Safe Model Polymorphism for Flexible Modeling

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

Safe Model Polymorphism for Flexible Modeling

Thomas Degueule[a], Benoit Combemale[a], Arnaud Blouin[b], Olivier Barais[c], Jean-Marc Jézéquel[c]

[a] *Inria, France*
[b] *INSA Rennes, France*
[c] *University of Rennes 1, France*

Abstract

Domain-Specific Languages (DSLs) are increasingly used by domain experts to handle various concerns in systems and software development. To support this trend, the Model-Driven Engineering (MDE) community has developed advanced techniques for designing new DSLs. However, the widespread use of independently developed, and constantly evolving DSLs is hampered by the rigidity imposed to the language users by the DSLs and their tooling, e.g., for manipulating a model through various similar DSLs or successive versions of a given DSL. In this paper, we propose a disciplined approach that leverages type groups' polymorphism to provide an advanced type system for manipulating models, in a polymorphic way, through different DSL interfaces. A DSL interface, aka. *model type*, specifies a set of features, or services, available on the model it types, and subtyping relations among these model types define the safe substitutions. This type system complements the Melange language workbench and is seamlessly integrated into the Eclipse Modeling Framework (EMF), hence providing structural interoperability and compatibility of models between EMF-based tools. We illustrate the validity and practicability of our approach by bridging safe interoperability between different semantic and syntactic variation points of a finite-state machine (FSM) language, as well as between successive versions of the Unified Modeling Language (UML).

Keywords: Metamodeling, Model typing, Type groups polymorphism

1. Introduction

Extending the time-honored practice of separation of concerns, Domain-Specific (*modeling*) Languages (DSLs) are increasingly used to handle different, complex problem or solution concerns in software-intensive system development [43]. In particular, problem-based DSLs target experts who focus on building parts of software or systems that address specific concerns (e.g., security policies or variability management). These DSLs help narrow the gap between problem and implementation concepts by providing experts with problem-level abstractions and tools to transform these abstractions into executable artifacts.

A DSL provides appropriate constructs based on abstractions relevant for a specific domain of expertise. As the concepts of the domain and the experts understanding of the domain evolve, the DSL evolve accordingly.

Separation of concerns is achieved through the use of different DSLs, each supported by specific tools dedicated to particular modeling activities (analysis, refactoring, code generation, etc.). DSLs are thus challenged by the socio-technical coordination required during software and system development throughout the entire life cycle (requirement, analysis, design, development, maintenance, and runtime management). The social coordination is intended to support the communication between the various domain experts involved in the development. These experts share (part of) the information to be used in various environments according to the different points of view on the system. The technical coordination requires composing and ensuring the consistency of the different artifacts, which implies their manipulation in different tools.

The constant evolution of DSLs and their necessary coordination raise the need of more flexibility for DSL users in their manipulation of the models through various modeling tools (either using different versions of a given DSL or different DSLs). This requires increased compatibility of modeling tools between different versions of a given DSL, and interoperability between the modeling tools used by different stakeholders.

Model-Driven Engineering (MDE) technologies assist domain experts in defining problem-space DSLs, without requiring strong skills in language implementation or compiler construction [33]. Using MDE technologies, new DSLs are typically first specified through metamodels that define their abstract syntax[1]. The MDE community has developed a rich ecosystem of interoperable, generative tools defined over standardized object-oriented metamodeling technologies such as EMOF [46]. These tools can generate supporting DSL tools such as parsers, code generators, and other integrated development environment services.

MDE technologies strongly rely on the *conformance relation*, which states that a model conforms to a metamodel if each of its elements is an instance of a meta-class defined in the metamodel. As a result, *a model conforms to a unique metamodel*: the one used to create it. This implementation-oriented view prevents *model polymorphism*, i.e., the ability to manipulate a given model through different interfaces, each one associated to a particular DSL for a specific domain of expertise or particular modeling tools. Therefore, it is not possible to safely manipulate a model using different versions or variants of a DSL.

In the last decade, significant efforts were devoted to model and metamodel co-evolution, mainly through the implementation of specific model transformations [60, 11]. However, we demonstrate in this paper that an important part of the required interoperability and compatibility appears between structurally similar DSLs, opening up the possibility to automatically provide more flexibility in their manipulation. To motivate this claim, we report in this paper the results

[1] A metamodel defines the abstract syntax of a DSL, and a model is a (graphical or textual) statement expressed in the DSL.

of an experimental study on the UML models publicly available on Github[2]. This study shows up to 93% of compatibility opportunities to load the models according to different versions of UML. We also identify different intermediate flexibility levels according to the objective (support of the application of read-only model transformations, or in-place model transformations that modify the model) and the usage context (support of the compatibility to apply any model transformations, or for a particular model transformation whose footprint can be computed [35]).

To address these limitations, we propose to circumvent the overly restrictive conformance relation standing between models and metamodels with a dedicated type system. We explore this principle through a disciplined approach that leverages type groups' polymorphism to provide an advanced type system for manipulating models, in a polymorphic way, through different DSL interfaces. A DSL interface specifies a set of features, or services, available on the model it types. In our approach, these interfaces are captured in *model types* and supported by a typing relation between models and model types [55] and a subtyping relation between model types [31]. Model types are structural contracts over a language. They are used to define a set of constraints over admissible models, where a model is a graph of meta-class instances referred to as objects. Subtyping relations define the safe structural substitutions from one DSL to another. This opens up the possibility to define model manipulation tools (e.g., transformations, checkers, compilers) that can be reused for different languages, provided that they implement the required interface. This is the approach followed by the Melange language workbench to support reusable DSL specifications [19].

In this paper:

- We list the properties of the conformance relation that are at the heart of the current MDE approaches and show how they hinder flexible modeling. We complete this study by analyzing the flexibility opportunities through an experimental study of the UML models publicly available on Github;
- We present the necessary concepts and relations for MDE that lift the current limitations by complementing the conformance relation with a typing relation based on an explicit structural language interface (aka. model type), therefore providing increased compatibility and interoperability to manipulate a model through different DSL interfaces;
- We describe a model-oriented type system that supports these concepts and relations to provide a safe mechanism of polymorphism for models. This type system ensures *safe structural substitutability* of models conforming to different languages;
- We extend Melange, a language workbench seamlessly integrated with the *Eclipse Modeling Framework* (EMF) ecosystem, with an implementation

[2] https://github.com/

of the model-oriented type system. We detail how the implementation of Melange successfully emulates type groups polymorphism and structural typing to provide model polymorphism seamlessly on top of the legacy EMF ecosystem;

- We illustrate the validity and practicability of our approach through a controlled experiment on a family of finite-state machine languages, and an uncontrolled experiment on the UML models collected on Github.

The remainder of this paper is organized as follows. Section 2 presents the limitations of the conformance relation in the use of modeling tools, both from a theoretical and experimental point of view. Section 3 introduces our proposal on superseding model conformance with a typing relation enabling better flexibility in the manipulation of models. Section 4 introduces Melange, an implementation of the proposed type system within the Eclipse platform, and atop EMF. Section 5 describes experimentations on controlled and uncontrolled case studies. Section 6 discusses related work. Section 7 concludes and discusses perspectives of our work.

2. On the Limits of the Conformance Relation

To state whether a model is a valid instance of a DSL, MDE relies on the *conformance relation* that stands between a model and its metamodel. The conformance relation plays a crucial role in MDE as it identifies *which* models are valid instances of a given DSL and *how* they should be manipulated. In this section, we first review the conformance relation definitions used in the literature, tools, and standards. Based on these definitions, we provide and detail theoretical limits of this conformance. We then conduct a systematic analysis of UML (Unified Modeling Language [51]) models gathered from the popular repository hosting service Github. Specifically, we show that the constraints enforced by the conformance relation can be relaxed to increase the level of flexibility in model manipulation without losing any guarantee in terms of safe model manipulation.

2.1. Limits of the Conformance Relation from a Theoretical Point of View

In MDE, the abstract syntax of DSLs is usually defined by a metamodel [53]. Based on this cornerstone artifact, concrete syntaxes, semantics, and various tools can be defined [56]. To determine whether a model is a valid statement of a DSL, MDE relies on the conformance relation that stands between a model and a metamodel. While no standard definition of the conformance relation exists, a recurring definition has emerged from the literature: a model conforms to a metamodel if every element of the model is an instance of one of the elements of the metamodel.

Different names have been given to this relation in the literature: "*sem*" (e.g., Bézivin and Gerbé [4]), "*instantiation*" (e.g., Atkinson and Kühne [1]) or "*conformance*" (e.g., Bézivin et al. [5]). All these relations are directly based

on the abstract syntax of DSLs, expressed in the form of a metamodel. In the following, we use the term *conformance relation* to refer to this relation between models and metamodels. Table 1 presents several definitions of the conformance relation from the literature over the last decade.

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.) [25]. Favre's definition is thus less strict than the other ones presented in this section. However, while this definition is sufficient for the study and understanding of MDE, it is not precise enough for DSLs tooling or automated checking of the validity of a model wrt. a DSL.

Other authors define the conformance relation through the instantiation relation that 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*" [29]; "*every object in the model has a corresponding non-abstract class in the metamodel*" [22]. Bézivin et al. do not directly refer to classes, but prefer the terms "*definition*" [4] or "*meta-element*" [5] (i.e., element of a metamodel). Such definitions authorize the definition of the abstract syntax of a DSL 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 all agree on one point: the conformance relation is based on the instantiation relation between objects and classes.

