these definitions are often highly specific to a certain technological space and are still not detailed enough to properly describe language composition. Hence, our definition is rather general.

5.2. Realizing Language Composition in MontiCore

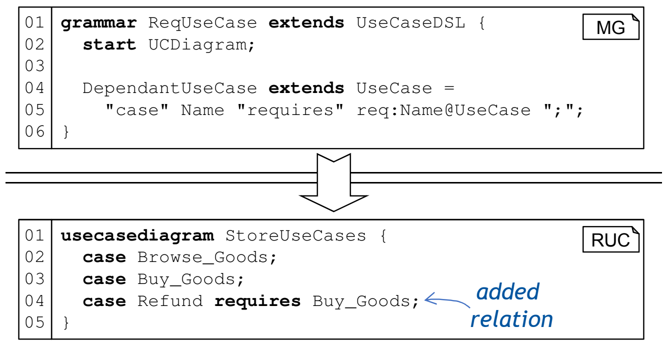
MontiCore supports various types of language composition, enabling the realization of those introduced in section 4. This offers distinct options for integrating models or their constituents.

5.2.1. Language Inheritance

The most basic composition technique is language inheritance. Here, a modeling language extends another, inheriting existing constructs. In MontiCore, the inheriting language adopts abstract and concrete syntax, well-formedness rules, and the generated infrastructure, allowing for reuse and extension. This way, existing language productions can be reused and new sentences added. Furthermore, the inheriting language can override existing productions and exchange the start rule.

5.2.2. Language Extension

Language extension uses the technical mechanism of inheritance to add new constituents to a language without altering the original definition. If a realization of language inheritance uses extension of nonterminals only and does not override any nonterminals, it does not remove any parts of the syntax because extension of a nonterminal in MontiCore adds novel right-hand sides to a given grammar rule for the nonterminal. Therefore, any language inheritance in which the inheriting language reuses the start rule of an inherited language and in which no nonterminals are overridden is a language extension. MontiCore does not distinguish language extension from language inheritance in all parts of the language infrastructure except for the grammar.



```
01 grammar ReqUseCase extends UseCaseDSL {
02   start UCDiagram;
03
04   DependantUseCase extends UseCase =
05     "case" Name "requires" req:Name@UseCase ";";
06 }
```

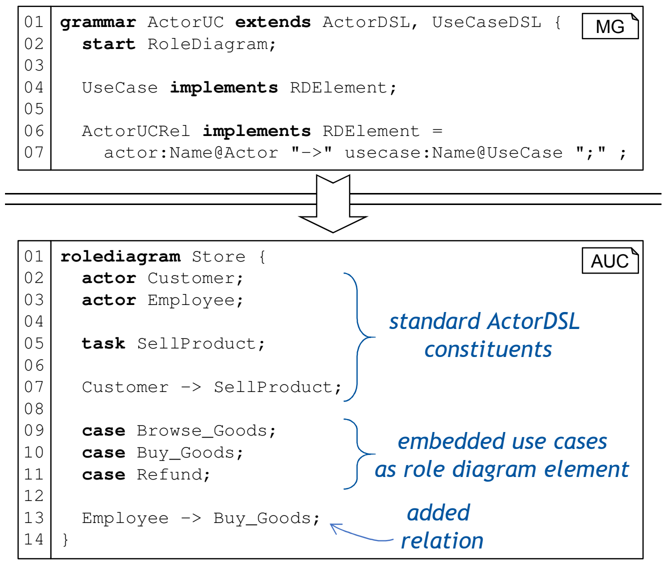
```
01 usecasediagram StoreUseCases {
02   case Browse_Goods;
03   case Buy_Goods;
04   case Refund requires Buy_Goods;
05 }
```

Figure 7 Extended UseCaseDSL for enabling additional relations between particular use cases.

Figure 7 (top) shows an example of conservative language extension. The grammar `ReqUseCase` extends the existing language definition of the `UseCaseDSL`. The starting nonterminal stays the same (l. 2). Additionally, we add a specific kind of use case (l. 4f.), extending the original one. At each position where such a use case could be used originally, a modeler can now use the new form as well, which adds a relation to previously defined use cases. Figure 7 (bottom) contains a corresponding model adhering to the extended language. The model represents simplified use cases for a store. The newly added production rule enables modeling use cases requiring other use cases. For instance, l. 4 describes that in order to get a `Refund`, the `Buy_Goods` case must also be involved in the corresponding transaction. Overall, language inheritance and extension are relatively straightforward ways to build new languages based on existing definitions. While extension requires each model of the original language to remain a valid instance of the newly created, inheritance is less restrictive and allows for arbitrary modifications. However, this could result in conflicts in the AST or the generated Java infrastructure, such as type or API clashes for overridden productions.

5.2.3. Language Embedding

Language embedding considers the integration of one or more existing languages into each other. Here, constructs of the embedded language components are accessible in a common language definition and suitably interwoven. In MontiCore, we realize language embedding via a common, unifying grammar. Similar to language inheritance, existing grammars are extended to leverage their productions in a new language. The main difference is that not only a single, but instead multiple grammars are extended, such that the sentences of the corresponding language definitions are jointly available. Thus, the particular model elements can be used together. An example is embedding expressions into an automaton language, where the expressions can function as guards for transitions.



```
01 grammar ActorUC extends ActorDSL, UseCaseDSL {
02   start RoleDiagram;
03
04   UseCase implements RDElement;
05
06   ActorUCRel implements RDElement =
07     actor:Name@Actor "->" usecase:Name@UseCase ";" ;
```

```
01 rolediagram Store {
02   actor Customer;
03   actor Employee;
04
05   task SellProduct;
06
07   Customer -> SellProduct;
08
09   case Browse_Goods;
10   case Buy_Goods;
11   case Refund;
12
13   Employee -> Buy_Goods;
14 }
```

MG
AUC
standard ActorDSL constituents
embedded use cases as role diagram element
added relation

Figure 8 Embedding the UseCaseDSL into the ActorDSL, allowing to relate actors with use cases.

Based on the running examples of the textual actor and use case modeling languages, Figure 8 (top) contains an example of how to embed one language into another. In this scenario, use cases are integrated into the `ActorDSL`, which, thus, reuses the `RoleDiagram` as starting nonterminal (l. 2). The constituents of a model are `RDElements` as prescribed in the `ActorDSL`. To further embed use cases, we let the corresponding nonterminal implement this respective interface (l. 4). This way, the definition of use cases remains unchanged, while they can be utilized the same as typical elements of the `ActorDSL`, resulting in a joint modeling opportunity. This already suffices to enable embedding constituents of both languages. To further establish a relation between the two languages' components, we can add the `ActorUCRel` (l. 6f.), allowing actors to refer to use cases similar to tasks. The bottom of Figure 8 presents a possible model containing elements of the original languages, `ActorDSL` (ll. 2-7) and `UseCaseDSL` (ll. 9-11), as well as the newly introduced relation (l. 13).

Language embedding significantly benefits from interfaces in the host language, allowing for implementation by the elements to be integrated. Thus, these interfaces serve as predefined hook points, facilitating extension and embedding. Without such explicit extension points, language embedding is also applicable but might require more overriding of existing productions.

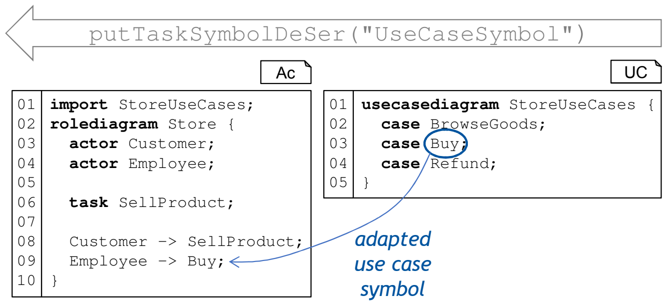
5.2.4. Language Aggregation An advanced composition technique is language aggregation. Here, the composed models stay separated artifacts while operating in a single, combined context. Model elements of these particular artifacts may refer to each other, mutually accessing or mutating each other's instances. In MontiCore, this is achieved by utilizing the symbol table. Symbols, as uniquely identifiable elements, can be resolved via their qualified names. Technically, language aggregation can be achieved in different forms:

Aggregation via shared grammar Language aggregation can be achieved by utilizing a shared grammar, specifying the symbols to define and refer to. Thus, all models adhering to this grammar share the same symbol kinds and symbol table structure. This enables exchanging symbols between the separated artifacts by referring to the corresponding constituents via their name. Furthermore, multiple, completely different, extending languages can reuse these symbol definitions and thus achieve an exchangeable information base between their independent models.

Aggregation via unifying grammar Alternatively, we can realize language aggregation via a unifying grammar. Similar to language embedding, the distinct constructs of the existing language definitions are made accessible to each other. While, in this case, the models' artifacts still remain separated, they technically belong to a common language definition that serves as the glue to handle all elements within the same context. In comparison to the previous approach, a newly created unifying grammar embeds both grammars of the languages to be aggregated. Thus, to enable mutual symbol usage of the separate worlds, the inheritance relation is the opposite of aggregation via a shared grammar that uses the fact that somewhere in the inheritance hierarchy, a common grammar is already extended in both languages.

