Composition Operators for Modeling Languages: A Literature Review

Jérôme Pfeiffer†, Bernhard Rumpe*, David Schmalzing*, Andreas Wortmann†
* Software Engineering, RWTH Aachen University, Aachen, Germany, www.se-rwth.de
† Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW),
University of Stuttgart, Stuttgart, Germany, www.isw.uni-stuttgart.de

***Abstract*—Efficiently engineering modeling languages demands their reuse through composition. Research in language engineering has produced many different operators to reuse and compose languages and language parts. Unfortunately, these operate on different dimensions of languages, produce diverse results, and are distributed across various technological spaces and publications, which hampers understanding the state of language composition for researchers and practitioners. To mitigate this, we report the results of a literature review on modeling language composition operators. In this review, we identify operators, their properties, and supported language dimensions, and relate them to categories of language composition. Through this, our survey draws a new, detailed map of modeling language composition operators that can guide researchers in software language engineering in identifying uncharted territory and practitioners in employing the most suitable composition operators.**

***Index Terms*—Software Language Engineering, Modeling Languages, Language Composition, Literature Review**

I. Introduction

Software is the primary driver of innovation for cyber-physical systems, the Internet-of-Things, or Industry 4.0. Software languages [1], [2] can facilitate this innovation by providing syntax and semantics that improve abstraction w.r.t the domain of investigation. For instance, a systems engineering language might include SI units and the respective computations to facilitate managing physical properties within its models In contrast, a logic-based query language might include modeling elements about logical reasoning and corresponding computations. Consequently, many domains have transitioned to using modeling languages tailored to their specific challenges, such as automotive [3], [4], avionics [5], biology [6], [7], chemistry [8], [9], construction [10], insurance [11], law [12], manufacturing [13], [14], medicine [15], [16], robotics [17], and systems engineering [18], [19]. But the engineering of modeling languages often demands expertise in multiple meta-languages, tools, or paradigms, such as describing abstract and concrete syntaxes [20], [21], [22], [23] as well as model transformations operating on instances of these syntaxes [24], [25], [26] to, *e.g.,* realize their semantics [27].

For software, we know reuse is a main driver for its proliferation, that is, reuse-in-the-large (*e.g.,* complete applications, frameworks, libraries) and reuse-in-the-small (individual modules or their fragments). As software languages are also software [28], they can benefit from reusing them or their parts in the engineering of new languages as well. Reusing language parts requires means for modeling language composition [29].

Ten years ago, Erdweg et al. uncovered five different categories of language composition [30]: language extension, language restriction, language unification, self-extension, and extension composition. But since then, much has happened in software language engineering: Various language workbenches [31] have emerged and perished, projectional editing has become popular [32], language engineering is moving toward becoming web-based [33], [34], and concepts to better understand and organize language reuse have been developed [35], [29]. Given the importance of software languages today and these developments, we aim to understand how the composition of languages and language parts has evolved in the last decade. This work, hence, aims to guide researchers in the field in directing their efforts towards research gaps in modeling language composition and to help practitioners in finding suitable modeling language composition operators efficiently. To this end, we investigate the following questions:

RQ1 Which kinds of modeling language composition operators are there, and how do they relate to the five categories of [30]?

RQ2 Which language definition dimensions (regarding syntax and semantics) are supported by the operators?

RQ3 Which properties do language composition operators have concerning being black-box, modular, additive, and closed under composition?

The findings of the presented study show that most composition operators can be classified as language extension. Furthermore, the composition of syntax is covered by all operators although the realization of syntax specifications differs between the operators and oftentimes depends on a technological space of a language workbench. The composition of semantics is supported by 10 out of 25 operators we found in our study. Besides, many operators are modular, but only one enables black-box composition. Language workbenches and composition operators are closely related. However, our findings show that three operators are described without a realization in the technological space of a language workbench. We, therefore, decided not to focus on the support of composition operators in the different technological spaces.

In the remainder, Sec. II recapitulates categories of language composition. Afterward, Sec. IV describes the research method and study design, and Sec. V reports our findings. Next, Sec. VI discusses related studies, and Sec. VII highlights

observations. Finally, Sec. VIII concludes.

II. Categories of Language Composition

Erdweg et al. [30] identified five categories of language composition to address the lack of precise language engineering terminology concerning composition. To this end, they assume that languages comprise context-free syntax, well-formedness rules, and semantics [36] and can be reused unchanged. The categories are:

1.) Language Extension. Language extension is a direct form of language composition that operates on the level of language definitions. It requires the reuse of a language without changes to extend that language. This, *e.g.,* can be achieved by creating a new grammar that inherits from a parent grammar to reuse its productions.
Example: Creating a grammar for timed state machines by extending an existing state machine grammar and adding productions for handling time to the new grammar.

2.) Language Restriction. Language restriction is a special case of language extension in which the extension restricts the language. Restriction can be, *e.g.,* achieved by extending a language definition with new context conditions that prevent the occurrence of certain model elements. Hence, models with these elements are restricted from the language.
Example: Removing fork nodes and join nodes from an activity diagram language to prevent modeling parallel activities.

3.) Language Unification. Language unification is a language composition "on equal terms" [30], *i.e.,* without direction, which requires that the definitions of both languages can be reused unchanged by adding glue code only.
Example: Unifying OCL with transitions of statechart, such that statechart transitions can fire based on OCL constraints in their guards.

4.) Self-Extension. Self-extension describes the embedding of languages into a host language by providing a host language model that encapsulates the embedded language's concepts. The concepts of the embedded language are realized only with the concepts of the host language.
Example: Having an object-oriented programming language and creating a program that adds new classes to the language. Other programs can then use these; hence, the language has been extended without modifying the language definition (e.g., the compiler). As this is not a property of a composition operator but of a language itself, we do not consider self-extension in the following.

5.) Extension Composition. Extension composition describes the capability of language extensions to work together. That is, whether language extensions can be composed, either through the incremental extension of a language or by the union of independent extensions. As extension composition is trivial for operators that are closed under composition and matter of a language workbench for operators not closed under composition, this also is not considered in the following.

However, the exact interpretation of these categories depends on what can be considered "glue code" and what "equal terms" are. Language composition operators that produce a new language by copying and merging elements of their input languages, *e.g.,* to create a new grammar by joining all productions of two input grammars, can be considered to be composing on "equal terms", but without glue code (unless the resulting grammar can be considered being the glue). Consequently, the lack of precision in the formulation of the initial categories leaves some freedom of interpretation that we leverage and explain in the description of our findings.

III. Terminology

For the analysis of language composition operators, we investigate the following important properties of operators (not of the composed language parts):

Modularity of Composition: An operator supports the modular composition of language fragments if the composed parts continue to exist as identifiable artifacts in the composite. Modular composition allows, for instance, to evolve or maintain the composed language fragments individually so that the composite can also automatically benefit from these changes.
Example: When composing a state machine language with a data type language defining the properties usable in the state machines through linking (*e.g.,* importing of data types), both parts continue to exist as uniquely identifiable languages with only little composition "glue" between them.
Counterexample: When merging two metamodels into a new metamodel, the result often is a single new metamodel artifact containing the elements from both input metamodels. Hence, changes to the input metamodels are not propagated to the composed new metamodel.

Closed under Composition: An operator is closed under composition if the operator's application on two instances of a type T (*e.g.,* grammars or metamodels) produces an instance of type T again. This ensures that the operator can be applied to the result of the composition again.
Example: The metamodel composition outlined above takes two input metamodels and produces another metamodel.
Counterexample: When restricting a language by composing an abstract syntax definition (*e.g.,* grammars or metamodels) with well-formedness rules to prevent instantiation of certain abstract syntax elements (*cf.* [37]), the inputs are of different types.

Additive or Restrictive: An additive (restrictive) operator can only add elements to (remove elements from) a language definition. Note that an additive operator on well-formedness rules can add restrictions to a language definition that ultimately reduce the language.
Example: The metamodel composition outlined above produces a new metamodel containing the elements of both input metamodels.
Counterexample: The restriction operator outlined above adds new elements to the language definition but reduces the resulting language.

Black-box (BB) or White-box: A black-box composition operator does not require detailed insights into the language definitions to be composed but operates on their well-defined interfaces. A white-box operator, on the other hand, needs

detailed insights into the definitions and their constituents.
Example: For languages with operational semantics, a corresponding black-box composition operator could only rely on the interfaces of the language interpreters to compose them.
Counterexample: Composing two grammars usually requires understanding both grammars completely to understand intended extension points.

In the following, we assume the following definitions from [29] to investigate language composition on the more detailed level of language definitions: (1) A *language* is a set of possible sentences; and (2) A *language definition* comprises concrete syntax (CS), abstract syntax (AS), context conditions (CoCos), and semantics (Sem) .

- action language (AL),
- interpreter (Int),
- code generator (CG),
- general-purpose language (GPL),
- grammar (Gr), and
- metamodel (MM).

In our analysis, we refer to grammar rules as "productions" and metamodel elements as "classes". For directed composition operators, we use the prefix "base" for the language, grammar, metamodel, *etc.*, that elements of the "client" language, grammar, metamodel, *etc.*, are composed into.

Furthermore, we understand semantics as the meaning of a model [27], which is constructed by its language's syntax to a well-understood semantic domain. The semantic domain can, *e.g.,* be mathematical theories, such as stream-processing functions [38] or Petri-nets [39], as well as sufficiently understood domains. Denotational, translational, or operational semantics can define such a semantic mapping.

IV. Research Method

To answer RQ1-RQ3, we conducted a systematic literature review [40], [41], [42] on modeling language composition[1] consisting of five phases: (1) First, we decided on our study's scope and research questions. (2) Based on this scope, we performed a literature search in the second phase to identify the initial corpus of our study. (3) We then removed irrelevant publications from our initial corpus by screening keywords, abstracts, and titles in the third phase. (4) Afterward, we analyzed the publications in detail, applying classification schemes based on our research questions and removing the remaining irrelevant publications from the corpus. (5) Finally, we extracted the data from the publications of the corpus to answer our research questions. As we also performed backward and forward snowballing [43], we applied phases two to five a second time for the additional identified literature.

A. Search strategy and data sources

To produce a corpus of relevant publications on modeling language composition, we first identified relevant search terms as follows: Synonymous or at least closely related to these is also the term "domain-specific language" or "DSL" for short. The basis for defining such languages and, thus, language composition mechanisms are usually metamodels or grammars. These terms, therefore, form the first part of our search clause. Then, conjugated with these terms, we consider the concept of composition in the second part of the search clause and include terms used in the context of language composition, namely integration, derivation, and extension. These led to the following search term:

```
("metamodel" OR "modelling language"
OR "modeling language" OR "software
language" OR"DSL" OR "domain-specific
language" OR "grammar")
AND
("composition" OR "integration" OR
"derivation" OR "extension").
```

For this study, we were interested in publications that explicitly present a language composition operator in detail. We, therefore, limited the text search to keywords, titles, and abstracts. Publications, where language composition is part of the contribution, should mention combinations of terms of our search clause in the keywords, title, or abstract. However, to avoid missing relevant publications due to this limitation, we have included synonyms for the term modeling language in the search clause. Since we are interested in how language composition has evolved since the classification of language composition in [30], we limit our search to the years after its publication. That is, we limited our search to publications published after the first of January 2012 up to the first of March 2022, our search date. For our search, we used the ACM Digital Library, IEEE Xplore, Springer, Scopus, and the Web of Science. We excluded Google Scholar for its well-known problems [44], [45]. Instead, we employed snowballing to identify potentially relevant literature that we might have missed in the initial search.

For the databases that did not support the search query as presented, we split the query into multiple queries and merged their results manually. For the ACM Digital Library, this resulted in three search queries, one for the keywords, one for the title, and one for the abstract. For Scopus, we similarly split the search query into three parts but could combine these parts using disjunctions. For Springer, we had to search for exact phrases, *i.e.,* perform a search for each conjunction of the search terms separately. Finally, for the Web of Science, we could reuse the search query as presented with minor modifications. Other limitations did not affect our query. Overall, applying the search query to the selected databases under the aforementioned constraints returned the 8.741 results presented in Figure 1.

Since we had to use multiple overlapping queries for some of the libraries, the search on these libraries already resulted in duplicate findings. When merging the multiple queries for a single library, we removed the duplicate findings for that library. The numbers of publications given per library are,

[1]Replication package is available at https://awortmann.github.io/language-composition/

therefore, without duplicates. However, across the different libraries, we again had duplicate findings.

To identify the corpus relevant to our study, we first removed the 2.703 duplicates resulting in 6.038 unique publications. We then applied inclusion and exclusion criteria (cf. Sec. IV-B) to keywords, titles, and abstracts to remove an additional 5.915 publications, resulting in 123 English, potentially relevant, peer-reviewed publications. These publications we then analyzed in detail during the classification phase (cf. Sec. IV-C) to understand if they were relevant to our study. Again applying our inclusion and exclusion criteria, this time to a deeper analysis of the publications resulted in 28 publications. Applying forward and backward snowballing (cf. Sec. IV-D) we added another 11 publications. This resulted in 39 publications relevant to our study.

