Leveraging Family Polymorphism in MDE

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

Leveraging Family Polymorphism in MDE

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

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

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

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

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

ABSTRACT

While Domain-Specific Modeling Languages (DSMLs) are increasingly used in industry, both their definition and tooling (*e.g.*, checkers, document or code generators, model transformations) still require significant development efforts that must be balanced with their limited number of users (by definition). Unfortunately, the current Model-Driven Engineering (MDE) technologies strongly rely on the conformance relation that bind a model to the unique DSML used to create it. Consequently, while most of the tools would be reusable for a family of close DSMLs, in practice it is not possible. In this paper, we propose to abstract the overly restrictive conformance relation with a typing relation allowing to manipulate a model through different DSMLs and uncover the need for model-oriented type systems. We introduce K3SLE, a new modeling framework built on top of the Eclipse Modeling Framework (EMF) that leverages family polymorphism to support the typing relation. Based on the typing relation, it natively provides model polymorphism, language inheritance and DSML evolution and interoperability. We demonstrate its use on representative use cases.

Categories and Subject Descriptors

D2.13 [**Software Engineering**]: Reusable Software; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*

General Terms

Theory, Languages

Keywords

Model-Driven Engineering, Domain-Specific Modeling Languages, Metamodeling, Model Typing, Conformance

1. INTRODUCTION

Extending the time-honored practice of separation of concerns, Domain-Specific Modeling Languages (DSMLs) are increasingly used to handle different, complex concerns in system, and software development. However, both the definition of a DSML and of its tooling (*e.g.* checkers, document or code generators, model transformations) require significant development efforts for, by definition, a limited audience.

Abstractly speaking, the metamodel of a DSML can be seen as the definition of a group of related types, that is, a set of constraints over admissible graphs of objects (models). With that view, illustrated in Figure 1, a model m is a graph of interconnected objects respecting the type constraints MT defined by its metamodel MM.

From a concrete point of view however, the situation is a little bit different. A metamodel MM typically implements MT with a set of classes in a given programming language, for instance Java in the case of the Eclipse Modeling Framework (EMF). Then a model is concretely made of a set of instances of the classes defined in MM, explicitly referring to the concrete implementation of these classes. This can be seen for example in the fact that the XML serialization provided by EMF explicitly embeds an URI which refers to MM.

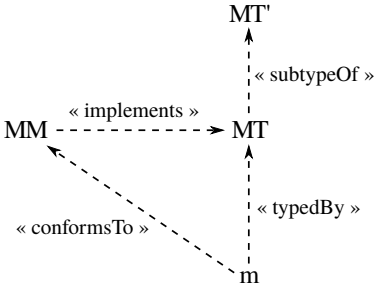

Figure 1: An approach to support typing in MDE

As pointed out in [22], many of the proposed reuse mechanisms in MDE depend on concrete metamodels. This implementation-oriented view prevents model polymorphism and the emergence of multi-viewpoint MDE.

This is not just an academic problem: If a code generator is written for class diagrams in UML2.1, how do we ensure that it still works for models defined with UML2.2? If an analysis tool for a specific safety viewpoint is defined, how can it be reuse for the same analysis on a model that merges a set of viewpoints without managing a cumbersome modeling language that merges all the potential system viewpoints?" How can an Airbus engineer send a state machine diagram edited with an Airbus domain specific tool (called SAM) to a colleague using UML Statecharts, which are conceptually the same but physically different? This has long been identified (*e.g.*, in Estublier et al. [12]) as a core limitation of MDE: no support for abstract architecture evolution and little reuse of components developed elsewhere.

In practice, the problem is twofold:

1. There is no way in existing modeling frameworks (*e.g.* EMF) to say that a model m is typed by more than one type group MT, or *e.g.*, that a given model transformation works for the intersection of MT_1 and MT_2.

2. The conformance between m and MM is fixed once and for all when the model is built. It means that a model "conforms" to a fixed set of concepts identified by a specific URI.

The programming language community overcame the first problem with the concept of *duck typing*[1], that can nicely be combined with static typing and family polymorphism [11] as in *e.g.* Python or Scala. Leveraging this idea, we already proposed the notion of Model Typing [33] to define type groups based subtyping relations among metamodels [17]. That made it possible to define model transformations that would work across different metamodels provided they are subtypes of one another.

In this paper, we propose to abstract the overly restrictive conformance relation with a typing relation allowing to manipulate a model through different DSMLs and uncover the need for model-oriented type systems uncoupled from the host language typing system. To improve the reuse of model-oriented artifacts (models, transformations, ...), we propose to build a type system with family polymorphism in MDE that allows any model m conforming to MM to be seen in a polymorphic way as typed by any super type MT' of MT (see Figure 1). We show that severing the *conformance* relation between m and MM gives the modeler all the needed flexibility to perform reuse and handle evolutions, still benefiting from full static typing to check correctness or even just for making auto-completion easier. We claim that the abstraction of the conformance relation with the typing relation should become the cornerstone of reuse in MDE.

Concretely speaking, we propose a new extension to the Kermeta language [24], called K3SLE, to separate the host language type system and the model-oriented type system in order to ease the manipulation of type groups. K3SLE can be seen as a layer on top of EMF and the Kermeta action language that enables model polymorphism, inheritance among DSMLs, as well as evolution and interoperability of DSMLs. K3SLE makes it possible to:

- define **metamodels** by importing their concrete implementations from Ecore and potentially a set of metamodel extensions expressed using **aspects**
- define **model types** (a.k.a type groups) from scratch or by extracting conceptual typing information from concrete metamodels
- define **inheritance relationships**, allowing reuse of abstract syntax and semantics accross different metamodels
- and finally, define well-typed **transformations** that can be polymorphically called on different metamodels.

We provide a translational semantics for K3SLE by compilation to the Java programming language, keeping full compatibility with legacy code generated by EMF. We validate our approach by demonstrating examples of model polymorphism through the reuse of the very same transformation on two different DSMLs. These DSMLs have a common super type and metamodel inheritance that can be used as a clean way to implement semantic variation points.

[1] "*When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck*" (James Whitcomb Riley)

We then discuss how this facilities can help DSML engineers to solve scenarios they commonly face.

The remainder of this paper is organized as follows. Section 2 presents background material on the conformance relation between models and metamodels and current limits. Section 3 introduces our proposal on superseding model conformance with a typing relation enabling better reuse facilities. Section 4 introduces the syntax, semantics and implementation of K3SLE, our new modeling framework built on top of EMF for manipulating type families. Section 5 illustrates the facilities provided by K3SLE on representative use cases and discusses their generalization. Section 6 discusses related work on reuse in MDE and typing in OOP. Finally, Section 7 concludes and discusses some perspectives of our work.

2. BACKGROUND ON THE CONFORMANCE RELATION

To state whether a model is a valid element of a DSML, MDE relies on the *conformance relation* which stands between a model and a metamodel (*i.e.* the DSML implementation). A model conforming to the metamodel of a DSML is a valid element of this language.

2.1 Conformance Relation in the Literature

Different names have been given to this relation in the literature: "*sem*", "*instantiation*", "*conformance*", or "*compliance*". All these relations are directly based on the abstract syntax of DSMLs. In the following, we use the term *conformance relation* to refer to this relation between models and DSMLs. Table 1 presents several definitions of the conformance relation from the literature over the last ten years.

