Melange: A Meta-language for Modular and Reusable Development of DSLs

Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, Jean-Marc Jézéquel

Melange: A Meta-language for Modular and Reusable Development of DSLs

Thomas Degueule
INRIA, France
thomas.degueule@inria.fr

Benoit Combemale
INRIA, France
benoit.combemale@inria.fr

Arnaud Blouin
INSA Rennes, France
arnaud.blouin@irisa.fr

Olivier Barais
University of Rennes 1, France
olivier.barais@irisa.fr

Jean-Marc Jézéquel
University of Rennes 1, France
jezequel@irisa.fr

Abstract

Domain-Specific Languages (DSLs) are now developed for a wide variety of domains to address specific concerns in the development of complex systems. When engineering new DSLs, it is likely that previous efforts spent on the development of other languages could be leveraged, especially when their domains overlap. However, legacy DSLs may not fit exactly the end user requirements and thus require further extension, restriction, or specialization. While current language workbenches provide import mechanisms, they usually lack an explicit support for such customizations of imported artifacts. In this paper, we propose an approach for building DSLs by safely assembling and customizing legacy DSLs artifacts. This approach is based on typing relations that provide a reasoning layer for manipulating DSLs while ensuring type safety. On top of this reasoning layer, we provide an algebra of operators for extending, restricting, and assembling separate DSL artifacts. We implemented the typing relations and algebra into the Melange meta-language. We illustrate Melange through the modular definition of an executable modeling language for the Internet Of Things domain. We show how it eases the definition of new DSLs by maximizing the reuse of legacy artifacts without introducing issues in terms of performance, technical ecosystem compatibility, or generated code volume.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Specialized application languages

Keywords Domain-specific languages, language reuse, language composition, model typing, melange

1. Introduction

Extending the time-honored practice of separation of concerns, Domain-Specific Languages (DSLs) are increasingly used to handle different, complex concerns in software-intensive system development. However, the definition of a DSL and the associated tools (*i.e.* checkers, simulators, generators) require significant development efforts for, by definition, a limited audience and a DSL that is doomed to evolve as the concepts in the domain and the expert understanding of the domain evolve.

Despite the wide range of domains in which DSLs are used and their constant evolution, many of them are close and share commonalities such as a particular action language or a common paradigm (*e.g.* the family of DSLs for statecharts [6]). Recent work in the community of Software Language Engineering focused on language workbenches that support the modular design of DSLs, and the possible reuse of such *modules* (usually using a scattered clause `import` linking separate artifacts) [26, 43]. Besides, particular composition operators have been proposed for unifying or extending existing languages [33]. However, while most of the approaches propose either a diffuse way to reuse language modules, or to reuse as is complete languages, there is still little support for easily assembling language modules with customization facilities (*e.g.* restriction) in order to finely tune the resulting DSL according to the language designer's requirements.

In this paper, we present Melange, a tool-supported meta-language in which legacy DSLs are assembled and customized to produce new ones. Melange provides specific constructs to assemble various abstract syntax and operational semantics artifacts into a DSL. DSLs can then be used as first-class entities to be reused, extended, restricted or adapted into other DSLs. Melange relies on typing relations that statically ensure the structural correctness of the produced DSLs, and subtyping relations between DSLs to reason about their substitutability. Newly produced DSLs are correct by construction, ready for production (*i.e.* the result can be deployed and used as is), and reusable in a new assembly.

We illustrate the benefits of the proposed language operators and type system by designing a new executable modeling language for the Internet Of Things domain. We show how the proposed approach eases the definition of new DSLs by maximizing the reuse of legacy artifacts without introducing issues in terms of performance, technical ecosystem compatibility, or generated code volume.

The remainder of this paper is organized as follows. Section 2 gives an overview of the approach, further detailed with the algebra for DSL assembly and customization (Section 3) and its support in the dedicated meta-language Melange (Section 4). Then we illustrate our approach with a significant case study in Section 5. Finally, Section 6 discusses related work, and Section 7 concludes and discusses several perspectives of our work.

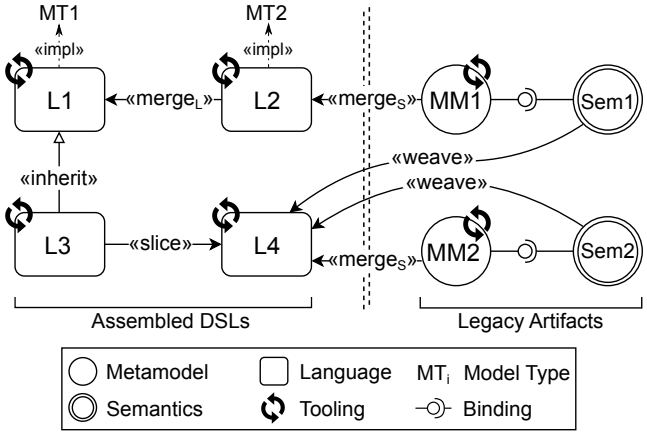


Figure 1: DSLs Assembly and Customization

2. Approach Overview

Domain-specific languages are typically defined through three main concerns: abstract syntax, concrete syntax(es) and semantics. Various approaches may be employed to specify each of them, usually using dedicated meta-languages [44]. The abstract syntax specifies the domain concepts and their relations and is defined by a metamodel or a grammar. This choice often depends on the language designer's background and culture. Examples of meta-languages for specifying the abstract syntax of a DSL include EMOF [1] and SDF [19]. The semantics of a DSL can be defined using various approaches including axiomatic semantics, denotational semantics, operational semantics, and their variants [36]. Concrete syntaxes are usually specified as a mapping from the abstract syntax to textual or graphical representations, *e.g.* through the definition of a parser or a projectional editor [46]. In this paper, we focus on DSLs whose abstract syntaxes are defined with metamodels and whose semantics are defined in an operational way through the definition of computational steps designed following the interpreter pattern [17]. Computational steps may be defined in different ways, *e.g.* using aspect-oriented modeling [23] or endogenous transformations [5]. In this paper, however, we only focus on the weaving of computational steps in an object-oriented (OO) fashion with the interpreter pattern. In such a case, specifying the operational semantics of a DSL involves the use of an action language to define methods that are statically introduced directly in the concepts of its abstract syntax [24]. It is worth noting that the proposed approach can easily be adapted to other kinds of operational semantics specification mechanisms, such as endogenous transformations in a functional way. We do not address in this paper the problem of concrete syntax composition and customization.

