Structural Model Subtyping with OCL Constraints

Artur Boronat
Department of Informatics
University of Leicester
United Kingdom
aboronat@le.ac.uk

Abstract

In model-driven engineering (MDE), models abstract the relevant features of software artefacts and model management operations, including model transformations, act on them automating large tasks of the development process. Flexible reuse of such operations is an important factor to improve productivity when developing and maintaining MDE solutions. In this work, we revisit the traditional notion of object subtyping based on subsumption, discarded by other approaches to model subtyping. We refine a type system for object-oriented programming, with multiple inheritance, to support model types in order to analyse its advantages and limitations with respect to reuse in MDE. Specifically, we extend type expressions with referential constraints and with OCL constraints. Our approach has been validated with a tool that extracts model types from (EMF) metamodels, paired with their OCL constraints, automatically and that exploits the extended subtyping relation to reuse model management operations. We show that structural model subtyping is expressive enough to support variants of model subtyping, including multiple, partial and dynamic model subtyping. The tool has received the ACM badge "Artifacts Evaluated – Functional".

CCS Concepts • **Software and its engineering → Model-driven software engineering**; **Formal language definitions**; **Formal software verification**;

Keywords Model subtyping, EMF, OCL, type theory.

ACM Reference Format:
Artur Boronat. 2017. Structural Model Subtyping with OCL Constraints. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136026

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136026

1 Introduction

Our aim in this work is to revisit the research question of whether type subsumption – i.e. the relation capturing that any inhabitant of a given subtype is also an inhabitant of a given supertype – is a valid mechanism for facilitating reuse of model management operations in MDE in order to analyse its advantages and limitations. We approach this topic by exposing a general problem involving reuse, which we then solve by using structural model subtyping.

In a typed setting, model management operations are applied to models that conform to metamodels, which define the abstract syntax of a modeling language. Additional well-formedness constraints can be added to the language, usually by encoding them in an OCL dialect. The notion of a pair $(\mathcal{M}, \Omega)$ formed by a metamodel $\mathcal{M}$ and its (OCL) constraints Ω is captured as a metamodel specification [4]. A very general form of reuse emerges when we want to learn whether we can extrapolate model management operations from one metamodel specification to another one. This involves learning whether two metamodel specifications, whose metamodels and constraints need not be related a priori, are *compatible* capturing the notion of subtype polymorphism in model management operations. This is a frequent problem in the evolution of modeling languages, where updates to their abstract syntax and constraints may need to preserve forward/backward compatibility of operations.

Depending on how typing is considered [24], we can distinguish semantics where typing (the *instanceOf* relation) is ontological, explicitly definable in the metamodeling language, or linguistic, implicitly defined by the metamodeling language. In addition, we consider that the semantics of object subtyping corresponds to (static) subclassing – generalization – in the first appproach, and to subsumption, in the second approach. Hence, we distinguish approaches where models are represented as graphs, where typing is ontological; or as terms, where typing is linguistic. On the one hand, graph transformation theory and well-known (meta)modeling environments, such as the Eclipse Modeling Framework (EMF) and the USE environment [18, 23], rely on the first representation, exploiting a set-theoretic representation to implement tools, e.g. for checking model typing and for analysing (OCL) constraints. On the other hand, type-theoretic approaches, such as [10, 26], rely on the second representation for exploiting inductive reasoning and higher-order functions.

To solve the aforementioned problem, the following results have been developed in this work:
(1) Extension of OO types, with multiple inheritance semantics [8], to model types with referential constraints, reusing a type-theoretic framework, with *linguistic typing* and a convenient semantics for subtyping. Our model types formalize the common features of EMF metamodels, including classes with attributes, cross-references and containments, multiple inheritance, data types and enumeration types, multiplicity, ordering, uniqueness and bidirectionality constraints for cross-references and containments.
(2) Attachment of the USE semantics of OCL constraints [27], with *ontological typing*, to extended OO types in order to reuse tool support for mechanizing the satisfaction of OCL constraints in our model types.
(3) Definition of a subtyping relation between metamodel specifications that couples extended OO subtyping, between the types in both metamodels, with the compatibility of their OCL constraints. Such a subtyping relation can be regarded as a structural refinement relation between metamodel specifications. The computation of the subtyping relation for two given metamodel specifications relies on the construction of an extension metamodel that reflects (linguistic) model subtyping using (ontological) subtyping, enabling reasoning about the compability of OCL constraints from both metamodel specifications. The extension metamodel also serves as the basis for reusing model management operations *as-is*, i.e. without introducing changes to the operation behaviour.
(4) A semi-decidable procedure that solves the proposed problem by using structural model subtyping and its implementation in a tool, facilitating flexibility in MDE by supporting dynamic, partial and multiple typing. The tool implementing this procedure has received the *ACM badge "Artifacts Evaluated - Functional"* and is available, with case studies, at https://mde-subtyping.github.io/web/.