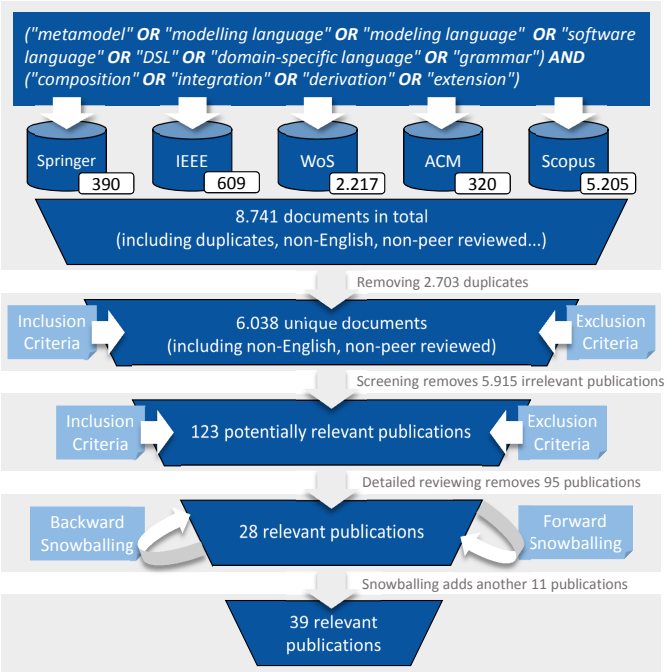


Figure 1: Our literature search and selection process

B. Screening papers for inclusion and exclusion

We applied the following inclusion and exclusion criteria to keywords, titles, and abstracts of the unique 6.038 publications.

Inclusion: (1) Peer-reviewed publications published in journals, conferences, and workshops. (2) Publications that are accessible electronically. (3) Publications where from the title, abstract, and keywords, we can deduce that the publication focuses on software language composition. (4) Publications that describe a language composition operators.

Exclusion: (1) Publications that are not available in English. (2) Publications that are not systematically peer-reviewed, such as monographs, slides, and websites. (3) Teasers and short papers of less than four pages, such as calls for papers, editorials, or curricula. (4) Publications that are secondary studies. (5) Pure case studies that apply but do not explain a language composition operator. (6) Publications on internal domain-specific languages (DSLs). (7) Publications that are not about DSLs, DSMLs or modeling languages

We included a publication if it met every inclusion criteria and did not meet any exclusion criteria. By screening the publications, we identify relevant publications and eliminate all non-relevant and further publications that meet our exclusion criteria, removing publications from the corpus that are not to be considered in our study. We first performed an initial screening, thereby pre-selecting and identifying potentially relevant publications using our inclusion and exclusion criteria based on keywords, titles, and abstracts only. Thereby, we included publications where in doubt.

C. Classifying studies

Removing 5915 publications in the screening phase, we obtained 123 potentially-relevant publications for further consideration. We divided these between authors for detailed analysis and classification and documented the results for each publication in a detailed questionnaire tailored to our research questions. Classifying studies with multiple reviewers entails aligning the reviewers' understanding of the matter at hand. Therefore, each author classified the same set of 20 (ca. 16%) randomly selected publications from our corpus. By comparing the results and discussing discrepancies, we developed and refined our shared understanding of the publications, the employed questionnaire, and the inclusion and exclusion criteria. We then split the remaining publications evenly among the authors, who analyzed and classified the publications in detail. However, we did not exclude any publication solely based on its comprehensibility. Instead, in cases where a single author was uncertain about the analysis or classification of a publication, we discussed the publication among all authors. The questionnaire and the classification scheme we conceived to answer our research questions. Possible answers to the questions of our questionnaire manifested as a simple yes or no, answers alongside a classification scheme, or free-text answers in cases where appropriate. During classification, we eliminated another 95 publications to obtain 28 relevant publications.

For each relevant publication, we extracted the presented language composition operators and classified these along the classification scheme revisited in Sec. II, where such a classification was appropriate. This classification partially enables us to answer **RQ1** regarding how language composition has evolved in recent years. Either the previously proposed classification scheme still holds, that is, we can classify all of the presented language composition operators, or there are new language composition operators that do not uphold this classification scheme. Where publications presented more than one language composition operator, we investigated each operator in isolation.

D. Snowballing

To identify relevant publications we might have missed, we applied forward snowballing [43] to [46], identifying all citations of this publication. For this search, we used semantic scholar[2], identifying 115 citations. Before looking at the publications in detail, we removed duplicates already considered in our initial search. After removing 44 duplicates, we applied our inclusion and exclusion criteria to the remaining 71 publications, thereby removing 4 publications for being short papers or non-peer-reviewed. For the remaining 67 publications, we decided on inclusion by looking first at the title, then the abstract, the location of the citation, and finally, the full publication to decide if the publication is relevant to our study. During these steps, we successively removed irrelevant publications, thereby removing 60 publications irrelevant to our study in total. Through forward snowballing of [46], we, therefore, identified 7 publications relevant to our study.

Performing backward snowballing, we applied a similar process, this time to references of publications in our corpus. Using semantic scholar, we identified 89 references in the initial search. From the search result, we removed 34 duplicates and one publication because of inaccessibility. Applying our inclusion and exclusion criteria, we removed another 31 publications for being published at a date outside the scope of our study (27 publications), non-peer-reviewed (1 publication), not available (1 publication), or short papers (2 publication) only. We again decided inclusion for the remaining 22 publications by looking at the title, abstract, place of citation, and the paper in full. We thereby identified 19 publications as irrelevant to our study, leaving 4 for inclusion. Applying backward snowballing again to the results of backward and forward snowballing did not yield any new results, wherefore we terminated the search. Therefore, we identified 11 publications relevant to our study through forward and backward snowballing. Together with the relevant publications of our initial search, this resulted in 39 publications that remained in the corpus of this study. From these, we extracted 25 unique language composition operators.

V. Findings

We categorized all 25 language composition operators into three of the five categories described in [30]. For these categories, we distinguish between operators that compose syntax only and those that compose both syntax and semantics. Overall we identified 9 cases of language extension composing syntax, 6 cases of language unification composing syntax, 4 cases of language extension composing syntax, 4 cases of language unification composing syntax and semantics, and 2 cases of language restriction composing syntax and semantics.

In the following, we detail our findings regarding **RQ1** - **RQ3**.

A. RQ1: Which Language Composition Operators Exist?

With this question, we aim to investigate which composition operators exist for language extension, language restriction, and language unification. We grouped our findings by the dimensions (syntax, semantics) of language definitions composed by the respective operators. Overall, we found operators composing definitions of syntaxes (*e.g.,* grammars, metamodels) and operators composing language modules comprising definitions of syntaxes and semantics (*e.g.,* code generators, interpreters, transformations). We did not find an operator that composes only semantics or syntax with semantics. Since none of the composition operators we found provided a formalized definition, we decided on a semi-formal description of each operator.

The tables presented in this section report one language composition operator per row. The tables include the language operators' publication, the constituents (AS, CS, Sem, ...), the kind of technical realizations (Gr, MM, GPL, ...) it addresses, the technological space it operates within, and its modularity, that is, whether it is closed under composition, additive, and black-box. Consequently, a paper reporting multiple operators is referenced in multiple rows. Also, this does not imply that a specific technological space comes with the described composition operator but that a composition operator was developed in the referenced technological space (*e.g.,* by a third party).

1) Language Extension Operators Composing Syntax: This section reports our findings on language extension operators capable of composing the syntax of two language definitions. Overall, we found operators that can compose either grammars, metamodels, or (RDF) graphs that describe abstract syntax, as well as operators that can compose grammars with metamodels. We summarized our findings regarding language extension composing syntax in Table I.

1.1) Grammar Embedding [47], [48], [49], [50], [51]
The operator takes a base grammar, a production of the base grammar, a client grammar, and a production of the client grammar. In the example of Figure 2 grammar `IOAutomaton` and its production `Automaton` is embedded into the production `BehaviorModel` of the base grammar `MAAutomaton`. Executing the operator produces a new grammar in which the selected production of the base grammar is augmented with an alternative of the selected client grammar production. To this end, the base grammar may feature dedicated extension points (*e.g.,* special kinds of productions) to denote the incompleteness of the grammar. Models conforming to the resulting grammar can use base grammar production instances as usual and the embedded client grammar production instances in place of the selected base grammar production in the same model (*cf.* Figure 2).

1.2) Grammar Inheritance [47], [48], [50], [51], [52], [53]
The operator takes base grammar and client grammar as input. This results in a client grammar where all productions from the base grammar are made available. Hence, the client grammar can reuse or override the productions of the base grammar

[2]https://www.semanticscholar.org/

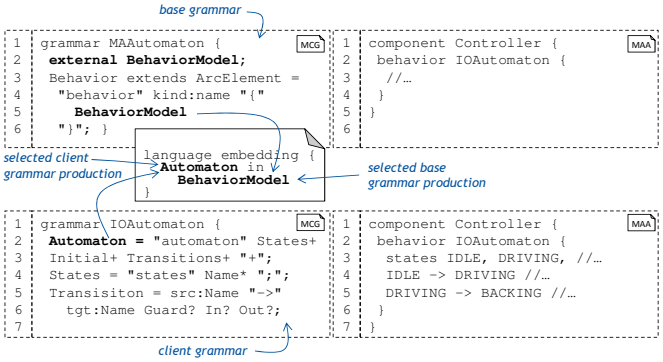


Figure 2: The operator for grammar embedding on the left and the effect on the model level on the right [48].

(*cf.* Figure 3). Models conforming to the resulting grammar can use instances of base and client productions in the same model (*cf.* Figure 3). An algebraic formalization of grammar inheritance is available from [53].

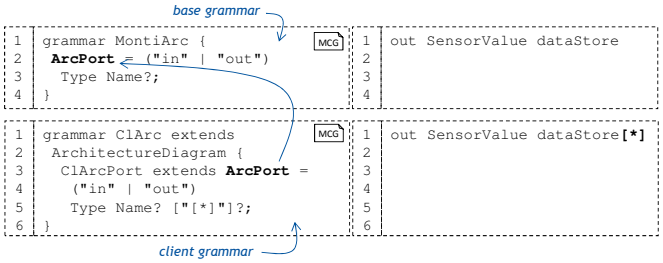


Figure 3: The operator for grammar inheritance on the left and the effect on the model level on the right [51].

1.3) Grammar Mixins [54]

The operator takes a base grammar (*cf.* l. 1 of Figure 4) and a client grammar, the mixin (*cf.* l. 2 of Figure 4), as input. Any grammar can be used as a mixin. It makes the abstract syntax, defined in a metamodel, of the client grammar available to be referenced in the client grammar's productions. However, they cannot be overwritten or extended. On the model level, this has the effect that models of the base grammar can leverage modeling concepts from the client grammar's metamodel within the same model.

1.4) Metamodel Embedding [55]

The operator takes a base metamodel, a client metamodel, a metaclass of the base metamodel, a metaclass of the client metamodel, the cardinality of the relation between the selected metaclasses, and two Boolean arguments denoting whether the relation between the classes should express a composition or aggregation. It composes both metamodels by introducing an association with the given cardinality and the relation between the client metaclass and the base metaclass (*cf.* Figure 5). The example given in the paper is that of an expression metaclass which is embedded into a cell class of sheet metamodel, which

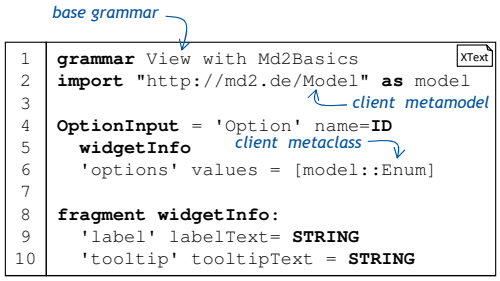


Figure 4: The operator for grammar mixins exemplified with a grammar `View` where a metamodel `Model` is imported as mixin [54].

then has the effect that cells now can define expressions (*cf.* Figure 5). An algebraic, partial, formalization of metamodel embedding is available from [55].

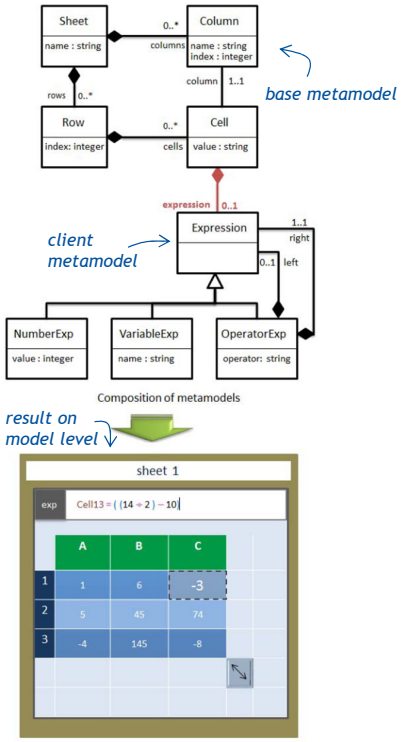


Figure 5: The operator for metamodel embedding that embeds expressions into sheet cells [55].