Figure 1 gives a high-level overview of our approach. On the right side are legacy language artifacts that must be reused and assembled to build new DSLs. Imported artifacts include abstract syntax and semantics, possibly with their corresponding tools and services (*e.g.* checkers, or transformations). These tools consist in manipulating models conforming to a particular metamodel. Similarly, semantic definitions directly access and manipulate model elements for execution or compilation purposes. Hence, abstract syntax and semantics are related one another through *binding* relations: semantics artifacts *require* a particular shape of abstract syntax, which is *provided* by a given metamodel. On the left side of Figure 1 are the newly built languages. *Assembly operators* (*merge*$_S$, *weave*) realize the transition from legacy artifacts to new DSLs. They import and connect disparate language artifacts, *e.g.* by merging different abstract syntaxes or by binding a given semantics to a new syntax. Naturally, the same artifacts can be reused in different assemblies. The output of assembly operators is encapsulated in a language definition. Once new assemblies are created, *customization operators* (*slice*, *merge*$_L$, *inherits*) offer the possibility to refine the newly built DSLs so as to meet additional requirements or to fit a specialized context. Both assembly and customization operators are captured in an algebra (*cf.* Section 3).

Assembling and customizing DSLs is a complex task that requires checking the composability of heterogeneous parts and the validity of the result. For example, based on Figure 1, it is clear that *Sem1* can be woven on *L4* only if it can be bound to its syntax *MM2*. Similarly, the intuitive meaning of inheritance, as found in most OO languages, implies the compatibility between the super- and sub- elements. It follows that the compatibility between *L1* and *L3* in Figure 1 must be ensured to guarantee that *L1*'s tooling can be reused for *L3*. What is missing here to guarantee these properties is an abstraction layer that would support reasoning about the compatibility between different languages artifacts. In our approach, we rely on the notion of model typing as introduced by Steel *et al.* [42] and further refined by Guy *et al.* [18]. Model types are structural interfaces over the abstract syntax of a language, defined by a metamodel. As such, they also take the form of a metamodel. They are linked one another by subtyping relations that specify if a model

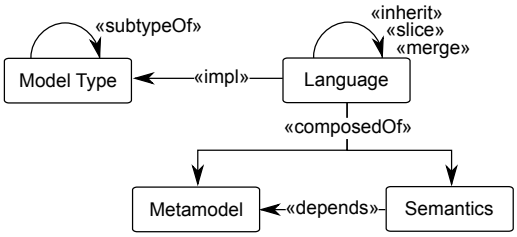


Figure 2: Model Typing Relations

conforming to a given metamodel can be manipulated through another metamodel. Several metamodels may *implement* the same model type, meaning that transformations and tools defined over a model type can be reused for all matching metamodels. Moreover, model typing allows to reason about the compatibility between different metamodels.

We further extend the concept of model typing by explicitly separating implementations of languages (*i.e.* abstract syntax, semantics, and tools) from their structural interfaces (*i.e.* model types exposing part of their features) as canonical representation of languages. As depicted in Figure 2, each language has at least one model type that captures its structural interface. Then, the associated type system enables reasoning about compatibility between different artifacts, *e.g.* to check whether a given semantics can be applied on a given abstract syntax, or to ensure that within an inheritance relation the sub-language remains compatible with the super-language.

3. An Algebra for DSL Assembly and Customization

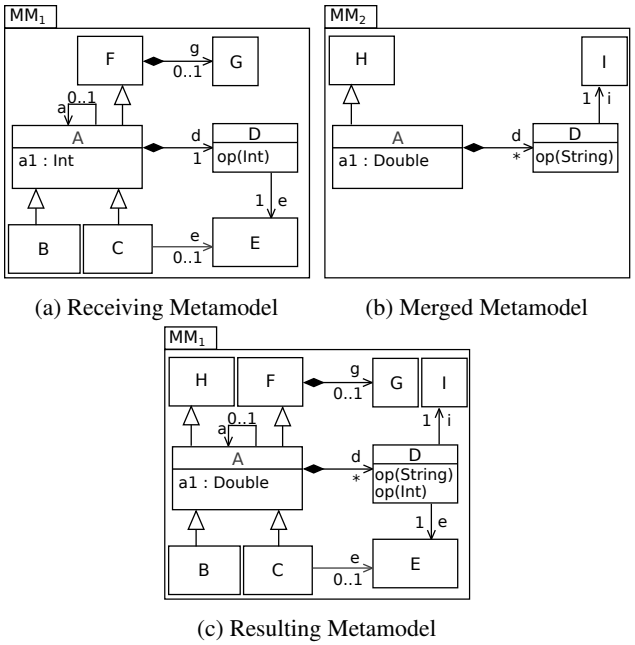
In this section, we introduce an abstract algebraic specification of operators for language assembly and customization. This specification is mainly intended to serve as a reference for the implementation of a concrete meta-language that would support the aforementioned approach. We first provide the definitions and concepts required to define the algebra (Section 3.1). Then, we introduce the operators for language assembly (Section 3.2) and customization (Section 3.3).

3.1 Language Definition

Based on the informal conceptual model of Section 2, we define a language $\mathcal{L}$ as a 3-tuple of its abstract syntax, semantics, and exact model type:

$$\mathcal{L} \triangleq \langle AS, Sem, MT \rangle$$

Including the exact model type of a language into the tuple allows to directly specify the impact of each of the operators of the algebra on the typing layer. As explained in Section 3.1.3, model types also indirectly support the reuse of languages tooling. In the following, for any language $\mathcal{L}$, we denote $AS(\mathcal{L})$ its abstract syntax, $Sem(\mathcal{L})$ its semantics, and $MT(\mathcal{L})$ its exact model type. On non-ambiguous cases, we simply refer to them as AS, Sem, and MT. The next sub-sections detail each of them.



(a) Receiving Metamodel (b) Merged Metamodel

(c) Resulting Metamodel

Figure 3: Syntax Merging Operator

3.1.1 Syntax and Syntax Merging

In our algebra, the abstract syntax AS of a language $\mathcal{L}$ is specified using a metamodel, *i.e.* a multigraph of classes and their relations. When assembling several abstract syntaxes, their concepts must be merged together so that the resulting abstract syntax is no less capable than its ancestors. Informally, this means that the abstract syntax resulting from the merge must incorporate concepts from all languages and merge the definitions of shared elements. In our specific case, merging several abstract syntaxes boils down to the problem of metamodel composition [10]. Figure 3 illustrates the syntax merging operator on a simple example. We use the terms *receiving metamodel*, *merged metamodel*, and *resulting metamodel* to refer to the three metamodels involved in the merging operation. Similarly, the terms *receiving language* and *resulting language* will be used throughout this section.

Depending on the meta-language used for defining metamodels, different merging operators may be employed with different policies for matching and merging rules, conflicts management, *etc.*. The choice of the concrete semantics of the syntax merging operator is left to the implementer of the algebra, and a concrete implementation is described in Section 4.3. In the remainder of this section, we denote $\circ$ the abstract syntax merging operator.

3.1.2 Semantics and Semantics Merging

The semantics Sem of a language $\mathcal{L}$ is defined by a sequence of aspect definitions A_i^t, where A is a class, t is a pointcut and i is the index of A in the sequence. In this case, the pointcut t specifies the concept of the language's abstract syntax (a meta-class) on which the aspect must be ultimately woven.