The Meta-Object Facility (MOF) specification [46] from the Object Management Group (OMG) and the Eclipse Modeling Framework [56] (EMF) are *de facto* technological standards in the MDE community. Many tools and frameworks are based on these standards, such as ATL [37], Kermeta [36], Epsilon [39], or Xtext [24] to name just a few. The way these standards define the relation between models and metamodels is thus central in today's tooling support of MDE.

MOF permits the definition of the abstract syntax of DSLs in the form of an object-oriented metamodel. MOF, however, does not give any indication regarding the relation that stands between a model and a metamodel. Besides, 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*" [47].

EMF does not give any definition either, but relies on two technologies to manipulate models: Java classes for instantiating elements of models and XML Schema for model serialization[3] [56]. 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

[3]EMF can also instantiate models reflectively or using a dedicated serialization mechanism provided by users of the framework. However, we only consider the case of XML Schema which is the default behavior of EMF.

Bézivin and Gerbé [4]	"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 and 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 [25]	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. [5]	"A model M conforms to a metamodel MM if and only if each model element has its metaelement defined in MM."
Gasević et al. [29]	"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. [50]	"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 and Rusu [22]	"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 Literature

XML Schema recommendation states that Conformance (i.e., validity) checking can be viewed as a multi-step process [59]: 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*" [59]. 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.

The three following major limitations of the conformance relation stand out from these definitions:

(1) The conformance relation is instantiation-based. The relation between a model and its metamodel is set up at the time the model is instantiated. In practice, a model is typically stored in an XML file, with an unchangeable and explicit URI that identifies the metamodel used to create it.

(2) The conformance relation is nominal. In the type system domain, nominal typing refers to a type system that relies on types' names to define explicitly the typing relations [26]. For instance, the Java language has a nominal type system: in "*class A extends B*", the keyword "*extends*" explicitly appoints B as the super type of A. By analogy, the conformance relation in MDE is nominal: a model explicitly refers to its "type" materialized in the URI that points to its metamodel.

(3) A model conforms to one and only one metamodel. This property is a consequence of the two previous ones. Because the conformance relation is instantiation-based, nominal, and only one metamodel is used to create a model, a model conforms to this metamodel only, throughout its lifetime.

These three properties make explicit the metamodel that must be used to manipulate a model, thereby avoiding unsafe manipulations. The immediate drawback is that, by requiring one particular metamodel, it is not possible to use another one (even a close one, e.g., a subsequent version) to manipulate a model. For instance, when a language evolves, the URI of its metamodel is usually updated to materialize the new version. So, models, tools, and transformations defined over the previous version must be subsequently updated, even when the two versions are forward-compatible [40]. Some tools (e.g., ATL [37]) rely on dynamic typing mechanisms and are thus less fragile to evolution. In this case, however, it is not possible to determine *statically* whether a model can be manipulated by a given tool. The gains in terms of flexibility thus occur at the cost of safety.

One can astutely cope with this limitation by sharing the same URI among the versions of the language, even though they actually describe different languages. This, however, implies that language tools cannot determine *a priori* whether they will be able to process a model. This limitation also extends to languages that share commonalities modelers want to handle seamlessly. In other words,

as we shall see in the next section, the strong insurances in terms of safety occur at the cost of flexibility.

2.2. Limits of the Conformance Relation from an Experimental Point of View: the Case of the UML Models on Github

UML is widely used for the object-oriented analysis and design of software systems (e.g., Budgen et al. [9]). Various tool providers provide their own implementation of the different versions of the UML specification. Within the Eclipse ecosystem, the Model Development Tools[4] sub-project (MDT) provides an EMF-based implementation of the UML2 specification, along with other closely related technologies such as the Object Constraint Language[5] (OCL). Each new revision of the UML specifications (every two or three years) leads to new major versions of the MDT-UML2 implementation. On UML metamodel changes, UML models need to be updated subsequently to take the novelties into account – even when changes in the metamodel do not directly impact them. To help modelers migrate their UML models, MDT-UML2 provides migration guides[6] that detail the changes for each new version. As the guides show, new versions usually *add*, *remove*, and *modify* elements, operations, and constraints of the metamodel, consequently breaking parts of the associated API. To cope with these changes, each guide usually describes a migration procedure that UML modelers should follow.

In this section, we present a systematic analysis of UML models hosted on Github. We show that, although the conformance relation prevents it, most UML models can be loaded using different versions of the MDT-UML2 metamodel without having to be explicitly migrated. This requires to manually bypass the nominal typing constraint (i.e., the metamodel URI) imposed by the conformance relation. All the material of the experimentation is available on the companion webpage of the paper[7].

2.2.1. Dataset Collection

We collect an initial dataset of UML models by crawling Github repositories and gathering all the files having either the *uml* file extension, or the *xmi* file extension and containing the term *uml*. This query retrieves a total of 8737 files at the date of 2014-07-07. We first remove: all the duplicates[8]; the models created using another modeling tool such as ArgoUML or Modelio, as they cannot be processed using MDT-UML2. At that point, we obtain 3647 models that specify one version of the MDT-UML2 metamodel as the URI of their metamodel. We then automatically process those models and prune from the

[4] http://www.eclipse.org/modeling/mdt/
[5] http://www.omg.org/spec/OCL/
[6] https://wiki.eclipse.org/MDT/UML2#Guides
[7] http://melange-lang.org/comlan15
[8] Duplicates were mined and removed using the *fdupes* program that relies on full MD5 hashes and byte-to-byte comparison, i.e., syntactically and semantically equivalent models may not be detected as duplicates in this phase

dataset those that cannot be loaded because (i) their XMI serialization is ill-formed or (ii) they exhibit external dependencies towards other metamodels or custom UML profiles that cannot be systematically resolved. Table 2 depicts the results of this selection. A model is considered "loadable" if there exist at least one version of the MDT-UML2 metamodel that can be used to load it. In our experiment, we consider the UML2.2 to 2.5 specifications that correspond to the four major revisions of MDT-UML2 (2.x to 5.x). Besides the UML metamodel, MDT provides a validation framework that defines a set of constraints derived from the UML specifications and implemented as Java code. This validation framework ensures that a given model conforms to its metamodel and the associated constraints (i.e., its static semantics). We systematically run the validation framework on each loaded model and obtain 1651 valid models. In the following, the set of valid models constitutes our base dataset for different analyses.

Extracted	Duplicates	Uniques	MDT-UML2	Valid
8737	2767	5970	3647	1651

Table 2: Initial Dataset Collection

We then identify for each valid model the precise metamodel version to which it conforms, i.e., the one used to create it. This information is directly extracted from their serialized form, as each of them contains an explicit URI referring to its precise metamodel. As depicted in Table 3, they cover all the major revisions of UML we are considering.

Version of UML	2.2	2.3	2.4	2.5	All
Conforming models	183	726	682	60	1651

Table 3: Distribution of the UML Versions of Valid Models

2.2.2. Analysis

We then analyze the resulting models with respect to the four major version of UML implemented in MDT-UML2, namely UML2.2 to 2.5. EMF normally relies on the URI stored in the XMI of a serialized model to determine the metamodel that should be used for loading and manipulating it. As an immediate consequence, it prevents loading the same model using different versions of a metamodel, as the URI is likely to change to reflect the current version of the metamodel (this is the case for UML). In this experiment, we purposely try to load each valid model with each different version of UML, even though EMF would usually prevent it. To do so, we change the URI stored in the models on-the-fly, just before invoking the parser. Then, for the models that are successfully loaded with a given metamodel, we run the validations associated to the latter. Table 4 sums up how many of the 1651 models can be loaded and validated with

each different version. If a model makes use of a feature that does not exist in the metamodel used to load it, the parser may crash. For instance, a model that makes use of a feature introduced in UML2.4 cannot be loaded using the UML2.3 version of the metamodel. Similarly, constraints introduced, modified, or removed from a specific version can influence the result of the validation.

UML Version	2.2	2.3	2.4	2.5
Validated	1502	1497	1502	1460
% of all valid models	90.98	90.67	90.98	88.43

Table 4: Valid Models per Version

Number of compatible versions	1	2	3	4
Number of models	119	127	32	1373
% of all valid models	7.21	7.69	1.94	83.16

Table 5: Compatible Versions per Model

As shown in Table 5, only 7.21% of the UML models are strictly tied to a specific version of their metamodel, while 83.16% of them may be loaded and validated using any version. Overall, as shown in Table 4, each version of the UML metamodel can load around 90% of all the valid UML models, regardless of the original metamodel they conform to. These results can be explained by several factors. First, the EMF parser relies on an XML parser that only traverses the nodes present in the serialized model. This means that any model making use of features that are left unchanged across several metamodel versions can be loaded using any of these versions. Second, the parser creates in-memory model elements that are instances of the meta-classes of the metamodel it was configured to use, regardless of the meta-classes that were used to serialize the model. This means that, for example, an `Association` serialized using UML2.3 may be later parsed using UML2.4, provided that the features of `Association` used in the model did not change in the meantime. This implies that the subsequent validation phase will run just fine, as the manipulated types are the expected ones, thereby avoiding run-time errors. Naturally, validation may fail if the model does not meet the updated set of constraints.