1.5) Metamodel Fragment Composition [56]

A metamodel fragment is a container for a metamodel that exposes contractually specified provided and required interfaces. A provided interface exposes metaclasses of the fragment's metamodel, whereas a required interface demands implementation by a metaclass of another metamodel fragment. For instance, the `Business Process Diagram` metamodel in Figure 6 demands a implementation of `IPerformer` that is fulfilled by the `IOrgElement` of the `Organization Model`. The Composition of a base metamodel fragment and

a client metamodel fragment is realized by mapping classes of the provided interface of the client metamodel fragment to the required interfaces of the base metamodel fragment. Thereby, the client metamodel fragment classes implement the required interfaces of the base metamodel fragment. The effect on the modeling level is similar to *Metamodel Embedding* where the mapped modeling concepts can be used in the models conforming to the base metamodel.

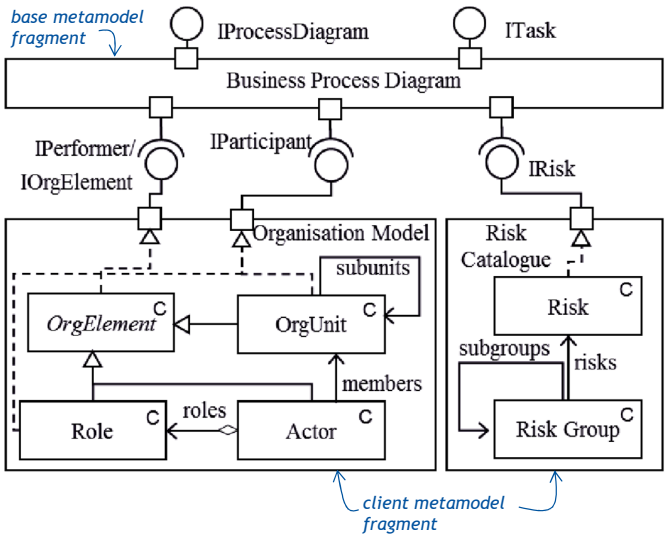


Figure 6: The operator for metamodel fragment composition applied to a business process diagram that is composed with an organization and a risk catalog metamodel [56].

1.6) Metamodel Mixins [57]

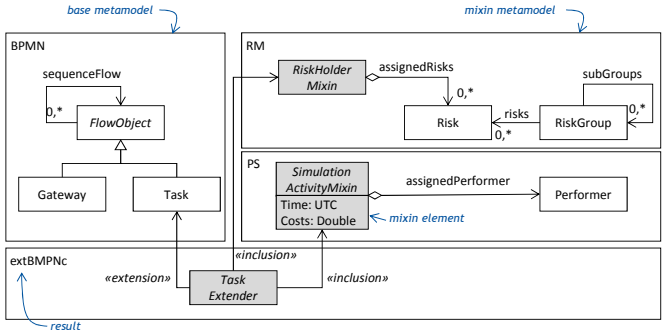


Figure 7: The operator for metamodel mixins that mixes `Simulation` and `Risk` into a `Task` metaclass [57].

This composition operator takes two metamodels, the base metamodel, and the mixin metamodel as input. They are composed by adding the elements of an abstract metamodel class (the "mixin element"), *e.g.,* `RiskHolderMixin` and `SimulationActivityMixin` (*cf.* Figure 7), of the mixin metamodel to a class of the base metamodel, *e.g.,* `Task` (*cf.* Figure 7). The result is a new metamodel comprising the base metamodel, and the mixin metamodel, and an additional class that extends the class of the base metamodel, and references the mixin class (*cf.* Figure 7). Note that the result of a mixin composition cannot be used as a mixin element again.

1.7) Metamodel Template Instantiation [58]

The template instantiation operator takes a metamodel template, which comprises abstract classes as extension points, and a "variant" metamodel as input together with renamings. For instance, in Figure 8 the extension points `SoundSource` and `Filter` of a base metamodel are implemented by the variant metaclasses `Oscillator` and `Filter`, respectively. The operator then merges both metamodels based on naming and thereby instantiates the metamodel template. The result is a new metamodel. Models that conform to this new metamodel can utilize the non-abstract concepts of the metamodel of the template, and, in addition, the instantiated abstract classes instantiated by the "variant" metamodel.

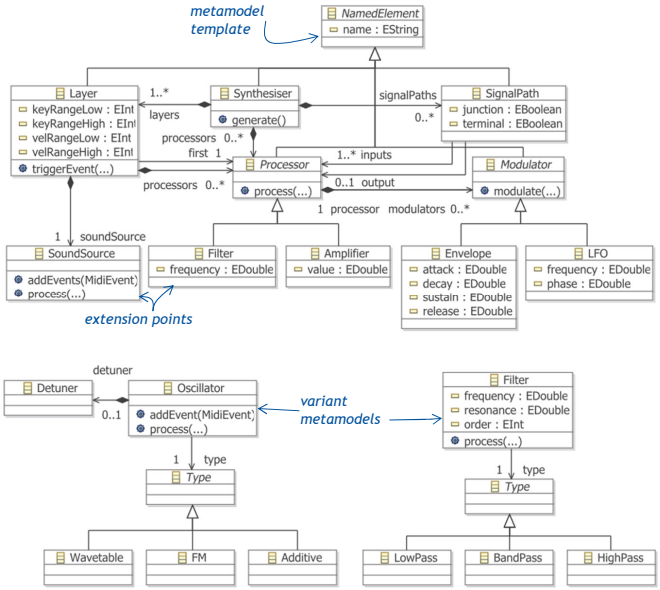


Figure 8: The input for the operator for metamodel template instantiation that takes the template classes `SoundSource` and `Filter` and instantiates them with Oscilator and Filter respectively [58].

1.8) Syntax Component Composition [59], [60]

The operator takes two language components, comprising grammars, Java well-formedness rules, and bindings between interfaces of provided and required extension points as input. The components are composed by relating the provided extension points of the client component to the required extension points of the base component. Figure 9 shows the operator exemplified with three language components that are composed according to their extension points via a feature diagram. This governs how the artifacts of the client language need to be composed with the artifacts of the base language component. The result is a language component again. Models

conform to the composed language can use all the concepts of the base language component, and, additionally, the concepts of the client component in place of the bound concepts of the required extensions.

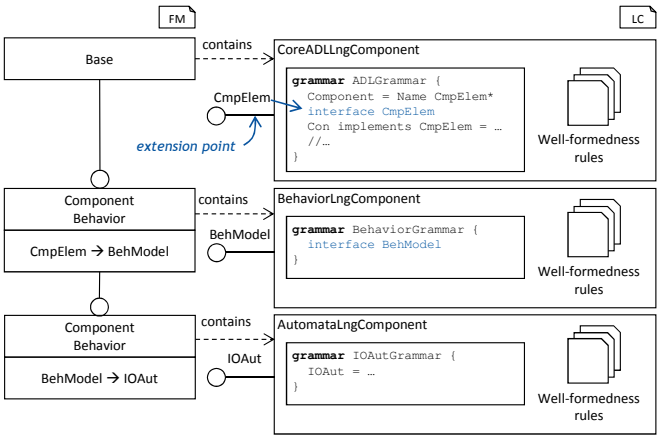


Figure 9: The operator for syntax component composition exemplified with three components that are bound to another [60].

1.9) Metamodel Facet Composition [61], [62]

Facets extend existing objects with a new type, fields and constraints [62]. They are instances of a metamodel that is called facet metamodel. Facet metamodels provide an interface restricting the compatible classes to the facet type. The operator for metamodel facet composition takes at least one metamodel, one facet metamodel, and a set of facet laws as input (*cf.* Figure 10). Facet laws govern the acquisition of facets by conditions automatically. After the composition, on the language level, both metamodels remain loosely coupled. On the model level, facet objects (instances of facet metamodels) can be added to metamodel objects. With this, objects are extended with the type and the properties of the facet.

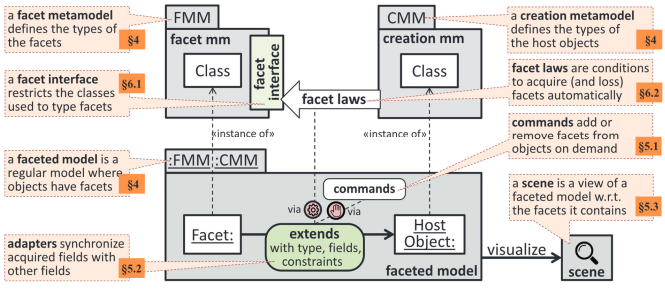


Figure 10: The operator for metamodel facet composition [62].

2) Language Unification Operators Composing Syntax: This section reports our findings on language unification operators capable of composing the syntax of two language definitions on roughly "equal terms" [30]. Some of these are forms of language coordination [29], *i.e.,* the composition is achieved mainly by adding glue code between the composed language' artifacts, some are forms of language integration [29], where the elements of the composed languages' artifacts, *e.g.,* their productions, are copied into a new artifact. With respect to [30], we consider this reusing the composed languages unchanged and argue that a new artifact is a form of glue code. Otherwise, operators, such as metamodel merging, where new language extensions are composed by copying elements of the input languages on equal terms, cannot be captured by the classification of [30]. We summarized our findings regarding language unification composing syntax in Table II.

2.1) Annotation-Based Language Unification [46]

The operator takes a textual base syntax definition (*e.g.,* grammars, abstract syntax tree (AST) classes, JSON, ...) and a client syntax definition as input. The base syntax definition needs to support the notion of annotations. It unifies elements of the base syntax definition with concepts from the client syntax definition based on annotations in the base syntax models. For instance, the annotations in Figure 11 reference elements specified in an XML file that defines a database table. Models of the composed language can use annotations of client language concepts in models of the base language.

```
1  @EntityRef("Person")          base syntax
2  public class Person {
3    @ColumnRef("Identifier")
4    private int id;
5
6    @ColumnRef("Identifier")    references to
7    private String name;        client syntax
8
9    @ColumnRef("Identifier")
10   private int age;
11   //…
12 }
```

Figure 11: The operator for annotation-based language unification exemplified with Java annotation unifying a database language with Java [46].

2.2) Grammar Unification [63], [64]

The operator takes multiple grammars and creates a new grammar that contains the union of productions of the input grammars. For instance, Figure 12 shows the unification of three grammars resulting into the grammar `result` (*cf.* l. 1-2).. Where productions of the input grammars have the same left-hand-side name, alternatives for this production are created. Models conforming to the resulting grammar may use grammar productions to conform to the composed grammar's productions in the same model.

2.3) Graph Merging [65]

The operator takes two graphs that represent the abstract

Table I: Language Extension Operators Composing Syntax

Name	Constituents	Tech. Space	Modular	Closed	Additive	BB
Grammar Embedding [47], [48], [49], [50], [51]	AS (Gr), CS (Gr)	Grammarware, Language Boxes, MontiCore	✓	✓	✓	✗
Grammar Inheritance [47], [48], [50], [51], [52], [53]	AS (Gr), CS (Gr)	MetaDepth, MontiCore, Grammarware	✓	✓	✓	✗
Grammar Mixins [54]	AS (Gr), CS (Gr)	Xtext	✓	✓	✓	✗
Metamodel Embedding [55]	AS (MM), CS (MM)	EMF	✓	✓	✓	✗
Metamodel Facet Composition [61], [62]	AS (MM), CoCos (OCL)	Metadepth	✓	✓	✓	✗
Metamodel Fragment Composition [56]	AS (MM)	CML	✓	✓	✓	✓
Metamodel Mixins [57]	AS (MM)	ADOxx	✓	✗	✓	✗
Metamodel Template Instantiation [58]	AS (MM)	EMF & SWAT	✗	✗	✓	✗
Syntax Component Composition [59], [60]	AS (Gr), CS (Gr), CoCos (GPL)	MontiCore	✓	✓	✓	✗

```
1 unification [Grammar g, Grammar decl, Grammar expr]
2                         returns [Grammar result]
3   {unify  = decl  + expr;}
4   {glue   = unify + 'factor: id;';}
5   {result = glue  + 'exprdecl: declist expr;';}
6 ;
```

Figure 12: The operator for grammar unification [64].

syntax of two languages, nodes of these graphs and connection rules between these nodes (*cf.* Figure 13). The graphs are merged by by applying these rules. Models of the languages remain separated and are unified by the graph interconnections on the metalevel. A logical formalization of graph merging is available from [65].

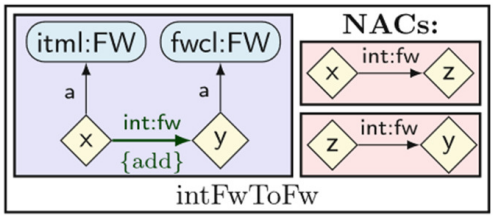


Figure 13: The operator for graph merging exemplified with a integration rule `int:fw` between the nodes `itml:FW` and `fwcl:FW`[65].

2.4) Language Aggregation [48], [51], [66]

Language aggregation takes two syntax definitions, *i.e.,* grammar [48], [51] or metamodel [66], and one named element of each as input to be aggregated. The named elements of both languages are aggregated by producing an adapter that coordinates the interaction between both. For the syntax definition based on grammars [48], [51], *e.g.,* this happens via an adapter between the symbol tables of both languages linking the corresponding defining symbols of one language with the using symbols of the other language. For metamodel-based language aggregation, an additional coordinating metaclass is produced [66]. Both adapters have to be implemented manually. In both cases, the result is a language in which the files containing the models remain separate but link to each other (*cf.* Figure 14). This is similar to importing classes in programming languages. An formalization of metamodel alignment using category theory is available from [67].