Aggregation via resolvers A possibility to aggregate languages without associating them via a combining grammar is the configuration of resolvers. Resolvers are responsible for traversing the symbol table to derive symbols from their corresponding qualified names. MontiCore supports attaching adapters to these resolvers to link symbols of different kinds, interpreting these in the required context. Ultimately, these adapters serve as the glue between the particular modeling languages provided via hand-coded extensions for predefined hook points.

Aggregation via symbol files As introduced in subsection 2.1, the symbol tables of models of a language can be persisted in symbol table files. Such symbol table files enable decoupling the tools of the individual languages completely. The aggregation in this approach is achieved by translating or re-interpreting the symbol table files of an aggregated language as if they were symbol table files of their own language. To do so, MontiCore offers reconfiguration of the algorithms that achieve the deserialization of each symbol kind individually.



```
01 import StoreUseCases;
02 rolediagram Store {
03   actor Customer;
04   actor Employee;
05
06   task SellProduct;
07
08   Customer -> SellProduct;
09   Employee -> Buy;
10 }
```

```
01 usecasediagram StoreUseCases {
02   case BrowseGoods;
03   case Buy;
04   case Refund;
05 }
```

Figure 9 Aggregating the UseCaseDSL and the ActorDSL, adapting use cases as tasks.

Figure 9 shows how two models of the `ActorDSL` and the `UseCaseDSL` are composed using language aggregation. Similar to the embedding example, we make use cases accessible as tasks in models of the `ActorDSL`. However, in this case, there is no unifying grammar for introducing productions that provide the glue between the separate language definitions. Instead, they are connected via their symbol tables by adapting use case symbols to task symbols. For this purpose, MontiCore provides generated symbol deserializers that allow loading a stored symbol table into memory and simultaneously translating their symbol kinds. As tasks and use cases are similar in nature, the corresponding deserializer manages this mapping without any customization. However, a manual adaption might be necessary if the symbols are too different. In this scenario, a language engineer registers the use case symbol kind to be loaded as a task via the provided `putTaskSymbolDeSer()` method. MontiCore generates these methods automatically inside the scope classes for all symbol kinds known to a language.

Ultimately, the two models remain separated artifacts that refer to each other via their names. On the left-hand side, the role diagram `Store` imports (l. 1) the use case diagram `StoreUseCases`. Syntactically, it remains a pure `ActorDSL` model. However, due to the symbol-kind adaptation, it can also refer to use cases in its relations. Thus, we can model the link from the actor `Employee` to the use case `Buy` (l. 9), referring to an element of the other diagram. The reference is automatically resolved in a shared context.

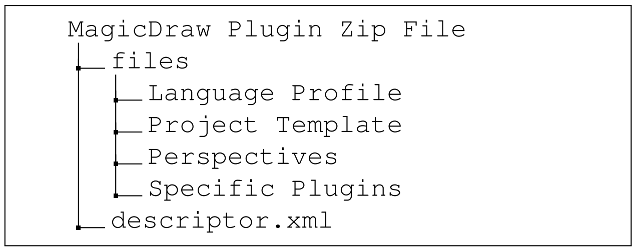


Figure 10 MagicDraw plugin hierarchy consisting of the software artifacts that define a language.

6. Realization of Language Components in MagicDraw

6.1. Language Component Models in MagicDraw

As discussed in subsection 2.2, one of the main functionalities of MagicDraw is the ability to provide editing capabilities for modeling languages using a customization engine. This allows MagicDraw to be adapted for a specific domain and to develop a language profile consisting of language components. We define a language component in MagicDraw as follows:

Definition 2 (Language component in MagicDraw) *A language component in MagicDraw is a reusable unit consisting of artifacts that, entirely or in part, encapsulate a (potentially incomplete) graphical language definition. A graphical language definition in MagicDraw consists of the abstract syntax, the graphical concrete syntax, well-formedness rules (context conditions), and the semantics of a software language.*

In other words, a language component in MagicDraw consists of the artifacts that are part of the realization of the graphical language. MagicDraw allows language designers to write and maintain graphical languages commonly used in industrial DSML engineering (Méndez-Acuña et al. 2016; Tolvanen & Kelly 2005). Moreover, it also provides a Java-based plugin mechanism that supports the integration of further automation and other custom functions. Figure 10 shows the software artifacts that are individually generated and bundled together in an archived plugin file that is subsequently installed in MagicDraw as a DSML. Once this plugin is installed, users can utilize the in-built hierarchical navigation tool, *Model Browser*, to manage their model data, including all language components, in a structured and organized manner. Software artifacts are realized in the form of a file system and are stored in a file system directory format as part of the compiled source code that is bundled and installed as a DSML plugin in MagicDraw.

Language Profile. The language profile of a MagicDraw plugin contains the artifacts of the language. These include the definition of language elements as *stereotypes* along with their configured properties, called *customizations*, and relations within a *UML profile*. Figure 4 shows an example of the UseCaseDSL stereotypes and their customizations. The `customizationTarget` property specifies to which stereotype the properties are applied. The `hiddenOwnedTypes` property specifies the list of stereotypes and metaclasses that will be hidden for this stereotype, meaning if it is configured to `Element`, no standard UML elements can be created for this DSML element. The language also contains artifacts for the configured tables, diagrams, and matrices that are exported as Extensible Markup Language (XML) files and bundled into the plugin. Context conditions are defined in the language profile in the form of *validationRules* on language elements, which are located within an *activeValidationSuite*, meaning these validation rules, written in Java, are checked for errors during design time. The language profile defines these configurations to hide general UML or SysML elements, which are beneficial in domain-specific modeling.

Project Template. An important component of the MagicDraw DSML is the project template. The project template is essentially a customized project pattern that is instantiated during the creation of a new model and serves as a starting point in modeling. Whenever a new project in MagicDraw is created, the template automatically applies the predefined model structure on the graphical modeling canvas. Often, the lack of such templates introduces rigorous manual efforts in the creation of models in complex systems where language users need to design their models from scratch. Such a project template is not only helpful for both novice and advanced users, but it is also important in improving the overall user experience for such users. The project template is generated as a MagicDraw `.mdzip` file and is bundled into the MagicDraw plugin.

Perspectives. Perspectives are a way of displaying a fixed set of DSML constructs or tool functionalities. They are predefined and show functionalities depending on the kind of users, novice or advanced. By defining perspectives, users can choose a suitable user experience mode for their modeling. Advanced users may require a greater number of functionalities, while novice users might find fewer functionalities easier to work with during modeling. Perspectives are created as `.umd` files and bundled with the MagicDraw plugin. Some examples of functionalities that can be configured are: MagicDraw menu items, MagicDraw toolbars, DSML diagram toolbars, and shortcuts.

Specific Plugins. Plugins in MagicDraw are a way of adding new functionalities in MagicDraw to enhance a DSML. The main parts of a plugin include a directory containing the compiled Java files, packaged as `jar` files, an XML file that describes the purpose of the plugin, and any other dependent files that are required by the plugin. An integrated development environment (IDE) such as Eclipse or IntelliJ can be used to compile and package jar files. Plugins can be used for a variety of purposes. As part of defining context conditions, a *validation plugin* is created to support binary validation rules for language elements that are not supported by MagicDraw. These can include checking naming conventions or performing type checks on certain language elements. A *visualization plugin* can be programmed to dynamically change the color of a graphical model element during design time. All the plugins are combined and bundled into the MagicDraw plugin.

Descriptor XML File. A descriptor XML file is used to reference all the artifacts and their configurations to help MagicDraw seamlessly install the plugin file. During the installation of the plugin file, MagicDraw detects the sets of artifacts and copies them into its own installation directory. On tool startup,

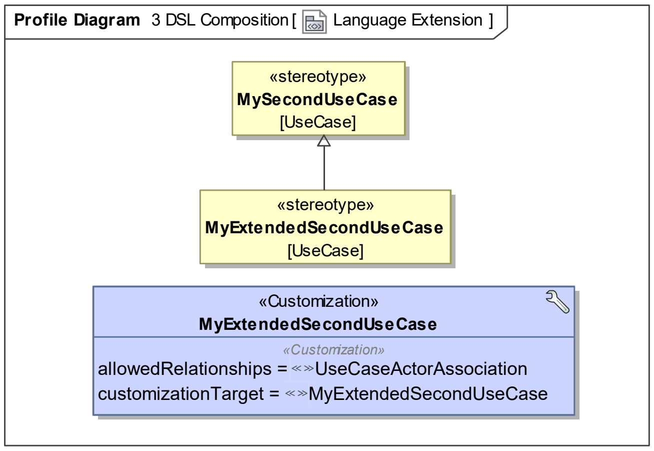

Profile Diagram 3 DSL Composition [Language Extension]
«stereotype»
MySecondUseCase
[UseCase]
«stereotype»
MyExtendedSecondUseCase
[UseCase]
«Customization»
MyExtendedSecondUseCase
«Customization»
allowedRelationships = «»UseCaseActorAssociation
customizationTarget = «»MyExtendedSecondUseCase

Figure 11 An example of the `MySecondUseCase` being extended and customized to `MyExtendedSecondUseCase` with an association dependency between a `UseCase` and an `Actor`.

MagicDraw loads all the necessary generated artifacts into memory so that the DSML is ready to use.