Favre considers every representation of a language as a metamodel, and thus builds the conformance relation between a model and any of these representations (abstract or concrete syntax, documentation, tool, *etc.*) [13]. Favre's definition is thus less strict than the other ones presented in this section. However, if this definition is sufficient for the study and understanding of MDE, it is not precise enough for DSMLs tooling or automated checking of the validity of a model wrt. a DSML.

Other authors define the conformance relation through the instantiation relation which stands between an object and the class from which it is built: "*every element of an Mm-level model must be an instance-of exactly one element of an Mm+1-level model*" [1]; "*metamodels and models are connected by the instanceOf relation*" [16]; "*every object in the model has a corresponding non-abstract class in the metamodel*" [10].

Bézivin *et al.* do not directly refer to classes, but prefer the terms "*definition*" [2] or "*meta-element*" [3] (*i.e.* element of a metamodel). Such definitions authorize the definition of the abstract syntax of a DSML under a different form than a set of classes.

With the exception of the definition given by Favre, definitions from the literature presented in Table 1 agree on one point: the conformance relation is based on the instantiation relation between objects and classes.

2.2 Conformance relation in de facto standards

The Meta-Object Facility (MOF) from Object Management Group (OMG) and the Eclipse Modeling Framework (EMF) are *de facto* technological standards in the MDE community. Several tools and frameworks are based on these standards, such as MOFLON[2],

[2] `http://www.moflon.org/`

Bézivin [2]	"Let us consider model X containing entities a and b. There exists one (and only one) meta-model Y defining the "semantics" of X. The relationship between a model and its meta-model (or between a meta-model and its meta-meta-model) is called the sem relationship. The significance of the sem relationship is as follows. All entities of model X find their definition in meta-model Y."
Atkinson et Kühne [1]	"In an n-level modeling architecture, M0, M1...Mn-1, every element of an Mm-level model must be an instance-of exactly one element of an Mm+1-level model, for all m < n - 1, and any relationship other than the instance-of relationship between two elements X and Y implies that level(X)=level(Y)."
Favre [13]	Favre does not give a definition for conformance relation, but presents it as a shortcut for the sequence of two other relations: "elementOf" (which stands between a model of a language and this language) and "representationOf" (which stands between a metamodel and modeling language).
Bézivin *et al.* [3]	"A model M conforms to a metamodel MM if and only if each model element has its metaelement defined in MM."
Găsevic *et al.* [16]	"metamodels and models are connected by the instanceOf relation meaning that a metamodel element (*e.g.*, the Class metaclass from the UML metamodel) is instantiated at the model level (*e.g.*, a UML class Collie)."
Rose *et al.* [30]	"A model conforms to a metamodel when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel. [...] For example, a conformance constraint might state that every object in the model has a corresponding non-abstract class in the metamodel."
Egea et Rusu [10]	"Namely, the objects of a "conformant" model are necessarily instances of the classes of the associated metamodel (possibly) related by instances of associations between the metamodel's classes."

Table 1: Definitions of the conformance relation in the litterature

Kermeta [3], ATL [4], Henshin [5], Epsilon[6], Xtext[7], or Xtend[8]. The way these standards define the relation between models and metamodels is thus central in today's tooling support of MDE.

The Meta-Object Facility (MOF) is a metalanguage dedicated to the definition of the abstract syntax of DSMLs under the form of an object-oriented metamodel [27]. However, the MOF specification does not give any indication on the relation which stands between a model and a metamodel. Since the MOF and UML specifications are strongly dependent on each other, we can also look for such a relation in UML. UML is said to be an instance of MOF and "*every model element of UML is an instance of exactly one model element in MOF*" [28]. However, the definition is not sufficient to clearly define a relation between MOF and metamodels, or between metamodels and models.

The Eclipse Modeling Framework (EMF) also does not give such a definition, but relies on two technologies to manipulate models [34]: Java classes for instantiation of elements of models; XML Schema for model serialization. On the Java side, one class is generated for each concept of the abstract syntax and models are sets of object instances of these generated classes. On the XML side, an XML Schema describes the structure of a metamodel for enabling the serialization of models as XML documents. The XML Schema recommendation states that Conformance (*i.e.,*validity) checking can be viewed as a multi-step process [37]: first, the root element of the document instance is checked to have the right contents; then, each sub-element is checked to conform to its description in a schema. Moreover, "*to check an element for conformance, the processor first locates the declaration for the element in a schema*" [37]. Thus, an element of an XML document without a corresponding declaration in a XML Schema does not conform to this schema, neither does the whole document. Metamodels being described through XML Schemas and models through XML documents, a model conforms to a given metamodel if all the elements of the model have a corresponding declaration in the XML Schema of the metamodel.

2.3 Definition of the Conformance Relation

Except for the definition from Favre, all the definitions from the literature cover the same definition: a model conforms to a metamodel if every element of the model is an instance of one the elements of the metamodel. Moreover, an EMF model (either described through Java objects or an XML document) cannot contain elements that are not instances of the elements of the metamodel (either described through Java classes or an XML Schema). Finally, the MOF specification does not give any clear definition of the conformance relation but suggests an instantiation relation between models and metamodels.

From our study of the literature and technological standards, we can define the conformance relation as follows:

DEFINITION 1. **(Conformance relation)** *A model m conforms to a metamodel MM iff,* $\forall o \in m$*, o is instance of C such that* $C \in MM$*.*

2.4 Limits of the Conformance Relation

From the above definition, we can retain three important limitations of the conformance relation:

(1) The conformance relation is a construction-based relation. Since the conformance relation relies heavily on the instantiation relation, the relation between a model and a metamodel is set up at the construction of the model.

(2) The conformance relation is hardcoded. The conformance relation is set up once and for all at the creation of the model. Moreover, the relation is kept throughout the life of a model, even through serialization. Indeed, during the serialization of a model, the URI of its metamodel is one of the meta-information that is serialized with the description of the model elements and their relations.

(3) A model conforms to one and only one metamodel. This point results from the two previous ones. Because the conformance relation is construction-based, hardcoded, and only one metamodel is used to create a model, a model conforms to this metamodel only. Indeed, an object is an instance of (*i.e.,*is built from) only one class, which defines the fields and operations of the object. A given class only belongs to one package, which itself belongs to one metamodel. Thus, one and only one metamodel exists to which a given model conforms, which is the metamodel defining all the classes from which the elements of the model are instances ("*There exists one (and only one) meta-model Y defining the "semantics" of X*" [2]). The conformance relation thus forbids polymorphism at the model level: a model has one and only one "form", the one defined by its unique metamodel.

[3] http://www.kermeta.org
[4] http://www.eclipse.org/atl/
[5] http://www.eclipse.org/henshin/
[6] http://www.eclipse.org/epsilon/
[7] http://www.eclipse.org/Xtext/
[8] http://www.eclipse.org/xtend/

The conformance relation, standing between a model and a metamodel, is used to state whether a model is a valid element of a DSML. The conformance relation is thus central to MDE, it used as a typing relation. The benefits of such a conformance relation are multiple and are used in various application domains such as to: check the validity of a model; open the model in a given editor; authorize or forbid passing a model as parameter of a model transformation; make proposals for code completion or recommenders.