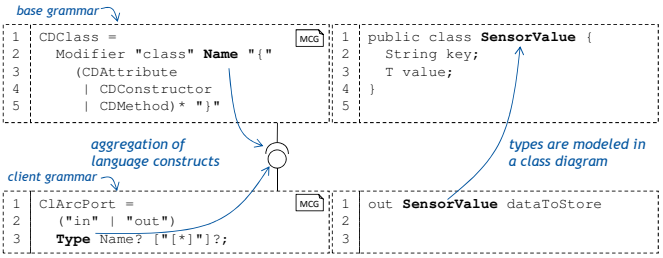


Figure 14: The operator for language aggregation exemplified with a grammar production `ClArcPort` that references a `Type` which is defined in a production `CDClass` [51].

2.5) Metamodel Alignment [67]

The operator takes multiple similar metamodels and a set of mapping metaclasses with unidirectional connections between the similar metaclasses of the metamodels as input. The metamodels are aligned by mapping a selection of their classes to the base metamodel using the mapping rules. For instance, in Figure 15 three metamodels M^1, M^2, M^3 are aligned using the mapping M^0. The resulting metamodel, hence, comprises all metaclasses of the input metamodels as well as the adapters and relations between the selected elements. The models of the composed languages remain separated but are linked to one another through their metamodels.

2.6) Metamodel Merging [68], [69]

The classes of two metamodels are merged based on the names of their elements. The result is a new metamodel featuring classes, attributes, and associations from both input metamodels (*cf.* Figure 16). For metamodel classes of the same name, their elements are merged as well. The effect achieved on the model level is similar to the effect of applying the operators *Language Aggregation* and *Metamodel Alignment*.

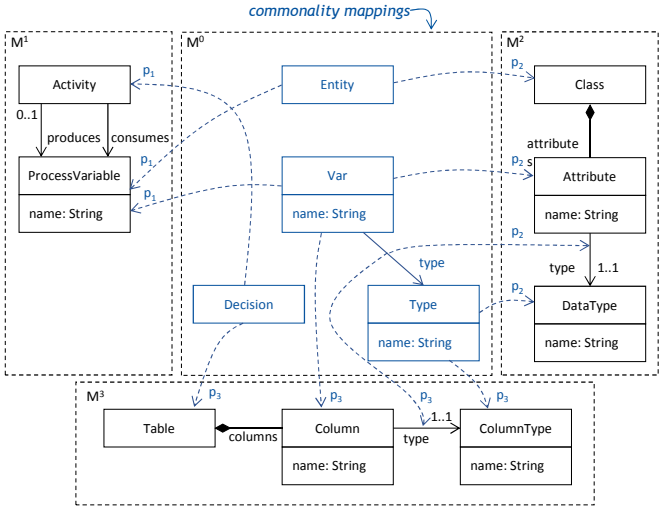


Figure 15: The operator for metamodel alignment exemplified [67].

An algebraic formalization of metamodel merging is available from [69].

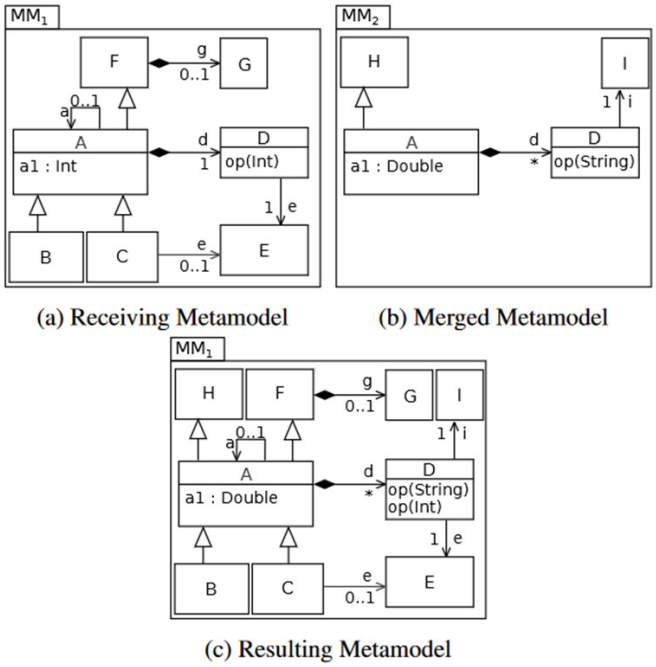


Figure 16: The operator for metamodel merging by merging MM_2 (b) into MM_1 (a) resulting in the metamodel MM_1 (c) [68].

3) Language Extension Operators Composing Syntax and Semantics: This section reports our findings on language extension operators capable of composing the syntax and semantics of two language definitions. Therefore, most of these operators expect a language definition container, such as a module or component, or expect that operational semantics are part of the abstract syntax definition (*e.g.,* a metamodel with semantics realized in its methods). Some of these apply to syntax without semantics as well, *e.g.,* where semantics is defined within the methods of a metamodel's classes. We summarized our findings regarding language extension composing syntax and semantics in Table III.

3.1) Abstract-Syntax-Driven Language Embedding [70]

An abstract-syntax-driven language definition is a language definition in which the abstract syntax artifacts also comprise (a) their concrete syntax in the form of annotations and (b) their semantics in the form of aspects carrying methods for some abstract syntax classes. For instance, Figure 17 defines `AddOp` together with an annotation that states the concrete syntax. The operator takes a base language definition and a client language definition, and a mapping from a client class to a class of the base language definition. The base language definition then is extended by providing a subclass (carrying CS and Sem) for a selected base class and registering it as an alternative for the selected base class (*cf.* Figure 17). Thus, new alternatives for syntax and semantics can be embedded into a base language definition. The effect achieved on the model level is that the mapped client language concepts are usable in place of the base language's concept including concrete syntax and semantics.

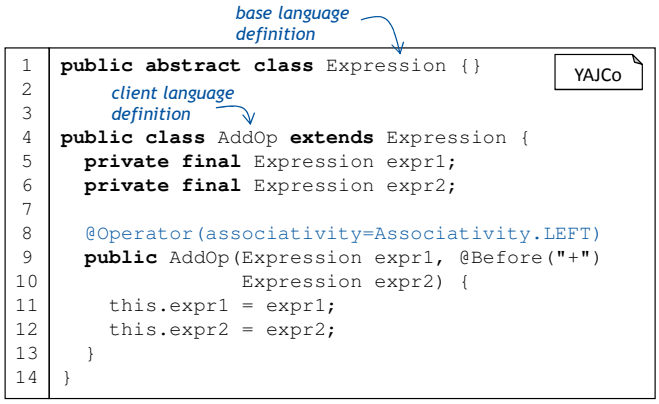


Figure 17: The operator for abstract-syntax-driven language embedding visualized by a subclass `AddOp` extending the abstract class `Expression` [70].

3.2) Language Component Embedding [71], [72]

A language component comprises a grammar that supports the integrated definition of CS and AS, Java well-formedness rules, and a code generator and is encapsulated with interfaces of required and provided extension points. Figure 18 shows an example of a language component for a `Transition System`. Provided extension points expose elements of the language component to the environment, whereas required extension points expose extension points of the language component's elements. The operator takes a base language component, a client language component, and a mapping

Table II: Language Unification Operators Composing Syntax

Name	Constituents	Tech. Space	Modular	Closed	Additive	BB
Annotation-Based Language Unification [46]	AS	N/A	✓	✗	✓	✗
Grammar Unification [63], [64]	AS (Gr), CS (Gr)	APEG, Enso	✓	✓	✓	✗
Graph Merging [65]	AS (RDF graph)	RDF	✗	✓	✓	✗
Language Aggregation [48], [51], [66]	AS (Gr/Symbols)	MontiCore	✓	✓	✓	✗
Metamodel Alignment [67]	AS (MM)	N/A	✓	✓	✓	✗
Metamodel Merging [68], [69]	AS (MM)	Melange	✗	✓	✓	✗

from the client language component's provided interfaces to the base language component's required interfaces. Based on these bindings, the grammars of the language components are embedded (using *grammar embedding*), their well-formedness rules joined, and the code generators embedded as well. Moreover, a new interface is synthesized from the interfaces of both language components and a new language component comprising the composed artifacts as well as the new interface is created. On the model level, the same effect as with *Grammar Embedding* is achieved regarding syntax. However, by composing also well-formedness rules and code generators, the semantics of the embedded concepts are also available.

```
1  dsl component TransitionSystem {                                  LC
2    grammar mc.FSM;          <- grammar reference
3    gen FSMG context fsm._gen.FSMGenerators;   <- generator context
4
5    provides production StateMachine;
6    requires optional production Istate;        } grammar extensions
7    requires mandatory production Itrans;
8
9    provides gen FSMMainGen for StateMachine with FSMG;
10   requires optional gen StateGen for IState with FSMG;
11   requires optional gen TransGen for ITrans with FSMG;
12                                                  } generator extensions
13   wfrs TransitionCorrect {
14     fsm._cocos.TransitionSourceStateExists;
15     fsm._cocos.TransitionTargetStateExists;
16   }                                              } provided sets of well-
17   wfrs TSCorrect {                                 formedness rules
18     fsm._cocos.AllStatesReachable;
19     fsm._cocos.NamesAreUpperCase;
20   }
21 }
```

Figure 18: An example for a language component for a `TransitionSystem` language [71].

3.3) Language Concern Composition [68], [73]

A language concern is a metamodel with behavioral semantics realized in its methods. The metamodel may expose required extensions as annotated interface classes. This operator takes a base language concern with required extensions, a client language concern as input, and a mapping from client concern metamodel classes to base metamodel required interface classes. It establishes interface implementation relations between the classes of the client metamodel and the required interfaces of the base metamodel according to this mapping (*cf.* Figure 19). For instance, Figure 19 shows a base language concern `FSM` that is composed with two client language concerns `AL` and `Exp` via two adapting metaclasses `BindAction` and `BindGuard`, respectively. The models of the composed language can then use the client's language concepts in place of the base language's concepts according to the mapping between both concerns including semantics. An algebraic formalization of language concern composition is available from [68].

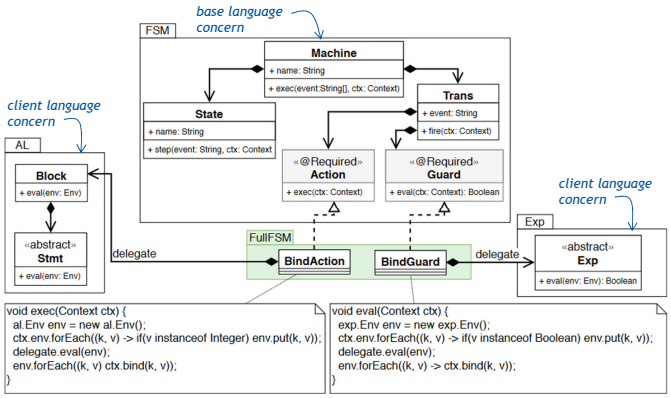


Figure 19: An example for the language composition operator language concern composition that extends a state machine language (`FSM`) with an action language (`AL`) and expression (`Exp`) [73].

3.4) Language Module Extension [74], [75], [76]

A language module [75], called language [76] or trait [74], comprises definitions of CS, AS, and computation rules (*e.g.,* for semantics or well-formedness rules), such that the definitions of semantics are unambiguously related to a single AS element (*cf.* Figure 20). The client language module can extend from a base language module to make all CS, AS, and computation rules of the base language modules available in the client, which then can override (some of) these elements. For instance, in Figure 20, the language `RobotTime` extends the existing language `Robot` and extends the rule `commands` to add time constraints. The result is that these new CS, AS, and computation rules are available on the model level.

4) Language Unification Operators Composing Syntax and Semantics: This section reports our findings on language unification operators capable of composing the syntax and semantics of two language definitions. As with language exten-

Table III: Language Extension Operators Composing Syntax and Semantics

Name	Constituents	Tech. Space	Modular	Closed	Additive	BB
Abstract-Syntax-Driven Language Embedding [70]	AS (GPL classes), CS (AS annotations), Sem (AS methods)	YAJCo	✓	✓	✓	✗
Language Component Embedding [71], [72]	AS (Gr), CS, (Gr) Sem (GPL), CoCos (GPL)	MontiCore	✓	✓	✓	✗
Language Concern Composition [73], [68]	AS (MM), Sem (MM)	ALEX, Melange	✓	✓	✓	✗
Language Module Extension [74], [75], [76]	AS (Gr), CS (Gr), Sem (AL)	LISA, Neverlang	✓	✓	✓	✗

base language module — client language module

LISA

```
1  language RobotTime extends Robot {
2    attributes double *time;
3    rule extends start {
4      compute {
5      START.time = COMMANDS.time;}
6    }
7    rule extends commands {
8      COMMANDS ::= COMMAND COMMANDS compute {
9        COMMANDS[0].time = COMMAND.time + COMMANDS[1].time;}
10     | epsilon compute {
11       COMMANDS.time = 0;};
12   }
13   rule extends command {
14     COMMAND ::= left  compute {
15       COMMAND.time = 1;};
16     COMMAND ::= right compute {
17       COMMAND.time = 1;};
18     COMMAND ::= up    compute {
19       COMMAND.time = 1;};
20     COMMAND ::= down  compute {
21       COMMAND.time = 1;};
22   }
23 }
```

Figure 20: An example for the language composition operator language module extension that extends a `Robot` language with time constraints [76].

sion operators composing syntax and semantics, most of these operators expect a language definition container and some can be applied to syntaxes without semantics. We summarized our findings regarding language unification composing syntax and semantics in Table IV.