The advice is the class A itself, consisting of attributes and methods. When a joinpoint is found, *i.e.* when a matching concept is found in the language, elements of the advice are inserted in the target meta-class. Since aspects are defined using classes in an OO manner, they may inherit from each other. To cope with possible specialization and redefinition of methods, aspects are ordered by hierarchy in a sequence:

$$Sem(\mathcal{L}) \triangleq (A_i^t \in Aspects) \text{ where}$$
$$\forall A_i^t \in Sem(\mathcal{L}), \exists c \in AS(\mathcal{L}) : c \text{ match } t$$
$$\forall A_i^t, A_j^t \in Sem(\mathcal{L}) : A_i^t \lhd A_j^t \implies i > j$$

where $match$ denotes the joinpoint matching relation and $\lhd$ denotes the class inheritance operator. For a language to be well-formed, each of its aspects must have a matching meta-class in its abstract syntax; this is what the first property ensures. Ordering the aspects that compose a semantics in a sequence lets the choice of linearization and/or disambiguation opened to the implementer when several aspects are in conflicts (*e.g.* insert the same method on the same target t). The merging of two semantics, denoted $Sem \bullet Sem'$, consists in producing a new semantics structure. As the definition shows, merging two semantics is equivalent to concatenating their sequences of aspects. As a result, this operator is not commutative and any redefinition of an aspect or method in Sem' overrides the previous definition in Sem:

$$Sem \bullet Sem' \equiv Sem \frown Sem'$$

where $\frown$ denotes the sequence concatenation operator. We also denote sig the *signature* of an aspect A. The signature of an aspect is a metamodel that exposes all the features (*i.e.* properties and methods) defined in an aspect and its dependencies, omitting the concrete method bodies. The signature of a semantic specification Sem is thus defined as the structural merge (through $\circ$) of the signature of the aspects that compose it:

$$sig(Sem) \triangleq \bigcup_{A_i^t \in Sem}^{\circ} sig(A_i^t)$$

3.1.3 Model Typing

Each language $\mathcal{L}$ has one exact model type MT. Like abstract syntaxes, model types are described with a metamodel. The exact model type of a language is its most precise structural interface, *i.e.* the model type that exposes all its features. Thus, the exact model type of a language exposes both its concepts and their relations (*i.e.* its metamodel) and the signature of its semantics (newly inserted features and methods). Hence, the exact type MT of a language $\mathcal{L}$ is defined as the structural merge of its abstract syntax and the signature of its semantics:

$$MT(\mathcal{L}) \triangleq AS(\mathcal{L}) \circ sig(Sem(\mathcal{L}))$$

Any change in either the abstract syntax or the signature of the semantics of a language will result in a different type. The issue of tooling is indirectly addressed through the reasoning capabilities provided by the model typing layer: if the result of the application of operators leads to a language $\mathcal{L}$ whose model type MT is a subtype of the model type MT' of another language $\mathcal{L}'$, then tools defined for $\mathcal{L}'$ can be reused as is for $\mathcal{L}$. In the following, we denote $<:$ the subtyping relation between model types.

3.2 Operators for Language Assembly

3.2.1 Syntax Merging

When building new languages, it is likely that previously defined language abstract syntax fragments may be reused as is. For instance, the syntactic constructs of a simple action language (*e.g.* with expressions, object manipulation, basic I/O) may be shared by any language encompassing the expression of queries or actions. This first scenario of language assembly thus consists in importing a fragment of abstract syntax from another language to reuse its definition. In such a case, the language resulting from the merge of the receiving language and the merged abstract syntax must incorporate all the concepts of both, while preserving the semantics of the receiving language. Also, its model type must be updated accordingly to incorporate the new syntactic constructs. Hence, we specify the merging of an abstract syntax into a language, denoted $\xleftarrow{m}$, as follows:

$$\mathcal{L} \xleftarrow{m} AS' = \langle AS \circ AS', Sem, MT \circ AS' \rangle$$

In most cases, the resulting model type $MT' = MT \circ AS'$ is a subtype of both AS' and MT since it incorporates the features of both. It is however worth noting that new elements introduced in a model type with the $\circ$ operator may break the compatibility with the super model type in some cases (*e.g.* the introduction of a new mandatory feature [18]). In the former case, when the compatibility can be ensured through subtyping, tooling defined over AS' and/or $\mathcal{L}$ (*e.g.* transformations, checkers) can be reused as is on the resulting language.

3.2.2 Semantics Weaving

Another scenario of language assembly consists in importing predefined semantics elements in a language. When different languages share some close abstract syntax, such as different flavors of an action language, their semantics are likely to be similar, at least for the common subparts (*e.g.* the semantics of integer addition is likely to remain unchanged). When the case arises, one would like to import the semantics definition of addition from one action language to another. We denote $\xleftarrow{w}$ the semantics weaving operator, which consists in weaving a semantics Sem' on a language $\mathcal{L}$. In such a case, the two semantics are merged and the exact type of $\mathcal{L}$ is updated to incorporate the syntactic signature of the new semantics:

$$\mathcal{L} \xleftarrow{w} Sem' = \langle AS, Sem \bullet Sem', MT \circ sig(Sem') \rangle$$

Following the previous definitions, this operator can be successfully applied only if there is a matching meta-class in AS for each aspect in Sem'. Since the two semantics are concatenated, Sem' may override any previous definition of Sem, meaning that the semantics merge operator may be employed either to augment or to override part of the semantics of the receiving language $\mathcal{L}$. The semantics weaving operator is thus particularly relevant for incrementally implementing semantic variation points [4].

3.3 Operators for Language Customization

In the previous subsection, we specified how the syntax merging and semantics weaving operators help to build new languages by assembling predefined fragments of abstract syntax and semantics. However, although the reuse of language artifacts significantly decreases the development costs, the resulting languages may not fit exactly the language designer's expectations. Thus, we introduce in this section an algebra for language customization. Customization may include specialization of the abstract syntax or semantics of a language for a given context, restriction to a subset of its scope or composition with (possibly part of) other language definitions. In a recent paper, Erdweg *et al.* propose a taxonomy of different composition operators between languages, including language extension, restriction, and unification [11]. The operators of our algebra closely match their taxonomy: the *inheritance* operator is similar to language extension, the *slicing* operator is similar to language restriction, and the *merging* operator is similar to language unification.