6.2. Realizing Language Composition in MagicDraw

All forms of language composition discussed in section 4 rely on the composition of the language components and, hence, the composition of their constituents themselves. In other words, a language in MagicDraw can be composed by composing the artifacts of the individual language components.

6.2.1. Language Inheritance Figure 4 of our running example shows the inheritance of language components for the UseCaseDSL. The `MyFirstUseCase` and `MySecondUseCase` stereotypes are defined as specific classifiers for the `UseCase` stereotype, which is a general classifier. Therefore, each instance of the specific classifiers naturally inherits the properties of the `UseCase` classifier. Newly configured attributes in the inherited stereotype with the same name as that of the base language element are distinguished internally using unique identifiers to eliminate problems of multiple inheritance (Butting, Eikermann, et al. 2020). For example, if an instance of `MyFirstUseCase` has an attribute named `attribute1`, then `attribute1` of `MyFirstUseCase` can be configured via the *Properties* tab of the *Specification* configuration window of the model element in MagicDraw. However, when a user wants to configure `attribute1` that belongs to an instance of `UseCase`, they would have to configure it in the *Tags* section under the *Specification* configuration window of the model element in MagicDraw. Context conditions specified on `UseCase` are valid for instances of the `MyFirstUseCase` and `MySecondUseCase` stereotypes.

6.2.2. Language Extension Figure 11 shows an example of language extension on a language component for the UseCaseDSL. The `MyExtendedSecondUseCase` stereotype extends the `MySecondUseCase` stereotype and inherits the `UseCase` metaclass. The metaclass for `MyExtendedSecondUseCase` can also be altered during an extension composition, but this could lead to undesired errors during modeling. The `MyExtendedSecondUseCase` stereotype is configured with an association relation between `UseCases` and `Actors`. However, since the `MySecondUseCase` stereotype disallows all relationships (Figure 4), the `UseCaseActorAssociation` is also naturally disallowed for all instances of the `MyExtendedSecondUseCase` stereotype. While restricting language elements is necessary for certain scenarios, the under-specification of these language elements is recommended to allow configuring relations in other variants of the UseCaseDSL. In our experience, instead of restricting all relations, allowing specific types of relations is far more beneficial in composing a language. Similar to language inheritance, context conditions are also naturally valid in language extensions.

6.2.3. Language Embedding Figure 12 shows an example of the configuration of a novel syntax in the composed language for embedding the ActorDSL into the UseCaseDSL. The UseCaseDSL and the ActorDSL are reused completely without change as read-only projects in MagicDraw, fostering the reusability of the individual languages and their components. The `UseCaseTask` is the novel syntax that inherits from both the host and embedded language's stereotypes, `UseCase` and `Task`. As a result of this inheritance, the `UseCaseTask` contains the metaclasses of both `UseCase` and `GeneralTask` stereotypes, `UseCase` and `Class`, respectively. `UseCaseTask` can, therefore, be either instantiated with a `UseCase` or a `Class`, depending on the user during model design. The `GeneralTask` stereotype is a specialization of the `Task` stereotype and, therefore, naturally inherits all its properties and features. The `UseCaseGeneralTaskGlue` attribute for the `UseCaseTask` stereotype acts as the integration glue and is configured by setting the property *showPropertiesWhenNotAppliedLimitedByElementType* value to the `UseCase` stereotype. This property ensures that the integration glue is visible to the `UseCase` stereotype. On the other hand, the `IsUseCaseAGeneralTask` attribute on the `GeneralTask` stereotype is configured to demonstrate that the integration glue (`UseCaseGeneralTaskGlue`) can only exist as a novel syntax, and the UseCaseDSL will be unaware of the ActorDSL constructs when only this attribute is set. Thus, any model elements of the `UseCase` stereotype cannot access the constructs of the ActorDSL without such an integration glue necessary for language embedding in MagicDraw.

Figure 13 illustrates language embedding using an example model containing the following model elements: *Customer* (`Actor`), *T1* (`Task`), and *UseCaseWithdraw* (`UseCase`). Initially, a *Customer* can only perform *T1*, whereas the *UseCaseWithdraw* is only configured to be a `UseCase` stereotype. The languages are reused completely unchanged, and *UseCaseWithdraw* is unaware of either *Customer* or *T1*, except only for being modeled on the same UML class diagram. Later, during modeling, the user decides that the *UseCaseWithdraw* description is simple enough for this use case to be considered a task. By configuration, the `UseCaseGeneralTaskGlue` attribute for *UseCaseWithdraw* is visible, and the user sets a value to it, e.g.,

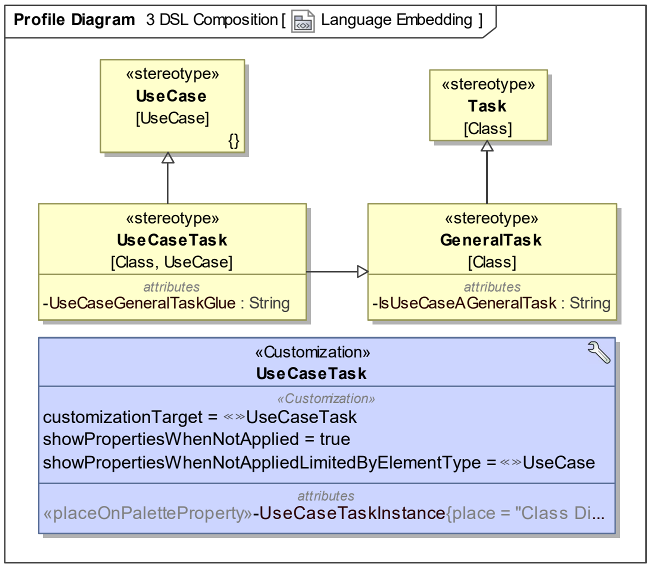

Profile Diagram 3 DSL Composition [Language Embedding]
«stereotype»
UseCase
[UseCase]
{}
«stereotype»
Task
[Class]
«stereotype»
UseCaseTask
[Class, UseCase]
attributes
-UseCaseGeneralTaskGlue : String
«stereotype»
GeneralTask
[Class]
attributes
-IsUseCaseAGeneralTask : String
«Customization»
UseCaseTask
«Customization»
customizationTarget = «»UseCaseTask
showPropertiesWhenNotApplied = true
showPropertiesWhenNotAppliedLimitedByElementType = «»UseCase
attributes
«placeOnPaletteProperty»-UseCaseTaskInstance{place = "Class Di...

Figure 12 The configuration for language embedding using a host UseCaseDSL, an embedded ActorDSL, and an integration glue (`UseCaseTask`) in MagicDraw. The `UseCaseTask` stereotype is the novel syntax that specifies the `UseCaseGeneralTaskGlue` attribute as the integration glue on the `UseCase` stereotype. The ActorDSL is embedded through the integration glue attribute of the `UseCaseTask` stereotype into the UseCaseDSL using MagicDraw's *showPropertiesWhenNotAppliedLimitedByElementType* property.

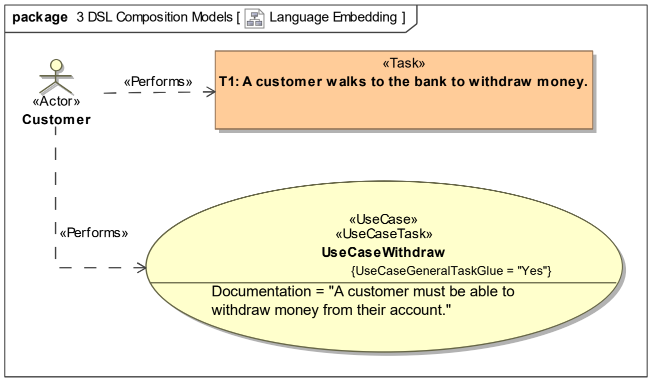

package 3 DSL Composition Models [Language Embedding]
«Actor»
Customer
«Performs»
«Task»
T1: A customer walks to the bank to withdraw money.
«Performs»
«UseCase»
«UseCaseTask»
UseCaseWithdraw
{UseCaseGeneralTaskGlue = "Yes"}
Documentation = "A customer must be able to withdraw money from their account."

Figure 13 An example of a model showing language embedding using an integration glue in MagicDraw. The attribute `UseCaseGeneralTaskGlue` acts as the integration glue between the ActorDSL and the UseCaseDSL. If this glue is missing, the model element *UseCaseWithdraw* is unaware of either the *Customer* or the `Task` *T1* model elements. Here, as the integration glue is set, *UseCaseWithdraw* is configured both with a `UseCase` and the `UseCaseTask` stereotype, therefore embedding the ActorDSL into the UseCaseDSL, allowing a *Customer* to now configure an outgoing *Performs* relationship to *UseCaseWithdraw*.

"Yes". This automatically adds the `UseCaseTask` stereotype to *UseCaseWithdraw* and embeds the ActorDSL constructs, meaning *UseCaseWithdraw* is now configured both as a use case and a task. The integration glue (`UseCaseGeneralTaskGlue`) creates the necessary interface between the UseCaseDSL and the ActorDSL, enabling the user to now set an outgoing *Performs* relationship from the *Customer* to the *UseCaseWithdraw*. On the class diagram, the model element *Customer* is shown with the respective relationships to both *T1* and *UseCaseWithdraw*. The setting of the integration glue ensures the ActorDSL is embedded into the UseCaseDSL. Without setting this integration glue, the individual ActorDSL and the UseCaseDSL remain independent and unaware of each other. New language infrastructure, such as the creation of novel context conditions, is valid in the composed language. The individual context conditions for the UseCaseDSL and the ActorDSL continue to remain valid independently. The introduction of new language infrastructure, therefore, does not affect the usage or the configuration of the individual languages.