In the following subsections, we introduce an example and discuss approaches for model typing, focussing on accepted notions of model subtyping for EMF metamodels and for graph transformation theory. In section 4, a syntactic representation for model types and structural model subtyping are introduced. In section 5, we map a semantics of metamodels specifications to our model types and extend the subtyping relation to metamodel specifications. In section 6, we present how the tool provides support for the theory developed. In section 7, we compare our approach to related work, finalizing the work with concluding remarks.

2 Example

As a running example, we are using the metamodel specifications $(\mathcal{M}_g, \Omega_g)$ and $(\mathcal{M}_{sm}, \Omega_{sm})$ for defining graphs and deterministic state machines, resp., shown in Figure 1. The model types described by both metamodels are structurally similar in that they both describe languages of graphs. On

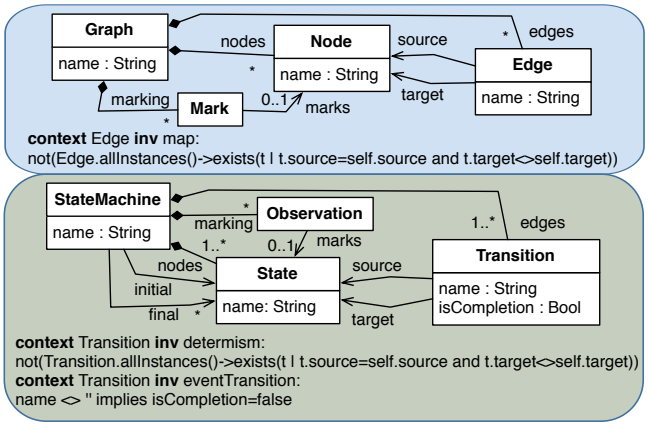

Figure 1. $(\mathcal{M}_g, \Omega_g)$ – top – and $(\mathcal{M}_{sm}, \Omega_{sm})$ – bottom

the one hand, $(\mathcal{M}_g, \Omega_g)$ characterizes the graph of a function defined over nodes. On the other hand, $(\mathcal{M}_{sm}, \Omega_{sm})$ characterizes deterministic state machines where transitions can be triggered by an event (indicated in the `name` attribute of the transition) or are triggerless, e.g. they are completion transitions. These metamodels provide a mechanism to mark nodes, and states, as already mapped, or as already observed.

We can envisage an operation $(\mathcal{M}_g, \Omega_g) \rightarrow (\mathcal{M}_g, \Omega_g)$ for applying a map in $(\mathcal{M}_g, \Omega_g)$ to a particular node by marking it. Specifically, this operation has been implemented as an ATL model transformation for $(\mathcal{M}_g, \Omega_g)$ that creates an object `Mark` for an unmarked node whose predecessor is marked. Our approach enables the reuse of that operation for simulating state machines conforming to $(\mathcal{M}_{sm}, \Omega_{sm})$.

Our structural subtyping mechanism has also been validated with the example used for model subtyping in [31] and with a second case study on the evolution of modeling languages. These case studies have been implemented and are documented at https://mde-subtyping.github.io/web/.

3 Approaches to Model Subtyping

In the recent literature on model transformation, there is an emerging interest in formalizing mechanisms for reuse of model transformations [9, 19, 25]. Approaches to reuse transformation logic involve mechanisms to facilitate its application in different contexts (by means of typing) or by extending the logic itself (by means of transformation rule extension). We are going to focus on the first one, while laying the foundations for the second one, in order to study when it is safe to reuse a model management operation.