4.1) Language Component Aggregation [77]

This operator takes two language components of the kind taken by *language component embedding* and a set of bindings between the interfaces of both components. It then unifies selected language symbols exposed through their interfaces according to the bindings passed to the operator. To realize this unification, it uses *language aggregation* between their symbols, joins their well-formedness rules, and synthesizes adapters between components' code generators. Figure 21) shows language component aggregation exemplified with two language components `CD` and `Aut`. Applying the operator results into a new composed language component. The effect on the model level, is, thus, the same as with *language aggregation* but additionally includes well-formedness rules and code generators.

4.2) Language Union [78]

Lang-N-Play defines AS, CS, and Sem by logic rules. Here,

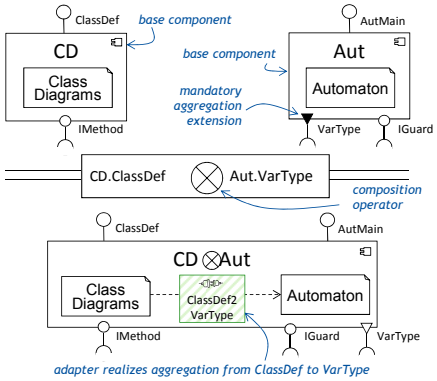


Figure 21: An example for the language composition operator language component aggregation that composes two language components `CD` and `Aut` and creates a new component as result [77].

language union is the merging of new rules, which can extend the language definition's syntax, well-formedness rules, or transformations, into language definitions. The rules to be merged do not need to be part of another language definition, but can also be specified in terms of a Prolog program. For instance, Figure 22) shows the union of a base grammar `lists` with client grammar rules for errors. Note that the rules to be merged into a language definition do not exist as a stand-alone artifact before and that the result of a union is a language definition that cannot be merged into another one directly.

base grammar

Lang-N-Play

client grammar

```
1 lists U {!
2   Expression e ::= myError,
3   Error er ::= myError,
4   (elementAt zero nil) --> myError,
5   (elementAt (succ V) nil) --> myError,
6 !}
```

Figure 22: An example for the language composition operator language union that enables lists to use error expression [78].

4.3) Metamodel Service Orchestration [79]

This operator can compose language definitions in the form of metamodels that carry semantics as methods and expose provider and consumer services contracts. The base language definition, therefore, comprises consumer interfaces without

implementations and the client language definition comprises provider interfaces with implementations (*cf.* Figure 23). At the "runtime" of the metamodel instances, links between them can be established by exchanging "XML-based messages" [79]. Hence, a composite metamodel is a "conceptual notion" [79] only as the metamodels are not woven together.

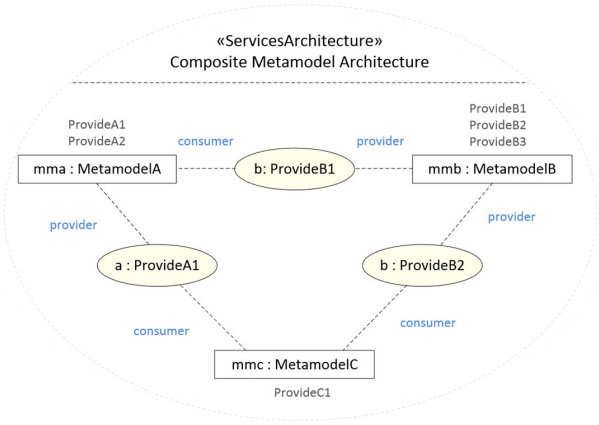


Figure 23: An example for the language composition operator for metamodel service orchestration consisting of three metamodels that are orchestrated based on their provideer and consumer interfaces [79].

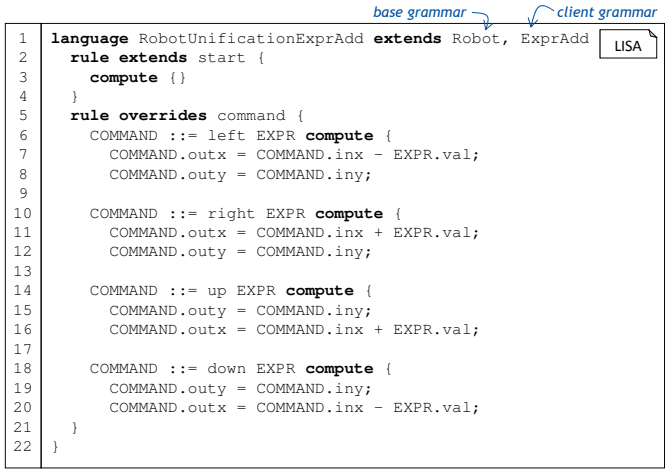
4.4) Object-Oriented Language Unification [76]

Here, a language module comprises definitions of CS, AS, and computation rules (*e.g.,* for semantics or well-formedness rules), such that the definitions of semantics are unambiguously related to a single AS element. This operator then takes two language modules and a set of glue rules overriding the rules of the input languages. For instance, in Figure 24 the language unifies two languages `Robot` and `ExprAdd`. As a result, it creates a new language module extending from both language modules featuring the overriding rules. Thus, the models that conform to this newly created language can use each other's syntax and semantics in an integrated way within the same model file.

5) Language Restriction Operators Composing Syntax and Semantics: The operators presented in this section can be applied to restrict the set of accepted models or semantic mapping of these models of a base language. Some do not require semantics and can be applied to pure syntax as well. We summarized our findings regarding language restriction composing syntax and semantics in Table V.

5.1) Language Module Restriction [50], [53], [75], [78]

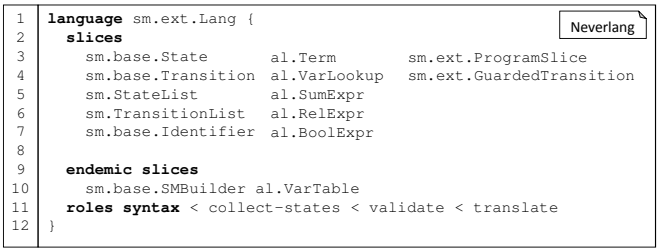
This operator takes a base language module and a list of language elements of the base language that shall be reused. The result is a new grammar that extends the base language module without inheriting all its elements. Thus language restriction, in this case, is achieved by not selecting the elements of the base language module to be restricted. For




```
1  language RobotUnificationExprAdd extends Robot, ExprAdd
2    rule extends start {
3      compute {}
4    }
5    rule overrides command {
6      COMMAND ::= left EXPR compute {
7        COMMAND.outx = COMMAND.inx - EXPR.val;
8        COMMAND.outy = COMMAND.iny;
9
10     COMMAND ::= right EXPR compute {
11       COMMAND.outx = COMMAND.inx + EXPR.val;
12       COMMAND.outy = COMMAND.iny;
13
14     COMMAND ::= up EXPR compute {
15       COMMAND.outy = COMMAND.iny;
16       COMMAND.outx = COMMAND.inx + EXPR.val;
17
18     COMMAND ::= down EXPR compute {
19       COMMAND.outy = COMMAND.iny;
20       COMMAND.outx = COMMAND.inx - EXPR.val;
21   }
22 }
```

Figure 24: An example for object-oriented language unification [76].

instance, in the language definition in Figure 25, one of the slices could be removed to restrict the language. The effect on the model level is that all language concepts of the base language without the restricted ones are available to be used. An algebraic formalization of language module restriction is available from [53].




```
1  language sm.ext.Lang {
2    slices
3      sm.base.State       al.Term        sm.ext.ProgramSlice
4      sm.base.Transition  al.VarLookup   sm.ext.GuardedTransition
5      sm.StateList        al.SumExpr
6      sm.TransitionList   al.RelExpr
7      sm.base.Identifier  al.BoolExpr
8
9    endemic slices
10     sm.base.SMBuilder al.VarTable
11   roles syntax < collect-states < validate < translate
12 }
```

Figure 25: An example for the language composition operator for language module restriction depicting a state machine language in Neverlang consisting of slices that reference modules. By removing slices, the language can be restricted [75].

5.2) Language Slicing [68]

In language slicing, elements of one metamodel (pattern) are removed from a base metamodel by matching of names of, *e.g.,* classes, attributes, methods, *etc.*. The resulting metamodel then yields fewer elements than its base metamodel. For instance, in Figure 26 the class `D` is sliced. Where the metamodels can carry semantics in the form of method implementations, this also slices semantics realizations. The effect on model level is the same as in *Language Module Restriction*. An algebraic formalization of language module restriction is available from [68].

Table IV: Language Unification Operators Composing Syntax and Semantics

Name	Constituents	Tech. Space	Modular	Closed	Additive	BB
Language Component Aggregation [77]	AS (Gr), CS (Gr), Sem (GPL), CoCos (GPL)	MontiCore	✓	✓	✓	✗
Language Union [78]	AS, CS, Sem (GPL)	Lang-N-Play	✗	✗	✓	✗
Metamodel Service Orchestration [79]	AS, CS, Sem (GPL)	N/A	✓	✗	✓	✗
Object-Oriented Language Unification [76]	AS (MM), CS (Gr), Sem (AS methods)	LISA	✓	✓	✓	✗

Table V: Language Restriction Operators Composing Syntax and Semantics

Name	Constituents	Tech. Space	Modular	Closed	Additive	BB
Language Module Restriction [50], [53], [75], [78]	AS (Gr), CS (Gr), Sem (AL)	Grammarware, Neverlang	✗	✗	✗	✗
Language Slicing [68]	AS (MM), Sem (Int)	GEMOC / Melange	✓	✓	✗	✗

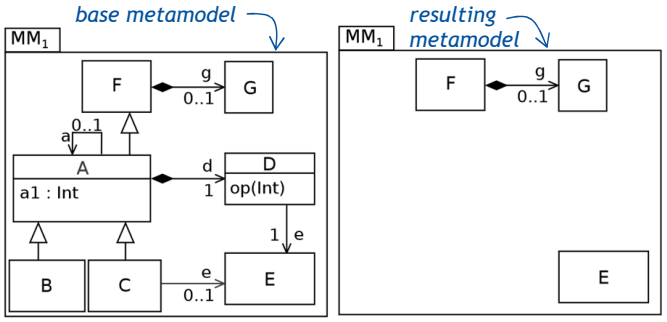


Figure 26: An example for the language composition operator for language slicing with the source left and the result on the right based on the input of metaclass `D` [68].

6) Summary of Findings and Practitioners Guide: Our findings show that most of the operators found work solely composing syntax. Thereby, using graphs, metamodels, grammars, or GPL classes for specifying the abstract syntax. For the desired composition effect, most operators are classified into language extension (13), followed by language unification (10). Only two operators, that we found can be used for language restriction. From our findings, we extracted a guide for practitioners (see Figure 27) that gives an overview of all composition operators available in literature today and to identify the most accurate operator for their needs. To this end, Figure 27 depicts different decisions that ultimately lead to a composition operator identified in our study. For all operators, we decide between the different abstract syntax realizations, whether semantics is required, if yes, how it is realized, and what the desired effect of applying the operator should be, *i.e.,* how it is classified. Where necessary to distinguish from other operators, we furthermore added decision nodes for properties or technological spaces. Otherwise, the properties and technological spaces are stated in the Tables I-V. For instance, for abstract syntax is realized in a metamodel, and semantics is not required, the desired effect should be classified as an extension, and all the investigated properties of our studies should be fulfilled, then the only appropriate operator is *Metamodel Fragment Composition*. For the paths that are not covered in Figure 27 we did not find an operator in our study. This lets researchers identify uncharted territory, *e.g.,* no composition operator composes metamodels including semantics defined in code generators.

B. RQ2: Which Language Dimensions are Supported by Composition Operators?

Most language composition operators only support a subset of language constituents or may only be realized in certain technological spaces. Therefore, they are only applicable to some implementations of language constituents. For example, some language composition operators may only support the composition of abstract or concrete syntax, whereas others also enable the composition of semantics (meaning [27]). Furthermore, some language composition operators operate on metamodels, whereas others operate on context-free grammars. With this research question, we aim to identify which language composition operators support which language constituents and which implementations. In particular, we are interested in language composition operators that support the composition of semantics and whether or not any language composition operators support a multitude of implementations.