6.2.4. Language Aggregation

Figure 14 shows an example of the configuration required to achieve language aggregation in MagicDraw. Similar to language embedding, we reuse the ActorDSL and the UseCaseDSL and their components in their entirety. Language aggregation in MagicDraw is realized by creating a UML dependency relationship along with a novel syntax. This ensures that language components are still decoupled, as each of the individual language components is reused without alterations. A composed language is created without any modifications to either the UseCaseDSL or the ActorDSL, as they are read-only projects in MagicDraw, and the respective models for each language are part of their individual artifacts. The `TaskBelongsToUseCase` dependency is created as a novel syntax in the composed language. The customization values for the properties *typesForSource* and *typesForTarget* define from which source stereotypes to which target stereotypes the dependencies are configurable. In our example, the source is configured to be of `Task` stereotype, and the target is configured to be of `UseCase` stereotype. This means an outgoing dependency from an instance of a `Task` to an instance of a `UseCase` can be created, but not the other way around. To achieve bidirectionality in the composed language, the properties *typesForSource* and *typesForTarget*, must be configured with both the `Task` and the `UseCase` stereotypes. This kind of configuration can also be created during the initial definition of the languages, meaning both the UseCaseDSL and the ActorDSL already exist, and a `TaskBelongsToUseCase` dependency is created to achieve the aggregation. In this scenario, a loose form of conceptual coupling exists between the languages, as certain constructs of either language may be known to the other language.

Figure 15 illustrates language aggregation using the same example model we described earlier in Figure 13 involving a *Customer* (`Actor`), *T1* (`Task`), and a *UseCaseWithdraw* (`UseCase`). Initially, the *Customer* can only perform *T1*, and the *UseCaseWithdraw* is configured to be a `UseCase` stereotype. The languages are reused unchanged, and *UseCaseWithdraw* is unaware of either the *Customer* or *T1*, except only for being modeled on the same UML class diagram. Later, the user

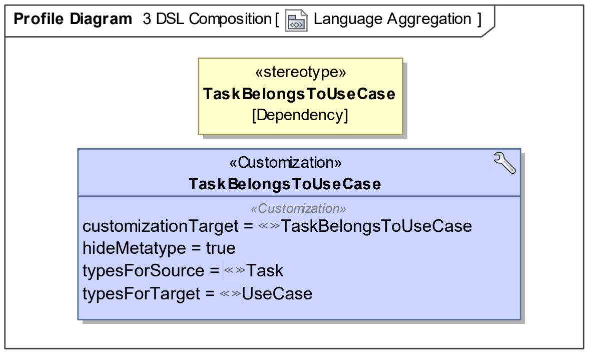


Figure 14 An example of the configuration for language composition using language aggregation in MagicDraw. The `TaskBelongsToUseCase` dependency is a novel syntax in the composed language. The *typesForSource* and *typesForTarget* property configurations in the customization of the novel syntax specify the direction of the dependency between the models of the ActorDSL and the UseCaseDSL. In this configuration, the dependency is configured to exist from a `Task` to a `UseCase` model element. Similar to language embedding, both languages are completely reused unchanged in language aggregation.

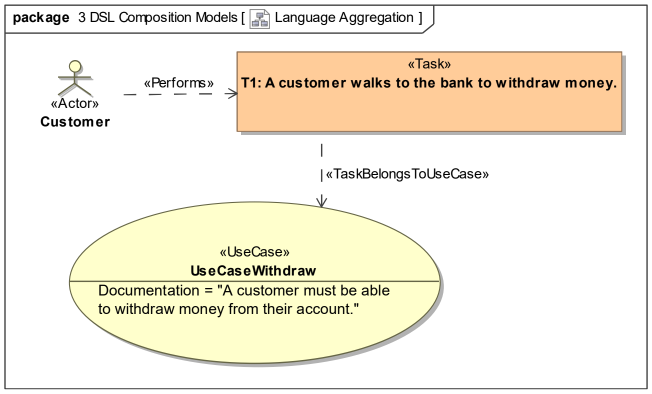

package 3 DSL Composition Models [Language Aggregation]
«Actor»
Customer
«Performs»
«Task»
T1: A customer walks to the bank to withdraw money.
«TaskBelongsToUseCase»
«UseCase»
UseCaseWithdraw
Documentation = "A customer must be able to withdraw money from their account."

Figure 15 An example of a MagicDraw model showing language aggregation using a dependency relationship. The `TaskBelongsToUseCase` dependency creates the necessary relation between task *T1* and the use case *UseCaseWithdraw*. When this dependency is not set, *UseCaseWithdraw* is completely unaware of either the *Customer* or *T1*. In this example, the direct dependency is only configured between *T1* and *UseCaseWithdraw*.

decides that task *T1* belongs to *UseCaseWithdraw*. As the `TaskBelongsToUseCase` dependency is already configured for *T1*, the user can create an outgoing dependency to indicate that *T1* belongs to *UseCaseWithdraw*. This dependency creates the necessary aggregation between the ActorDSL and the UseCaseDSL. As the `Actor` stereotype is configured with a `Performs` dependency to the `Task` stereotype, there exists no direct dependency between *UseCaseWithdraw* and the *Customer*. However, *UseCaseWithdraw* can trace back to the *Customer*, meaning the set of dependencies that are recorded between model elements in the MagicDraw can be used to derive the link between the *Customer* and *UseCaseWithdraw*. To directly connect *Customer* and *UseCaseWithdraw*, a new dependency must be created in the composed language. Without this dependency configuration, the languages are unaware of each other, and their respective models continue to remain in their individual artifacts. Similar to language embedding, new language infrastructure, such as the creation of novel context conditions on the composed language, can be created.

7. Unified Concept for Language Components

In section 5 and section 6, we discussed in detail the definitions of language components in their respective technological spaces of textual and graphical modeling. To achieve a seamless unification of the language component aspects between the two kinds of representation, it is important to consider all cross-cutting concerns. These concerns address the reusability of the definition of the language, both syntactically and semantically. We observe that the individual definitions of language components in MontiCore and MagicDraw are principally similar in nature. This is also supported by considerable similarities regarding the frameworks' abstract syntax synthesis, fostering equivalent composition techniques. A key difference in the definition of the language component across both technological spaces is the specific format of the software artifacts. While both definitions leverage software artifacts, MontiCore focuses primarily on the creation of grammar files, whereas in MagicDraw, the language is defined directly on the modeling tool, and subsequently, the artifacts are bundled into another artifact, the plugin file. To this end, we discuss the mutual definition of a language component independent of the technological spaces.

7.1. Mutual Definition of a Language Component

A language component consists of reusable units that encapsulate a language definition, partially or in whole. Further, a language component consists of software artifacts that contribute to the definition of a language. Examples of such software artifacts could be grammar files, e.g., in MontiCore, or files in a file system directory bundled together in the form of a plugin, e.g., in MagicDraw. Therefore, we define a language component independent of a technological space as follows:

Definition 3 (Language component) *A language component is a reusable unit encapsulating a potentially incomplete language definition valid for its respective technological space, including but not limited to textual or graphical representation. A language definition consists of reusable software artifacts comprising the realization of the syntax and semantics of a software language.*

In other words, the mutual definition of a language component does not distinguish the different software artifacts in different technological spaces that belong and contribute to the language definition. We consider realizing software artifacts in the context of the file system. Therefore, all the software artifacts are primarily located in a file system directory format as part of the compiled source code. Even though the description of the abstract or concrete syntax is realized differently in the different technological spaces, the underlying compiled source code is stored as files. Additional artifacts or applications (in some cases, the modeling tool or the language workbench itself) facilitate completing the language definition. Thus, they can also be categorized as parts of language components. For the actual composition of these language components in a unified manner, we consider the following mutual notions of language composition.

7.2. Mutual Notions of Language Composition

Aspects of mutual notions of language composition cover composition techniques described earlier in this paper (cf. section 5 and section 6). Additionally, certain characteristics and threats to validity of these mutual notions must be considered for language composition to be described independently of their technological spaces.

Language inheritance and extension are closely related. Both forms of language composition rely on inheritance concepts well-defined in the object-oriented programming world. Language inheritance should provide reuse, extension, and overriding of all language elements for a language definition, such as nonterminals in a textual language and stereotypes in a graphical language. Subsequently, in language extension, a composed language should add novel parts to a reused language. The main difference between these two variants is that with language inheritance, the models of the original language no longer need to be valid models of the resulting language definition. Language extension, on the other hand, conserves all original sentences (hence the notion of conservation extension) and only introduces new constituents.

In contrast to language inheritance and extension, which reuses a single language specification, language embedding reuses at least two languages, a host language, and an embedded language. Language embedding is achieved by a joint inheritance from the involved languages. This can be achieved by providing an integration glue, as novel syntax, in the composed language that extends the syntax of both the host language and the embedded language. Here, the embedded language is embedded into the host language at predefined *extension points*. An extension point is configured in the host language to provide for the extension for the constituents of the embedded language. In other words, embedding realizes or subsequently implements the extension point.