Regarding typing, working with models in MDE processes can be done at two levels of abstraction: a white-box approach (called intra-resource in [34]) where model internals are exposed so that side effects can be analyzed as model transformations act upon models; and a more coarse (black-box) approach (called inter-resource in [34]) where models

are treated as first-order citizens in model management scenarios [6]. Depending on the desired level of rigour, the former may be regarded as a building block to enable a sound basis for the latter, i.e. by building model management systems based on safe model transformations. Refining this classification around the notion of intra-resource typing, we find approaches that build on a notion of model type, typically considering model subtyping or metamodel adaptation, out of which we are interested in subtyping only.

Model subtyping can be dealt with as a subsumption relation or as model type matching, by generalizing the homologous notions in OO programming languages [2]. Steel et al. [31] proposed a type system with type groups in order to formalize the type of a metamodel. Substitutability in this type system is facilitated by a model type matching relation, which generalizes the matching relation on type groups presented in [7] to model types, where a model type `M'` matches a model type `M`, denoted `M' <# M`, iff for each class `C` in `M`, there is a class `C` in `M'` such that the signature of its operations is preserved[1]. The type system proposed relies on class names to match object types, which implies that the model type for state machines of Fig. 1 would not be considered a subtype of the model type for graphs without an explicit adaptation.

In [31], the authors discuss that a name-independent structural type system is not able to capture classes with no contents that are solely used for adding structure to a concept taxonomy, e.g. by denoting final states in a state machine. From a design point of view, the use of empty classes may be helpful for modeling new concepts at an early stage in the development process. However, for implementation purposes, this may be considered a design smell, as it violates the single-responsibility principle. In addition, there are alternative modeling techniques that permit skipping that case without losing expressive power. For example, one may consider clustering a hierarchy of empty children classes into the superclass by adding a segregative attribute (e.g. a boolean attribute `isFinal`) or by designating different types of classes by using references (e.g. by using a reference `finalStates` from the class `StateMachine` to the class `State`).

Guy et al. improved this notion of subtyping as isomorphic model subtyping in [19] and introduced non-isomorphic model subtyping for enabling model adaptation by means of renaming maps. Guy et al. [19] also discuss the notion of partial and total subtyping in order to facilitate reuse of model transformations in practical scenarios. Partial model subtyping aims at enabling the safe reuse of a model transformation even if only the part of the model type that is used in the model transformation is present. In this context, Sen et al. [29, 30] conceptualized this situation as the notion of effective model type of a transformation: the minimal subset of the elements of the input metamodel that is used in the transformation.

Model transformations based on graph transformation theory rely on the theory of typed attributed graphs with type node inheritance [12, 16]. Type checking in this theory is achieved by constructing a graph morphism between a graph (the model) and the type graph (the metamodel) that preserves the structure of the graph. Model subtyping is supported in graph transformations by means of the notion of abstract production rule where nodes in a graph pattern in the rule may correspond to abstract nodes (similar to an abstract superclass). From a theoretical point of view, given a graph and an abstract production rule that can be applied to it, it has been shown that a unique concrete production rule can be constructed so that the effects of the transformation on the graph are equivalent to the application of abstract production rule directly on the graph. This means that the usual theory for typed attributed graph transformation can be applied for graph grammars with abstract production rules, with a notion of object subtyping. Using this theory, the type graphs representing the metamodels in Figure 1 denote different types of graphs that are not related unless adaptation mechanisms are used explicitly.

4 Model Types and Subtyping

One of the contributions of this work consists in giving an interpretation to metamodels (and object-oriented models), where refinement extension is achieved by (static) subclassing, with model types where refinement is achieved by subtyping (subsumption). To distinguish both kinds of type representations we are going to use $\mathcal{M}^{\rightarrow}$ to denote metamodels and $\mathcal{M}^{<}$ to denote model type expressions. This distinction is necessary in order to consider OCL constraints in model types, as discussed in the following section.

In this section, we first introduce model type expressions $\mathcal{M}^{<}$ for representing model types $[\![\mathcal{M}^{<}]\!]$ and, subsequently, define them. Then, we define a notion of model subtyping based on subsumption. In section 5, we show how a metamodel $\mathcal{M}^{\rightarrow}$ is related to its model type expression $\mathcal{M}^{<}$.

4.1 Model Type Expressions

Models in software engineering have a dual interpretation, namely as "a related collection of instances of metaobjects, representing (describing or prescribing) an information system, or parts thereof, such as a software product or as semantics", or as "a semantically closed abstraction of a system or a complete description of a system from a particular perspective" [1]. That is, as syntax or as semantics [20]. Since metamodels are also models, we differentiate a metamodel from a model type by saying that a metamodel denotes a unique model type [3, 5].