This is similar to a common situation in the programming world where, for instance, the same `.cpp` file may be processed using various versions of a C++ compiler, implementing various versions of the C++ standard. A compiler implementing the C++03 standard may *try* to load a `.cpp` file written in C++11, but will ultimately raise an error if the file makes use of constructs specific to C++11. In this case, the gains in terms of flexibility occur at the price of safety, since it is no more possible to determine *a priori* if the program can be safely manipulated.

Admittedly, models found on Github may not always represent the state of the modeling practice in the industry. However, we find that the models we analyzed have an average size of 102 model elements and that their footprints [35] cover, on average, 40% of the UML metamodel. Moreover, we find that among the 1651 footprints extracted from the valid models, only 382 are unique. This is due to the fact that, for instance, UML models describing a class diagram share the same or similar footprints (the subset of UML necessary for expressing class diagrams). The interested reader can refer to the companion webpage for exploring the experimentation data and results.

2.3. Opportunities of Flexible Modeling Beyond the Conformance Relation

From this experiment, we envision that flexibility and safety in the manipulation of models are two frictional forces that must be constantly balanced. Safe manipulation of a model is highly dependent on the nature of the manipulations one would like to apply to it. Other authors have studied this notion of *usage context* in modeling [40]. While the conformance relation ensures that manipulating a model through its metamodel is always safe, regardless of the context, it hinders flexible modeling by preventing the manipulation of models in other contexts where safety could still be ensured.

From our study of the UML models on Github, we see that flexibility can be drastically improved when the set of expected manipulations (in this case, parsing and validation) can be bounded. Naturally, the method employed in our experiment is not satisfactory, as it ultimately amounts to some kind of "Russian roulette": most of the time, models can be loaded safely, but when it is not the case the parser unpredictably crashes – and the only way to find out is to actually try.

What is missing here is a relation that would state whether a model can be safely manipulated with respect to a set of expected manipulations or not. This relation would prevent unsafe manipulations and ensure that the right level of flexibility can be achieved for a given context. As an illustration, in the UML experiment, the only requirement is the read-only access to the subset of the metamodel that is actually used by the model (i.e., its footprint), hence the substantial reuse opportunities. Conversely, applying a transformation with side effects (e.g., an in-place transformation) to the model would require read-write access to its footprint, thereby strengthening the constraints on the model and restraining the opportunities of flexible manipulation. If the impact of all manipulations cannot be easily determined, the relation must ensure complete access to the entire model, which would further reduce the opportunities of flexibility, as the relation would have to take the whole metamodel into account. Finally, the guarantees that must be ensured through model manipulation may go up to behavioral substitutability, where every desirable property of the manipulating program should be kept [57].

As we are interested in model manipulation, we need to be able to express the minimal contract required to safely manipulate a model in a given context. In MDE, this contract is usually materialized in the metamodel defining the abstract syntax of a DSL. As we have shown, however, metamodels and the

conformance relation are too restrictive to achieve the right level of flexibility. To alleviate these limitations, we propose to explicitly reify the contract imposed by a given context. This contract may be manually written or automatically inferred, for instance by extracting the metamodel footprint of a given model transformation [35]. This relaxes the constraints imposed by the conformance relation by enabling the safe manipulation of models fulfilling the appropriate contract, regardless of the metamodel used to create them.

We propose to reify such contracts as explicit structural interfaces expressed in the form of model types that abstract the possibly arbitrarily complex implementation techniques used to construct a DSL. Section 3 presents the structural interfaces we propose, along with the associated type system that supports flexible modeling through model polymorphism.

3. A Type System for Flexible Modeling

As shown in the previous section, the conformance relation hinders flexible modeling by preventing the manipulation of the same model in different modeling contexts or environments. In this section, we propose a model-oriented type system for flexible modeling that alleviate this limitation by enabling *model polymorphism*, i.e., the possibility to consider and manipulate a model under different forms. This type system relies on *explicit structural interfaces* captured in *model types* [55] that materialize the contract models must fulfill to be manipulated in a given context. Specifically, we show how to ensure *structural substitutability* between models that conform to different languages.

Section 3.1 introduces the necessary concepts and relations for separating implementations of languages from their structural interfaces and expressing such contracts. Section 3.2 presents the different subtyping relations we use to check the substitutability between model types and enable model polymorphism.

3.1. Reifying Model Types as Structural Language Interfaces for Reasoning on Model Polymorphism

The implementation techniques employed to define the different concerns of a DSL (e.g., syntax, semantics) are diverse and arbitrarily complex. For instance, different formalisms can be used to define a metamodel (e.g., an entity-relationship diagram, a class diagram, the MOF formalism) and to define the operational semantics on top of a metamodel (e.g., ATL transformations [37], Kermeta aspects [36], or simply Java code). To ease reasoning on language implementations, we propose to reify the concept of structural language interface and to explicitly separate language implementations from their interfaces. A structural language interface is similar to a metamodel as it exposes a set of concepts and their features and specifies *how* the models matching this interface must be manipulated, i.e., what is the contract they must fulfill. Contrary to metamodels, language interfaces are inherently abstract and cannot be used to instantiate models. We choose to use model types [55] as the formalism for expressing structural language interfaces. The benefit of model types compared

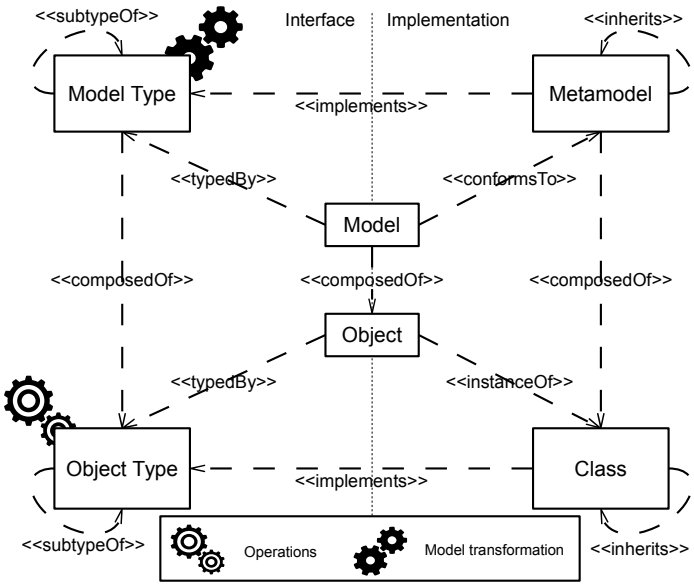
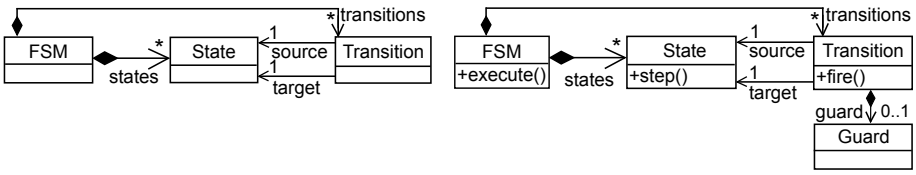


Figure 1: Separating Implementations and Interfaces of Languages

to metamodels is that the same model can be manipulated through various model types thanks to subtyping relations [31]. Explicitly separating interfaces from implementations permits to use interfaces as first-class entities. Therefore, they can be used to explicitly state what is the contract a model must fulfill in order to be manipulated in a given context or environment.

Figure 1 presents an overview of the concepts and relations of the proposed type system and how they seamlessly integrate with the existing modeling concepts. These concepts and relations are detailed hereafter and are explained using an illustrative example consisting of two variants of a finite-state machine (FSM) metamodel (Figure 2): a simple FSM (*Fsm*, Figure 2a) and an executable FSM with simple guards on transitions (*GuardFsm*, Figure 2b). These two variants define the concept of *FSM* composed of *State*s and *Transition*s. A *Transition* has a reference to its *source* and *target* states. Regarding *GuardFsm*, an FSM model can be executed (operations *execute*, *step*, and *fire*) and transitions can declare a *Guard*.



(a) The *Fsm* Metamodel (b) The *GuardFsm* Metamodel

Figure 2: The Metamodels of Two Variants of a Finite-state Machine Language

Object Types. Mainstream object-oriented programming languages provide developers with the concept of explicit interface (e.g., the `interface` keyword in Java). Such explicit interfaces permit the definition of types as contracts, usually consisting of a set of method signatures. This enables instances of a certain class to be manipulated through these interfaces, provided that the class implements the appropriate interfaces. An *object type* is an explicit structural interface: it exposes a subset of the features defined in the implementing class and thus available on its instances. We use the term *object type* instead of *interface* to avoid any confusion with other uses of the term *interface* in this work. We graphically denote the concept of object type using the class representation supplemented with the symbol OT. Figure 3 shows an example of an object type of the *Transition* meta-class from the *GuardFsm* metamodel, that exposes only its *fire* method.