3.3.1 Language Merging

Situations arise where two independent languages must be composed to form a more powerful one. For instance, a finite-state machine language may be defined as a basic language of states and labeled transitions combined to an action language for expressing complex guards and actions. The resulting language may in turn be merged with a language for expressing classifiers where the state machines would describe their behavior. To support this kind of scenario, we introduce the language merging operator, denoted $\uplus$. The output of this operator is a new language that incorporates both the syntactic and semantic definitions of its two operands. In this case, the receiving language is augmented with the merged language to produce the resulting language. Since the merged language can override part of the semantics of the receiving language, order matters and commutativity can not be ensured.

$$\mathcal{L} \uplus \mathcal{L}' = \langle AS \circ AS', Sem \bullet Sem', MT \circ MT' \rangle$$

3.3.2 Language Inheritance

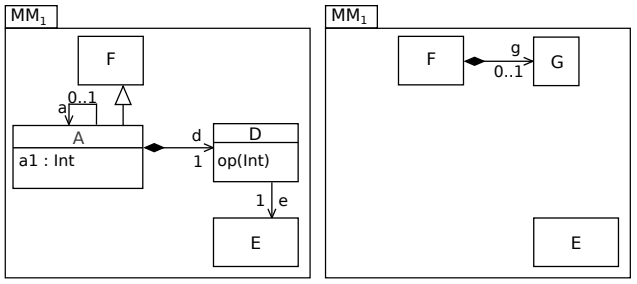
In essence, the language inheritance operator is similar to the language merging operator, as both aims to combine the definitions of two languages. The language inheritance operator, denoted $\oplus$, differs from the language merging operator in that it does not consider the two languages on equal terms: a *sub-language* inherits from a *super-language*. Moreover, the language inheritance operator ensures that the sub-language remains compatible with its super-language. Whatever the subsequent operators applied to the sub-language, it must remain compatible with the super-language, otherwise an error is reported. Concretely, it means that the exact model type of the sub-language must remain a subtype of the exact model type of the super-language: the $MT{<:}MT'$ property is conservative, meaning that any operators apply on $\mathcal{L}$ must not violate it. In a sense, the language inheritance operator supports a form of language design-by-contract, as the language designer is assured that tools defined over $\mathcal{L}'$ will be reused untouched on $\mathcal{L}$.

$$\begin{aligned}\mathcal{L} \oplus \mathcal{L}' = \langle & AS \circ AS', Sem' \bullet Sem, MT'' \rangle \text{ where} \\ & MT'' = MT \circ MT' \text{ and} \\ & MT''{<:}MT'\end{aligned}$$

Note that in this case, the semantics Sem and Sem' are concatenated in reverse order $Sem' \bullet Sem$. The sub-language first inherits the abstract syntax and semantics of its super-language, and may then override part of the inherited artifacts to refine its definition further.

3.3.3 Language Slicing

Model slicing [3, 41] is a model comprehension technique inspired by program slicing [48]. The process of model slicing involves *extracting* from an input model a subset of model elements that represent a *model slice*. Slicing criteria are model elements from the input model that provide entry points for producing a model slice. The slicing process starts by slicing the input model from model elements given as input (the slicing criteria). Then, each model element linked (*e.g.* by inheritance or reference) to a slicing criterion is sliced, and so on until no more model elements can be sliced. For instance, model slicing can be used to extract the static metamodel footprint MM' of a model operation defined over a metamodel MM, *i.e.* extracting the elements of MM used by the operation [22]. Model slicing can be positive or negative. Positive model slicing consists of slicing models according to structural criteria. These criteria are the required model elements from which the slice is built. For instance, based on the simple metamodels of Figure 3, one may want to slice the MM_1 metamodel using as slicing criterion the reference a of the class A. This slicing consists of statically extracting all the elements of MM_1 in relation with a (a included). The result of this slicing is depicted by Figure 4a: the class A that contains a is sliced; the super class of A (F) is sliced; the A's references with a lower cardinality greater than 0 are sliced (only the mandatory references and attributes are sliced); the target classes of these references (*e.g.* D) are sliced. This slicing process continues recursively until no more elements can be sliced. We extended the model slicing principles proposed by Blouin *et al.* [3] to support



(a) Positive slicing on MM_1 with $A.a$ as input (b) Negative slicing on MM_1 with D as input

Figure 4: Slice Examples

negative slicing. Negative slicing consists of considering the slicing criteria as model elements not to have in the slice. For instance, a negative slicing of MM_1 with the class D as slicing criterion produces the slice depicted by Figure 4b: a clone, that will be the output slice, of MM_1 is created; The class D is removed from this clone; all the classes that have a mandatory reference to D are removed (class A); subclasses of the removed classes are also removed (classes B and C). This slicing process continues recursively until no more elements can be removed.

Model slicing can be used to perform language restriction. For instance, a language designer may want to shrink a legacy metamodel to its sub-set used by a set of model operations of interest [22]. This consists of a positive slicing from a set of operations. A language designer may also want to restrict the features of a language (*e.g.* removing specific features of a programming language) for education purposes or to reduce its expressiveness [11]. This consists of a negative slicing from unwanted elements.

In the context of language engineering, we leverage the slicing operation to permit language designer to slice a language according to some slicing criteria, as formalized as follows. Given a language $\mathcal{L}_1 \triangleq (AS_1, Sem_1, MT_1)$. Slicing $\mathcal{L}_1$ using slicing criteria c consists of slicing positively or negatively (resp. denoted Λ^+ and Λ^-, or Λ^+_- when considering both operators) its abstract syntax AS_1 using c to produce a new abstract syntax AS_2, such that $AS_2 \subseteq AS_1$. Then, the aspects A^t_i, that compose Sem_1, that only refer to elements defined in AS_2 are extracted to form Sem_2, as formalized as follows:

$$\begin{aligned}\Lambda^+_-(\mathcal{L}_1, c) = \langle AS_2, Sem_2, MT_2\rangle, \text{ where:}\\ AS_2 \triangleq \lambda^+_-(AS_1, c),\ AS_2 \subseteq AS_1,\\ Sem_2 \triangleq \left\{A^t_i \in Sem_1, fp(A^t_i, AS_1) \subseteq AS_2\right\},\\ MT_1 <: MT_2,\end{aligned}$$

The footprint operation (denoted fp) extracts the metamodel elements of AS_1 used in the aspects a. The choice of applying a positive ($\Lambda+$) or negative ($\Lambda-$) slicing is made by the language designer during the language design according to her requirements. The abstract syntax slicing operation [3] (denoted λ^+, λ^-, or λ^+_-) slices a given abstract syntax AS_1 according to slicing criteria c to produce an output abstract syntax AS_2. Because of the strict slicing that extracts metamodel elements by assuring the conformance, the model type MT_1 is a sub-type of the output MT_2.

4. Melange: A Meta-Language for DSL Assembly and Customization

Melange[1] is an open-source meta-language and framework for DSL engineering. Instead of providing its own dedicated meta-languages for the specification of each part of a DSL (abstract syntax, type system, semantics, *etc.*), Melange relies on other independently-developed components to provide such features. The abstract syntax of DSLs is specified using the Ecore implementation of the EMOF standard provided by the Eclipse Modeling Framework (EMF)[2]. The choice of Ecore is motivated by the success of EMF both in the industry and academic areas. This allows Melange to possibly integrate a wide range of existing DSLs: over 300 Ecore metamodels exists in the "metamodel zoo" [35], over 9000 on Github. For semantics specification, Melange relies on the Xtend programming language[3] to express operational semantics with the definition of aspects. The algebra introduced in Section 3 has been implemented within Melange, providing features for assembly and customization of legacy DSLs artifacts. Overall, Melange is tightly integrated with the EMF ecosystem. Newly built DSLs can thus benefit from other EMF-based components such as Xtext [14] for defining their textual syntax or Sirius[4] for their graphical representation. Melange is bundled as a set of Eclipse plug-ins.