Besides, by preventing a model from being related to several metamodels, and thus to several DSMLs, the conformance relation restrains reuse between DSMLs. For example, a metamodel is used to "type" parameters of a model transformation. A model conforms to one and only one metamodel. So, it is not possible to substitute a model conforming to a metamodel MM' to a model conforming to a metamodel MM expected by a model transformation T. That, even if MM and MM' are conceptually close. T only accepts as parameters models conforming to MM. T is thus not reusable through several DSMLs.

3. TYPING RELATION

To address the previous limitations, we propose to consider abstract views of model implementations (i.e. metamodels) as first class entities. Such an abstract view, aka. language interface, is used to *type* a model with respect to a set of required constraints, while the metamodel is kept for the model construction. We call an interface of a modeling language a *model type*.

In this section, we present an approach to deal with model types as first-class entities. We introduce the different concepts which are necessary to take into account when defining a relation between models and model types. We define such a relation as a typing relation, which lifts the limitations from the conformance relation. We define model types as a set of object types. The model typing relation is defined through the specific *exact* type of a model. The subtyping relation is specified between model types. Finally, the implementation relations is defined between metamodels and model types.

3.1 Approach overview

In order to abstract from the instantiation relation, on which the conformance relation rely, we propose to separate the concepts of interface and of implementation of the models, as it is done for interface and implementation of objects. Types (either object or model types) thus exhibit an interface through which it is possible to manipulate (or check the validity) the entities they type (objects or models). Classes and metamodels provide the implementation of objects and models, and allow to create them. Similarly to an object that has only one class but has a set of types, a model has only one metamodel (the one from which the model has been created), but has a set of model types.

Among the different types of an object or of a model, we consider the *exact* type, which is the most precise type about the object or the model. To build the model typing relation between a model and its set of model types, we rely on these *exact* types of models and model elements, and on model subtyping relations.

Figure 2 presents an overview of the concepts and relations that we reify in a modeling framework supported by a dedicated model-oriented type system.

A model is created from a metamodel and typed by a set of model types. A model *m* consists of objects, instances of the classes of the metamodel of *m*, and typed by the object types of the model types of *m*.

Consequently, a metamodel provides the implementation of all the models which are created from it. This implementation

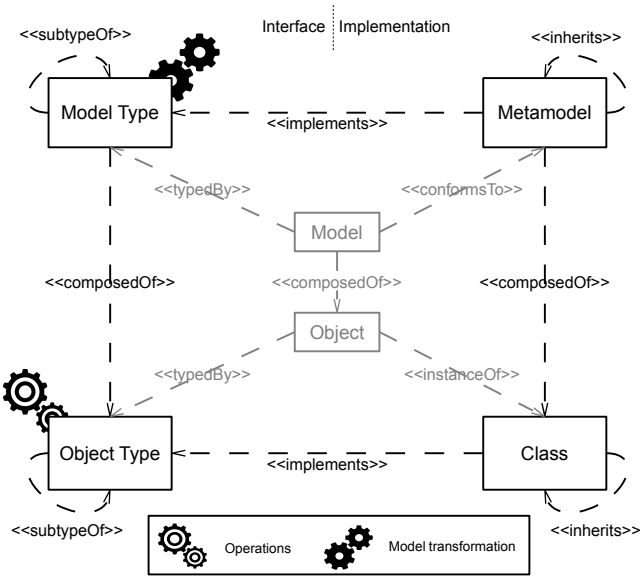

Figure 2: Approach overview to support typing in MDE

consists of the classes (including structural features and operations) of model elements (*i.e.,* objects). An inheritance relation stands between two metamodels: the super-metamodel and the sub-metamodel. Inheritance relation allows to reuse the structure (classes) and the behavior (operations) from one metamodel to another.

A model type exhibits the interface through which it is possible to manipulate the models typed by the model type. A model type consists of object types, themselves exhibiting the interface of model elements. Model transformations are defined with respect to model types, *i.e.,*the interface of models. Model transformations manipulate models typed by given model types, and not models created from given metamodels. A subtyping relation stands between two model types. A subtyping relation states the substitutability of models typed by the subtype to models typed by the supertype.

A metamodel implements one or more model types by providing the implementation corresponding to the interface these model types exhibit.

We precisely define in the next subsection the concepts of model type and subtyping relation that we introduced. Next section presents a new meta-language supporting this concepts.

3.2 Model Interfaces

To support types, *i.e.,*interfaces, for models, which would allow to build a typing relation which is not construction based, we rely on object types, *i.e.,*types of model elements. We define *exact* model types which are the basis on which we compute the model typing relation. *Exact* model types are themselves built on *exact* object types of model elements.

3.2.1 Object Type

An *object type* exhibits the interface of a set of objects through which it is possible to manipulate them. It exhibits the structural features (attributes and references) and behavioral features (operations) that can be accessed on the objects.

DEFINITION 2. **(Object Type)** *An object type T is a set of declarations of structural and behavioral features.*

We define the *exact* type of an object as the most precise type of this object, exhibiting *all* of its structural and behavioral features. It is possible to extract the *exact* type of an object from the class which has been used to create (*i.e.,* instantiate) this element.

DEFINITION 3. **(Exact Object Type)** *The* exact *type of object instances of the class C is the object type $Ex(C)$ such that:*

- *forall operations of C, there is a corresponding operation signature in $Ex(C)$;*
- *forall structural features (attributes and references) of C, there is a corresponding structural feature signature in $Ex(C)$.*

3.2.2 Model Types

A *model type* exhibits the interface through which it is possible to manipulate models typed by this model type. Such an interface consists of object types, which themselves exhibits the interface through which it is possible to manipulate model elements (*i.e.,* objects).

DEFINITION 4. **(Model Type)** *A model type MT is a set of object types.*

As for objects, among all the model types typing a given model, there is one particular model type: the *exact* model type of the model. The *exact* model type of a model is the type corresponding to the metamodel from which the model has been created. That is, it contains all the object types corresponding to the classes of the metamodel, and is thus the most precise type of the model. We can thus extract the *exact* model type of a model directly from its metamodel.

DEFINITION 5. **(Exact Model Type)** *The* exact *model type $Ex(MM)$ of models conforming to the metamodel MM is a model type such that $\forall T \in Ex(MM) \Leftrightarrow C \in MM$ such that $T = Ex(C)$*

3.2.3 Model Typing Relation

Based on the reification of the concept of model type as a first-class entity, we can define the typing relation which stands between a model and its model types. This typing relation is more flexible than the conformance relation, since it allows a model to have several model types, *i.e.,* it allows subtype polymorphism of models.

To state whether a model is typed by a given model type, we use the *exact* model type of the model, *i.e.,*the model type extracted from the metamodel from which the model have been created, and the *total isomorphic* subtyping relation introduced in [17] (denoted <:) checking the substitutability between two model types.

By definition of the subtyping relation, a model m typed by a model type MT is also typed by all the supertypes of MT. The *exact* model type MT_m of a model m being the most precise type of m, there exists no model type which is both a subtype of MT_m and a type of m. MT_m is thus the subtype of *all* the types of m. We can therefore define the set of types of a model as the set of supertypes of its *exact* model type.