Similar to language embedding, language aggregation also composes multiple languages, with models remaining in their individual artifacts. While these languages can be reused in their entirety, in this form of language composition, adjustments need to be made to one of the existing languages so that one language can refer to the other language and access the visible constructs. This requires a re-configuration and a regeneration of the existing language infrastructure. A kind of reference must be established between the two languages, for instance, via a symbol table infrastructure in a textual technological space or via relationships in the form of associations in the graphical technological space. This reference establishes the interface between the accessible model elements of the languages. Therefore, in language aggregation, the involved languages exhibit a rather loose form of conceptual coupling.

The *reusability* of language components allows independent languages to be bundled together and foster building a library of languages. These are particularly helpful in developing complex variants of languages that are *flexible* enough to allow extensions and adaptations for composed languages. Some of the forms of language composition describe building language components in a way that they act as *interface providers* for allowing different realizations of a language. The *grouping* of individual language components ensures that a common combination of such language components can be reused directly for a language definition. Further, the realization of language components should be *under-specified*, wherever possible, to avoid problems of language restriction later on during extensions.

8. Discussion

In this paper, we present individual definitions of language components and the different forms of language composition in the technological spaces of textual and graphical modeling. Additionally, we also present a mutual definition of a language component and the mutual notions in language composition that is valid in both of these technological spaces. Defining and realizing such notions of language composition independent of technological spaces is a complex endeavor. While this study aims to provide a unified notion of language components, the realization of these concepts is certainly not the "go-to" solution. Instead, we aim to provide guidance for further work on language components and composition of languages that essentially targets existing and upcoming tools and language workbenches, including those that provide hybrid support for both textual and graphical modeling. Language engineers who compose languages or variants of languages should, therefore, be aware of all the concepts and decide on techniques that are best suited for their language development.

As observed from best practices in software engineering, monolithic definitions of large artifacts often lead to problems in the maintenance and reuse of existing assets that have been developed. To this extent, the introduction of reuse through modularity has gained traction in software language engineering (SLE) and component-based software engineering (CBSE) communities in recent years. Software components consist of interfaces providing the necessary interaction of the component to its environment (Broy et al. 1998). Our approach does not specifically proclaim explicit interfaces for language components, which are considered an overhead in developing, evolving, and maintaining them. Interfaces require the inclusion of all foreseeable points of extension, which is a hindrance to the reusability of language components or even parts of them. Further, they often lead to inconsistencies between the explicit interface and the underlying implementation. Instead, our intention is to describe the composition of languages through the introduction of novel syntax and reusing existing languages completely in addition to under-specifying language components so that an in-depth knowledge of the internals of a language component is not essential. Such internals, therefore, help in realizing language composition in a black-box manner, where only the essential and novel parts in a composed language are exposed and implemented by language engineers. While explicit interfaces can be beneficial in complex systems, our approach ensures that the language component definitions are less complex in nature, fostering independent reusability.

Definitions of language components consist of software artifacts. The only difference between the definitions in different technological spaces is the specific format in which the software artifacts are generated and stored. These artifacts are a result of the compiled source code during the definition of the DSMLs. We consider the modeling tool or the language workbench itself a language component, as it contributes to the definition of a language. However, the specific tools may be independent of consideration for aspects of language composition. All forms of language composition reuse at least one language in some way or the other. This fosters the composition of linguistic assets at a finer granularity level than the complete languages themselves (Bertolotti et al. 2023). While language inheritance and language extension reuse a single language, language embedding and language aggregation reuse at least two languages. Language embedding ensures that the embedded language is embedded into the host language at a single point, which makes the accessible constructs of the embedded language available to the host language. Language aggregation introduces novelty or adjustments in one of the existing languages to provide a kind of reference between the languages. This could be seen as a rather loose form of conceptual coupling. However, in all forms of language composition, the reusability of independent language components allows language engineers to build a library of languages that are flexible enough to build variants of languages easily.

In a hybrid technological space that uses a combination of textual and graphical representation, such as in projectional editing, the mutual notions of language composition are also applicable. MontiCore focuses on textual DSMLs and is widely used to derive proofs of concepts for various domains that make use of textual languages. Its concepts and results are not only used in academia but have been translated into successful domain-specific industrial projects. MagicDraw, on the other hand, focuses on graphical DSMLs, with consideration for many industrial scenarios, and therefore has a more practical approach for modeling scenarios with practitioners. This union of language composition aspects using these two technological tools creates an exhaustive complement of research in academia and its application in the industry. The results of this study show that language components and the different forms of language composition are applicable to other language workbenches and modeling tools that support language development. In particular, we have also completed a preliminary evaluation of these notions in Enterprise Architect, where the results comply with the concepts presented in this paper.

In this paper, we mutually define the concept of a language component that is valid for textual and graphical technological spaces. The underlying principle is that language components do not distinguish between the software artifacts that belong in different technological spaces. These software artifacts are realized in the context of a file system, which is implicitly present in a file system directory as part of the compiled source code. The forms of language composition are applicable to the language components as well as to the DSMLs. Language inheritance reuses a single language and provides mechanisms to reuse, extend, and override language elements. Language extension is achieved by adding novel parts to a reused language. The already constructed models of the original language continue to exist and remain valid in their respective original software artifacts. Language embedding reuses at least two languages but provides an integration glue, as a novel syntax, on the composed language that enables the embedding of an embedded language into a host language. This integration glue permits the extension of the host language with the constructs of the embedded language. In theory, an integration glue can be used to glue together completely incompatible components. A famous example of this is during the Apollo 13 mission, when the lunar module's supply of lithium hydroxide filters was exhausted, and the space crew had to create an adapter using items such as plastic bags and duct tape to reuse filters from the command module (Orloff et al. 2006). A corresponding example in software engineering is fostering compatibility using wrapper classes that encapsulate the functionality of other components (Arnold et al. 2000). However, in practice, an integration glue should be compatible between the grammars, in the textual space, and the graphical elements, in the graphical space, to prevent ambiguities during composition. Unifying grammars in MontiCore and novel integration glue, along with inheritance in MagicDraw, are ways to technically realize language embedding. Language aggregation also reuses at least two languages and needs adjustments on one of the existing languages, which requires sufficient reconfiguration and regeneration of the existing language infrastructure. Here, a kind of reference must be established in the respective technological space to provide a kind of interface for accessing constructs of the other language. Symbol tables in MontiCore and associations in MagicDraw are ways to technically realize language aggregation in textual and graphical modeling.

To achieve independent language composition, a library of language components is beneficial. Such language components should be reusable in any technological space: textual, graphical, or projectional. These language components should not be strictly specified, as it often leads to restrictions when composing a new language. For instance, disallowing all kinds of relationships in the original language leads to relationships being disallowed in the composed language as well. We discuss the realization of the concepts using a textual language workbench, MontiCore, and a graphical modeling tool, MagicDraw. Since all language workbenches serve the same primary purpose of language development and the concepts discussed in this paper are generalized, the overall notation of language components can be applied to other language workbenches or modeling tools. The running example languages of UseCaseDSL and ActorDSL cover aspects of language components such that they can be used to demonstrate the results of the study in real-world projects. However, as part of further work, we aim to validate the proposed realization techniques for other language workbenches and modeling tools.

We bring our perspectives on the aspects of language components and forms of language composition both from a researcher's and practitioner's point of view. As language engineers, we have used MontiCore and MagicDraw to compose modular languages in several domains over the years, which has subsequently been used extensively by a wide variety of users and domain experts. We argue in this paper that the concepts and realization of the language components and the various forms to compose languages are valid in both the textual and graphical technological spaces. We validated our study with a wide variety of language engineers, researchers, and practitioners from research groups as well as across many business lines within Siemens AG with different modeling needs. These researchers and practitioners have years of experience either in developing or using DSMLs that are suited for their specific domains, such as in the healthcare, energy, IT, or digital industries. We researched various methodologies to realize language components and the forms of language composition to provide an overview of methods, concepts, and their realizations, which is beneficial for all kinds of language engineers and users, both novice and advanced, and serves as an important step in developing component-based independent language composition techniques in the large.

Threats to Validity The validity of this study is the extent to which the language component definition and the forms of language composition, both in concept and in implementation, are free from systematic errors or bias. To this extent, we have explored the definition of a language component in two technological spaces, textual and graphical, and also in terms of the realization of these language components. Further, we discuss the various forms of language composition and how they are implemented in both these technological spaces. We discuss validity threats for our study and ways in which our conclusions might be wrong.

Among the available textual language workbenches and graphical modeling tools, our study is based on two well-known, actively developed, advanced, and OMG standard-compliant tools. MontiCore is an open-source language workbench for engineering textual DSMLs that is used in academia with its methods and concepts also translating into industrial applications (Hölldobler et al. 2021). MagicDraw, on the other hand, is a popular commercial modeling tool supporting UML and SysML standards and is widely used in the industry for developing graphical DSMLs and has support for many frameworks such as SysML, UAF, SoaML, and Enterprise Architect in the form of plugins (MagicDraw Enterprise 2022).