Regarding language constituents, our primary question is if any language composition operators support both the composition of syntax and semantics. While all identified operators support the composition of language syntax in one form or another, only 10 (40%) of the 25 identified language composition operators support the composition of semantics. For the composition of abstract syntax, composition operators mainly operate on grammars (40%) and metamodels (40%). On these, the composition of languages' abstract syntax is well-understood. Other composition operator utilizes GPL classes [70] or employs RDF graphs [65] for abstract syntax composition. For the remaining language composition operators, the respective publications claimed to support the composition of abstract syntax. However, these did not specify how the abstract syntax is defined. On both grammars and metamodels, operators exist that support language extension (*cf.* Table I and Table III), unification (*cf.* Table II and Table IV), restriction (*cf.* Table V). There are language composition operators such as language extensions, aggrega-

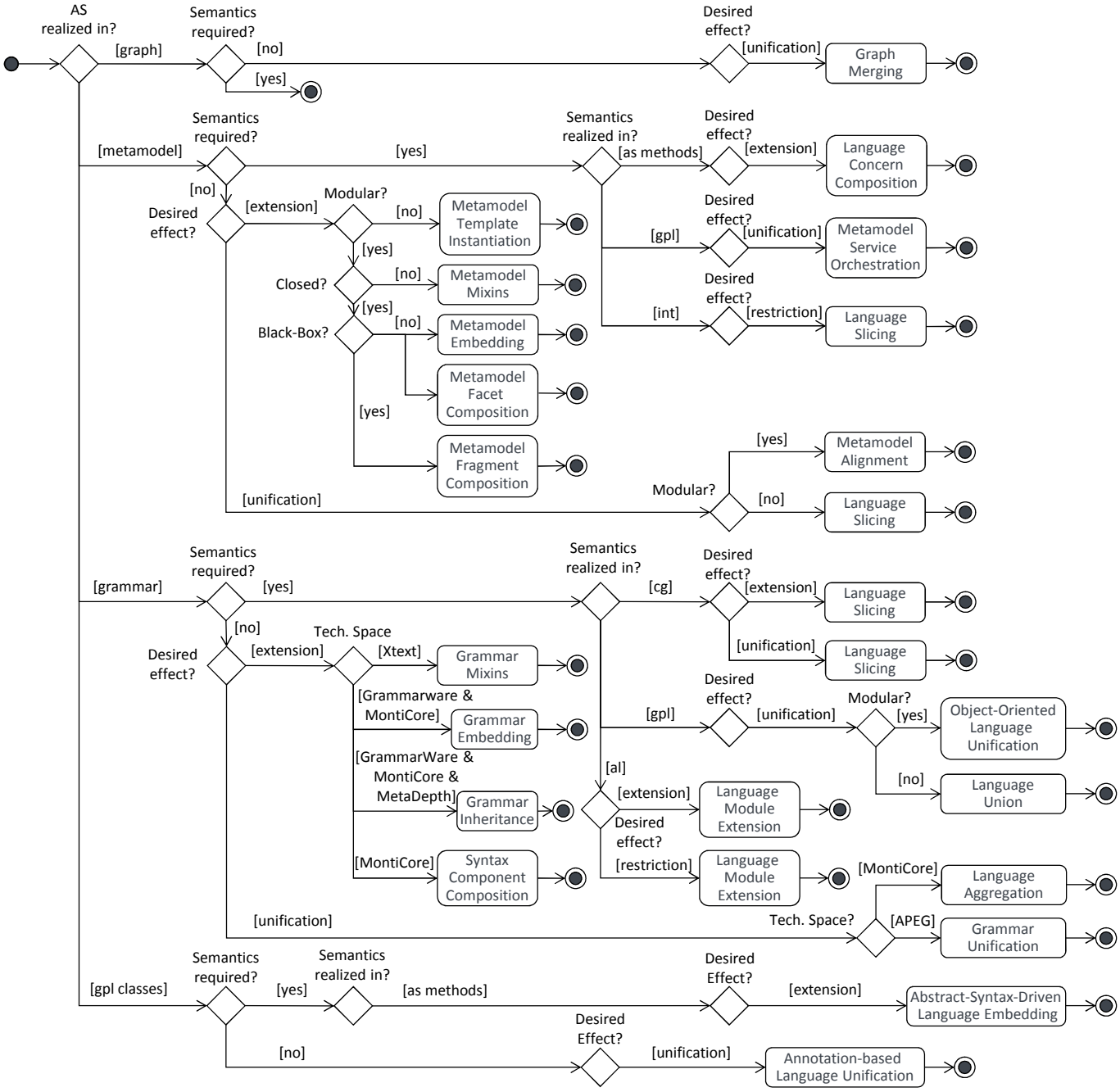


Figure 27: A guide to choosing a suitable composition operator for specific needs. Whenever an option for a decision is not available, we found no suitable operator in our study for it.

tion, and language embedding for the composition of abstract syntax defined by grammars, and language merging and slicing for the composition of abstract syntax defined by metamodels. For the definition of language semantics, language definitions rely on code generators, internal and external interpreters, and aspects. For all these, at least one operator exists that supports their composition. Furthermore, all of these can be combined with the common concepts for abstract syntax definition, *i.e.,* grammars and metamodels.

Besides the combined support for language syntax and semantics, we were also interested in whether language composition operators support different implementations, *e.g.,* the composition of grammars with metamodels. Such composition operators could help to bridge the gap between technological spaces and foster the reuse of legacy languages. Furthermore, the composition of grammars with metamodels would enable language developers to benefit from both concepts, utilizing their strengths and mitigating their weaknesses. Through

such composition operators, for example, language developers could utilize grammars' close connection of concrete and abstract syntax to define the syntax-heavy part of a language, such as expressions, and utilize metamodels for the structural parts. However, in our study, we did not identify any operator that supports the composition of grammars with metamodels.

Besides the composition of grammars with metamodels, language composition operators should also support the composition of language semantics to ensure that also the meanings of their models can be composed. Otherwise, composing modeling languages is reduced to composing syntax and "gluing" the different semantics carrying artifacts together manually again. This entails that modeling language reuse through composition is limited to experts in the corresponding technological spaces. However, the findings of our corpus suggest that the semantics of modeling languages are largely realized through interpretation or translation (compiling), *i.e.,* the semantics are specified as code in the interpreter or through transformations that produce artifacts carrying the semantics of the models. This entails that the semantics of a modeling language often is implemented in code and polluted with technical details to an extent that complicates analyzing and composing their semantics realizations.

C. RQ3: Which properties do language composition operators have concerning being black-box, modular, additive, and closed under composition?

With this question, we aim to investigate the properties of the composition operators that we identified in our review. Thereby, we aim to identify how much knowledge about language specifics and internals is necessary to apply composition operators. Furthermore, we want to find out how modular the composed language is, *i.e.,* whether the composed languages continue to exist as identifiable and changeable artifacts from which changes are propagated to the composed language. Besides, we want to investigate which operators are additive and which are restrictive. Finally, we are interested in the operability of the operators, *i.e.,* whether they are closed under composition. Summarized, in this section, we aim at answering the following research questions:

RQ 3.1 How much knowledge about the internals of a language is necessary to perform the composition, *i.e.,* what are existing black-box approaches to language composition?

RQ 3.2 How modular are existing composition operators?

RQ 3.3 Are the existing operators additive or restrictive?

RQ 3.4 Are the existing operators closed under composition?

RQ 3.1: How much knowledge about language internals is necessary for composition?: With this research question, we want to investigate, how much knowledge about the internals of the languages to be composed is necessary. This includes knowledge about the technological space, the realization of the constituents, *e.g.,* is it important to know whether the abstract syntax is specified in a metamodel or a grammar, and also the knowledge about the internal structure of these artifacts, *e.g.,* is it important to know the right-hand side of a grammar production that we want to compose. A black-box composition operator for composing metamodels is the operator *Metamodel Fragment Composition* [56]. The operator introduces metamodel fragments as units of composition with contractually specified provided interfaces and required interfaces. A metamodel fragment encapsulates metamodel elements that contribute to either fragment implementation or fragment interface definition. To support information hiding, a fragment defines a set of interfaces, which hide the internal implementation of a fragment. By binding interfaces of two fragments, metamodels can be composed.

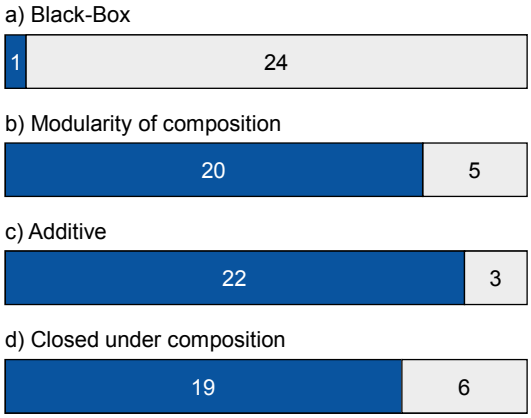


Figure 28: The number of operators with the properties of being a) black-box (1), b) modular (20), c) additive (22), and d) closed under composition (19).

The operators *Language Component Embedding* [71], [72] and *Language Component Unification* [77] are white-box composition operators employing language components representing language fragments that can be reused. Language components can comprise syntax and semantics. These constituents can then be provided for other components to be reused by the component's interface. The interface points of multiple components can then be bound to each other for the underlying language artifacts to be composed. Thus, the language engineer does neither require knowledge about the artifacts to realize the language nor their internal structure. For the automated composition of the realizing artifacts, the authors present composition operators for context-free grammars, well-formedness rules, and code generators realizing the semantics. However, when composing code generators via this mechanism, the language engineer needs to write glue code, thus, he needs white-box knowledge in this case. Hence, both operators are only black-box when considering the composition of syntax and context conditions, otherwise, they are white-box.

Another borderline white-box composition operator is *Language Module Extension* based on Neverlang [75]. In Neverlang, languages are defined at multiple levels of abstraction. The lowest level is modules. Modules can declare syntax and arbitrary many roles. Roles implement the semantics for the respective syntax. Slices, the next higher level of abstraction

are language components that comprise multiple modules that belong to one another. For example, a module containing a syntax definition for a while loop, a module with a role for checking whether the condition evaluates to a Boolean and a role for generating executable code. Slices then can be reused in languages where multiple slices are imported and composed. This seems like a black-box operator at first but requires white-box knowledge as there are no explicit bindings of roles or syntax rules in the language definition. Thus, when composing two slices containing syntax and roles, the language engineer has to know, which syntax rule fits another one, and which role has to be executed in which order.

RQ 3.2: How modular are the composition operators?: This research question aims at finding out, whether the result of the composition is modular. The result is modular if the composed parts continue to exist as identifiable artifacts in the composite. Figure 28 b) shows the findings of our literature review regarding this question. 80%, *i.e.,* 20 of the operators, are modular, whereas 20%, *i.e.,* 5 operators, are not. In the following, we present a modular and a non-modular composition operator. A modular composition operator is *Grammar Embedding* [47], [48], [49]. This operator takes a production of a host grammar and a production of a client grammar as input. The result is a new grammar with an extended production that embeds the client grammar's production into the host grammar's production. Both composed productions exist and are identifiable after the embedding. The involved grammars may be imported and the host production be extended by a new alternative on its right-hand side containing the client's production. Thus, all of the operator's input continues to exist after the composition.

A non-modular operator is *Metamodel Merging* [52], [69]. It takes two metamodels as input and produces one merged metamodel comprising all concepts of both metamodels and merged concepts that are shared across both metamodels. However, the result, *i.e.,* the merged metamodel, no longer provides information about the two source metamodels or which concepts originate from which metamodel.

RQ 3.3: Are the composition operators additive or restrictive?: We consider a language composition operator as additive when the operator only adds language constituents to a language definition, whereas a restrictive operator removes constituents. All of the found language composition operators classified as language extension or language unification are additive. For instance, *Abstract-Syntax-Driven Language Embedding* supports language unification by extending an existing language with concepts that enable the embedding of concepts of a second language. However, there are operators that are additive in the language definition but can be restrictive on the language that results. *Syntax Component Composition* is additive in the way that it enables to add grammar productions and context conditions to a language definition. Since context conditions add constraints based on the abstract syntax of a language adding context conditions can restrict a language. There are two restrictive language composition operators. *Language Slicing* gets a metamodel class and a metamodel and removes the class and all its dependencies from the metamodel recursively and, thereby, restricts the language and its definition.

RQ 3.4: Are the composition operators closed under composition?: We consider a language composition operator as closed under composition if it takes two language fragments of the same type (*e.g.,* two grammars) and produces another fragment of that type, such that the result of the composition again can serve as input for another composition. This facilitates reuse and language evolution. Figure 28 d) shows our results on composition operators that are closed under composition. Most of the operators (76%) are closed under composition. One of the operators that is not closed under composition is *Metamodel Mixins* [57]. The operator for metamodel mixin takes two inputs, a parent element and a mixin element. A parent element is a compound metamodel class that can contain other elements such as attributes or reference other metamodel classes. A mixin element is a compound element that must be abstract to be reused. The result of the operator is an extended metamodel with the mixins mixed in the parent element. However, the result of the operator, the extended metamodel, cannot be reused as a mixin element in another composition step.

VI. Related Work

Systematic mapping studies [80] and systematic literature reviews [81] are common methods for investigating state-of-the-art and open challenges in software engineering. Several mapping studies and literature reviews examining the current state of DSLs, their reuse, and variability exist today.