DEFINITION 6. **(Model Typing Relation)** *The model typing relation (:) is a binary relation from the set of all models $\mathcal{M}$ to the set of all model types $\mathcal{MT}$, such that $(m, MT) \in$: (also denoted $m : MT$) iff MT_m<:MT, where $MT \in \mathcal{MT}$, $m \in \mathcal{M}$ and MT_m is the* exact *model type of m.*

3.2.4 Implementation Relation

An implementation relation stands between metamodels and model types. This relation specifies that a given metamodel *MM* provides the implementation of the features declared in a given model type MT. It means that any model created from *MM* can be manipulated through MT. Formally, a metamodel *MM* implements a model type MT if the exact type of *MM* is a subtype of MT, which leads to the following definition of model type implementation:

DEFINITION 7. **(Implementation Relation)** *The implementation relation ($<\bullet$) is a binary relation between a metamodel and a model type, such that $(MM, MT) \in <\bullet$ (also denoted $MM <\bullet MT$) iff $Ex(MM)$<:MT*

The previous definitions are independent of different choices that can be made in the implementation of a particular model type checker [17]. In particular the subtyping and implementation relations can be declared or inferred (*e.g.*, through *structural typing*), and the typing relation can be checked statically or dynamically.

4. K3SLE: A MODELING FRAMEWORK FOR MODEL TYPING

In this section, we present K3SLE, a modeling framework that supports the concepts defined in the previous section: model types, metamodels, as well as the typing and subtyping relations. The framework includes a dedicated meta-language, which is a new extension to the Kermeta language. It leverages family polymorphism in order to enable model polymorphism, transformation reuse and evolution and interoperability of DSMLs, together with several other facilities such as language inheritance or transformation footprinting. K3SLE comes with a model-oriented type system that statically and structurally types the different constructs of its meta-language independently of the host typing system. The textual editor support of K3SLE benefits from static typing by providing early detection of errors and facilities such as autocompletion. The choice of structural typing is motivated by the open-world nature inherent to MDE: since DSMLs are created independently and continuously evolving, all possible relationships between all possible DSMLs cannot be initially fixed. Finally, K3SLE comes with its own compiler that generates Java code, thus enabling full interoperability with legacy EMF code.

In the following subsections we present the meta-language provided by K3SLE (abstract syntax, concrete syntax and semantics) as well as its support on top of EMF to offer the expected facilities for manipulating of models.

4.1 Abstract Syntax

The abstract syntax of K3SLE supports the necessary concepts for the definition of model typing relations. As a consequence, the different relations depicted in Figure 2 are materialized in its metamodel (Figure 3). This metamodel is complemented by context conditions expressed as OCL constraints that are not presented in this paper for the sake of conciseness. *ModelTypingSpace* is the root of the metamodel and thus contains the definition of *Metamodel*s, *ModelType*s and (model) *Transformation*s. The model type checking process operates within a given scope defined by a model typing space.

A *Metamodel* defines the implementation of a DSML and is composed of a set of *Class*es. It can be supplemented with behavioral *Aspect*s woven on the elements it contains [19]. Aspect models allow to insert new classes, methods, properties, or constraints

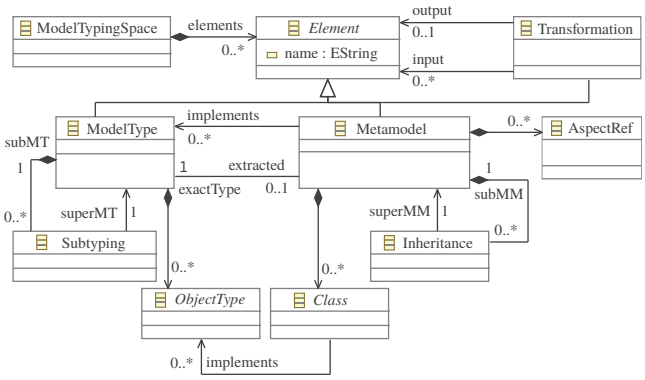

Figure 3: Excerpt of the abstract syntax K3SLE

into any previously created metamodel. Besides, an *Inheritance* relationship has been defined between metamodels. Finally, according to Definition 7, a metamodel can implement multiple model types.

A *ModelType* defines the interface of a DSML and is thus used to type models. It is composed of a set of *ObjectType*s. The exact model type (reference *exactType*) of a metamodel is automatically *extracted* to facilitate the model typing resolution. A subtyping relationship is also defined between model types. The subtyping and inheritance relationships have been reified as classes to smoothly introduce, in future improvements, different semantics for subtyping (*e.g.* with/without static semantics checks [35]) and inheritance (*e.g.* with/without aspects).

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

The concepts of *Class* and *ObjectType* are sufficient to abstractly reason about the implementations and interfaces of meta-elements. It opens up the possibility for manipulating artifacts defined using different modeling frameworks. However, in our current implementation, we only support metamodels specified using Ecore files. Ecore provides the following language constructs for specifying a DSL metamodel: package, classes, properties, multiple inheritance, and different kinds of associations between classes. The semantics of these object-oriented constructs is close to a standard object model that is shared by various languages (*e.g.* Java). We chose Ecore because it is a *de facto* standard allowing the interoperability with other tools. Consequently, the *Class* and *ObjectType* meta-classes are concretely mapped to Ecore elements.

4.2 Concrete Syntax

For the sake of concision, we do not detail here the whole grammar used to define the concrete syntax of K3SLE. Instead, we illustrate its core usage throughout the simple example described in Listing 1.

```
1  metamodel MMA {
2    ecore "MMA.ecore"
3    exactType MTA
4  }
5  metamodel MMB {
6    ecore "MMB.ecore"
7    exactType MTB
8    aspect AspectFoo
9    aspect AspectBar
10 }
11 metamodel MMC inherits MMA {
12   exactType MTC
13   aspect AspectBaz
14 }
15 modeltype MTX {
16   ecore "MTX.ecore"
17 }
18 transformation void foo(MTX m) { ... }
19 transformation MTC bar(MTA m) { ... }
```

Listing 1: Example of the concrete syntax of K3SLE

Similarly to the abstract syntax, the 3 main elements of the concrete syntax are *metamodels*, *model types*, and *transformations*. The concrete syntax of a metamodel consists of importing the associated Ecore file (*e.g.* line 2) and declaring the identifier of its exact type (*e.g.* line 3). The second metamodel, *MMB*, additionally imports a set of aspects woven on its abstract syntax (lines 8-9). A metamodel, such as *MMC*, can be defined by inheriting from another one (line 11). In this case, both the abstract Ecore syntax and the aspects of the super-metamodel are inherited.

In addition to the exact types of the metamodels, other model types can be declared by importing an Ecore file (line 16). Note that although the syntax is the same, the import of an Ecore file has a different meaning within a metamodel or a model type: in the former case it defines the implementation (*i.e.* metamodel) of a language; in the latter it defines the interface (*i.e.* model type) of a language. Neither the subtyping nor the implementation relationships are explicitly written in the concrete syntax. Instead, they are inferred by the model-oriented type system at compile time, as described in Section 4.3.