In this section, we present the Melange meta-language through its abstract syntax (Section 4.1), concrete syntax (Section 4.2), implementation choices (Section 4.3) and integration with the EMF ecosystem (Section 4.4).

4.1 Abstract Syntax

The abstract syntax of Melange (the metamodel depicted in Figure 5) includes the concepts and relations discussed in Section 2. This abstract syntax is supplemented with static semantics rules expressed as OCL constraints not presented here for the sake of conciseness. *LanguagesSpec*, the root of Melange's abstract syntax, defines a meta-program that (i) specifies an assembly of DSLs (ii) delimits the scope for the inference and checking of model typing relations.

A *Language* is defined by its *Metamodel* and its associated *Semantics*. A *Metamodel* is composed of a set of *Class*es. A *Semantics* consists of a set of *Aspect*s used to weave behavior into its meta-classes. This mechanism relies on

[1] `http://melange-lang.org`

[2] `https://www.eclipse.org/modeling/emf/`

[3] `http://www.eclipse.org/xtend/`

[4] `https://eclipse.org/sirius/`

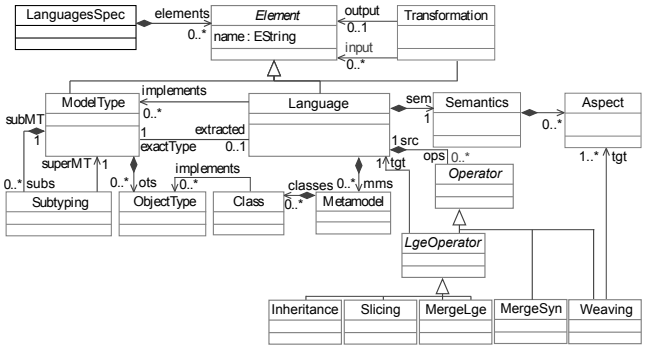


Figure 5: Excerpt of the Abstract Syntax of Melange

static introduction and is inspired by the concept of open classes [24]. As specified in Section 3, both assembly and customization operators can be applied on languages. The *MergeSyn* operator is used to import and merge a metamodel into a given language, whereas the *Weaving* operator is used to weave a given aspect on the abstract syntax of a language. For the customization part, a language can inherit (*Inheritance* operator) from a "super" language. The *MergeLge* operator allows language designers to merge one language into another one. Finally, the *Slicing* operator permits to slice a language given a specific slicing criterion.

A *ModelType* defines an interface to manipulate models. It consists of a set of *ObjectType*s, thereby defining a group of interrelated types. Model types can be created from scratch, or automatically inferred from a concrete language. In the latter case, the language explicitly references this new model type as its $exactType$. Model types are linked one another by subtyping relations: if MT' is a subtype of MT, then there is one and only one *Subtyping* instance that references MT' as its *subType* and MT as its *superType*.

A *Transformation* defines a model transformation that takes *Element*s as input and may produce an *Element* as output. This means that transformations can operate on *Metamodel*s, *ModelType*s, or *Transformation*s themselves.

4.2 Concrete Syntax

Melange provides a textual editor that allows designers to import, manipulate, and reason about DSLs. For the sake of conciseness, we do not detail the whole grammar of the concrete syntax of Melange. Instead, we illustrate its typical use through the mere examples introduced in Section 3 to illustrate the proposed language operators. In a single Melange file, multiple languages can be designed. For instance, the *L1* and *L2* languages are designed by importing respectively the *MM1.ecore* and *MM2.ecore* metamodels (Lines 1 to 6). The *L3* language (Lines 7 to 9) is equivalent to the merging result depicted by Figure 3c. *L3* inherits from *L1*, *i.e. L3* reuses the abstract syntax *MM1.ecore*. This abstract syntax is then merged with the *L2* language to form *L3* (Line 8). The *L4* language (Lines 10 to 12) is designed by performing a positive slicing on the *L1* using as input the reference *a* of the class *A*. Similarly, *L5* is designed (Lines 13 to 15) by performing a negative slicing on *L1* using as input the class *D*. Finally, *L6* (Lines 16 to 18) inherits from *L5*. The aspect *ExecuteAspect* is then woven into *L6*.

```
1  language L1 {
2    syntax "MM1.ecore"
3  }
4  language L2 {
5    syntax "MM2.ecore"
6  }
7  language L3 inherits L1 {
8    merge L2
9  }
10 language L4 {
11   slice+ L1 using [MM1.A.a]
12 }
13 language L5 {
14   slice- L1 using [MM1.D]
15 }
16 language L6 inherits L5 {
17   with ExecutableAspect
18 }
```

Listing 1: Concrete Syntax of Melange

4.3 Implementation Choices

The algebra introduced in Section 3 can be implemented in various ways. We report here on additional choices we made in its concrete implementation within Melange. The algebra does not require a particular formalism for expressing metamodels. In our implementation, we rely on the Ecore implementation of the EMOF standard provided by EMF to specify the abstract syntax of DSLs. Different operators for metamodel merging have been proposed in the literature (*e.g.* [10, 28]). Interestingly, the UML2.0 specification introduces the notion of *PackageMerge* that specifies "*how the contents of one package are extended by the contents of another package*" [37]. Informally, the UML specification states that "*a resulting element will not be any less capable than it was prior to the merge*". Matching of elements of both sides mostly occurs based on name equality. When a match is found between two elements, the resulting package incorporates both sides of its definition. We choose to use a slightly improved version of the *PackageMerge* operator as defined in the UML specification and refined by Dingel *et al.* [9]. To meet our requirements, we adapt the *PackageMerge* operator by trading its UML specificities with EMOF specificities, while preserving its general spirit. For example, we do not consider the concept of *Profile* and adapt the concept of UML *Association* to the concept of EMOF *Reference*. The *PackageMerge* operator specifies a set of constraints that must be ensured for the merge to succeed. If one of the constraints is violated, the merge is ill-formed and an error is reported. It follows that operators of the algebra that relies on the abstract syntax merging operator share the same property: if the constraints are violated the operation is invalid and an error is reported to the user, otherwise the result is guaranteed to be well-formed. On the semantics part, we choose to use

the Xtend programming language supplemented with annotations we developed to specify the operational semantics of DSLs through the definition of aspects. Xtend compiles directly to Java code, providing a seamless integration with other artifacts generated using the EMF framework. A simple example of aspect used to weave executability in the *State* meta-class of a FSM language is given in Listing 2. The *_self* variable refers to the element on which the aspect is ultimately woven (a *State* object in this case) and allows the aspect to access all its features (*outgoingTransitions* in this case). Here, the *ExecutableState* aspect inserts a *step* method in the *State* meta-class to fire the appropriate transition given an input character *c*. Note that aspects may also declare new attributes that are introduced in the target meta-classes.

```
1  @Aspect(className = fsm.State)
2  class ExecutableState {
3    def void step(char c) {
4      val t = _self.outgoingTransitions
5              .findFirst[input == c]
6      if (t == null) throw new Exception
7      else t.fire
8    }
9  }
```

Listing 2: Weaving Executability with Aspects

The *@Aspect* annotation specifies the pointcut of the aspect, while the rest of the class definition defines its advice (new methods and attributes to be inserted). Since pointcuts and advices are not clearly separated, the process of rebinding a set of aspects to a new abstract syntax consists in copying the aspects while updating their pointcuts to target the appropriate concepts of the new abstract syntax.