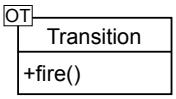


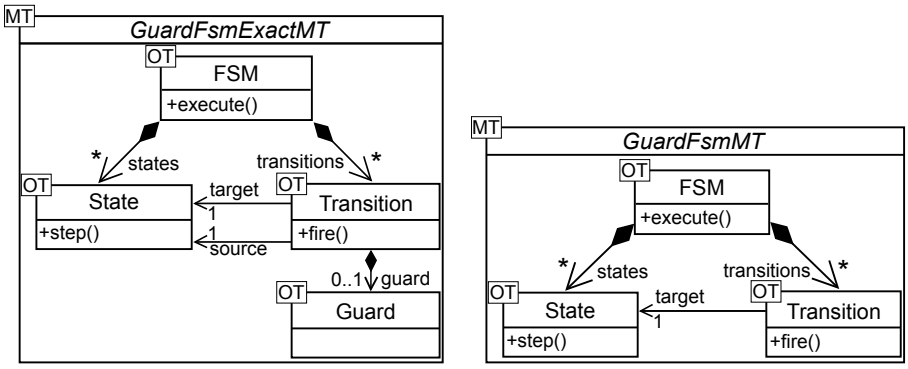
Figure 3: An Object Type of the *Transition* Class

Model Types. A *model type* is an explicit structural interface that defines the contract a model must fulfill to be manipulated through the model type. Model types consists of a set of object types and their relations, and a model can be typed by multiple model types. A model type thus defines a *group of interrelated types*. To avoid unsafe manipulations, the consistency of type groups must be ensured, i.e., types of different groups must not be mixed [23]. Among all the model types of a given model, its *exact model type* contains all the exact object types corresponding to the classes of its metamodel. The exact model type of a model is thus its most precise type. It can be directly extracted from the metamodel used to create the model.
Formally, the exact model type $Ex(MM)$ of models conforming to a metamodel MM is a model type such that $\forall T \in Ex(MM), \exists C \in MM$, and $\forall C \in MM, \exists T \in Ex(MM)$ such that $T = Ex(C)$.

Figure 4 illustrates the concept of model type using the FSM example. We graphically denote the concept of model type using the class representation supplemented with the symbol MT. The structure of a model type is graphically denoted by a class diagram of its object types. Figure 4a depicts the exact model type (named *GuardFsmExactMT*) of *GuardFsm*. *GuardFsmExactMT* exposes all the features of all the object types of *GuardFsm*.

Figure 4b shows an example of a model type (*GuardFsmMT*) of *GuardFsm*. *GuardFsmMT* does not expose all the features of all its object types. The *source* feature of the *Transition* object type and the *Guard* object type are omitted. Therefore, *GuardFsmMT* cannot be considered as the exact model type of *GuardFsm* but only as one of its model type.

Model Typing Relation. We call *model typing relation*, the typing relation that stands between a model and its model types. This typing relation brings



(a) The Exact Model Type of *GuardFsm* (b) A Model Type of *GuardFsm*

Figure 4: Examples of Model Types of the *GuardFsm* Metamodel

flexibility against the standard conformance relation since it allows a model to have several model types. While each model element in a model is instance of only one specific class (defined in its metamodel), it can be typed by multiple object types (defined in different model types). However, because model types form a group of interrelated types, their consistency must be ensured and types of different groups must not be mixed [23]. Consequently, a model m is typed by a model type MT if all the model elements in the model are typed by an object type defined by MT.
Formally, 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 $m : MT$ iff $\forall o \in \mathcal{M}, \exists t \in MT$ such that o is typed by t, where $m \in \mathcal{M}$, and $MT \in \mathcal{MT}$.

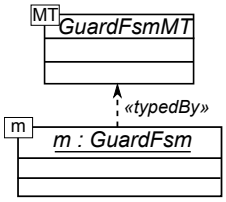


Figure 5: Model Typing Relation Example

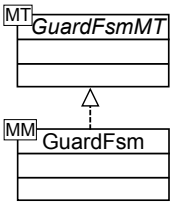


Figure 6: Implementation Relation Example

Figure 5 illustrates the model typing relation using the $GuardFsmMT$ model type depicted in Figure 4a. We graphically denote a model using the standard object instance representation tagged with m. In Figure 5, m conforms to the metamodel $GuardFsm$ and is thus typed by the model type $GuardFsmMT$.

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 conforming to MM can be manipulated through MT.

Formally, a metamodel MM implements a model type MT if for each object type in MT there is a corresponding meta-class in MM implementing it.

Figure 6 illustrates the implementation relation using the model type depicted in Figure 4b. We graphically denote the implementation relation that stands between a metamodel and a model type using the standard implementation representation. Figure 6 depicts a model type *GuardFsmMT* of *GuardFsm*. In this example, we graphically denote the concept of metamodel by tagging a class with the symbol MM. Using the proposed definition, *GuardFsm* implements the model type *GuardFsmMT* since all object types of *GuardFsmMT* have a corresponding implementation (a class) defined in *GuardFsm*.

The two previous definitions (model typing and implementation relations) are independent of different choices in the implementation of a model type checker [31].

3.2. Supporting Flexible Modeling with Model Polymorphism

In the previous section, we introduced the necessary concepts and relations for explicitly separating DSLs implementations from their structural interfaces. We capture these interfaces using model types, which express the contract a model must fulfill to be manipulated in a given context. In order to enable the manipulation of the same model through different interfaces, i.e., model polymorphism, a subtyping relation is needed to state in which cases substitutability is safe. In the remainder of this paper, we focus on the *total isomorphic subtyping* relation introduced by Guy et al. This relation takes the form $\mathcal{MT} \times \mathcal{MT} \rightarrow Boolean$ where $\mathcal{MT}$ is the set of all model types. It states whether models typed by a model type can be substituted safely to models typed by another model type, in any context. This relation, denoted <:, checks subgraph isomorphism between the concepts (i.e., object types) of two model types, ensuring that all the concepts of the super-model type have a matching concept in the sub-model type [31]. The total isomorphic subtyping relation strengthen the model type matching relation proposed by Steel and Jézéquel [55] to prevent model manipulation from instantiating an element without its mandatory properties. Naturally, our approach remains applicable using other subtyping relations, such as the ones introduced by Guy et al. to support adaptation between two model types or partial subtyping. Furthermore, the concepts presented can be applied to support behavioral substitutability, e.g., by taking into account contracts expressed as pre- and post-conditions [57], or simulation relations based on event structures [42]. In this paper, however, we focus on safe *structural* substitutability, and leave the deeper issue of behavioral substitutability to future work.

While the total isomorphic subtyping relation states whether structural substitutability between two model types is safe, regardless of the context, it hinders other scenarios of flexible modeling that arise when considering the context. For instance, as envisioned in Section 2.3, the additional constraint introduced by Guy et al. on elements instantiation can be relaxed when it is known that the manipulations have no side effects on their input models and do not instantiate new elements. To gather such information, one can extract

the static footprint of the transformations she would like to apply [35]. In this case, the contract a model must fulfill in order to be manipulated through these transformations can be reified as a model type corresponding to the footprint. The substitutability between two model types in the context of a transformation t can then be formulated as $MT' <: fp(t, MT)$ where $MT, MT' \in \mathcal{MT}$ and $fp(t, MT)$ is the footprint of the transformation t on MT.

Statically inferring the footprint of a transformation may not always be possible when the transformation is black-box or because of the cost of static analysis; and is imprecise at best. To improve the preciseness of footprints, one can rely on dynamic footprinting [35]. In this case, the transformation is invoked on a model of interest while recording a trace of its execution. Doing so, the footprint is much more precise but the benefits of static checking are lost, and substitutability between model types cannot be checked statically.

4. Melange: Contributing a Type System atop EMF for Flexible Modeling

Melange[9] is an open-source language workbench bundled as a set of Eclipse plug-ins. Melange provides support for executable and aspect-oriented metamodeling, along with operators for DSL assembly and customization [19]. In this section, however, we focus on its support for flexible modeling through the type system introduced in Section 3 and a dedicated mechanism named the *MelangeResource*. We briefly present the syntax of Melange for defining languages and model types (Section 4.1). We then put the emphasis on its support for model polymorphism (Section 4.2) and seamless integration with the EMF ecosystem (Section 4.3).

4.1. Language and Model Type Definition in Melange

Melange relies on various meta-languages for expressing the different concerns that compose a DSL, in particular Ecore [56] (provided by the EMF framework) for defining their abstract syntax in the form of metamodels and Xtend[10] for defining their operational semantics as a set of aspects [36]. The choice of Ecore is motivated by the success of EMF both in the industry and academic areas.