A model transformation (lines 18-19) can specify multiple input parameters and a return type. The aforementioned concrete syntax has been implemented using Xtext, an open-source framework for implementing textual DSLs and their associated tools [9]. From a grammar specification, Xtext is able to generate a complete textual editor within Eclipse. Associated to Xtext, Xbase is a general-purpose expression language build on top of the Xtext framework and Eclipse that comes with its own compiler that generates Java code [9]. The body of tranformations is written using Xbase, thus leveraging all the features of this language (control structures, lambda expressions, *etc.*) and its tooling (type checking, syntax highlighting, auto-completion, *etc.*). Our implementation is freely available on the companion webpage[9].

4.3 Behavioral Semantics

The behavioral semantics of K3SLE is implemented by means of a compiler that transforms an instance of the K3SLE metamodel in Java code, providing a complete interoperability with legacy EMF code. Since the type system of Java does not support type groups polymorphism in a safe and reusable way [11], we present in this section the different artifacts generated by the K3SLE compiler on top of the EMF artifacts that allow to consider a metamodel as a group of related types, and thus provide model polymorphism.

4.3.1 Overview of the compiler

Once a valid instance of the K3SLE metamodel created, *e.g.* using the textual Xtext editor, the compiler first completes the resulting abstract syntax tree (AST) by infering the exact types of the different metamodels and resolving the subtyping hierarchy. The implementation relationships between metamodels and model types are also inferred in this phase, thus building a complete AST conforming to the metamodel of Figure 3.

[9]http://diverse.irisa.fr/software/k3sle/

We implemented the K3SLE compiler using the Xtext framework. Xtext comes with a metamodel of the Java language and is able to automatically generate Java code from an AST conforming to it. Thus, the second phase of compilation consists in a transformation from the concepts of the metamodel of Figure 3 to the concepts of the Java metamodel. This mapping is described in depth in the next subsection. Once the target Java model is built, the associated Java code is automatically generated by the Xtext framework. Figure 4 summarizes this compilation process, combined with the legacy EMF compilation chain.

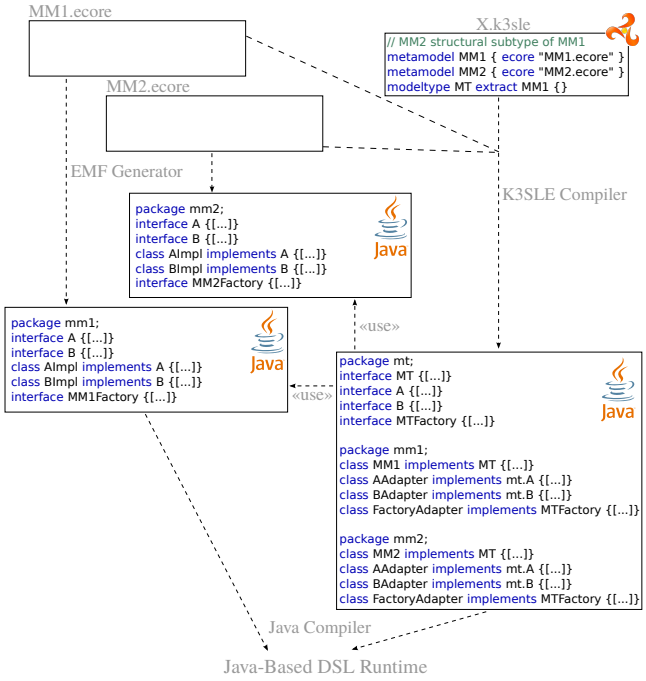

Figure 4: Overview of K3SLE compilation process

4.3.2 Notation

The denotational semantics of the K3SLE meta-language towards Java constructs is described using the following notation and operators. Let MT be a model type composed of a set of object types T with their methods signatures and relationships, and MM a metamodel composed of a set of classes A with their own attributes, operations, and relationships. We also consider $\bullet$ the Java class extension operator (keyword *extends*) and $\circ$ the Java interface implementation operator (keyword *implements*). The description of the semantics is based on the following Java artifacts generated by the EMF framework for a given metamodel:

- $Factory^{Emf}_{Java_I}$ is the generated factory that create instances of its meta-classes
- $A^{Emf}_{Java_I}$ is the Java interface generated for each meta-class

Finally, $Adapter^{K}_{T}$ denotes a Java class adapter that implements the interface T and delegates method calls to the adapted class K.

4.3.3 Model type compilation

For each model type, the compiler generates a set of Java interfaces as such $MT = \{MT^{K3}_{Java_I}, \{A^{K3}_{Java_I}\}, Factory^{K3}_{Java_I}\}$ where:

1. $MT^{K3}_{Java_I}$ is a Java interface for the model type itself. It serves as an interface to interact with the concrete resource used by EMF to manipulate the model.
2. $A^{K3}_{Java_I}$ is a generated interface for each exposed element A. It exposes properties accessors and mutators and provided operations for the element.
3. $Factory^{K3}_{Java_I}$ is a generated interface that defines its abstract factory. This factory is an *adapter* interface for mapping the *client* model type for a particular metamodel. A concrete factory is generated for each pair (MT/MM).

These interfaces are the main support for substituability both at the model and model element levels. The other artifacts, either generated by EMF or by the K3SLE compiler, then implement – or are adapted to – them, so that the substituability between concrete implementations is ensured.

4.3.4 Metamodel compilation

For each metamodel MM, the compiler generates a set of Java classes as such $MM = \{MM^{K3}_{Java_C} \circ \{MT^{K3}_{Java_I}\}\}$ where:

1. $MM^{K3}_{Java_C}$ is a class for the metamodel itself that directly implements all the Java interfaces $MT^{K3}_{Java_I}$ of the model types MT_i it implements.

For each model type MT implemented by a metamodel MM, the compiler generates a set of Java classes as such $MM/MT = \{\{Adapter^{K3}_{Java_C}\}, Factory^{K3}_{Java_C}\}$ where:

1. $Adapter^{K3}_{Java_C} = Adapter^{A^{Emf}_{Java_I}}_{A^{K3}_{Java_I}}$ is a class that represents a concrete adapter from each model element $A^{EMF}_{Java_I}$ it contains towards the corresponding object type interface $A^{K3}_{Java_I}$.
2. $Factory^{K3}_{Java_C} = Adapter^{Factory^{Emf}_{Java_I}}_{Factory^{K3}_{Java_I}}$ that delegates the creation of the model elements to the EMF factory $Factory^{Emf}_{Java_I}$ and wraps the newly created model elements into the generated adapters.

4.3.5 Inheritance compilation

Inheritance (denoted $\Diamond$) allows to reuse structure and behavior between metamodels. It is implemented as an inclusion mechanism, *i.e.*, when a metamodel inherits from another one, a new Ecore file and its associated EMF code is generated. This newly generated code may then be adapted to any defined model type using the adaptation mechanism described in Section 4.3.4. Multiple inheritance is also supported with a merge without conflict. Any conflict during the structural merge [29] of the inherited metamodels raises a typing error.