We also made the following choices in the priorities given to each operator. The inheritance operator has the highest priority, followed by the merge and slice operator (in order of appearance), ending with the aspect weaving operator. First, languages may inherit part of their definition from a super-language. As a consequence, the type system ensures that the sub-typing relation between the two languages is kept, otherwise an error is reported. Then, other artifacts may be assembled, merged or sliced on top of the inherited definition. Finally, aspect weaving comes last to support both the redefinition of imported parts and the addition of "glue code" to make the different parts fit together. As an example, when two merged languages exhibit no common subparts, a new aspect can be woven to connect them in a meaningful way by adding structural references between their abstract syntax, or by inserting some additional code to make their respective interpreters cooperate, *e.g.* through context translation. Finally, for each language declaration, Melange infers its corresponding exact model type. The embedded model-oriented type system automatically infers the subtyping hierarchy through structural typing. This hierarchy is used to ensure the subtyping relation when inheritance is involved and is displayed to the user in a dedicated Eclipse view.

4.4 Compilation Scheme and Integration with EMF

From a Melange program, such as the one depicted in Listing 1, the Melange compiler first reads and imports the external definitions and assembles them according to the rules of the algebra. Once the new DSLs are assembled, customization operators are applied. Then, the compiler completes the resulting model by inferring the subtyping hierarchy among the model types inferred for each language. The implementation relations between metamodels and model types are also inferred in this phase, leading to a complete Melange model conforming to the metamodel of Figure 5. Then, it generates a set of artifacts for each declared language: (i) an Ecore file describing its abstract syntax (ii) a set of aspects describing its semantics attached to the concepts of its abstract syntax (iii) an Ecore file describing its exact model type and (iv) an Eclipse plug-in that can be deployed as is in a new Eclipse instance to support the creation and manipulation of models conforming to it. To generate the runtime code for the new artifacts, Melange relies on the EMF compiler (a *genmodel* generating Java code from an Ecore file), and the Xtend compiler (generating Java code from the aspects file). For each language definition, the Java code generated by both compilers is associated to a plug-in. Since Melange reuse the formalism for language definition of EMF, along with its compilation chain, it is fully interoperable with the EMF ecosystem. Newly created DSLs may thus benefit from other tools of the EMF ecosystem such as Xtext for the definition of a textual editor or Sirius for a graphical representation.

5. Case Study

In this section, we illustrate how the proposed algebra implemented within Melange can be used by language designers to assemble legacy DSLs. We then discuss the results, the integration of the proposed operators in an existing language workbench, and the development overhead. All the materials of the case study are available on the companion webpage[5].

5.1 Language Requirements

To illustrate Melange, we design an executable modeling language for the Internet of Things (IoT) domain, *i.e.* for embedded and distributed systems. This language is inspired by general-purpose executable modeling languages (*e.g.* Executable UML [32] or fUML [40]) and IoT modeling languages (*e.g.* ThingML [15]). This language enables the modeling of the behavior of communicating sensors built on top of resource-constrained embedded systems, such as low-power sensor and micro-controller devices (Arduino[6], Raspberry Pi[7], *etc.*). Such a language aims at providing appropriate abstractions and dedicated simulators, interpreters, or compilers depending on the targeted platforms. To illustrate the ben-

[5] http://melange-lang.org/sle15

[6] http://www.arduino.cc/

[7] https://www.raspberrypi.org/

efits of Melange, the resulting language will be built as an assembly of other popular languages. We consider the three following requirements while designing this language:

i) *The language has to provide an IDL (Interface Definition Language) to model the sensor interfaces in terms of provided services.* Examples of popular languages that provide the appropriate abstractions include the class diagram of (f)UML, the SysML block definition diagram [16], or MOF, as they all provide an OO interface definition language.

ii) *The language must support the modeling of concurrent sensor activities.* Various languages may be employed to model this concern. For instance, process modeling languages such as the (f)UML/SysML activity diagram or BPEL/BPMN are good candidates.

iii) *The primitive actions that can be invoked within the activities must be expressed with a popular language IoT developers are familiar with.* Such a language can be shared by the community and embedded on a set of devices used in the IoT domain. Even though the C language is the common base language of most embedded platforms, its lack of abstraction hinders its exploitation in a modeling environment. Instead, we choose the Lua language[8]. Lua is a dynamically-typed language commonly used as an extension or scripting language. Lua is notably popular in the IoT domain since it is compact enough to fit on a variety of host platforms.

5.2 Language Design using Melange

With the aim of validating Melange, the experimental protocol consists in selecting three publicly-available implementations of existing EMF-based languages to support these three requirements. For the structural part, we use the Ecore language itself as an implementation of EMOF. EMOF provides structural modeling capabilities similar to the UML class diagram. For the activity modeling part, we reuse materials from the *Model Execution Case* of the TTC'15 tool contest[9]. The case foresees the specification of the operational semantics of a subset of the UML activity diagram language with transformation languages. For the action language part, we reuse an existing implementation of the Lua language developed using Xtext. We provide an operational semantics of the Lua language using Xtend and a set of active annotations.

The new language has to provide three perspectives: i) Capturing the services offered by IoT devices, ii) Defining the behavior of these services through a model of an internal process describing the workflow of activities, and iii) Modeling activity implementations. Each activity can execute an action defined using the Lua language. This action language is extended to integrate a new primitive to send messages containing data. These messages are used to invoke services on other devices. The resulting language is built using the Melange assembly definition depicted in Listing 3. The definition is decomposed in multiple languages to ease the description of the process. In a real situation, this definition can be shortened.

```
1  language ActivityLang {
2   syntax "platform:/resource/Activity.ecore"
3   with OperationalSemanticsActivityAspect
4  }
5
6  language LuaLang {
7   syntax "platform:/resource/xtext/Lua.ecore"
8   with org.k3.lua.OperationalSemanticsAspect
9  }
10
11 language EcoreLang {
12  syntax "platform:/resource/Ecore.ecore"
13 }
14
15 language ActivitySlice{
16  slice+ ActivityLang using ['OpaqueAction',
17   'MergeNode','DecisionNode', 'InitialNode',
18   'JoinNode','ForkNode','ActivityFinalNode']
19 }
20
21 language ActivityEcoreLang{
22  merge ActivitySlice
23  merge EcoreLang
24  with fr.inria.diverse.glue.EOperationAspect
25 }
26
27 language ActivityELuaLang{
28  merge ActivityEcoreLang
29  merge LuaLang
30  with fr.inria.diverse.glue.ExpressionAspect
31 }
32
33 language  LuaExtensionLang{
34  syntax "platform:/resource/LuaExt.ecore"
35  with org.luaext.OperationalSemanticsAspect
36 }
37
38 language FinalLang inherits ActivityELuaLang{
39   merge LuaExtensionLang
40   with fr.inria.diverse.lua.ExpressionAspect
41 }
```