Our model types are used as classifiers of objects and their operations are applied to models, even if they act on

[1] Noting that properties are encoded as pairs of generator-mutator methods in their approach.

the internals of the model representation. For example, a model transformation may consist of a sequence of updates of property values or alter the structure of the model.

In [8], record types are used to model classes of objects, and thus object types, and union types define tagged values. That is, whereas an record type `(name: String)` denotes objects with a property `name`, a union type `[red : [], blue: [], green: []]` defines the type of those values that contain a single literal, defining an enumeration type. In the following, we extend the syntax for types presented in [8], by including referential types and by including notation both for the types themselves and for their name. The reason for this is to enable the definition of recursive structures through referential types.

Definition 4.1. (Extended syntax for types) Given the countable sets $\mathcal{B} = \{Oid, Bool, String, Int, Float\}$ of base type names b, $\mathcal{E}$ of enumeration type names ϵ, $\mathcal{C}$ of class names c, $\mathcal{P}$ of property names p, the set τ of types over $\mathcal{B}$, $\mathcal{E}$, $\mathcal{C}$ and $\mathcal{P}$, is defined as follows

$$\begin{aligned}
\textbf{Type} \ni \tau &::= \alpha \mid \varsigma \mid \mu \mid \tau \rightarrow \tau \\
\textbf{TName} \ni \alpha &::= b \mid \epsilon \mid \texttt{Void} \mid \texttt{Any} \mid \rho \\
\textbf{RefTName} \ni \rho &::= (ref)?(unique)?(ordered)?\ c[n..m](\#p)? \\
\textbf{ObjectType} \ni \varsigma &::= (\varrho) \mid () \\
\textbf{UnionType} \ni \mu &::= [\varrho] \mid [] \\
\textbf{PropSetType} \ni \varrho &::= p : \alpha \mid \varrho, \varrho
\end{aligned}$$

where $n \in \mathbf{N}$ and $m \in \mathbf{N} \cup \{*\}$.

[] denotes the bottom type as the least informative type and `Void` denotes its name, () denotes the most general type and `Any` denotes its name, and $\tau \rightarrow \tau$ denotes the type of a function. Optional sentential forms are denoted with grouping parentheses and a question mark.

We will use the boolean predicates $isRef(\rho)$, $isCmt(\rho)$, $isOrdered(\rho)$ and $isUnique(\rho)$ to check whether a property type expression is a cross reference type expression or a containment reference type, whether it is ordered and whether it is unique, respectively. In addition, the projections $c(\rho)$, $lower(\rho)$, $upper(\rho)$, and $op(\rho)$ obtain the class name, the lower bound, the upper bound and the opposite property name (when present), respectively.

$p_1 : \alpha_1, \ldots, p_n : \alpha_n$ corresponds to a set of structural features, where attributes are defined as properties of the form $p : b$ or $p : \epsilon$, cross-references pointing to a class c are defined as properties of the form

$$p : ref\ (unique)?\ (ordered)?\ c[n..m](\#p)?$$

and containments pointing to a class c are defined as properties of the form $p : (unique)?\ (ordered)?\ c[n..m](\#p)?$.

Definition 4.2. (Model Type Expression) Given finite sets $\mathcal{B}$ of base type names b, $\mathcal{C}$ of class names c, $\mathcal{E}$ of enumeration names ϵ and $\mathcal{P}$ of property names p, a model type expression $\mathcal{M}^<$ is defined as a tuple $(type, roots)$ where: $type$ is an injective function mapping each class name c to a corresponding object type ς of the form $(p_1 : \alpha_1, \ldots, p_n : \alpha_n)$, specifying the structure of the objects of class c, each enumeration type name ϵ, denoting an enumeration of literals $l_1, \ldots, l_n$, to a union type of the form $[l_1 : [], \ldots, l_n : []]$, `Any` to (), and `Void` to []; $roots \subseteq \mathcal{C}$ are the names that designate the root metaclasses in the metamodel inducing the union type $[\bigcup_{c \in roots}(c : unique\ c[0..*])]$, which we denote by $\mu(\mathcal{M}^<)$.