Previous work on compositional reusable language design (Butting, Eikermann, et al. 2020; Gupta et al. 2021) for both MontiCore and MagicDraw was evaluated. We note that the realization part of our study is currently tool-specific or vendor-locked. This could introduce unintended risks related to generalizing language composition aspects, as not all workbenches and tools provide a similar set of modeling concepts or functionalities. However, other language workbenches or tools can provide similar outcomes as in the study, given they provide means for language composition and introduce aspects for assembling language components. Our work is a first step towards a universal understanding of language components. A beneficial future work would be to compare the derived definition against other language workbenches and frameworks, such as MPS (Voelter & Pech 2012), Xtext (Efftinge & Völter 2006), or Spoofax (Kats & Visser 2010), to identify potential differences and further refine the definition.

To illustrate the foundational aspects of our study, we developed two example languages: UseCaseDSL and ActorDSL. Although these example languages are not very complex in nature, they are sufficient to cover the distinct language composition concepts described in this study. Further, these languages are commonly used for demonstrating proof of concepts in many DSMLs and are applicable to real-world projects as well (Tiwari & Gupta 2015). Both the example languages created for this study are based on UML and SysML constructs. The UseCaseDSL is used to create simplified UML-like use case diagrams used for modeling simple use cases and their relationships. The ActorDSL is used to create and model the actors and their tasks. Each of these languages has hook points that allow for easy extensions, as described in section 3. While each language is standalone and represents entirely separate domain concepts, the composition of both provides for the description of discrete modeling techniques.

We describe the concepts of language components and the forms of language composition in a way that, in each of the individual technological spaces, the findings support reusability. However, in a complex scenario with many domain concepts, language components from one domain may not always be completely reusable in another entirely independent domain. The definitions of the language components and the forms of language composition show that the individual definitions are similar and are applicable to both the textual and graphical technological spaces. A minor threat to generalizability remains concerning DSLs utilizing projectional editing. However, this threat can be neglected, as, from a language engineering perspective, projectional editing largely combines the concepts found in textual and graphical modeling analyzed in this paper. Additionally, we were able to replicate the results of the study across more complex and practical languages belonging to various domains, which illustrates that the concepts can be generalized. While some of these practical implementations are industry-specific, such as designing DSMLs for Siemens Healthineers and Siemens Digital Industry, other implementations were carried out as research for publicly funded projects with academic partners, such as SpesML (Gupta et al. 2022b). With constantly evolving workbenches and tools, it is important to ensure that the concepts of our study are validated and verified for other language workbenches and modeling tools as well. While our motivation for this study is derived from previously researched language composition aspects, we clarify that we are not the exclusive users of these workbenches, modeling tools, and the concepts described in this study.

9. Related Work

To easily develop, manage, and evolve complex systems (and software), it is important to build them using individual reusable units. Such units should capture all the details needed for the system to operate while also providing the necessary interfaces to allow the composition of larger languages. A modular approach for model-based systems engineering (MBSE) alleviates concerns for such complex systems (Herrmann et al. 2009). Compositional modeling aspects described in (Broy & Rumpe 2007; Rumpe & Wortmann 2018) detail modular aspects in interacting systems. Compositional approaches consider not only the modeling language but the models, their respective software components, and their artifacts as well (Talcott et al. 2021; Butting, Eikermann, et al. 2020). These approaches are well described in the textual technological space (Butting et al. 2021; Butting, Pfeiffer, et al. 2020; Hölldobler et al. 2018) with consideration for the different forms of language composition. (Völter & Visser 2010) discuss in their study the primary requirements and variation points in two established language workbenches, Spoofax and JetBrains Meta Programming System (MPS).

An important related work on language composition is described by (Erdweg et al. 2012) in their work on language composition mechanisms that is described for the textual technological space. In their paper, the described mechanisms are: "language extension, language restriction, language unification, self-extension, and extension composition". Their notion of "language extension" also requires the reuse of a base language completely unchanged, meaning the extension itself is not purposeful when considered independent from the base language. Such language extensions also subsume language restrictions that prohibit the use of certain language constructs. Their described notion of "language unification" matches our notion of language aggregation, where two independent languages are reused unchanged with the introduction of a novel syntax, which they refer to as *glue code*. In contrast, our notion of language embedding introduces an integration glue that helps embed one language completely unchanged into another language at predefined extension points. Thus, language embedding can be considered principally similar to their notion of "self-extension". However, in their definition of "self-extension", a language can be extended by the programs of the language itself through extension styles such as string embedding or pure embedding. MontiCore does not, by default, provide such a self-extension mechanism but instead provides the unification of two languages at a single point. The notion of "extension composition" through incremental extension is also supported by both MontiCore and MagicDraw. One distinction to their work is that our concepts of language composition are extended to the graphical technological space, which, to the best of our knowledge, is currently missing in the literature. This is primarily because reusing languages is still considered more effort than building completely new languages (Méndez-Acuña et al. 2016), and therefore, a conceptual framework for language composition in the graphical technological space would foster the modular development of reusable language components. Our work, thus, fundamentally builds on the mechanisms of language compositions described by (Erdweg et al. 2012) and presents a unified approach to language composition detailed in the two technological spaces of textual and graphical languages.

Language-related patterns are common in the literature (Spinellis 2001; Mernik et al. 2005; Drux, Jansen, & Rumpe 2022), but there is a lack of relations in identifying language components that can be reused with such patterns. Various studies (Şutîi et al. 2018; Vallecillo 2010) have also discussed focus on the reusability aspects of language implementations. (Cheng et al. 2015) discuss how specific forms of language composition can be restrictive, subsequently introducing challenges in language reuse.

A single definition of a language component that is valid in different technological spaces is still a relatively untouched research area. We observe that definitions of language components or forms of language composition have been exclusively described in the technological space of textual modeling (Butting et al. 2019; Haber et al. 2015). These studies describe the constituents of a language component as software (and hardware) artifacts that realize a system's functionality. The definition of a language component in this paper is similar to the one presented in (Clark et al. 2015), whereas our definition does not suggest that language components should have required and provided interfaces. (Leduc et al. 2020) in their study introduce the "language extension problem (LEP)", that allows defining a family of languages wherein a new language can be added by adding novel syntax or semantics. The LEP describes five constraints: "extensibility in both dimensions, strong static type safety, no modification or duplication, separate compilation, and independent extensibility", which are in some form or shape addressed by the concepts described in our work. Specifically, language components can be extended independently, new semantics can be introduced to the syntax, and existing semantics ensure it describes the complete syntax. Reusing language components and their artifacts ensures that modifications to the originals are unnecessary and components can be compiled separately.

The systematic engineering of languages in the textual space, including in practice, has been well-researched (Sprinkle et al. 2010; Krahn et al. 2006; Meyers et al. 2012). Similarly, the systematic development of languages in various domains in the graphical space has been studied for practitioners in industrial scenarios (Gupta et al. 2022b). The implementation of modularisation and composition concepts with MPS have been researched (Voelter 2011), and it is based on a projectional editor. Although possibilities to combine the development of languages in two technological spaces exist, e.g., integration of a textual syntax inside a graphical modeling tool (Drux, Jansen, Rumpe, & Schmalzing 2022; Seidewitz 2014), often the creation of such supplementary units is tedious, homogeneous, horizontal, or vertical and only provides a loose relation to our study. An example is the embedding of generated eclipse modeling framework (EMF) based textual editors and eclipse graphical modeling framework (GMF) based graphical editors with tree-based editors that are generated with EMF (Scheidgen 2008). Our study, however, provides mutual and common notions of language components and the different forms of language composition that are valid in both the textual and the graphical technological spaces. Other studies (Maro et al. 2015; Engelen & van den Brand 2010) propose ways to facilitate model transformations by providing both a textual and a graphical notation. On the other hand, we propose ways to compose languages using language components that can be exclusively used in the individual technological spaces, meaning the concepts can be interchangeably used in the textual and graphical spaces. In the projectional space, a model can be projected using various notations, including textual, graphical, or a combination of both (Voelter 2010). Therefore, the aspects described in our study also apply to this technological space.

Graphical modeling and UML tools such as Enterprise Architect (Enterprise Architect 2022), IBM Engineering Systems Design Rhapsody (IBM Rhapsody 2022), and MetaEdit+ (Tolvanen & Rossi 2003) have been evaluated using a variety of parameters intended to study the reusability of language components (Khaled 2009; Ozkaya 2019). Textual language workbenches such as MontiCore (Hölldobler et al. 2021), Spoofax (Kats & Visser 2010), EMFText (Heidenreich et al. 2009), and Xtext (Efftinge & Völter 2006) allow text-based modeling language development and have also been evaluated for modular language development (Erdweg et al. 2013).

In studies on pattern-based modeling (Bottoni et al. 2010), research approaches provide language-independent formalization of a pattern to assist in the composition and analysis of conflicts expressed in a textual form. The concept of reusability has also been discussed through generic model transformations (Sánchez Cuadrado et al. 2011) where templates are defined to map the concepts with their meta-models. Certain languages can also be defined through multiple levels of modeling. Such kinds of multi-level modeling paradigms (Lara et al. 2014) achieve separation of concerns across different stages in language development and usage of more complex languages. In their paper, (Butting et al. 2022) take a deeper look into composition aspects of a textual language, which can be described by referring to model element names in other languages using strongly kind-typed symbol tables. In the graphical space, (Gupta et al. 2021) provide a detailed look into how languages are developed in the industry for a variety of software and system domains and how the reusability of graphical language components is vital in reducing repetitive efforts needed to compose graphical DSMLs in the large.