Melange comes with a textual editor that enables the definition of DSLs using a dedicated syntax. The minimal definition of the *GuardFsm* language introduced in Figure 2b is given in Listing 1. The **`syntax`** keyword specifies the Ecore file that defines its abstract syntax. The **`with`** keyword is used to weave the aspects defining its operational semantics. Finally, the **`exactType`** keyword automatically extracts from its implementation its exact model type *GuardFsmMT* which exposes both the features defined in its metamodel and the features woven by aspects. Every language definition in Melange must specify its exact type using the **`exactType`** keyword. The interested reader can refer to

[9] http://melange-lang.org
[10] http://xtend-lang.org/

Degueule et al. [19] for more details on operational semantics definition and aspect weaving in Melange. Listing 2 illustrates the explicit definition of the exact model type of *Fsm* depicted in Figure 2a. The **modeltype** keyword explicitly defines a model type to enable fine-grained control over the exact required set of features for a specific purpose (e.g., opening a model in a specific environment or applying a given transformation). Nonetheless, *FsmMT* and *GuardFsmMT* share the same nature: they are structural language interfaces expressed in the form of a model type.

```
1  language GuardFsm {
2    syntax "GuardFsm.ecore"
3    with ExecutableFsm
4    with ExecutableState
5    with ExecutableTransition
6    exactType GuardFsmMT
7  }
```

Listing 1: The *GuardFsm* Language

```
8  modeltype FsmMT {
9    syntax "FsmMT.ecore"
10 }
11 language CustomFsm
12   implements FsmMT {
13   [...]
14 }
```

Listing 2: The *FsmMT* Model Type

From a Melange specification, such as the one presented in Listings 1 and 2, the type system of Melange automatically infers the subtyping relations among the declared model types and the implementation relations between languages and model types, as specified in Section 3. By default, the model-oriented type system of Melange relies on the total isomorphic subtyping relation introduced by Guy et al. [31]. However, one can easily contribute another subtyping relation to Melange (e.g., the contract-aware subtyping relation of Sun et al. [57]) by implementing a new `match` method[11]. Naturally, every language directly implements its exact model type (e.g., *GuardFsm* implements the *GuardFsmMT* model type). Subtyping relations among model types are also inferred in this phase. For instance, in this case the type checker infers that *GuardFsmMT* is a subtype of *FsmMT* since all the features of the latter are also present in the former. It is worth noting that our implementation of the type system relies on structural typing by default: the type system analyses the structure of metamodels and model types to determine the typing relations, without requiring the user to explicitly specify the typing relations (as with nominal typing). However, users can require that a language implements a specific set of model types using the **implements** keyword (e.g., Line 12 of Listing 2). In this case, the type checker reports an error to the user as long as the language does not implement one of its interfaces. This form of nominal typing enables a simple kind of design-by-contract in the case where the language designer knows, at design time, in which environments models conforming to it should be manipulated. Finally, Melange enables to fix simple structural dissimilarities by allowing users to rename concepts in a metamodel or model type using a

[11] https://github.com/diverse-project/melange/blob/master/plugins/fr.inria.diverse.melange.lib/src/main/java/fr/inria/diverse/melange/lib/MatchingHelper.xtend

`renaming` clause. As we are interested in flexibility between structurally similar languages, we do not detail this mechanism further.

4.2. Implementation of the Support for Model Polymorphism

With the conformance relation, models can only be manipulated through their original metamodel. The core idea of model polymorphism is to supersede the conformance relation with typing relations that enable the manipulation of models through different interfaces materialized by model types. We detail in this section the mechanism we use to enable model polymorphism on top of the EMF framework. EMF is a Java framework, which does not provide any support for type groups polymorphism and structural typing. We detail hereafter how Melange successfully emulates such concepts atop the EMF framework.

For each meta-class in a Ecore metamodel, EMF generates a corresponding Java interface that materializes its features (e.g., an attribute is implemented as a pair of getters/setters). It also generates a Java class implementing the interface where actual values of the persistent features are stored in memory. Therefore, each model element in a model is an instance of the Java class generated from its meta-class. The whole model is a graph of objects that can be manipulated using the generated Java types. Along the generated Java types, EMF also generates a dedicated factory [27], responsible for the creation of new model elements. Any transformation that encompasses the creation of new model elements must use the generated factory.

Melange uses a similar mechanism for materializing model types at the Java level. For each model type, Melange leverages the EMF generator to generate the corresponding Java interfaces. These interfaces are used to manipulate a model in the same way one would manipulate a model through its metamodel. From a user's point of view, there is no difference between manipulating a model through a metamodel or through a model type. The Melange's compiler also generates an abstract factory declaring methods for creating new elements of the model type. Because model types are inherently abstract, no classes implementing them are generated. Instead, concrete implementations are provided by the languages that implement a model type, thereby enabling the manipulation of models written in a given language through the model types it implements, i.e., model polymorphism.

The set of all possible model types implemented by a given language are, however, not known at the creation of a language; typing relations are only inferred *a posteriori* by the structural type checker of Melange. So, there is no direct relation between the Java classes that form a language and the Java interfaces that form a model type. To solve this problem, we employ the *adapter* design pattern [27] to create the implementation relations between classes and interfaces *a posteriori*. The general idea is to generate a set of adapters for each pair $\langle$*language, implemented model type*$\rangle$ (one per object type in the model type). Each adapter implements an object type and delegates the implementation of its features to the corresponding class in the implementing language. Along the adapters for model elements, Melange generates a concrete factory that provides the implementation of the abstract factory of the model type for a given

language. For each creation method in the abstract factory, the concrete factory delegates to the factory of the underlying language and encapsulates the result in an appropriate adapter. Melange uses a generative approach to generate the code of adapters by the time the implementation relation between languages and model types are inferred. The generation of these adapters is safe since the type checker ensures that each object type and feature in a model type has a corresponding implementation in the language.

The generated adapters ensure several properties. First, the graph of all adapters for manipulating a given model through a given model type is built lazily. Only the adapter of the root model element is initially created. Then, adapters of the other elements are built on demand when navigating references from one object to the other. For example, in Figure 4b, *State* adapters are only generated when navigating the *states* reference from *FSM* to *State*. Second, because a model type constitutes a family of types, generated adapters ensure that the semantics of type groups is respected, i.e., elements of different type groups cannot be mixed together. This constraint is inherited from family polymorphism [23] to ensure safe model polymorphism. Adapters also support dynamic dispatching by default. For instance, when a model type exposes an operation in one of its object type, calling this operation on a model will dynamically dispatch to the implementation provided by the actual language of the model. To limit memory overhead, the framework ensures that, for each model element, there is at most one adapter in memory – they are cached and retrieved whenever needed. Finally, adapters support model polymorphism through both direct manipulation and reflective manipulation of models using the reflective API provided by EMF [56]. Supporting polymorphism through the reflective API is especially important as many tools of the EMF ecosystems (e.g., Sirius[12], ATL [37]) rely heavily on reflection.

4.3. Seamless Integration with EMF

Melange aims at providing model polymorphism for EMF-based languages and tools in a seamless and non-intrusive way: the models, their metamodels, and the transformations manipulating them must remain unchanged. To do so, Melange provides a dedicated mechanism, named the *MelangeResource*, that allows to specify in which context (i.e., through which model type) a model must be loaded. The same model can thus be loaded in different environments if its metamodel implements the appropriate model types.

EMF relies on the concepts of *resources* and *resource sets* to load serialized models in memory and save them back as persistent document [56]. A resource represents a persistent model and, when loaded, provides access to the model elements it contains. Resources are created using dedicated *resource factories* responsible for loading a particular kind of models stored in a particular format. Each resource is identified by a unique Unified Resource Identifier (URI) [3] that locates it on the file system (e.g., `file:/path/Model.guardfsm`). When

[12] https://eclipse.org/sirius/

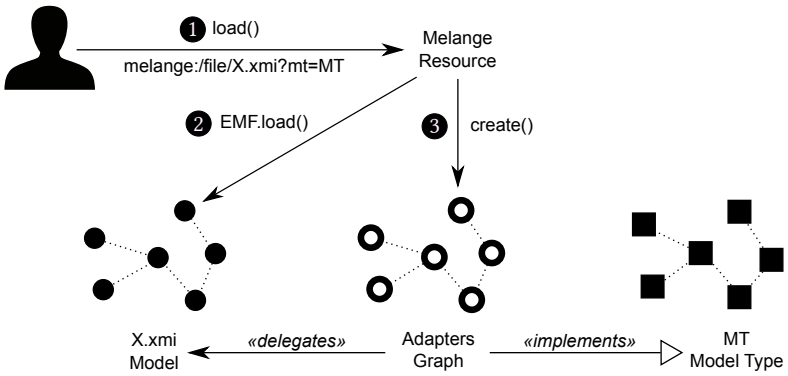


Figure 7: Leveraging the Adapter Pattern to Polymorphically Load a Model as a Specific Model Type

a user requests the loading of a model, the file extension and protocol of its URI are analyzed to determine the appropriate resource factory that must be used to instantiate a new resource representing the model in memory. Then, the resource factory identifies the metamodel of the loaded model (from the nominal reference stored in the serialized model) and uses the metamodel's factory to create the appropriate AST nodes and obtain the graph of objects representing the model in memory.

Melange provides a specialized resource mechanism named the *MelangeResource* to seamlessly support model polymorphism on top of the EMF framework. Melange contributes a new *protocol parser* to EMF that automatically delegates model loading to the *MelangeResource* when the protocol of its URI is `melange` (e.g., `melange:/file/path/Model.guardfsm`). When the user specifies the `melange` protocol the *MelangeResource* is used. In this case, the user can adjoin an additional query string parameter named `mt` to the URI. The `mt` parameter specifies the model type that is *expected*, regardless of the actual metamodel of the model. The *MelangeResource* ensures that the model can safely be loaded through this model type based on the typing relation inferred earlier. Internally, the *MelangeResource* instantiates the appropriate adapters that enable the manipulation of the model as typed by the expected model type. For example, the URI `melange:/file/Model.guardfsm?mt=FsmMT` specifies that the model stored in the `Model.guardfsm` file should be loaded as a (i.e., typed by) *FsmMT* model type. If no implementation relation between the metamodel of the loaded model and the requested model type exists, an error is reported to the user.