In our approach, a model may consist of heterogeneous root objects. When considering models in a purely graph-theoretic sense, one needs to consider all classes as roots. The model type expression corresponding to the state machine metamodel of Fig. 1 has the root class name `StateMachine` and type is defined as follows:

```
type(StateMachine) = (id: Oid, name: String,
  nodes: unique ordered State[1..*],
  edges: unique ordered Transition[1..*],
  marking: unique ordered Observation[0..*],
  initial: ref State[1..1],
  final: ref unique ordered State[0..*])
type(State) = (id: Oid, name: String)
type(Transition) = (id: Oid,
  name: String, isCompletion : Bool,
  source: ref State[1..1], source: ref State[1..1])
type(Observation) = (id: Oid, marks: ref State[0..1])
```

4.2 Semantics of Model Type Expressions

We borrow the semantic domain from [8], where $\mathbf{V}$ is the universal value domain of all computable values. We consider that $\mathbf{V}$ contains the set of typed object identifiers O and refer to object identifiers of objects that are instance of a particular class c as O_c.

We refine the semantics for type expressions as model type expressions in order to consider referential types and the different types of constraints definable in a metamodel. The semantics of a referential type expression ρ of the form

$$p : (ref)?\ (unique)?\ (ordered)?\ (c \mid \texttt{Any})[n..m](\#p')?$$

is given by the expression $[\![\rho]\!]_r$ where r is an optional parameter referring to the record containing the field p. The expression $[\![\rho]\!]_r$ is defined as follows:

- If ρ denotes a referential type (with prefix ref) of c that is ordered, then the type is $[\![List(O_c)]\!]$, where $List(O_c)$ can be regarded as the space of injective functions $\mathbf{N} \rightarrow O_c$ (where $\mathbf{N} \subseteq \mathbf{V}$ is the set of natural numbers and the index determines the ordering). If it is not ordered, then the type is $[\![Bag(O_c)]\!]$, where $Bag(O_c)$ can be regarded as the space of non-injective functions $\mathbf{N} \rightarrow O_c$, where the index is only used to distinguish different occurrences of the same value.
- Otherwise, ρ denotes a containment type (an object type) of class c and if it is ordered, then the type is $[\![List(type(c))]\!]$, where $List(type(c))$ can be regarded as the space of injective functions $\mathbf{N} \rightarrow type(c)$. In this

case, the domain $type(c)$ of such functions is the class of objects of type c and not just a set of references. Likewise, if it is not ordered the type is $[\![Bag(type(c))]\!]$.

Also there may be constraints regarding multiplicities, uniqueness and bidirectionality that restrict the corresponding referential types. Such constraints are encoded as follows:

- For referential types, assuming we have a collection of identifiers ic (either a list or a bag), multiplicity constraints $[n..m]$ are encoded as follows $|ic| \geq n \ \wedge \ (|ic| \leq m \ \vee \ |ic| = *)$. The constraint is encoded likewise for containment types.
- For referential types, assuming we have a collection of identifiers ic (either a list or a bag), the uniqueness constraint is encoded as follows $|ic| = |set(ic)|$, where set is the operator that extracts a set from a list or a bag by filtering out duplicates. The constraint is encoded likewise for containment types.
- For referential types, when the parameter r in $[\![\rho]\!]_r$ is present, the bidirectionality constraint $p : refc[n..m]\#p'$, where $p(r) = ic$[2] is encoded as follows $\forall i' \in ic(\exists r' \in \mathbf{R}(r' \in [\![\texttt{type}(c)]\!] \ \wedge \ id(r') = i' \ \wedge \ id(r) \in p'(r')))$. For containment types, the constraint is encoded likewise but we fetch the object identifier of the contained object by inspecting the field id.

For a property $p : \alpha$ defined in an object type ς, a full encoding of the semantics denoted by the expression $[\![\alpha]\!]_r$, where $r \in [\![\varsigma]\!]$, can be obtained by enumerating the 16 combinations of constraints definable in referential and containtment type expressions in addition to the cases for attributes.