Studies on the Engineering of DSLs

A systematic mapping study on DSLs [82] investigates, which techniques and methods are used when working with DSLs. Regarding the composition of DSLs, their mapping study states that the development and tooling for single DSLs is well-studied, but research on the interaction and integration of multiple DSLs is still ongoing. Another systematic mapping study on DSLs [83] points out that there is a lack of research in the direction of validation and maintenance. We think, that by giving DSL developers a clear understanding of the existing landscape of composition operators, maintenance could be improved by better modularization and reusability of DSLs. However, both mapping studies aim at giving a general overview of techniques for DSL development and which DSLs are created with which tools and for which domain, instead of being focused on the reuse and composition of DSLs. For managing variability in DSLs, the concept of language product lines has emerged. A systematic literature review on language product lines [84] analyses the capabilities of currently existing approaches. The paper identifies three dimensions of variability: (1) Abstract syntax variability describes the capability to select suitable language constructs for a particular user, (2) concrete syntax variability to the support selecting a different representation of the same construct, and (3) semantic variability is supported when different interpretations for a

language construct can be selected. For capturing all of these dimensions of variability, different feature model representations are proposed in the literature. One possibility is feature models supporting functional variability where one feature is associated with one language that comprises all dimensions. Another one is multi-dimensional variability with concern-specific features, where each feature comprises either abstract syntax, concrete syntax, or semantics, but it is not mandatory to comprise all aspects at once. The third possibility is multi-dimensional variability with concern-specific subtrees. In this feature tree, all dimensions of variability are one abstract feature and the concerns are then children of the respective dimension feature. Besides, for supporting language modularization the review differentiates between two techniques: (1) In endogenous modularity relationships between languages are defined as part of the language definition itself. Usually, this is done via import statements. By importing another language definition, all of its language constructs are available. (2) In exogenous modularity, however, the relationship between languages is defined externally in third-party artifacts that are then input for the composition process of both languages. For future research, the paper identifies the analysis, evaluation, and evolution of language product lines as open challenges. Since the paper investigates language product lines and the modularity of languages that should be incorporated into these, the composition of languages or identifying language composition operators and their specifics is not the main focus of the literature review.

Studies on Language Workbenches

A comparison of language workbenches [31], extracts a feature model on the realization of constituents of modeling languages, *i.e.,* notation, semantics, validation, editor, and composability, in different language workbenches. The results of the comparison on composability show that most of the investigated language workbenches support incremental extension and language unification across all constituents of their respective language specifications. To evaluate their results, they performed various benchmarks in the course of the language workbench challenge. Although the comparison partly investigated composability, this is not the main focus of the study and is also limited to the scope of language workbenches, whereas our review also included operators conceptualized without having a specific technological space of a language workbench in mind.

The survey performed in [85] provides an overview of DSL implementation aspects and classifies 14 language workbenches according to these implementation approaches. Besides the implementation aspects for language structure, *i.e.,* syntax, language semantics, language validation, and language editors, they consider language composability as one implementation aspect of DSLs. The investigated language workbenches support composability in different means. For instance, Xtext supports only language extension and from only one grammar and thus, does not support language extension composition or unification. Enso realizes composability by defining its merge operator between two grammars resulting in a new combined language. Although the authors mention different composition operators, they do not argue, why and in which way they support the classifications of [30]. Furthermore, their investigations on composition operators are limited to the scope of language workbenches and potentially missing operators described on a conceptual level only.

Language Composition Classifications

We based the classification of extracted composition operators based on [30]. However, there are other classifications in the literature, that we could have used instead. For instance, classify between language extension and specialization [86]. The former describes the extension of a base language with additional concepts, and the latter the specialization by restricting a base language. We consider both classifications in this paper but adopt the more precise differentiation in language extension and call specialization restriction instead. Besides, papers describing composition operators also name new composition classifications like merging [54], [57], inheritance, slicing, and mixin [65], [68], [69].

However, we only found a few implementations of each classification and found that they could be classified as language extension and unification according to the definition of Erdweg et. al.. A literature study on model composition [87] classifies studies into model-driven development-oriented, aspect-oriented modeling, collaborative programming, and domain-specific languages. Only the last classification is focused on language composition that propagates the composition to the model level. Nonetheless, the study's scope is model composition, and therefore they do not provide any new language composition classifications.

VII. Discussion

This section summarizes our observations, identifies future challenges for language composition, and discusses threats to the validity of our literature review.

A. Observations

Regarding the classification of the composition operators that we found, most of the operators belong to the classification of language extension. All of the identified composition operators across all classifications support the composition of syntax. However, there are differences in the realization of syntax specifications between the operators. Language extension composing syntax is entailed in the composition of grammars, metamodels, and GPL classes. The operators classified into language restriction are even more restrictive by supporting grammars and metamodels only. In that regard, we observed that there are no composition operators for restricting syntax solely. Consequently, there are many unexplored paths in our guide for practitioners that are not covered by current publications on language composition operators, and, thus, are open to future research (*cf.* Figure 27). The specification of a language's constituents depends on the technological

space. We observed that two-thirds of the composition operators are specific to a single technological space or at least are presented that way. The operators that were described without a technological space were either pure conceptual work using pseudo code or technology-unspecific description techniques, *e.g.,* MOF models as metamodels. Since composition operators and language workbench are strongly intertwined, investigating how the composition operators can be realized using different language workbenches would be a great opportunity for future research. Regarding modularity, most operators are modular. This is also the case for closed under composition. The operators that are not closed under composition and do not compose language artifacts of the same type; instead, these operators use the language as one input and a second input specifying the operation or the element to be modified, *e.g.,* removed. Another observation is that, besides the operators for reduction, all operators are additive. We found that black-box capabilities are rare with only one operator supporting this property. Taking a look at the different classifications for the operators shows that there exists one black-box operator for syntax extension only. No black-box composition operator exists regarding the extension of syntax with semantics, unification, and restriction. Since only five of the found operators provide formal descriptions, *i.e.,* making an effort to describe their composition operators mathematically, it is unclear whether some of these cases describe different mechanisms or whether they are essentially the same but in slightly different contexts, for instance, at different levels of abstraction. Also, our findings indicate the usefulness or fitness of using a language composition operator along the classification of Erdweg et. al. and the properties that we observed. However, the suitability for a specific purpose or use case cannot be indicated as they may feature other properties outside of the scope of this paper, *e.g.,* whether they can be applied fully automatically or whether the application can produce conflicts that need to be resolved (manually).

In summary, many extension operators for external modeling languages focus on abstract syntax, either specified through grammars or metamodels of which all are additive and none is usable in a black-box fashion. There also are some operators for the extension of modeling languages featuring syntax and semantics. The prevalent means to compose semantics seems to be by including methods realizing the semantics in the abstract syntax and the use of the same (mostly object-oriented) composition mechanisms for both syntax and semantics. This eases the composition as fewer different composition mechanisms need to be understood then. For the unification of modeling languages including syntax and semantics, only four operators were identified and these use different mechanisms each. Likewise, for the restriction of modeling languages, only two operators were identified, one removing metamodel elements and the other removing grammar rules. A side effect of the diversity of modeling language composition operators being spread across different technological spaces is that there currently is no technological space supporting all kinds of language composition operators as outlined in Sec. II.

Of course, the pure technical availability of a modeling language composition operator is not sufficient for its adoption. This requires also awareness of their existence by practitioners as well as support in sufficiently sophisticated language workbenches. For the former, surveys or interviews with practitioners would be necessary, for the latter, a detailed analysis of implementations in the different workbenches. Both of which should be the subject of future research.

For this survey we opted to reuse the classification schema of [30], to ensure some comparability with their initial work on language composition. Naturally, there are other means to classify modeling language composition operators as well, such as the distinction into merging, inheritance, and slicing [68] or referencing, embedding, and extension [88]. Such classifications usually stem from a specific technological background, such as metamodeling [68], and, hence, are partly applicable to general modeling language composition.

B. Challenges

Our observations regarding existing language composition operators and their properties lead to challenges for future investigations:

1.) Black-box Composition
Furthermore, composition operators should be easy to learn and apply. Black-box composition of languages can help with that as the implementation details of the languages to be composed remain hidden. In our study, we could only identify one black-box composition operator. Beyond these, further black-box operators supporting various kinds of language composition operators may be developed.

2.) Heterogeneous Composition
Language composition is a means of language (part) reuse. As such, language composition operators should support the composition of languages and language parts across different technological spaces (*e.g.,* embedding a Neverlang language in an Xtext language). However, none of the identified operators supports such heterogeneous language reuse.

3.) Automated Composition
Another challenge is the automation of the composition, *i.e.,* how much-handwritten extension is necessary after the composition took place. This is especially important for black-box approaches because otherwise, white-box knowledge is needed after the composition to finish integrating all parts of the composed languages. For instance, the composition operator *Language Component Embedding* [71], [72] is black-box regarding the composition of syntax and context conditions, but requires a handwritten adaptation of the generator after the composition for the generators of both languages to be integrated, and, thus, ultimately, is white-box.

4.) Alignment of Operators
In summary, we identified 36 publications reporting on 25 distinct language composition operators in the past ten years.

Hence, much effort is put into the development of new composition operators. We took a detailed look into the functioning of the operators on the language definition level and distinguished the operators based on that. However, in the future, their effect on model level could be investigated further to align operators on that level, too. From that operators could be further generalized to be reusable across technological spaces. Furthermore, another literature review could investigate what the most frequently used operators are. With this knowledge, we could tackle the challenges mentioned before for these operators first.

5.) Formalization

In this paper, we informally described 25 different composition operators. This is, because only 8 of 25 and only 6 out of 39 papers provide a formal description of their presented operator. As we are conducting a secondary study, we can only report on what is available. Where formal descriptions were provided, these usually build upon either a specific algebraic or logical theory built up in the corresponding publication. Future investigations could be formalizing existing operators to target exact definitions and relations between the various approaches and operators.

C. Threats to Validity

We identify threats to validity according to the four basic types of validity threats [89]: Our study is subject to threats to construct validity (research design), internal validity (data extraction), and conclusion validity (reliability). Threats to external validity (generalizability) are irrelevant as the results of our study cannot be generalized to other domains besides software languages and their composition.

Construct validity: the presented findings are only valid for our sample of papers. Thus, we ensured to include as many relevant papers as possible. To achieve this, we included the ACM digital library, SCOPUS, WOS, Spring Link, and IEEE Explore and only very carefully under the given exclusion criteria, excluded publications. Furthermore, we did not restrict our search query to only "language composition" but also included other terms for language like "DSL" or "grammar" for the definition of a language's syntax in the first part of our conjunction. For the second part, we also have chosen three more synonymous terms for "composition". This enabled capturing related publications without focussing on the exact, very specific, partly ambiguous "language" and "composition" terminology. Another threat to research design validity arises from the definition of the criteria of inclusion and exclusion. During the screening, we only considered the title, abstract, and keywords. To prevent excluding relevant publications based on the lack of investigation, we included papers we were uncertain of temporarily. In the subsequent phase, the complete papers were read and inclusion or exclusion was decided ultimately. Furthermore, in this step, all authors of this paper read 25% of the potentially relevant papers, filled out the analyses sheet, and discussed these papers together by comparing each other's sheets. This helped us get more confident in the analysis of the subsequent papers and improved our understanding of the questions in the analysis sheet. Besides, our review also is subject to the so-called publication bias, *i.e.,* it can report on published results only. Thus, we can only report on composition operators found in the literature. Furthermore, in cases where authors claim that their composition operators apply to certain language constituents without proving their claim, we have to trust their scientific ethics.

Internal validity: copes with problems arising during data extraction. Of course, our study relies on the quality of the primary studies. The most important threat regards the terminology used in the different publications to describe the composition operators. To deal with this issue we discussed terminology among the authors during the classification phase and agreed on the names in Sec. V-A. Another threat to internal validity is the description of the operators. Some papers provide a fine-grained description, whereas others only provide a cursory introduction to their operator. To counteract this issue we designed a detailed questionnaire that was required to be filled out for each operator. We excluded all publications that did not provide a sufficient description to fill out our questionnaire.

Conclusion validity: Threats here are making wrong conclusions and a lack of replicability. Regarding the former, we have discussed various issues that could lead to wrong conclusions in the context of threats to internal validity. For the study's replicability, we detailed the complete research method in Sec. IV, which enables replicating every phase of this mapping study.

VIII. Conclusion

We investigated the state of language composition based on the categories identified by Erdweg et al [30] ten years ago through a systematic literature review. Our findings amplify their categories as we did not find a language composition operator outside of these categories. Based on a corpus of 36 relevant publications, we identified 25 composition operators. We detailed the identified operators regarding the language constituents they consider, the technological space they operate within, and whether they are modular, closed, additive, or require white-box expertise for their application. We found some operators considering both syntax and semantics, which suggest that there are approaches to holistic language reuse.

Based on the insights about the identified operators, we identified four important challenges for easing language reuse even further, which are (1) composition across technological spaces, (2) in a black-box fashion, (3) fully automated, as well as (4) improving our understanding of the commonalities and differences of operators to guide research on language composition and enable investigation of language extension composition.

Overall, the findings reported in this study draw a map of composition operators, that can guide practitioners in identifying the language composition mechanisms they need for specific challenges based on our classification related to

Erdweg et. al. and the properties we analyzed. However, there are still unexplored paths in our map that are up to future investigations on the topic.

ACKNOWLEDGEMENTS

Funding: The authors of the University of Stuttgart were supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) [grant number 441207927].

REFERENCES

[1] A. Kleppe, *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[2] K. Hölldobler, B. Rumpe, and A. Wortmann, "Software language engineering in the large: towards composing and deriving languages," *Computer Languages, Systems & Structures*, vol. 54, pp. 386–405, 2018.

[3] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. Wenckstern, and A. Wortmann, "SMArDT modeling for automotive software testing," *Software: Practice and Experience*, vol. 49, no. 2, pp. 301–328, February 2019.

[4] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, 2007.