As a result, $MM_1 \Diamond \{\{A\}, MM_2, MM_3\} \equiv MM_1 = \{A\} \bigoplus \{A_2\} \bigoplus \{A_3\}$, where $MM_2 = \{A_2\}$, $MM_3 = \{A_3\}$, and $\bigoplus$ denotes the class merge operator without conflict.

Note that inheritance is a specific composition operator and not a typing facility. However, by definition the exact model type of a metamodel is a sub-model type of the exact model types of the metamodels that it inherits.

4.3.6 Dealing with the conformance relation and URIs on model loading

The model loading mechanism integrated in K3SLE relies on the model loading operation provided by the EMF framework.

However, whereas the EMF loading operation explicitly relies on the URI of the loaded model that identifies a unique metamodel, K3SLE relaxes this contraint by allowing the loading of a model according to a specific interface (*i.e.* a model type). More precisely, the loading operation checks if it *can* load a model according to a given model type: a model is loadable if the exact type of its metamodel implements this model type. This mechanism can be used to filter specific informations of the model at load-time, thus providing a basic mechanism for multi-viewpoints management.

5. USE CASES

In this section, we illustrate different facilities provided by the abstraction of the conformance relationship standing between models and metamodels with the typing relationship standing between models and model types in the context of MDE. We then discuss how the end developer of domain-specific languages benefits from these new facilities and how they can facilitate the definition of model transformations or the management of unpredicted evolutions of an existing metamodel.

The use cases use two slightly different metamodels of a simple executable finite state machine. The first one, *Fsm*, defines the concepts *FSM*, *State* and *Transition* ; the latter, *TimedFsm*, is an enriched metamodel with time constraints on transitions. Figure 5 depicts both the abstract syntax (on the left) and the operational semantics expressed using aspects (on the right) of these metamodels. The concepts specific to *TimedFsm* (*Transition*'s attribute *time* and its associated *timeIsOk* method) are highlighted in red. All the other concepts are shared between the two metamodels.

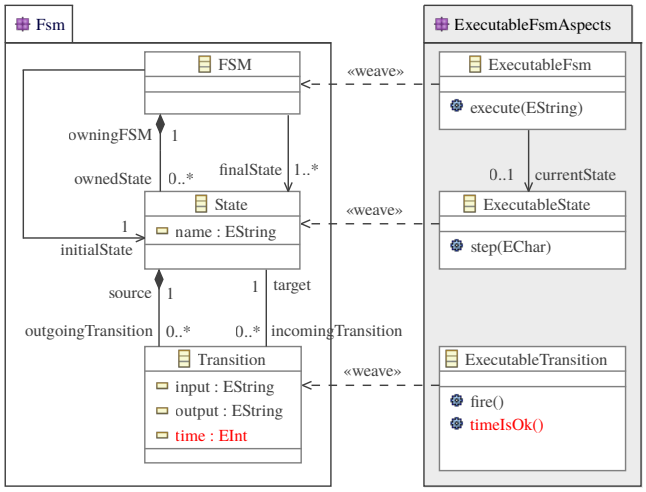

Figure 5: Metamodel of a simple executable finite-state machine

5.1 Model Polymorphism

The separation of the language implementation (*i.e.* metamodel) from the language interface (*i.e.* model type) permits to consider a given model differently depending on the used model type: a model can have multiple types. This enables polymorphism at the model level, providing both substituability and dynamic binding. In Listing 2, the same transformation is applied on the two metamodels of Figure 5.

```
1  metamodel Fsm {
2    ecore   "FSM.ecore"
3    exactType FsmMT
4    aspect fsm.ExecutableFSM
5    aspect fsm.ExecutableState
6    aspect fsm.ExecutableTransition
7  }
8  metamodel TimedFsm {
9    ecore  "TimedFSM.ecore"
10   exactType TimedFsmMT
11   aspect timedfsm.ExecutableTimedFSM
12   aspect timedfsm.ExecutableTimedState
13   aspect timedfsm.ExecutableTimedTransition
14 }
15 transformation execute(FsmMT fsm) {
16   val root = fsm.contents.head as fsmmt.FSM
17   root.execute("some_word")
18 }
19 @Main transformation main() {
20   val m1 = Fsm.load("ExFsm.xmi", FsmMT)
21   val m2 = TimedFsm.load("ExTFsm.xmi", FsmMT)
22   execute.call(m1)
23   execute.call(m2)
24 }
```

Listing 2: Model Polymorphism

The two metamodels *Fsm* and *TimedFsm* are defined by importing their abstract syntax (Ecore models, lines 2 and 9) and a set of aspects (lines 4-6 and 11-13). Each aspect defines the operational semantics of the targeted meta-class. Their respective exact types *FsmMT* and *TimedFsmMT* are also extracted (lines 3 and 10).

Because *TimedFsmMT* is structurally a subtype of *FsmMT*, any model typed by *TimedFsmMT* can be supplied to an operation expecting a model typed by *FsmMT*. In our example, two models conforming to the two metamodels are loaded in the transformation *main* (lines 20-21) w.r.t. the same model type. The transformation *execute* is then applied on both of them (lines 22-23), thanks to the substituability facility provided by model polymorphism.

Thanks to the dynamic binding of aspects methods, the behavioral semantics provided by aspects woven on *Fsm* (resp. on *TimedFsm*) are used when the model effectively passed to the transformation conforms to *Fsm* (resp. *TimedFsm*). Thus, the execution of a model conforming to *Fsm* will go through the appropriate states and transitions whereas the execution of a model conforming to *TimedFSM* will take into account the time constraints associated.

This simple example shows how model polymorphism simplifies the development of model manipulation operations: the overly restrictive conformance relationship would have forbidden the call of a simple transformation on models conforming to two different yet similar metamodels whereas the use of model interfaces coupled with models substituability and dynamic binding enables the expected behavior.

5.2 Language Inheritance

A closer look at the previous use case reveals that it can be nicely rewritten using language inheritance to represent the variation points (time contraints) between the two metamodels. Indeed, the *TimedFsm* metamodel defines the exact same concepts as *Fsm*, with some additional information on the *Transition* metaclass to support time constraints.

In such a case, defining an explicit inheritance relationship between the two metamodels allows the reuse of both concepts and semantics from one metamodel to another. In Listing 3, the *TimedFsm* metamodel inherits both the concepts and semantics of the *Fsm* metamodel. It also imports a specialized aspect

ExecutableTimedTransition on its *Transition* metaclass that extends the *ExecutableTransition* aspect and *overrides* its *fire*() method, calling the new *timeIsOk*() method in order to check time constraints. If the latter returns *true*, it simply calls the parent *fire*() method.

```
1  metamodel Fsm {
2    ecore "FSM.ecore"
3    exactType FsmMT
4    aspect fsm.ExecutableFSM
5    aspect fsm.ExecutableState
6    aspect fsm.ExecutableTransition
7  }
8  metamodel TimedFsm inherits Fsm {
9    exactType TimedFsmMT
10   // Redefinition of the fire() method
11   // in Transition to take time into account
12   aspect timedfsm.ExecutableTimedTransition
13 }
14 transformation execute(FsmMT fsm) {
15   val root = fsm.contents.head as fsmmt.FSM
16   root.execute("some_word")
17 }
18 @Main transformation main() {
19   val m1 = Fsm.load("ExFsm.xmi", FsmMT)
20   val m2 = TimedFsm.load("ExTFsm.xmi", FsmMT)
21   execute.call(m1)
22   execute.call(m2)
23 }
```

Listing 3: Language inheritance

The remainder of the code is strictly the same as in Listing 2, but the inheritance mechanism greatly simplifies the definition of the *TimedFsm* metamodel. This mechanism can help in defining language families by enabling the definition of variants of a specific language, *e.g.* here with the insertion of time constraints, while maintaining the compatibility with the model manipulation operations previously defined on the super-metamodel.