For defining model types, we consider the universal value domain $\mathbf{V}$ of all computable values, the domain $\mathbf{R}$ of records, the domain μ of union values (variants) and the set $\mathbf{F}$ of continuous functions f from $\mathbf{V}$ to $\mathbf{V}$. We refine the semantics for type expressions as model type expressions in order to consider referential types and the different types of constraints definable in a metamodel as follows:

$$[\![b]\!] = \mathcal{D}_b \in \mathbf{V} \qquad [\![\epsilon]\!] = [\![type(\epsilon)]\!]$$
$$[\![\texttt{Void}]\!] = [\![[\,]]\!] \qquad [\![\texttt{Any}]\!] = [\![()]\!]$$
$$[\![(\varrho)]\!] = \bigcap_{p:\alpha \in \varrho} \{r \in \mathbf{R} \mid p(r) \in [\![\alpha]\!]_r\} \in \mathbf{V}$$
$$[\![[\varrho]]\!] = \bigcup_{p:\alpha \in \varrho} \{\langle p, v\rangle \in \mathbf{U} \mid v \in [\![\alpha]\!]\} \in \mathbf{V}$$
$$[\![()]\!] = \mathbf{R} \in \mathbf{V} \qquad [\![[\,]]\!] = \{\bot\} \in \mathbf{V}$$
$$[\![\tau_1 \to \tau_2]\!] = \{f \in \mathbf{F} \mid v \in [\![\tau_1]\!] \Rightarrow f(v) \in [\![\tau_2]\!]\} \in \mathbf{V}$$

where the notation $\langle p, q\rangle$ denotes pairs of values p and q.

4.3 Model Subtyping

In this section, we extend the object subtyping (subsumption) relation [8] to model types. Model subtyping emerges from object subtyping and it is implicitly defined when a pair of model type expressions are given.

To extend the subtyping relation, we need to introduce how referential types and containment types are related. We do so by introducing an incompatibility relation $\nprec \subseteq \mathbf{RefTName} \times \mathbf{RefTName}$, which captures the subtyping violations (regarding multiplicities, ordering, uniqueness and bidirectionality) between two property type expressions. That is, an object type expression ς_1 containing a referential type expression $p : \alpha_1$ that is incompatible with a referential type expression $p : \alpha_2$, with the same name p, of another object type expression ς_2 means that ς_1 does not denote a subtype of the type denoted by ς_2. The incompatibility relation $\nprec$ is defined as follows:

$$\begin{aligned}\rho_1 \nprec \rho_2 \iff & \ lower(\rho_1) < lower(\rho_2) \\ & \vee \ ((upper(\rho_1) \neq * \ \wedge \ upper(\rho_2) \neq *) \\ & \quad \Rightarrow upper(\rho_1) > upper(\rho_2)) \\ & \vee \ (upper(\rho_1) = * \Rightarrow upper(\rho_2) \neq *) \\ & \vee \ (not(isOrdered(\rho_1)) \ \wedge \ (isOrdered(\rho_2))) \\ & \vee \ (not(isUnique(\rho_1)) \ \wedge \ (isUnique(\rho_2))) \\ & \vee \ op(\rho_1) \neq op(\rho_2)\end{aligned}$$

Given two model types $\mathcal{M}^{<}_{sub}$ and $\mathcal{M}^{<}_{super}$, the extended subtyping relation $\leq:_\beta \ \subseteq \ \mathbf{Type} \times \mathbf{Type}$, where $\beta \subseteq C \times C$ captures the names of those object types that have been related already, is defined as follows:

$$b \leq:_\beta b$$
$$Int \leq:_\beta Float \qquad \text{(base type names)}$$
$$\epsilon_1 \leq:_\beta \epsilon_2 \iff \texttt{type}(\epsilon_1) \leq:_\beta \texttt{type}(\epsilon_2) \qquad \text{(enum type names)}$$
$$c \leq:_\beta \texttt{Any}$$
$$\texttt{Void} \leq:_\beta c$$
$$c_1 \leq:_\beta c_2 \iff \texttt{type}(c_1) \leq:_\beta \texttt{type}(c_2) \qquad \text{(class names)}$$
$$\rho_1 \leq:_\beta \rho_2 \iff (c(\rho_1), c(\rho_2)) \in \beta \ \vee \ (not(\rho_1 \nprec \rho_1) \Rightarrow \texttt{type}(c(\rho_1)) \leq:_{\beta \cup \{(c(\rho_1),(c(\rho_2)))\}} \texttt{type}(c(\rho_2))) \qquad \text{(referential types)}$$
$$(\varrho) \leq:_\beta () \qquad [\,] \leq:_\beta (\varrho)$$
$$(\varrho_1) \leq:_\beta (\varrho_2) \iff \forall p : \alpha_2 \in \varrho_2 \ (\exists p : \alpha_1 \in \varrho_1(\alpha_1 \leq:_\beta \alpha_2)) \qquad \text{(object types)}$$
$$[\,] \leq:_\beta [\varrho] \qquad [\varrho_1] \leq:_\beta [\varrho_2] \iff \forall p : \alpha_1 \in \varrho_1(\exists p : \alpha_2 \in \varrho_2(\alpha_1 \leq:_\beta \alpha_2)) \qquad \text{(union types)}$$
$$\sigma_1 \to \tau_1 \leq:_\beta \sigma_2 \to \tau_2 \iff \sigma_2 \leq:_\beta \sigma_1 \ \wedge \ \tau_1 \leq:_\beta \tau_2 \qquad \text{(function types)}$$