The benefit of using a Melange URI is that neither the model nor its metamodel or the transformation code has to be changed. Only the *inputs* of the transformations are modified: the URI of the models to be manipulated. Listing 3 depicts the typical code used to load a model using the EMF framework, with or without the *MelangeResource*. The only visible difference is the URI used to load the model. The model polymorphism mechanism and its runtime support (i.e., the adapters and the specialized resource system) are completely

transparent for the user. In this case, the root of the model (Line 5) is returned as typed by the *FSM* object type defined in the *FsmMT* model type, even though the concrete type of the root of the model is the meta-class *FSM* defined in *GuardFsm*.

```
1 ResourceSet rs = new ResourceSetImpl();
2 //String oldUri = "file:/Model.guardfsm";
3 String uri = "melange:/file/Model.guardfsm?mt=FsmMT";
4 // Requests the model serialized at the given URI
5 Resource res = rs.getResource(URI.createURI(uri), true);
6 // Retrieve the first element of the model (ie. its root)
7 // getContents() is a generic function of EMF,
8 // the cast thus cannot be avoided
9 FSM root = (FSM) res.getContents().get(0);
```

Listing 3: Loading a Model using a Melange URI

5. Experiments

In this section, we evaluate our approach for safe model polymorphism on the two axes of flexible modeling we consider: compatibility between subsequent versions of the same language and interoperability between structurally similar languages. We show that the type system described in Section 3 and implemented in Section 4 within Melange provides safe and seamless model polymorphism for EMF-based languages and tools. Section 5.1 shows how the high level of flexibility envisioned for UML models in Section 2.2 can be achieved with the *MelangeResource*. Section 5.2 shows how model polymorphism supports flexible model loading and manipulation within a family of related DSLs describing variants of finite-state machines.

5.1. Flexible Loading of UML Models

In Section 2.2, we showed that the conformance relation hinders many loading opportunities for UML models: 93% of the models we analyzed could be loaded and validated using at least two versions of the UML metamodel when forcing EMF to bypass the conformance relation. The technique used was however not satisfactory as the parser were unpredictably crashing when a model could not be loaded with a given version of UML.

In this experiment, we aim to achieve the same scores in terms of flexibility while avoiding the "Russian roulette" downside. To this end, we leverage the information we gathered from the kind of manipulations applied to the loaded models. Only the meta-classes corresponding to the model elements they contain are used by the parser. If changes between two versions of the UML metamodel do not affect the subset of the metamodel used in a model, it can be loaded using both of them. Similarly, when invoking the validator, only the elements created by the parser are visited.

In consequence, while the footprint statically computed for the loading mechanism and the validator potentially corresponds to the entire metamodel,

we can restrict this footprint to the meta-classes corresponding to the elements contained in the model (i.e., dynamic footprinting [35]). In other words, the *contract* an UML model must fulfill in order to be safely loaded and validated by a given UML version can be captured in a model type corresponding to the subset of the meta-classes actually required to type the elements it contains.

We use Kompren [7] to generate a pruner [54] for the different versions of the UML metamodel. Then, we visit each model to extract the meta-elements it uses and prune the resulting metamodel with respect to each version of the UML metamodel we consider. As a result, we obtain the effective footprint of each model on each version of UML. The resulting footprint corresponds to the dynamic footprint of the loading and validation phase of UML models. Then, when trying to validate a model with respect to a particular version of UML, we use the subtyping relation introduced in Section 3.2 to find whether the target UML metamodel is a subtype of the computed footprint. When it is the case, the model can be safely loaded and validated; otherwise, the program prevents subsequent crashes and reports an error to the user. The experiment results show that using this technique, we obtain the exact same scores as those envisioned in Section 2.2. All the experimental materials as well as the slicers and the *MelangeResource* can be retrieved from the companion webpage.

5.2. Model Polymorphism for a Family of DSLs

Finite-state machine (FSM) languages are a typical example of DSLs used in a wide range of contexts (e.g., systems and software engineering [32], language processing [48], user interfaces [6]). This leads to a rich diversity of implementations that exposes syntactic and semantic variation points [12]. Syntactic variants comprise flat or composite states, presence of temporal constraints or complex guards, etc. Semantic variants comprise different models of execution, e.g., with or without run-to-completion [30]. DSL designers usually design dedicated environments for each of these variants. Because of the conformance relation, there is a strong coupling between models and the modeling environments used to create them. So, DSL users cannot benefit from tools and transformations (e.g., editors, simulators, or code generators) integrated in different environments.

To solve this problem, a language designer can leverage model polymorphism as promoted in this work: a language designer can design a generic tool or transformation based on a *model type*. The model type makes explicit the required set of features that a model must provide (i.e., the contract it must fulfill) in order to be manipulated by this specific tool. So, any model typed by this model type can then benefit of this tool, regardless of the actual language that was originally used to create it.

In this case study, we show the benefits of model polymorphism on a family of FSM languages. On the syntactic side, the FSM languages differ in their support for composite states, time constraints, both, or none. On the semantic side, they differ on whether they implement a run-to-completion or simultaneous events processing semantics. Altogether, by combining syntactic and semantic variations, our final family comprises eight variants. These languages are representative of

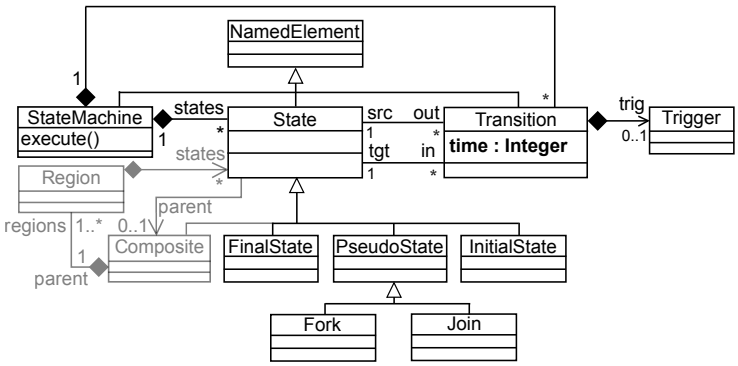


Figure 8: Excerpt of the Metamodels of the FSM Language Variants

the actual variation points that are exposed by popular languages enabling the expression of FSMs such as UML, Rhapsody, and classical statecharts [12].

Figure 8 presents the variations in the abstract syntax of the languages of the FSM family. The gray part (the `Region` and `Composite` classes) is specific to the hierarchical FSM metamodels. The attribute `time` of `Transition`, formatted in bold font, is specific to the timed FSM variants. For the sake of conciseness, the other attributes are not represented.

We define each language in Melange. We create four Ecore metamodels representing the four syntactic variants of the family. The two semantic variants consist of two sets of aspects we wrote in Xtend and that define the operational semantics associated to run-to-completion and simultaneous events processing (see Section 4.1). Listing 4 illustrates the definition of two of the eight variants. Melange takes care of assembling a particular syntax with a particular semantics [19] to produce the expected language. Each language declares its **`exactType`** (Lines 7 and 18), i.e., the precise model type that represents the contract implemented by the models conforming to it.

```
// Flat state machine complying to the
// run-to-completion policy, e.g. UML/Rhapsody
language FlatFsmRtc {
    syntax    "metamodels/FlatFsm.ecore"
    with      rtc.ExecutableStateMachine
    with      rtc.ExecutableState
    exactType FlatFsmRtcMT
}

// Hierarchical state machine complying
// to the simultaneous events processing
// policy, e.g. Classical statecharts
language HierarchicalFsmSimultaneous {
    syntax    "metamodels/HierarchicalFsm.ecore"
    with      simultaneous.ExecutableStateMachine
    with      simultaneous.ExecutableState
    with      simultaneous.ExecutableTransition
    exactType HierarchicalFsmSimultaneousMT
}
```

Listing 4: Two of the eight variants of finite-state machine languages

The type checker of Melange automatically infers the subtyping hierarchy among the exact model types, and the implementation relations between metamodels and model types. Based on the resulting hierarchy, the code generation phase generates the adaptation code supporting model polymorphism between the compatible variants (see Section 4.2).

Then, we implement two typical transformations on FSMs: *execute* checks whether a given sequence of events is recognized by a particular FSM model; *flatten* produces an equivalent FSM model without composite states. The former is defined over the most general model type `FlatFsmRtcMT` and can thus be polymorphically invoked on models conforming to any of the eight variants, taking into account the semantic variations thanks to the dynamic dispatch. The latter is defined over the model type of hierarchical FSMs and can thus be polymorphically invoked on models conforming to the four hierarchical variants.

```
1  // Delegate the execution of the state machine "fsm"
2  // to the "execute" method of its operational semantics.
3  // StateMachine is the root object type of FlatFsmRtcMT
4  public void execute(StateMachine fsm, String input) {
5    // Dynamically dispatched on the actual
6    // language's implementation of execute()
7    root.execute(input);
8  }
9
10 List<String> models = new ArrayList<String>();
11 models.add("melange:/file/Model.flat?mt=FlatFsmRtcMT");
12 models.add("melange:/file/Model.timed?mt=FlatFsmRtcMT");
13 models.add("melange:/file/Model.hiearchical?mt=FlatFsmRtcMT");
14 models.add("melange:/file/Model.timedhierarchical?mt=FlatFsmRtcMT");
15 ResourceSet rs = new ResourceSetImpl();
16
17 // Load the model pointed by the given URI,
18 // retrieve its root StateMachine, and execute it
19 for (String uri : models) {
20   Resource res = rs.getResource(uri, true);
21   StateMachine root = (StateMachine) res.getContents().get(0);
22   execute(res, "{x;y;z;o;p;q}");
23 }
```