5.3 Discussion

The facilities presented in the two previous use cases pave the way for more easily tackle a wide range of scenarios that are commonly faced by DSMLs engineers. As pointed out in the introduction, the conformance relation prevents the reuse of tooling within the same family of languages composed of conceptually close DSMLs. Model polymorphism solves this problem by enabling the use of the same tools on different metatamodels, provided that they implement the same interface (*i.e.* model type). Taking the example of the finite-state machines family, the different variants (*e.g.* composite states, final states, pseudo-states) may be designed by means of inheritance from a common simple state-machine metamodel on which different tools (*e.g.* interpreters, pretty-printers) are defined. The different variants would then natively benefit from this generic tooling, and may even refine their semantics if needed thanks to dynamic binding.

In the case of independently designed DSMLs, reuse is still possible if they share some commonalities. This is for example the case for state diagrams produced with an Airbus domain-specific tool (called SAM) and UML statecharts which are conceptually the same. Although they were designed independently, the definition (or extraction) of a common interface can provide interoperability between them. For complex cases, this may require the definition of adaptation mechanisms that are beyond the scope of this paper.

DSMLs are by definition doomed to evolve with the domain they represent. Similarly, widely used modeling languages such as MOF or UML are also subject to evolution. It is difficult for engineers to deal with this evolution, as all efforts concentrated around a language are lost with subsequent versions. With the use of model interfaces, the different tools can most of the time be reused. An example of model transformation reuse accross two subsequent versions of UML is available on the companion webpage.

6. RELATED WORK

This section synthesizes an overview of related work. In particular, it focuses on model transformation reuse and advanced typing in object-oriented programming language.

6.1 Model Transformation Reuse

The increasing trend to create new DSMLs, from scratch or by adapting existing ones, causes the emergence of families of DSMLs, *i.e.* a set of DSMLs sharing common aspects but specialized for a particular purpose. The emergence of a DSML family raises the need to reuse common tools among a given family [22, 21].

Several approaches have been proposed over the last decade for model transformation reuse. These approaches can be divided into two categories: approaches for model transformation reuse without adaptation (*i.e.*, reuse between isomorphic metamodels) and approaches allowing adaptations (*i.e.*, structural heterogeneities).

6.1.1 Reuse without Adaptation

Model transformation reuse without adaptation was first proposed by Varró *et al.* who introduced *variable entities* in patterns for declarative transformation rules [36]. These entities express only the needed concepts (types, attributes, *etc.*) to apply the rule, allowing any tokens with these concepts to match the pattern and thus to be processed by the rule. Later, Cuccuru *et al.* introduced the notion of semantic variation points in metamodels [5]. Variation points are specified through abstract classes defining a *template*. Metamodels can fix these variation points by binding them to classes extending the abstract classes. Patterns containing *variable entities* and *templates* can be seen as kinds of model types where the variability has to be explicitly expressed and thus anticipated.

Cuadrado *et al.* propose a notion of substitutability based on model typing and model type matching [31]. Instead of using an automatic algorithm to check the matching between two model types, they propose a DSL to manually declare the matching.

De Lara *et al.* present the *concept* mechanism, along with *model templates* and *mixin layers* leveraged from generic programming to MDE [8]. *Concepts* are really close to model types as they define the requirements that a metamodel must fulfill for its models to be processed by a transformation, under the form of a set of classes. The authors also propose a DSL to bind a metamodel to a *concept* and a mechanism to generate a specific transformation from the binding and the generic transformation defined on the *concept*.

6.1.2 Reuse through Adaptation

Adaptation allows the reuse of model transformations between metamodels in spite of structural heterogeneities. Two approaches exist. The first one adapts models conforming to a metamodel *MM* into models conforming to a metamodel MM' on which is defined the transformation of interest. The second one adapts a transformation defined on MM' to obtain a valid transformation on *MM*.

Kerboeuf *et al.* present an adaptation DSL named *Modif* which handles deletion of elements from a model (which conforms to a metamodel *MM*) to make it substitutable to an instance of the metamodel MM' [20]. For this, a trace of the adaptation is saved to be able to go back from the result of the transformation (conforming to MM') to the corresponding instance of *MM*.

Garcia *et al.* proposed to semi-automatically adapt a transformation with respect to metamodel changes [15]. A classification of metamodel evolutions is proposed as well as automatic adaptations of the transformation for some of them.

Sánchez, Wimmer *et al.* extended the binding mechanism presented for *concepts* by De Lara *et al.* [8] to go further than strict structural mapping by renaming, mapping, and filtering metamodel elements [32, 38]. These adaptations are possible through an hybrid approach that mixes model and transformation adaptations. This approach allows the same kind of adaptations than the injection of derived attributes in Kermeta aspects [24]. Contrary to Kermeta these adaptations are automatically generated.

These approaches permit to go further than reuse between isomorphic metamodels. However, some heterogeneities still cannot be handled automatically, such as different representations of similar information. Besides, to our best knowledge, the problem of DSLs and model transformations specialization as well as the problem of expressing model transformations dependencies have not been addressed yet.

For a detailed survey of the various approaches for model transformation reuse, we refer the reader to the recent study realized by Kusel *et al.* [22]. As noticed by the authors, a major current barrier to model transformation reuse is the insufficient abstraction from metamodels while the reuse mechanisms depend on concrete metamodel types (even approaches with generic types still depend on the internal structure of the metamodels). In this paper, we address this limitation by an explicit separation of the implementation and the interface of a given language, leading to a unified typing theory for models. The interface (*i.e.* the model type) is then used to declare the typing and subtyping relations, making them independent of the conformance with a particular implementation (*i.e.* metamodel).

6.2 Typing in OOP

Object-oriented type systems garner a considerable interest in the last decades in providing advanced typing mechanisms for programming language. This section discusses the relation between our proposal and seminal work on typing in the OOP domain. Based on a set of basic case studies from the modelling community, we show how we uncouple the typing system at the model level and the host language typing system to propose a typing relation at the model-level allowing to manipulate a model through different DSMLs and uncover the need for model-oriented type systems.

Nominal typing relies on types' name to explicitly define their typing relation. By analogy, the current MDE technologies are based on nominal typing where a model is bound to its unique DSML by using the name, more precisely the URI, of this last. In opposite, structural typing relies on the structure of types to define, both at design and run times, typing relations. Structural characteristics used to define such a relation can be for instance the return type and the parameters' type of operations, or the attributes' type of classes. Structural subtyping may be useful to bind two independent type hierarchies having some similar operations. If nominal type system prevents to bind two independent type hierarchies, a classical solution is to use the *Adapter* design pattern [14] to group them under the same hierarchy. However, implementing this pattern involves a substantial development effort. The scope of structural typing in current OOP languages is limited to the class level. In our work, we aim at reifying this concept for MDE with a major difference: because models are a first-class concept in MDE, our work concerns a higher level than classes, *i.e.* the model level, to propose a model-oriented type system.