The semantics of subtyping is given by set inclusion of domains [8, Theorem Semantic Subtyping].

[2] $p(r)$ denotes the projection of the value of field p in record r.

Theorem 4.3 (Extended semantic subtyping)**.**

$$\tau \leq:_{\emptyset} \tau' \Rightarrow [\![\tau]\!] \subseteq [\![\tau']\!]$$

The proof is similar to that of Theorem Semantic Subtyping in [8] while considering the 16 types of constraints imposable on referential types and noting that β acts as a memory of already paired class names for traversing referential types effectively.

We generalize the notion subtyping to the notion of model subtyping through the object type of the root objects of a model as follows:

Definition 4.4 (Syntactic Model Subtyping)**.**

$$\mathcal{M}_1^{<} \leq: \mathcal{M}_2^{<} \iff \mu(\mathcal{M}_1^{<}) \leq:_{\emptyset} \mu(\mathcal{M}_2^{<}).$$

5 Metamodel Specifications and Subtyping

In this section, we are going to use the notion of metamodel specification to refer to metamodels $\mathcal{M}^{\rightarrow}$ enriched with a set of OCL constraints Ω. We are going to abuse the notation $\mathcal{M}^{\rightarrow}$ to refer to any OO representation for metamodels where object subtyping is represented ontologically and where its semantics corresponds to subclassing (e.g. USE class diagrams, EMF metamodels), abstracting away from the syntactic specificities of each OO representation. Thus, our metamodels $\mathcal{M}^{\rightarrow}$ may refer to EMF metamodels or to USE class diagrams interchangeably.

Next, we introduce a representation for metamodels $\mathcal{M}^{\rightarrow}$ and recall their semantics as object models $\sigma(\mathcal{M}^{\rightarrow})$, given in [27], by slightly adapting their syntax. Then, we show how a metamodel $\mathcal{M}^{\rightarrow}$ determines a model type expression $\mathcal{M}^{<}$ by means of a syntax transformation $decl : \mathcal{M}^{\rightarrow} \mapsto \mathcal{M}^{<}$, which is used to triangulate $\sigma(\mathcal{M}^{\rightarrow})$ with $[\![\mathcal{M}^{<}]\!]$, reflecting the notion of model subtyping from model type expressions to metamodels. Finally, we extend the syntactic subtyping relation to metamodel specifications, defining its semantics.

5.1 Syntax of Metamodel Specifications

As the the typical representation of metamodels (normally depicted as class diagrams) is not aware of some type-theoretic constructs (such as union types) and the object subtyping relation is explicitly defined as a subclassing relation, we provide an alternative representation for metamodels $\mathcal{M}^{\rightarrow}$. Given the countable sets $\mathcal{B}$ of base type names b, $\mathcal{E}$ of enumeration type names ϵ, $\mathcal{C}$ of class names c, $\mathcal{P}$ of property names p, a metamodel $\mathcal{M}^{\rightarrow}$ is a tuple (Class, $\prec$, Enum, Cons) where: Class $\subseteq \mathcal{C}$ is a countable set of class names; $\prec \subseteq \mathcal{C} \times \mathcal{C}$ is a subclassing relation (allowing for multiple inheritance); Enum $\subseteq \mathcal{E}$ is a countable set of enumeration type names; and Cons is the set of constraints defined by a metamodel.