Listing 5: Polymorphically Invoking the *execute* Transformation

Listing 5 shows the pseudo-code required for specifying the *execute* transformation in Java and calling it. The `fsm` parameter of the transformation is typed by the root object type `StateMachine` of `FlatFsmRtcMT`, and can thus be polymorphically invoked on models conforming to any of the eight variants. In this case, the execution semantics is directly woven using the aspects depicted in Listing 4: the *execute* methods simply delegates to the appropriate *execute* method of `StateMachine` through dynamic dispatch. However, manipulations in *execute* may be arbitrarily complex and use all the features depicted in Figure 8. The *MelangeResource* is automatically invoked by the use of a `melange:` URI and transparently instantiates the appropriate adapter to enable the manipulation of the different models through the common interface `FlatFsmRtcMT`.

```
1  module FlattenFsm;
2  create OUT : FlatFsm from IN : CompositeFsmMT;
3
4  rule SM2SM {
5    from sm1 : CompositeFsmMT!StateMachine
6    to   sm2 : FlatFsm!StateMachine
7  }
8  -- Initial states of composite states become regular states
9  rule Initial2State {
10   from is1 : CompositeFsmMT!InitialState (
11     not is1.parentState.oclIsUndefined() )
12   to is2 : FlatFsm!State (
13     stateMachine <- is1.stateMachine,
14     name <- is1.name )
15 }
16 -- Resolves a transition originating from a composite state
17 rule T2TB {
18   from t1  : CompositeFsmMT!Transition,
19        src : CompositeFsmMT!CompositeState,
20        trg : CompositeFsmMT!State,
21        c   : CompositeFsmMT!State (
22          t1.source = src and
23          t1.target = trg and
24          c.parentState = src and
25          not trg.oclIsTypeOf(CompositeFsmMT!CompositeState) )
26   to t2 : FlatFsm!Transition (
27     name <- t1.name,
28     stateMachine <- t1.stateMachine,
29     source <- c,
30     target <- trg )
31 }
```

Listing 6: Excerpt of a Generic ATL *flatten* Transformation

Listing 6 depicts an excerpt of the *flatten* transformation written in ATL [37]. Most of the transformation rules are omitted for the sake of conciseness. In this case, the transformation requires as input a model typed by the `CompositeFsmMT` model type and produces a corresponding flattened state machine conforming to the `FlatFsm` metamodel. The *flatten* transformation thus accepts models conforming to any of the four hierarchical variants. One can write an ATL transformation on a model type in the exact same way than on a metamodel. When invoking the transformation the use of a Melange URI automatically invokes the *MelangeResource* so that the ATL engine transparently manipulates the models through the appropriate adapters. Making an ATL transformation generic only requires to change the type of its input models to the appropriate model type.

5.3. Discussion

Through the two presented experiments, we illustrate the benefits of safe model polymorphism on the two axes of flexible modeling we consider: compatibility between subsequent versions of a same language; interoperability between structurally similar languages. Our framework and its implementation in Melange through the *MelangeResource* allows to state in which cases a model can be safely manipulated, without requiring any work from the language designers or users. As mentioned in Section 3.2, our framework is parameterized by a particular subtyping relation between model types. In our experiments, we use the total isomorphic subtyping relation introduced by Guy et al. [31]. As a result, our experiments only focus on structural substitutability and do not consider the harder problem of behavioral substitutability. For instance, while the *MelangeResource* ensures that the *flatten* transformation presented in Section 5.2 can safely be applied on a model conforming to different languages, it cannot state whether its behavioral properties will be preserved (e.g., are the resulting state machine models flat and semantically equivalent to the inputs models?). Other experiments involving augmented model types on which behavioral contracts are expressed in the form of invariants, pre-, and post-conditions to assess property preservation through the use of the contract-aware subtyping relation introduced by Sun et al. [57] are left for future work. Finally, we envision that the overhead in terms of time and memory consumption implied by our generative approach relying on adapters is highly dependent on the size of the considered models and metamodels. A deeper analysis of the cost of our approach is left for future work.

6. Related Work

6.1. Model Transformation Reuse

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 and Pataricza who introduced *variable entities* in patterns for declarative transformation rules [58]. These entities only express the concepts (types, attributes, etc.) required to apply the rule. This allows tokens with these concepts to match the pattern and be processed by the rule. Later, Cuccuru et al. introduced the notion of semantic variation points in metamodels [14]. 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. Such patterns can be viewed as model types whose variability has to be explicitly expressed.

Sánchez Cuadrado and García Molina propose a notion of substitutability based on model typing and model type matching [52]. Instead of using an automatic algorithm to check the matching between two model types, they propose a DSL to manually declare the matching.

In this paper, we go further than these works by enabling the full automation of the reuse process between structurally similar languages, when no adaptation is required. When the subtyping relation states that two languages match each others, adapters are automatically generated and transparently used through the use of the *MelangeResource*. Unlike other approaches, neither the models, the metamodels, nor the transformations rules have to be changed.

6.1.2. Reuse through Adaptation

Adaptation enables 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.

De Lara and Guerra present the notion of *concept*, along with *model templates* and *mixin layers* [17, 49]. These notions are borrowed from the idea of generic programming, as found in mainstream programming languages (e.g., C++, Java). *Concepts* are similar to model types as they define the requirements that a metamodel must fulfill for its models to be processed by a transformation. However, in their current form, *concepts* do not benefit from subtyping relations between different concepts, and a metamodel must be explicitly bound to each *concept* of the expected transformations. 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*. Cuadrado et al. extend this binding mechanism to go further than strict structural mapping by renaming, mapping, and filtering metamodel elements [13]. *Concepts* are inspired from parametric polymorphism (aka. generics) while model types are inspired from inclusion polymorphism (aka. subtype polymorphism). Parametric and inclusion polymorphism serve similar purposes and are known to be complementary [10]. In our approach, the transformation of interest does not need to be changed, but the generation of adapters and their use at runtime implies an inevitable overhead.

De Lara et al. introduce *a-posteriori typing* for MDE, which allows to uncouple the creation type of a model from its classification types and to reclassify models after their creation to enable their manipulation in other contexts [18]. A-posteriori typing goes beyond our approach by enabling instance-level classification, which inherits the benefits and drawbacks of dynamic typing. However, their implementation relies on the MetaDepth tool, while our approach can be seamlessly used in any EMF-based tool.

Kerboeuf and Babau present an adaptation DSL named *Modif* which handles deletion of elements from a model (that conforms to a metamodel MM) to make it substitutable to an instance of the metamodel MM' [38]. 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 and Díaz proposed to semi-automatically adapt a transformation with respect to metamodel changes [28]. A classification of metamodel evolutions is proposed as well as automatic adaptations of the transformation for some of them. These approaches go beyond our scope by supporting complex metamodel changes. In the case of structurally similar languages, however, our approach is much more practical.

As illustrated with the different UML versions (see Section 2.2), metamodels evolve and their models have to be migrated. Di Ruscio et al. proposes an approach for supporting the co-evolution of metamodels and their models [20]. Model migration is supported by manually-written adaptations. This principle can notably be used to chain model transformations that use different metamodels [2]. Because of the expressive power of the adaptations, this approach is theoretically more expressive than our proposal. The adaptations, however, are manually written while the core idea of our proposal is to fully automate the adaptation when the languages are close enough.

These approaches permit to go further than reuse between isomorphic metamodels. Our framework Melange only supports renaming (when the same concept is given two different names in two languages) and does not address more complex scenarios.

6.2. Advanced Typing System in Object-oriented Programming

Object-oriented type systems garner a considerable interest in the last decades in providing advanced typing mechanisms for programming languages. This section discusses the relation between Melange and seminal work on typing in object-oriented programming (OOP).

Nominal typing relies on types' name to explicitly define their typing relation. By analogy, current MDE technologies are based on nominal typing where a model is bound to an unique DSL through its URI. Conversely, structural typing relies on the structure of types to define typing relations. Structural subtyping is useful to bind two independent type hierarchies having similar operations. If a nominal type system prevents to bind two independent type hierarchies, a classical solution is to use the *Adapter* design pattern [27] to group them under the same hierarchy. However, this pattern involves a substantial development effort. Moreover, the scope of structural typing in current OOP languages is limited to the class level. Our implementation of the *MelangeResource* aims at providing the benefits of structural typing at the type groups level, while ensuring their consistency to avoid unsafe manipulations [23]. To ease the developer's work, Melange automatically generates the set of adapters required to emulate such behavior.

Several advanced typing mechanisms have been proposed to enhance reuse capabilities in OO programming languages. Scala [45] and gBeta [34] propose to support family polymorphism through path-dependent types. Nystrom et al. [44] introduce the concept of nested inheritance, a mechanism that addresses several limitations of the standard inheritance and code reuse mechanisms. Lämmel and Ostermann [41] demonstrate how the use of type classes can simplify the

extension and integration of legacy code. Besides reuse, Diekmann and Tratt propose an approach to compose language units [21]. Similarly, object algebras demonstrate how to solve the expression problem [15, 16]. 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. 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 [8]. 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.