Several advanced typing mechanisms have been proposed to enhance the reuse in OO programs. Scala [26] and gBeta [18] propose to support family polymorphism through path dependent types. Nystrom *et al.* [25] introduce the concept of Nested Inheritance, a mechanism that addresses some of the limitations of ordinary inheritance and other code reuse mechanisms. Lammel *et al* [23] demonstrates how the use of type classes can simplify the extension and integration of legacy code. In the same vein, object algebras demonstrate how we can solve the expression problem [6, 7]. The basic idea is to create a family of objects via an abstract factory. New objects can be added to the family by extending the factory as per usual, and new operations can be added by overriding the factory methods. Based on MDE use cases, we discuss the need to uncouple the typing system at the model level and the host language typing system. Our final goal is to propose a typing relation at the model-level to manipulate a model through different DSMLs and uncover the need for model-oriented type systems. We also show the need for family polymorphism support in the model-oriented type system.

In another way, pluggable type system promotes the ability to define and use an *ad-hoc* type system on top of an exiting OOP language providing its own type system [4]. Pluggable type system is supported by several classical OOP languages such as Java and Xtend through the use of annotations and their dedicated processors. This mechanism permits to extend the type system of these OOP languages to perform specific checks. Our implementation relies on this concept to bind our typing system to Java type system. Technically, we defined our own pluggable type system on top of Xtend to provide developers with a model-oriented type system and allow a smooth integration with EMF legacy code.

7. CONCLUSION AND PERSPECTIVES

In this paper, we proposed to abstract the overly restrictive conformance relation that stands between models and metamodels with the typing relation between models and model types. We claim model types should become the cornerstone of reuse in MDE. We advocate to uncouple the model-oriented typing system from the host language typing system typically used by modeling frameworks such as EMF.

We introduced K3SLE as a new modeling framework on top of the EMF framework and the Kermeta action language to support and validate these ideas. We have shown how it can leverage family polymorphism to allow model polymorphism, inheritance among DSMLs as well as DSML evolution and interoperability.

K3SLE can be seen as a first step towards *modeling in the large*, where (model type) transformations, (model type) attributes, behavioral subtyping are supported first-hand. Another related perspective is to develop the notion of language interface (including required interfaces), in relation with new possible usages such as language composition.

On the implementation side, many choices have been made in Section 4 for our prototype (*e.g.* static typing rather than dynamic, use of the Adapter pattern), but other variations could be investigated.

Beyond the MDE domain, we believe that the notion of model type could be used as a lightweight and easy to understand view on complex hierarchies of type families that are excessively hard to understand when directly expressed with plenty of mutually dependent generic parameters as in *e.g.* Java. Of course, some experimental studies should also be needed for evaluating the usability of the concept of language interface, as well as the proposed language for modeling in the large.

8. REFERENCES

[1] C. Atkinson and T. Kühne. Profiles in a strict metamodeling framework. *Science of Comp. Program.*, 44(1):5–22, 2002.

[2] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proc. of ASE'01*, pages 273–280. IEEE Computer Society, 2001.

[3] J. Bezivin, F. Jouault, and D. Touzet. Principles, standards and tools for model engineering. In *Proc. of ICECCS'05*, pages 28–29, 2005.

[4] G. Bracha. Pluggable type systems. In *Proc. of OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[5] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard. Templatable metamodels for semantic variation points. In *Proc. of ECMDA-FA'07*, pages 68–82, 2007.

[6] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses - practical extensibility with object algebras. In *Proc. of ECOOP'12*, pages 2–27. Springer, 2012.

[7] B. C. d. S. Oliveira, T. Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *Proc. of ECOOP'13*, pages 27–51. Springer, 2013.

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

[9] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: implementing domain-specific languages for java. In *Proc. of GPCE'12*, pages 112–121. ACM, 2012.

[10] M. Egea and V. Rusu. Formal executable semantics for conformance in the MDE framework. *Innovations in Software and Systems Engineering*, 6(1-2):73–81, 2010.

[11] E. Ernst. Family polymorphism. In *Proc. of ECOOP'01*, pages 303–326. Springer, 2001.

[12] J. Estublier and G. Vega. Reuse and variability in large software applications. In *Proc. of ESEC/FSE'05*, pages 316–325. ACM, 2005.

[13] J.-M. Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon. *Dagstuhl Reports*, 2004.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] J. Garcia and O. Díaz. Adaptation of transformations to metamodel changes. In *Desarollo de Software Dirigido por Modelos*, pages 1–9, 2010.

[16] D. Gasević, N. Kaviani, and M. Hatala. On Metamodeling in Megamodels. In *Proc. of MODELS'07*, pages 91–105, 2007.

[17] C. Guy, B. Combemale, S. Derrien, R. H. Steel, James, and J.-M. Jézéquel. On Model Subtyping. In *Proc. of ECMFA'12*, 2012.

[18] A. Igarashi, C. Saito, and M. Viroli. Lightweight family polymorphism. In *Programming Languages and Systems*, pages 161–177. Springer, 2005.

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

[20] M. Kerboeuf and J.-P. Babau. A DSML for reversible transformations. In *Proc. of DSM'11*, pages 1–6, 2011.

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

[22] A. Kusel, J. Schonbock, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *Software & Systems Modeling*, pages 1–36, 2013.

[23] R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *Proc. of GPCE'06*, pages 161–170, 2006.

[24] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *Proceedings of the 8th International Conference on the Unified Modeling Language*, Proc. of MODELS/UML'05, pages 264–278, 2005.

[25] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. *SIGPLAN Not.*, 39(10):99–115, Oct. 2004.

[26] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. of ECOOP'03*, pages 201–224. Springer, 2003.

[27] OMG. Meta Object Facility (MOF) 2.0 Core Specification, 2006.

[28] OMG. *Unified Modeling Language (OMG UML), Infrastructure*, 2011.

[29] R. Reddy, R. France, S. Ghosh, F. Fleurey, and B. Baudry. Model composition-a signature-based approach. In *Aspect Oriented Modeling (AOM) Workshop*, 2005.

[30] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model migration with epsilon flock. In *Proc. of ICMT '10*, pages 184–198. Springer, 2010.

[31] J. Sánchez Cuadrado and J. García Molina. Approaches for model transformation reuse: Factorization and composition. In *Proc. of ICMT'08*, pages 168–182, 2008.

[32] 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.

[33] J. Steel and J. M. Jézéquel. On model typing. *Software and Systems Modeling*, 6(4):401–413, 2007.

[34] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2 edition, 2009.

[35] W. Sun, B. Combemale, S. Derrien, and R. B. France. Using model types to support contract-aware model substitutability. In *Proc. of ECMFA'13*, pages 118–133. Springer, 2013.

[36] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *Proc. of UML'04*, pages 290–304, 2004.

[37] W3C. XML Schema Part 0: Primer Second Edition. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`. [Online; accessed 17-October-2013].

[38] 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.