10. Conclusions

As problems in various domains grow in complexity, so does the need for describing models in different technological spaces using concepts of reusability and modularization in the form of language components. Individual technological spaces present individual definitions and notions of language components, and their respective synonyms are often vendor-locked or tied to specific language workbenches or graphical modeling tools. Within this paper, we explore the individual definitions for the notion of language components in two technological spaces, textual and graphical, using language workbenches and modeling tools that support language development and language composition. We also investigate the similarities and differences between the concepts in these two technological spaces and provide mutual definitions and notions of language components and how they can be composed into complex languages. Our approach is described and validated in the technological spaces of textual and graphical modeling. In the textual space, we explored the composition of languages primarily using language aggregation approaches via a shared grammar, a unifying grammar, resolvers, and strongly kind-typed symbol table files. In the graphical space, we explored the composition of languages primarily using language embedding approaches by embedding an embedded language into a host language. To ensure that the concepts of language components and the described forms of composition are valid in both these technological spaces, we described mutual definitions and notions that ultimately address cross-cutting concerns of reusability and modularization. We realized language components in the textual space by defining a (potentially incomplete) language using a context-free grammar that is configured with a hook point, at which language components can be composed to develop a complex language. In the graphical space, we described these language components as UML or SysML stereotypes that are configured with extensions to ensure that language constructs can be embedded into a host language. The realization in these technological spaces has been demonstrated and validated with two simple yet commonly used language examples, use cases and actors that perform certain tasks. Using language components from different languages in different technological spaces to compose complex languages is interesting for both researchers and practitioners. Software artifacts should be easily transferable to the individual language workbenches or graphical modeling tools to prevent building a new language from scratch every time that consists of similar syntax and semantics, primarily to assist in software product line engineering. Further, this is beneficial in developing a family of languages with reusable units that alleviates any cross-cutting concerns. While the concepts have been described using MontiCore, a textual language workbench, and MagicDraw, a graphical modeling tool, the results of this study show that these concepts can be applied to language workbenches and modeling tools that support language development. As part of further work, we intend to extend and validate our approaches to other modeling tools and language workbenches to provide a more holistic way of describing the mutual notions of language composition. We believe describing notions of language components and language composition, in general, is beneficial to language engineers in composing complex languages or families of languages that are reusable and can be the foundation for language reuse across different technological spaces.

References

Arnold, K., Gosling, J., & Holmes, D. (2000). *The Java Programming Language* (3rd ed.). USA: Addison-Wesley Longman Publishing Co., Inc. Retrieved from https://dl.acm.org/doi/10.5555/556709 doi: 10.5555/556709

Bertolotti, F., Cazzola, W., & Favalli, L. (2023). On the granularity of linguistic reuse. *Journal of Systems and Soft-*

ware, *202*, 111704. Retrieved from https://doi.org/10.1016/j.jss.2023.111704 doi: 10.1016/j.jss.2023.111704

Bottoni, P., Guerra, E., & de Lara, J. (2010). A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Inf. Softw. Technol.*, *52*(8), 821–844. Retrieved from https://doi.org/10.1016/j.infsof.2010.03.005 doi: 10.1016/j.infsof.2010.03.005

Broy, M., Huber, F., Paech, B., Rumpe, B., & Spies, K. (1998). Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)* (p. 43-68). Springer. Retrieved from https://doi.org/10.1007/10692867_2 doi: 10.1007/10692867_2

Broy, M., & Rumpe, B. (2007, Februar). Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, *30*(1), 3-18. Retrieved from https://doi.org/10.1007/s00287-006-0124-6 doi: 10.1007/s00287-006-0124-6

Butting, A. (2023). *Systematic Composition of Language Components in MontiCore*. Shaker Verlag. Retrieved from http://www.se-rwth.de/phdtheses/Diss-Butting-Systematic-Composition-of-Language-Components-in-MontiCore.pdf

Butting, A., Eikermann, R., Hölldobler, K., Jansen, N., Rumpe, B., & Wortmann, A. (2020, October). A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology*, *19*(3), 3:1-16. Retrieved from https://doi.org/10.5381/jot.2020.19.3.a4 doi: 10.5381/jot.2020.19.3.a4

Butting, A., Eikermann, R., Kautz, O., Rumpe, B., & Wortmann, A. (2018, September). Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM. Retrieved from https://doi.org/10.1145/3233027.3233037 doi: 10.1145/3233027

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

Butting, A., Hölldobler, K., Rumpe, B., & Wortmann, A. (2021, July). Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen (Ed.), *Composing Model-Based Analysis Tools* (p. 217-234). Springer. Retrieved from https://doi.org/10.1007/978-3-030-81915-6_10 doi: 10.1007/978-3-030-81915-6_10

Butting, A., Michael, J., & Rumpe, B. (2022, October). Language Composition via Kind-Typed Symbol Tables. *Journal of Object Technology*, *21*, 4:1-13. Retrieved from http://dx.doi.org/10.5381/jot.2022.21.4.a5 doi: 10.5381/jot.2022.21.4.a5

Butting, A., Pfeiffer, J., Rumpe, B., & Wortmann, A. (2020, October). A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (p. 35-46). ACM. Retrieved from https://doi.org/10.1145/3365438.3410934 doi: 10.1145/3365438

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

Cheng, B. H. C., Combemale, B., France, R. B., Jézéquel, J.-M., & Rumpe, B. (Eds.). (2015). *Globalizing Domain-Specific Languages*. Springer. Retrieved from https://doi.org/10.1007/978-3-319-26172-0 doi: 10.1007/978-3-319-26172-0

Clark, T., Brand, M. v. d., Combemale, B., & Rumpe, B. (2015). Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages* (p. 7-20). Springer. Retrieved from https://doi.org/10.1007/978-3-319-26172-0_2 doi: 10.1007/978-3-319-26172-0_2

Combemale, B., Kienzle, J., Mussbacher, G., Barais, O., Bousse, E., Cazzola, W., ... Wortmann, A. (2018). Concern-oriented language development (COLD): Fostering reuse in language engineering. *Comput. Lang. Syst. Struct.*, *54*, 139–155. Retrieved from https://doi.org/10.1016/j.cl.2018.05.004 doi: 10.1016/j.cl.2018.05.004

Combemale, B., Kienzle, J., Mussbacher, G., Barais, O., Bousse, E., Cazzola, W., ... Wortmann, A. (2018). Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, *54*, 139 - 155. Retrieved from http://www.se-rwth.de/publications/Concern-Oriented-Language-Development-COLD-Fostering-Reuse-in-Language-Engineering.pdf

Drux, F., Jansen, N., & Rumpe, B. (2022, October). A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology*, *21*(4), 4:1-13. Retrieved from https://doi.org/10.5381/jot.2022.21.4.a4 doi: 10.5381/jot.2022.21.4.a4

Drux, F., Jansen, N., Rumpe, B., & Schmalzing, D. (2022, June). Embedding Textual Languages in MagicDraw. In *Modellierung 2022 Satellite Events* (p. 32-43). Gesellschaft für Informatik e.V. Retrieved from https://doi.org/10.18420/modellierung2022ws-006 doi: 10.18420/modellierung2022ws-006

Efftinge, S., & Völter, M. (2006). oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit* (Vol. 32). Retrieved from http://voelter.de/data/workshops/EfftingeVoelterEclipseSummit.pdf

Engelen, L., & van den Brand, M. (2010). Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science*, *253*(7), 105–120. Retrieved from https://doi.org/10.1016/j.entcs.2010.08.035 doi: 10.1016/j.entcs.2010.08.035

Enterprise architect. (2022). Retrieved from https://sparxsystems.com/

Erdweg, S., Giarrusso, P. G., & Rendel, T. (2012). Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications* (pp. 1–8). Retrieved from https://doi.org/10.1145/

2427048.2427055 doi: 10.1145/2427048

Erdweg, S., Storm, T. v. d., Völter, M., Boersma, M., Bosman, R., Cook, W. R., ... others (2013). The state of the art in language workbenches. In *International Conference on Software Language Engineering* (pp. 197–217). Retrieved from https://doi.org/10.1007/978-3-319-02654-1_11 doi: 10.1007/978-3-319-02654-1_11

Forsythe, Charles. (2013). *Instant FreeMarker Starter*. Packt Publishing Ltd.