As our approach aims at providing model polymorphism atop the legacy EMF ecosystem implemented in Java, we cannot leverage advanced mechanisms such as path-dependent types. Instead, we provide our own mechanism based on the automatic generation of adapters. In future work, we plan to investigate the trade-offs between different implementation techniques, beyond adapters. The scope of these related works is limited to the class level. In this work instead, we focus on the definition of languages as a group of interrelated types. This implies the definition of typing relations between languages as introduced in Section 3.

7. Conclusion and Future Work

In this work we propose to overcome the limitations of the conformance relation that hinder the flexibility DSL users would expect between structurally similar DSLs. We describe a model-oriented type system that provides a safe mechanism of polymorphism for models. This type system is integrated into Melange, an open-source and freely available language workbench seamlessly integrated with the EMF ecosystem. Through an experiment on UML models gathered from Github and a case study on a family of syntactically and semantically variant FSM languages, we show the benefits of safe model polymorphism on the two axes of flexible modeling we consider: compatibility between subsequent versions of the same language and interoperability between structurally similar languages.

In future work, we will investigate the use of model types and the proposed type system as a support for defining explicit viewpoints on top of complex languages and models. In this context, model types would be used to filter the appropriate information for supporting a specific stakeholder in a particular task on the system under development. Finally, we plan to extend our work to investigate the support of behavioral substitutability, and increase the level of confidence when reusing model transformations.

Acknowledgement

This work is partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n611337 (HEADS

project), the ANR INS Project GEMOC (ANR-12-INSE-0011), the ITEA2 project MERgE, and the French LEOC project Clarity. We thank the anonymous reviewers for their insightful comments, which helped us improve the manuscript.

References

[1] C. Atkinson and T. Kühne. Profiles in a Strict Metamodeling Framework. *Science of Computer Programming (SCP)*, 44(1):5–22, 2002.

[2] F. Basciani, D. Di Ruscio, L. Iovino, and A. Pierantonio. Automated Chaining of Model Transformations with Incompatible Metamodels. In *Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS'14)*, pages 602–618. Springer, 2014.

[3] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform resource identifiers (URI): Generic Syntax. https://www.ietf.org/rfc/rfc2396.txt, 1998.

[4] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, pages 273–280, 2001.

[5] J. Bézivin, F. Jouault, and D. Touzet. Principles, Standards and Tools for Model Engineering. In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 28–29, 2005.

[6] A. Blouin and O. Beaudoux. Improving Modularity and Usability of Interactive Systems with Malai. In *Proceedings of the 2nd Symposium on Engineering interactive computing systems (EICS'10)*, pages 115–124, 2010.

[7] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and Generating Model Slicers. *Software and Systems Modeling (SoSyM)*, pages 1–17, 2012.

[8] G. Bracha. Pluggable Type Systems. In *Proceedings of the International Workshop on Revival of Dynamic Languages at OOPSLA'04*, 2004.

[9] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical Evidence about the UML: a Systematic Literature Review. *Software: Practice and Experience (SPE)*, 41(4):363–392, 2011.

[10] L. Cardelli and P. Wegner. On nderstanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[11] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-driven Engineering. In *Proceedings of the 12th International Enterprise Distributed Object Computing Conference (EDOC'08)*, pages 222–231, 2008.

[12] M. L. Crane and J. Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS'05)*, pages 97–112. 2005.

[13] J. S. Cuadrado, E. Guerra, and J. de Lara. A Component Model for Model Transformations. *IEEE Transactions on Software Engineering*, 40 (11):1042–1060, 2014.

[14] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard. Templatable Metamodels for Semantic Variation Points. In *Proceedings of the 3rd European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA'07)*, pages 68–82, 2007.

[15] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the Masses – Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*, pages 2–27. Springer, 2012.

[16] B. C. d. S. Oliveira, T. Storm, A. Loh, and W. R. Cook. Feature-Oriented Programming with Object Algebras. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*, pages 27–51. Springer, 2013.

[17] J. De Lara and E. Guerra. Generic Meta-modelling with Concepts, Templates and Mixin Layers. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, pages 16–30, 2010.

[18] J. De Lara, E. Guerra, and J. Sánchez Cuadrado. A-posteriori Typing for Model-Driven Engineering. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 156–165, 2015.

[19] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, pages 25–36, 2015.

[20] D. Di Ruscio, L. Iovino, and A. Pierantonio. Coupled Evolution in Model-driven Engineering. *IEEE Software*, 29(6):78–84, 2012.

[21] L. Diekmann and L. Tratt. Eco: a Language Composition Editor. In *Proceedings of the 7th International Conference on Software Language Engineering (SLE'14)*, pages 82–101. 2014.

[22] M. Egea and V. Rusu. Formal Executable Semantics for Conformance in the MDE Framework. *Innovations in Systems and Software Engineering*, 6 (1-2):73–81, 2010.

[23] E. Ernst. Family Polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 303–326. 2001.

[24] M. Eysholdt and H. Behrens. Xtext: Implement your Language Faster than the Quick and Dirty Way. In *Proceedings of the International Conference on Object-Oriented Programming Systems Languages and Applications Companion (OOPSLA'10 Companion)*, pages 307–309. ACM, 2010.

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

[26] R. B. Findler, M. Flatt, and M. Felleisen. Semantic Casts: Contracts and Structural Subtyping in a Nominal World. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, pages 365–389. Springer, 2004.

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

[28] J. Garcia and O. Díaz. Adaptation of Transformations to Metamodel Changes. In *Desarollo de Software Dirigido por Modelos*, pages 1–9, 2010.

[29] D. Gasević, N. Kaviani, and M. Hatala. On Metamodeling in Megamodels. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, pages 91–105, 2007.

[30] A. Gill et al. *Introduction to the Theory of Finite-state Machines*, volume 16. McGraw-Hill New York, 1962.

[31] C. Guy, B. Combemale, S. Derrien, J. R. Steel, and J.-M. Jézéquel. On Model Subtyping. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA'12)*, pages 400–415. 2012.

[32] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming (SCP)*, 8(3):231–274, 1987.

[33] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 471–480, 2011.

[34] A. Igarashi, C. Saito, and M. Viroli. Lightweight Family Polymorphism. In *Programming Languages and Systems (TOPLAS)*, pages 161–177. 2005.

[35] C. Jeanneret, M. Glinz, and B. Baudry. Estimating Footprints of Model Operations. In *Proceedings of the 33rd Internal Conference on Software Engineering (ICSE'11)*, 2011.

[36] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. *Software and Systems Modeling (SoSyM)*, 14(2):905–920, 2015.

[37] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming (SCP)*, 72(1):31–39, 2008.

[38] M. Kerboeuf and J.-P. Babau. A DSML for Reversible Transformations. In *Proceedings of the 11th Workshop on Domain-Specific Modeling (DSM'11)*, pages 1–6, 2011.

[39] D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Transformation Language. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT'08)*, pages 46–60. Springer, 2008.

[40] T. Kühne. On Model Compatibility with Referees and Contexts. *Software and Systems Modeling (SoSyM)*, 12(3):475–488, 2013.

[41] R. Lämmel and K. Ostermann. Software Extension and Integration with Type Classes. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 161–170, 2006.

[42] F. Latombe, X. Crégut, B. Combemale, J. Deantoni, and M. Pantel. Weaving Concurrency in Executable Domain-specific Modeling Languages. In *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, pages 125–136. ACM, 2015.

[43] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)*, 37(4): 316–344, 2005.

[44] N. Nystrom, S. Chong, and A. C. Myers. Scalable Extensibility via Nested Inheritance. In *Proceedings of the 19th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'04)*, volume 39, pages 99–115, New York, NY, USA, Oct. 2004. ACM.

[45] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, pages 201–224. 2003.

[46] OMG. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/spec/MOF/2.0/, 2006.

[47] OMG. Unified Modeling Language (UML), Infrastructure Specification. http://www.omg.org/spec/UML/, 2011.

[48] E. Roche and Y. Schabes. *Finite-state Language Processing*. MIT press, 1997.

[49] L. Rose, E. Guerra, J. De Lara, A. Etien, D. Kolovos, and R. Paige. Genericity for model management operations. *Software & Systems Modeling*, 12(1):201–219, 2013.

[50] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model Migration with Epsilon Flock. In *Proceedings of the 4th International Conference on Model Transformation (ICMT'10)*, pages 184–198, 2010.

[51] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[52] J. Sánchez Cuadrado and J. García Molina. Approaches for Model Transformation Reuse: Factorization and Composition. In *Proceedings of the 1st International Conference on Model Transformations (ICMT'08)*, pages 168–182, 2008.

[53] D. C. Schmidt. Model-driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[54] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model Pruning. In *Proceedings of the 12th Internal Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, 2009.

[55] J. Steel and J. M. Jézéquel. On Model Typing. *Software and Systems Modeling (SoSyM)*, 6(4):401–413, 2007.

[56] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[57] W. Sun, B. Combemale, S. Derrien, and R. B. France. Using Model Types to Support Contract-aware Model Substitutability. In *Proceedings of the 9th European Conference on Modelling Fundations and Applications (ECMFA'13)*, pages 118–133. 2013.

[58] D. Varró and A. Pataricza. Generic and Meta-transformations for Model Transformation Engineering. In *Proceedings of the 7th International Conference on UML Modelling Languages and Applications (UML'04)*, pages 290–304, 2004.

[59] W3C. XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, may 2001.

[60] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, pages 600–624. 2007.