Listing 3: Assembling the IoT Language with Melange

1. The abstract syntax of the three languages (the activity diagram from TTC15, Lua, and Ecore) are imported into Melange to form languages respectively called *ActivityLang*, *LuaLang*, and *EcoreLang* (Lines 1 to 13).
2. To design the *ActivitySlice* language, *ActivityLang* is sliced to preserve only the activity diagram part without the action language concepts (Lines 15 to 19). To do so, we manually identified the classes of interest (Lines 16 to 18).
3. *ActivitySlice* and *EcoreLang* are merged (Lines 21 to 25). The *EOperationAspect* then binds *EOperation* and *Activity* (Line 24). This step creates a language *ActivityEcoreLang* that enables the modeling of objects. Such objects can be for example a temperature sensor in a specific room

[8] http://www.lua.org/

[9] http://www.transformation-tool-contest.eu/

instance with an operation *getTemperature*. The implementations of the operations are defined through activity diagram definitions.

4. *ActivityEcoreLang* and *LuaLang* are then merged to form a new language *ActivityELuaLang* (Lines 27 to 31). The *Expression* classes from both languages are linked one another by a new aspect (Line 30).
5. The new language $LuaExtensionLang$ is designed to supplement Lua with message sending capabilities to support synchronization between several complex objects (Lines 33 to 36).
6. A new language *FinalLang*, which inherits from *ActivityELuaLang*, is then created (Lines 38 to 41). *ActivityELuaLang* is merged with the *LuaExtensionLang* and provides a specific glue linking the *ActivityELuaLang* semantics with the *LuaExtension* semantics.

Besides, each language is implicitly associated with its automatically-inferred exact model type. The type checking algorithm of Melange can thus infer the subtyping hierarchy among the different languages. For instance, in this case, the resulting *FinalLang* language subtypes all the other languages because it incorporates all their features [18]. Consequently, tools and transformation defined on *e.g.* the Ecore language can be reused to manipulate models created with the *FinalLang* language. In the end, we obtain a new executable modeling language for IoT resulting from the composition of three legacy languages for which reuse was unforeseen. Additionally, most of the previously defined tools can be reused as is.

5.2.1 Discussion

A critical point concerns the ability of Melange to be integrated into an existing ecosystem. The integration using Melange of three existing EMF languages allows a language designer to obtain a new EMF language. If we do not consider the imposed methodology for defining the language semantics (the use of the interpreter pattern [17]), no modification of these languages was required to support that composition. This illustrates how Melange can be integrated into an existing language workbench without any change in the legacy abstract syntaxes. All the Melange operators are used for this case study. Although this does not guarantee that these operators are sufficient, it highlights that all of them are required when a language designer needs to compose existing languages.

Another major point is the possible overhead in term of performance and lines of code that stem from the use of Melange. Compared to a top-down approach where the IoT language is built from scratch by an expert in language design, we observe no additional concepts integrated into the abstract syntax definition. At the semantics level, glue code is injected for the implicit conversion of the interpreter pattern context resulting from the composition of the various contexts stemming from various operational semantics. At runtime, no additional cost in terms of performance were observed to the use of the language resulting from the composition. Table 1 sums up the results.

Table 1: Comparison of Melange and a Top-down Approach for the IoT Language

	Melange	Top-down
Metaclasses (#)	104	104
LoC for the glue (#)	27	0
Efficiency (sec)	30,0	25,9

Performance comparison is obtained by loading and executing a model with 10 objects that contains one operation with a workflow with 1000 basic actions that do mainly 10 numeric operations. The comparison was done on the same laptop designed with an Intel i7 with 16Gb of memory, a Linux 64bit operating system and an Oracle Java 8 virtual machine.

Nevertheless, these results may be moderated by the following threats to validity. First, all the languages must be designed in the same technical ecosystem. Melange does not provide any support for integrating heterogeneous languages in terms of technical ecosystem. Second, Melange can not compose any language semantics. The composition can be done if and only if the semantics is operational and defined following the interpreter pattern (*e.g.* through static introduction or a visitor). Third, concepts with different names in different languages may represent the same concept. In such a case, adaptation mechanisms are required to align them before composition. Melange provides a simple renaming mechanism that allows to rename concepts, but lacks a powerful mechanism for realizing complex adaptations. Finally, the same person implemented the language using Melange and using a traditional top-down approach. This person is an expert in language design and modeling technologies. Besides, the top-down language design has been reviewed by three experts in language design and is publicly available on the companion webpage.

6. Related Work

A DSL allows developing software for a particular application domain quickly and effectively, yielding programs that are easy to understand, reason about, and maintain [21]. There may be, however, a significant overhead in creating the infrastructure needed to support a DSL. Numerous works proposed to create reusable and composable language units to tackle this issue. Methodologies have been proposed for building DSLs embedded within an existing, higher-order, and typed programming language [20]. Techniques have been then designed for building modular interpreters and tools for such embedded DSLs. Different techniques have been studied for addressing the challenge of language extension and composition, such as projectional editing [45]. Spoofax, however,

relies on meta-languages for defining syntaxes and semantics, which are inherently modular and composable [47]. Although basic import mechanisms are supported, they usually lack a powerful support for customization. More recently, an overview of the support provided by language workbenches has been provided [12]. In the grammar world, several techniques demonstrated the possibility to create language units using attribute grammars [25, 34, 39]. MontiCore applied modularity concepts for designing new DSLs by extending an existing one, or by composing other DSLs [29]. MontiCore reifies as a first-class object the concept of language inheritance to allow language feature reuse. Other works propose to leverage concepts from the component-based software engineering community to modularly develop DSLs [43, 50].

In the MDE domain, several meta-tooling platforms propose mechanisms for improving language design modularity. Ledeczi *et al.* propose to compose domain-specific design environments using MDE technologies [31]. Melusine [13], Xtext [14], or MPS [2] are frameworks supplemented with IDEs for building textual DSLs. In both the MDE and grammar domains, the increasing trend to create new DSLs, from scratch or by adapting existing ones, causes the emergence of families of DSLs. A family of DSLs is a set of DSLs sharing common aspects but specialized for a particular purpose. The emergence of a family of DSLs raises the need to reuse common tools among a given family [27, 30] and the need to create language composable units. To ease the language unit composition, Steel *et al.* [42] and De Lara *et al.* [8] propose to define a clear contract and a typing system that can be used for composing language units. De Lara *et al.* present the *concept* mechanism, along with *model templates* and *mixin layers* leveraged from generic programming to MDE [7]. *Concepts* are close to model types [42] as they define the requirements a metamodel must fulfill for its models to be processed by a transformation, under the form of a set of classes. Sánchez, Wimmer *et al.* go further than strict structural mapping by renaming, mapping, and filtering metamodel elements [38, 49]. Erdweg *et al.* proposed a taxonomy to ease the positioning of approach related to language composition [11]. According to this classification, our algebra supports the language extension, restriction, and unification operators. Additionally, we do not consider that restriction is only a matter of additional validation rules. Instead, we prune the language from the unwanted parts so that only the necessary concepts are kept.