Gupta, R., Jansen, N., Regnat, N., & Rumpe, B. (2022a, October). Design Guidelines for Improving User Experience in Industrial Domain-Specific Modelling Languages. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3550356.3561595 doi: 10.1145/3550356

Gupta, R., Jansen, N., Regnat, N., & Rumpe, B. (2022b, June). Implementation of the SpesML Workbench in MagicDraw. In *Modellierung 2022 Satellite Events* (p. 61-76). Gesellschaft für Informatik. Retrieved from https://doi.org/10.18420/modellierung2022ws-008 doi: 10.18420/modellierung2022ws-008

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

Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., & Wortmann, A. (2015). Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)* (p. 19-31). SciTePress. Retrieved from https://doi.org/10.5220/0005225000190031 doi: 10.5220/0005225000190031

Harel, D., & Rumpe, B. (2004, October). Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, *37*(10), 64-72. Retrieved from https://doi.org/10.1109/MC.2004.172 doi: 10.1109/MC.2004.172

Heidenreich, F., Johannes, J., Karol, S., Seifert, M., & Wende, C. (2009). Derivation and refinement of textual syntax for models. In *European Conference on Model Driven Architecture-Foundations and Applications* (pp. 114–129). Retrieved from https://doi.org/10.1007/978-3-642-02674-4_9 doi: 10.1007/978-3-642-02674-4_9

Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., & Völkel, S. (2009, July). Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineering in Research and Practice (SERP'09)* (p. 172-176). Retrieved from http://www.se-rwth.de/publications/Scaling-Up-Model-Based-Development-for-Large-Heterogeneous-Systems-with-Compositional-Modeling.pdf

Hölldobler, K., Kautz, O., & Rumpe, B. (2021). *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag. Retrieved from https://doi.org/10.2370/9783844080100 doi: 10.2370/9783844080100

Hölldobler, K., Rumpe, B., & Wortmann, A. (2018). Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, *54*, 386-405. Retrieved from https://doi.org/10.1016/j.cl.2018.08.002 doi: 10.1016/j.cl.2018.08.002

Ibm rhapsody. (2022). Retrieved from https://www.ibm.com/products/systems-design-rhapsody/

Kats, L. C., & Visser, E. (2010). 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). Retrieved from https://doi.org/10.1145/1932682.1869497 doi: 10.1145/1932682.1869497

Khaled, L. (2009). A comparison between UML tools. In *2009 second international conference on environmental and computer science* (pp. 111–114). Retrieved from https://doi.org/10.1109/ICECS.2009.38 doi: 10.1109/ICECS.2009.38

Krahn, H., Rumpe, B., & Völkel, S. (2006). Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)* (p. 150-158). Jyväskylä University, Finland. Retrieved from http://www.se-rwth.de/staff/rumpe/publications20042008/Roles-in-Software-Development-using-Domain-Specific-Modeling-Languages.pdf

Lara, J. D., Guerra, E., & Cuadrado, J. S. (2014, dec). When and How to Use Multilevel Modelling. *ACM Trans. Softw. Eng. Methodol.*, *24*(2). Retrieved from https://doi.org/10.1145/2685615 doi: 10.1145/2685615

Leduc, M., Degueule, T., Van Wyk, E., & Combemale, B. (2020). The software language extension problem. *Software and Systems Modeling*, *19*, 263–267. Retrieved from https://doi.org/10.1007/s10270-019-00772-7 doi: 10.1007/s10270-019-00772-7

Magicdraw enterprise. (2022). Retrieved from https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/

Maro, S., Steghöfer, J.-P., Anjorin, A., Tichy, M., & Gelin, L. (2015). On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (pp. 1–12). Retrieved from https://doi.org/10.1145/2814251.2814253 doi: 10.1145/2814251

Méndez-Acuña, D., Galindo, J. A., Degueule, T., Combemale, B., & Baudry, B. (2016). Leveraging software product lines engineering in the development of external dsls: A systematic literature review. *Computer Languages, Systems & Structures*, *46*, 206–235. Retrieved from https://doi.org/10.1016/j.cl.2016.09.004 doi: 10.1016/j.cl.2016.09.004

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, *37*(4), 316–344. Retrieved from https://doi.org/10.1145/1118890.1118892 doi: 10.1145/1118890.1118892

Meyers, B., Cicchetti, A., Guerra, E., & De Lara, J. (2012). Composing textual modelling languages in practice. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling* (pp. 31–36). Retrieved from https://doi.org/10.1145/2508443.2508449 doi: 10.1145/2508443.2508449

Orloff, R. W., Harland, D. M., Orloff, R. W., & Harland, D. M. (2006). Apollo 13: The seventh manned mission: in-flight abort 11–17 April 1970. *Apollo: The Definitive Sourcebook*, 361–392. Retrieved from https://doi.org/10.1007/0-387-37624-0_16 doi: 10.1007/0-387-37624-0_16

Ozkaya, M. (2019). Are the uml modelling tools powerful enough for practitioners? a literature review. *IET Software*, *13*(5), 338–354. Retrieved from https://doi.org/10.1049/iet-sen.2018.5409 doi: 10.1049/iet-sen.2018.5409

Rumpe, B. (2016). *Modeling with UML: Language, Concepts, Methods*. Springer International. Retrieved from https://doi.org/10.1007/978-3-319-33933-7 doi: 10.1007/978-3-319-33933-7

Rumpe, B., & Wortmann, A. (2018). Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan (Ed.), *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday* (p. 383-406). Springer. Retrieved from https://doi.org/10.1007/978-3-319-95246-8_23 doi: 10.1007/978-3-319-95246-8_23

Sánchez Cuadrado, J., Guerra, E., & Lara, J. d. (2011). Generic model transformations: write once, reuse everywhere. In *International Conference on Theory and Practice of Model Transformations* (pp. 62–77). Retrieved from https://doi.org/10.1007/978-3-642-21732-6_5 doi: 10.1007/978-3-642-21732-6_5

Scheidgen, M. (2008). Textual modelling embedded into graphical modelling. In *European Conference on Model Driven Architecture-Foundations and Applications* (pp. 153–168). Retrieved from https://doi.org/10.1007/978-3-540-69100-6_11 doi: 10.1007/978-3-540-69100-6_11

Seidewitz, E. (2014). UML with meaning: executable modeling in foundational UML and the Alf action language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology* (pp. 61–68). Retrieved from https://doi.org/10.1145/2692956.2663187 doi: 10.1145/2692956.2663187

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of systems and software*, *56*(1), 91–99. Retrieved from https://doi.org/10.1016/S0164-1212(00)00089-3 doi: 10.1016/S0164-1212(00)00089-3

Sprinkle, J., Rumpe, B., Vangheluwe, H., & Karsai, G. (2010). Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)* (p. 57-76). Springer. Retrieved from https://doi.org/10.1007/978-3-642-16277-0_3 doi: 10.1007/978-3-642-16277-0_3

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. Retrieved from https://doi.org/10.1016/j.cl.2017.07.004 doi: 10.1016/j.cl.2017.07.004

Talcott, C., Ananieva, S., Bae, K., Combemale, B., Heinrich, R., Hills, M., … Vangheluwe, H. (2021, July). Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen (Ed.), *Composing Model-Based Analysis Tools* (p. 45-70). Springer. Retrieved from https://doi.org/10.1007/978-3-030-81915-6_4 doi: 10.1007/978-3-030-81915-6_4

Tiwari, S., & Gupta, A. (2015). A systematic literature review of use case specifications research. *Information and Software Technology*, *67*, 128–158. Retrieved from https://doi.org/10.1016/j.infsof.2015.06.004 doi: 10.1016/j.infsof.2015.06.004

Tolvanen, J.-P., & Kelly, S. (2005). Defining domain-specific modeling languages to automate product derivation: Collected experiences. In *International conference on software product lines* (pp. 198–209). Retrieved from https://doi.org/10.1007/11554844_22 doi: 10.1007/11554844_22

Tolvanen, J.-P., & Rossi, M. (2003). Metaedit+ defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 92–93). Retrieved from https://doi.org/10.1145/949344.949365 doi: 10.1145/949344.949365

Vallecillo, A. (2010). On the combination of domain specific modeling languages. In *European Conference on Modelling Foundations and Applications* (pp. 305–320). Retrieved from https://doi.org/10.1007/978-3-642-13595-8_24 doi: 10.1007/978-3-642-13595-8_24

Voelter, M. (2010). Implementing feature variability for models and code with projectional language workbenches. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development* (pp. 41–48). Retrieved from https://doi.org/10.1145/1868688.1868695 doi: 10.1145/1868688.1868695

Voelter, M. (2011). Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering* (pp. 383–430). Retrieved from https://doi.org/10.1007/978-3-642-35992-7_11 doi: 10.1007/978-3-642-35992-7_11

Voelter, M., & Pech, V. (2012). Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)* (pp. 1449–1450). Retrieved from https://doi.org/10.1109/ICSE.2012.6227070 doi: 10.1109/ICSE.2012.6227070

Völter, M., & Visser, E. (2010). Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (pp. 301–304). Retrieved from https://doi.org/10.1145/1869542.1869623 doi: 10.1145/1869542.1869623

Wirth, N. (1996). Extended Backus-Naur Form (EBNF). *ISO/IEC*, *14977*(2996), 2–21.

About the authors

Arvid Butting is a researcher working on 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 him at butting@se-rwth.de.

Rohit Gupta is a researcher in model-based software and systems engineering at Siemens AG in Munich, Germany. You can contact him at rg.gupta@siemens.com.

Nico Jansen is a research assistant at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, software architectures, and model-based software and systems engineering. You can contact him at jansen@se-rwth.de.

Nikolaus Regnat is a researcher and a key expert in model-based software and systems engineering at Siemens AG in Munich, Germany. You can contact him at nikolaus.regnat@siemens.com.

Bernhard Rumpe is a professor heading the Software Engineering department at the RWTH Aachen University, Germany. His main interests are rigorous and practical software and system development methods based on adequate modelling techniques. This includes agile development methods as well as model-engineering based on UML/SysML-like notations and domain-specific languages. You can contact him at rumpe@se-rwth.de or visit https://www.se-rwth.de/staff/rumpe/.