[5] B. Annighoefer, M. Halle, A. Schweiger, M. Reich, C. Watkins, S. H. VanderLeest, S. Harwarth, and P. Deiber, "Challenges and ways forward for avionics platforms and their development in 2019," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*. IEEE, 2019, pp. 1–10.

[6] G. Hinkel, H. Groenda, L. Vannucci, O. Denninger, N. Cauli, and S. Ulbrich, "A domain-specific language (DSL) for integrating neuronal networks in robot control," in *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, 2015, pp. 9–15.

[7] T. C. Son, E. Pontelli, D. Ranjan, B. Milligan, and G. Gupta, "An agent-based domain specific framework for rapid prototyping of applications in evolutionary biology," in *International Workshop on Declarative Agent Languages and Technologies*. Springer, 2003, pp. 76–96.

[8] M. R. Lakin and A. Phillips, "Domain-specific programming languages for computational nucleic acid systems," *ACS Synthetic Biology*, vol. 9, no. 7, pp. 1499–1513, 2020.

[9] W. R. Saunders, J. Grant, and E. H. Müller, "A domain specific language for performance portable molecular dynamics algorithms," *Computer Physics Communications*, vol. 224, pp. 119–135, 2018.

[10] D. Elshani, A. Lombardi, A. Fisher, S. Staab, D. Hernández, and T. Wortmann, "Knowledge Graphs for Multidisciplinary Co-Design: Introducing RDF to BHoM," *Linked Data in Architecture and Construction*, 2020.

[11] M. Voelter, S. Koščejev, M. Riedel, A. Deitsch, and A. Hinkelmann, "A domain-specific language for payroll calculations: an experience report from datev," in *Domain-Specific Languages in Practice*. Springer, 2021, pp. 93–130.

[12] L. T. Van Binsbergen, L.-C. Liu, R. van Doesburg, and T. van Engers, "eFLINT: a domain-specific language for executable norm specifications," in *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2020, pp. 124–136.

[13] A. Lüder and N. Schmidt, "AutomationML in a Nutshell," in *Handbuch Industrie 4.0 Bd. 2*. Springer, 2017, pp. 213–258.

[14] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in Industry 4.0: an extended systematic mapping study," *Software and Systems Modeling*, vol. 19, no. 1, pp. 67–94, January 2020.

[15] O. Boiarskyi and S. Popereshnyak, "Automated system and domain-specific language for medical data collection and processing," in *International Scientific Conference "Intellectual Systems of Decision Making and Problem of Computational Intelligence"*. Springer, 2021, pp. 377–396.

[16] Y. Gu, R. Tinn, H. Cheng, M. Lucas, N. Usuyama, X. Liu, T. Naumann, J. Gao, and H. Poon, "Domain-specific language model pretraining for biomedical natural language processing," *ACM Transactions on Computing for Healthcare (HEALTH)*, vol. 3, no. 1, pp. 1–23, 2021.

[17] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *International conference on simulation, modeling, and programming for autonomous robots*. Springer, 2014, pp. 195–206.

[18] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, Tech. Rep., 2006.

[19] S. Wolny, A. Mazak, C. Carpella, V. Geist, and M. Wimmer, "Thirteen years of SysML: a systematic mapping study," *Software and Systems Modeling*, vol. 19, no. 1, pp. 111–169, 2020.

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

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

[22] S. N. Voogd, K. Aslam, L. Van Gool, B. Theelen, and I. Malavolta, "Real-Time Collaborative Modeling across Language Workbenches–a Case on Jetbrains MPS and Eclipse Spoofax," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2021, pp. 16–26.

[23] M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen, "Lessons learned from developing mbeddr: a case study in language engineering with MPS," *Software & Systems Modeling*, vol. 18, no. 1, pp. 585–630, 2019.

[24] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "ATL: a QVT-like transformation language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 719–720.

[25] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2008, pp. 46–60.

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

[27] D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of "Semantics"?" *IEEE Computer*, vol. 37, no. 10, pp. 64–72, October 2004.

[28] J.-M. Favre, D. Gasevic, R. Lämmel, and E. Pek, "Empirical language analysis in software linguistics," in *International Conference on Software Language Engineering*. Springer, 2010, pp. 316–326.

[29] T. Clark, M. v. d. Brand, B. Combemale, and B. Rumpe, "Conceptual Model of the Globalization for Domain-Specific Languages," in *Globalizing Domain-Specific Languages*, ser. LNCS 9400. Springer, 2015, pp. 7–20.

[30] S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language composition untangled," in *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, ser. LDTA '12. New York, NY, USA: Association for Computing Machinery, 2012.

[31] S. Erdweg, T. Van Der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems & Structures*, vol. 44, pp. 24–47, 2015.

[32] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: A controlled experiment," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 763–774.

[33] M. Barash, "Example-driven software language engineering," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, 2020, pp. 246–252.

[34] L.-E. Lafontant and E. Syriani, "Gentleman: a light-weight web-based projectional editor generator," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020, pp. 1–5.

[35] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel *et al.*, "Concern-oriented language development (COLD): Fostering reuse in language engineering," *Computer Languages, Systems & Structures*, vol. 54, pp. 139–155, 2018.

[36] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," *Future of Software Engineering (FOSE '07)*, pp. 37–54, May 2007.

[37] T. Degueule, T. Mayerhofer, and A. Wortmann, "Engineering a ROVER Language in GEMOC STUDIO & MONTICORE: A Comparison of Language Reuse Support," in *Proceedings of MODELS 2017. Workshop EXE*, ser. CEUR 2019, September 2017.

[38] M. Broy and K. Stølen, *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer Science & Business Media, 2012.

[39] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, no. 3, pp. 223–252, 1977.

[40] D. Budgen and P. Brereton, "Performing systematic literature reviews in software engineering," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 1051–1052.

[41] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," Technical report, ver. 2.3 ebse technical report. ebse, Tech. Rep., 2007.

[42] Z. Stapic, E. G. López, A. G. Cabot, L. de Marcos Ortega, and V. Strahonja, "Performing systematic literature review in software engineering," in *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin, 2012, p. 441.

[43] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014, pp. 1–10.

[44] M. Boeker, W. Vach, and E. Motschall, "Google scholar as replacement for systematic literature searches: good relative recall and precision are not enough," *BMC medical research methodology*, vol. 13, no. 1, pp. 1–12, 2013.

[45] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in industry 4.0: an extended systematic mapping study," *Software and Systems Modeling*, vol. 19, no. 1, pp. 67–94, 2020.

[46] M. Nosál, M. Sulír, and J. Juhár, "Language composition using source code annotations," *Computer Science and Information Systems*, vol. 13, no. 3, pp. 707–729, 2016.

[47] A. Butting, K. Hölldobler, B. Rumpe, and A. Wortmann, "Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques—The MontiCore Approach," in *Composing Model-Based Analysis Tools*. Springer, 2021, pp. 217–234.

[48] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, and A. Wortmann, "Composition of heterogeneous modeling languages," in *International Conference on Model-Driven Engineering and Software Development*. Springer, 2015, pp. 45–66.

[49] L. Diekmann and L. Tratt, "Parsing composed grammars with language boxes," in *Workshop on Scalable Language Specifications*, 2013.

[50] A. Johnstone, E. Scott, and M. van den Brand, "Modular grammar specification," *Science of Computer Programming*, vol. 87, pp. 23–43, 2014.

[51] A. Haber, M. Look, A. N. Perez, P. M. S. Nazari, B. Rumpe, S. Völkel, and A. Wortmann, "Integration of heterogeneous modeling languages via extensible and composable language components," in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, 2015, pp. 19–31.

[52] B. Meyers, A. Cicchetti, E. Guerra, and J. De Lara, "Composing textual modelling languages in practice," in *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, 2012, pp. 31–36.

[53] J. Andersen, C. Brabrand, and D. R. Christiansen, "Banana algebra: Compositional syntactic language extension," *Science of Computer Programming*, vol. 78, no. 10, pp. 1845–1870, 2013.

[54] C. Rieger, M. Westerkamp, and H. Kuchen, "Challenges and opportunities of modularizing textual domain-specific languages." *MODELSWARD*, pp. 387–395, 2018.

[55] A. Abouzahra, A. Sabraoui, and K. Afdel, "A metamodel composition driven approach to design new domain specific modeling languages," in *2017 European Conference on Electrical Engineering and Computer Science (EECS)*. IEEE, 2017, pp. 112–118.

[56] S. Živković and D. Karagiannis, "Towards metamodelling-in-the-large: Interface-based composition for modular metamodel development," in *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2015, pp. 413–428.

[57] S. Źivkoviź and D. Karagiannis, "Mixins and extenders for modular metamodel customisation," in *Proceedings of the 18th International Conference on Enterprise Information Systems*, 2016, pp. 259–270.

[58] H. Berg and B. Møller-Pedersen, "Type-safe symmetric composition of metamodels using templates," in *International Workshop on System Analysis and Modeling*. Springer, 2012, pp. 160–178.

[59] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic composition of independent language features," *Journal of Systems and Software*, vol. 152, pp. 50–69, 2019.

[60] Butting, Arvid and Eikermann, Robert and Kautz, Oliver and Rumpe, Bernhard and Wortmann, Andreas, "Controlled and extensible variability of concrete and abstract syntax with independent language features," in *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, 2018, pp. 75–82.

[61] J. de Lara, E. Guerra, J. Kienzle, and Y. Hattab, "Facet-oriented modelling: open objects for model-driven engineering," in *Proceedings of the 11th acm sigplan international conference on software language engineering*, 2018, pp. 147–159.

[62] J. D. Lara, E. Guerra, and J. Kienzle, "Facet-oriented Modelling," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–59, 2021.

[63] T. van Der Storm, W. R. Cook, and A. Loh, "Object grammars," in *SLE*. Springer, 2012, pp. 4–23.

[64] L. V. Reis, V. O. Di Iorio, and R. S. Bigonha, "An on-the-fly grammar modification mechanism for composing and defining extensible languages," *Computer Languages, Systems & Structures*, vol. 42, pp. 46–59, 2015.

[65] B. Braatz and C. Brandt, "A framework for families of domain-specific modelling languages," *Software & Systems Modeling*, vol. 13, no. 1, pp. 109–132, 2014.

[66] N. Essadi and A. Anwar, "Coordination between heterogeneous models using a meta-model composition approach," *Advances in Science, Technology and Engineering Systems Journal*, vol. 4, 01 2019.

[67] P. Stünkel, H. König, Y. Lamo, and A. Rutle, "Towards multiple model synchronization with comprehensive systems." in *FASE*, 2020, pp. 335–356.

[68] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A meta-language for modular and reusable development of dsls," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, 2015, pp. 25–36.

[69] F. Rabbi, Y. Lamo, and L. M. Kristensen, "A model driven engineering approach for heterogeneous model composition," in *International Conference on Model-Driven Engineering and Software Development*. Springer, 2017, pp. 198–221.

[70] S. Chodarev, D. Lakatoš, J. Porubän, and J. Kollár, "Abstract syntax driven approach for language composition," *Open Computer Science*, vol. 4, no. 3, pp. 107–117, 2014.

[71] A. Butting, J. Pfeiffer, B. Rumpe, and A. Wortmann, "A compositional framework for systematic modeling language reuse," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 35–46.

[72] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Modeling language variability with reusable language components," in *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*, 2018, pp. 65–75.

[73] M. Leduc, T. Degueule, and B. Combemale, "Modular language composition for the masses," in *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, 2018, pp. 47–59.

[74] W. Cazzola and E. Vacchi, "Language components for modular dsls using traits," *Computer Languages, Systems & Structures*, vol. 45, pp. 16–34, 2016.

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

[76] M. Mernik, "An object-oriented approach to language compositions for software language engineering," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2451–2464, 2013.

[77] J. Pfeiffer and A. Wortmann, "Towards the black-box aggregation of language components," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2021, pp. 576–585.

[78] M. Cimini, "On the effectiveness of higher-order logic programming in language-oriented programming," in *International Symposium on Functional and Logic Programming*. Springer, 2020, pp. 106–123.

[79] H. Berg, "Service-oriented design of metamodel components," in *ICSOFT*, 2012, pp. 70–79.

[80] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*, 2008, pp. 1–10.

[81] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.

[82] L. M. do Nascimento, D. L. Viana, P. Neto, D. Martins, V. C. Garcia, and S. Meira, "A systematic mapping study on domain-specific languages," in *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 2012, pp. 179–187.

[83] T. Kosar, S. Bohra, and M. Mernik, "Domain-specific languages: A systematic mapping study," *Information and Software Technology*, vol. 71, pp. 77–91, 2016.

[84] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry, "Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review," *Computer Languages, Systems & Structures*, vol. 46, pp. 206–235, 2016.

[85] E. Negm, S. Makady, and A. Salah, "Survey on domain specific languages implementation aspects," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 11, 2019.

[86] D. Spinellis, "Notable design patterns for domain-specific languages," *Journal of systems and software*, vol. 56, no. 1, pp. 91–99, 2001.

[87] A. Abouzahra, A. Sabraoui, and K. Afdel, "Model composition in Model Driven Engineering: A systematic literature review," *Information and Software Technology*, vol. 125, p. 106316, 2020.

[88] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, "DSL Engineering - Designing, Implementing and Using Domain-specific Languages," 2013.

[89] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.