7. Conclusion and Future Work

While current language workbenches provide import mechanisms, they usually lack an explicit support for customization and safe composition of imported artifacts. This paper proposes an approach for building DSLs by safely assembling and customizing legacy DSLs artifacts. We propose different operators for assembling (merge_S/*weave*), restricting (*slice*), extending (*inherits*), and merging (merge_l) DSLs. The use of typing and subtyping relations that provides a reasoning layer for DSLs manipulation is also promoted. The approach is implemented in Melange, an EMF-based meta-language. We illustrate and discuss this work by designing a new executable modeling language for IoT showing that: all the proposed operators are relevant for designing a new language based on the composition and the specialization of three legacy DSLs; the use of Melange does not introduce specific technical issue compared to a traditional top-down approach.

In our future work, we will investigate to what extent Melange can be used to provide agile modeling for DSL users, *i.e.* the safe reuse of model transformations that can work across several DSLs, and the specification of viewpoints.

Acknowledgments

This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011), the ITEA2 Project MERgE, and the French LEOC Project Clarity.

References

[1] MOF, 2.0 core final adopted specification, 2004.

[2] Language and IDE modularization and composition with MPS. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Proc. of GTTSE'13*, pages 383–430, 2013.

[3] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and generating model slicers. *Software and Systems Modeling (SoSyM)*, pages 1–17, 2012.

[4] F. Chauvel and J.-M. Jézéquel. Code generation from UML models with semantic variation points. In *Model Driven Engineering Languages and Systems*, pages 54–68. 2005.

[5] B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux. Essay on semantics definition in MDE-an instrumented approach for model verification. *J. of Sw*, 4(9):943–958, 2009.

[6] M. L. Crane and J. Dingel. UML vs. classical vs. Rhapsody statecharts: Not all models are created equal. In *Model Driven Engineering Languages and Systems*, pages 97–112. 2005.

[7] J. De Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *Proc. of MODELS'10*, pages 16–30, 2010.

[8] J. de Lara, E. Guerra, and J. Sánchez-Cuadrado. Abstracting modelling languages: A reutilization approach. In *Advanced Information Systems Engineering*, pages 127–143. 2012.

[9] J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *SoSym*, 7(4):443–467, 2008.

[10] M. Emerson and J. Sztipanovits. Techniques for metamodel composition. In *OOPSLA Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[11] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proc. of the Workshop on Language Descriptions, Tools, and Applications*, page 7, 2012.

[12] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *Software language engineering*, pages 197–217. 2013.

[13] J. Estublier, G. Vega, and A. Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *Proc. of MODELS'05*, pages 69–83, 2005.

[14] M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proc. of OOPSLA '10 Companion*, pages 307–309, 2010.

[15] F. Fleurey, B. Morin, A. Solberg, and O. Barais. MDE to manage communications with and between resource-constrained systems. In *Proc. of MODELS'11*, pages 349–363, 2011.

[16] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. 2014.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[18] C. Guy, B. Combemale, S. Derrien, J. R. Steel, and J.-M. Jézéquel. On model subtyping. In *Modelling Foundations and Applications*, pages 400–415. Springer, 2012.

[19] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *ACM Sigplan Notices*, 24(11):43–75, 1989.

[20] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[21] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proc. of ICSE'11*, pages 471–480. ACM, 2011.

[22] C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *Proc. of ICSE'11*, 2011.

[23] J.-M. Jézéquel. Model driven design and aspect weaving. *Software & Systems Modeling*, 7(2):209–218, 2008.

[24] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, pages 1–16, 2013.

[25] U. Kastens and W. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.

[26] L. C. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM sigplan notices*, volume 45, pages 444–463, 2010.

[27] D. Kolovos, L. Rose, and N. Matragkas. A research roadmap towards achieving scalability in model driven engineering. In *BigMDE'13*, 2013.

[28] D. S. Kolovos, R. F. Paige, and F. A. Polack. Merging models with the epsilon merging language (EML). In *Model driven engineering languages and systems*, pages 215–229. 2006.

[29] H. Krahn, B. Rumpe, and S. Völkel. Monticore: a framework for compositional development of domain specific languages. *JSTT*, 12(5):353–372, 2010.

[30] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *SoSyM*, pages 1–36, 2013.

[31] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, Nov 2001.

[32] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. 2002.

[33] M. Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9):2451 – 2464, 2013.

[34] M. Mernik and V. Zumer. Reusability of formal specifications in programming language description. In *8th Annual Workshop on Software Reuse, WISR8*, pages 1–4, 1997.

[35] Metamodel Zoos. `http://www.emn.fr/z-info/atlanmod/index.php/Zoos`, last access: april 2015.

[36] P. D. Mosses. The varieties of programming language semantics and their uses. In *Perspectives of System Informatics*, pages 165–190. Springer, 2001.

[37] *Unified Modeling Language 2.0, Infrastructure*. OMG, 2005.

[38] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Generic model transformations: Write once, reuse everywhere. In *Proc. of ICMT'11*, pages 62–77, 2011.

[39] J. a. Saraiva. Component-based programming for higher-order attribute grammars. In *Proc. of GPCE*, pages 268–282, 2002.

[40] E. Seidewitz. UML with meaning: Executable modeling in foundational UML and the Alf action language. *Ada Lett.*, 34 (3):61–68, Oct. 2014.

[41] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model Pruning. In *Proc. of MODELS'09*, 2009.

[42] J. Steel and J. M. Jézéquel. On model typing. *SoSyM*, 6(4): 401–413, 2007.

[43] E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 2015.

[44] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, and G. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Proc. SPLASH*, pages 95–111, 2014.

[45] M. Voelter. Language and IDE modularization, extension and composition with MPS. *Generative and Transformational Techniques in Software Engineering*, 2011.

[46] M. Voelter, B. Kolb, and J. Warmer. Projecting a modular future. 2014.

[47] M. Völter and E. Visser. Language extension and composition with language workbenches. In *Proc. of the OOPSLA companion*, pages 301–304. ACM, 2010.

[48] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[49] M. Wimmer, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, J. Cuadrado, E. Guerra, and J. de Lara. Reusing model transformations across heterogeneous metamodels. In *Proc. of MPM'11*, 2011.

[50] 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*, pages 413–428. Springer, 2015.