The constraints in Cons are given as propositions of the form: $isAbstract(c)$ holds if class c is abstract; $p(c)$ holds if class c contains a structural feature with name p; $l(\epsilon)$ holds if l is a literal of the enumeration type denoted by ϵ; $type(c, p) = t$, where $t \in \mathcal{B} \sqcup$ Enum $\sqcup$ Class defines the type of a structural feature, which is an attribute if $t \in \mathcal{B} \sqcup$ Enum and a reference if $t \in$ Class; $cont(c, p)$ holds if the structural feature p is a containment reference; for $n \in \mathbf{N}$ and $m \in \mathbf{N} \cup \{*\}$, $lower(c, p) = n$ and $upper(c, p) = m$, such that $n \leq m$, denote the lower and upper bounds of references; $unique(c, p)$ holds if the structural feature is unique (when $upper(c, p) > 1$); $ordered(c, p)$ holds if the structural feature is ordered (when $upper(c, p) > 1$); $opposite(c, p, c', p')$ holds if the structural feature $p(c)$ is defined as opposite of $p'(c')$, defining a bidirectional association.

5.2 Mapping Metamodels to Their Model Types

The interpretation $\sigma(\mathcal{M}^{\rightarrow})$ of a metamodel $\mathcal{M}^{\rightarrow}$ is usually given in terms of the set of metamodel-conformant models, which are normally represented as object diagrams. In this subsection, we map a semantics $\sigma(\mathcal{M}^{\rightarrow})$ to $[\![\mathcal{M}^{\rightarrow}]\!] \subseteq \mathbf{V}$ in our semantic domain in order to enable structural model subtyping for metamodels $\mathcal{M}^{\rightarrow}$.

We focus on the abstract syntax of object diagrams by using attributed graphs [16], which are defined in terms of E-graphs. An E-graph has two different types of nodes, graph and data nodes, and three kinds of edges, namely regular edges, and edges for node and edge attribution. An E-graph [16, Def. 8.1] is defined as a tuple

$$(N_G, N_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{F, NA, EA\}}),$$

where: N_G and N_D are the sets of graph and data nodes, resp.; E_G, E_{NA} and E_{EA} are the sets of graph, node attribute and edge attribute, resp.; and source and target functions are defined as

$source_G, target_G : E_G \rightarrow N_G$,
$source_{NA} : E_{NA} \rightarrow N_G$,
$target_{NA} : E_{NA} \rightarrow N_D$,
$source_{EA} : E_{EA} \rightarrow E_G$, and
$target_{EA} : E_{EA} \rightarrow N_D$.

Let $DSIG = (S_D, OP_D)$ be a data signature with attribute value sorts $S'_D \subseteq S_D$, **AG** is defined as the set of attributed graphs of the form (G, D), where G is an E-graph and D is a $DSIG$-algebra.

To define the model type that corresponds to each metamodel we need to relate the semantics of a metamodel $\mathcal{M}^{\rightarrow}$ to a model type $[\![\mathcal{M}^{<}]\!]$. Given the set **Union** of union values, such link will be achieved by means of a map

$$\uparrow^{<}_{\rightarrow} : \mathbf{AG} \rightarrow \mathbf{Union}$$

that obtains a union value from an attributed graph (or object diagram) and then we generalize $\uparrow^{<}_{\rightarrow}$ to all possible attributed graphs definable by a metamodel, mapping the semantics of metamodels to the semantics of model types.

$\uparrow^{<}_{\rightarrow}$ is defined by the equation

$$\uparrow^{<}_{\rightarrow}(G) \triangleq \langle root = oc \rangle,$$

where oc consists of objects $\{id = id(n), ps\}$ defined from nodes $n \in N_G$, where ps is defined as the set union of properties defined from edges in E_G and E_{EA} as follows:

- $p = v$ where $v = target(e)$ such that $e \in E_{NA}$ and $source(e) = n$ and $name(e) = p$;
- $p = ic$ where $ic = \{id(target(e)) \mid e \in E_G, source(e) = n, name(e) = p\}$.

We denote the function space **AG** $\rightarrow$ **Union** as $\Uparrow^{<}_{\rightarrow}$. The interpretation of a metamodel $\mathcal{M}^{\rightarrow}$ is given as the set $\sigma(\mathcal{M}^{\rightarrow})$ of system states definable using $\mathcal{M}^{\rightarrow}$. Ignoring the representational gap, we assume that $\sigma(\mathcal{M}^{\rightarrow}) \subseteq \mathcal{P}(\mathbf{AG})$.[3] Hence, the model type of a metamodel $\mathcal{M}^{\rightarrow}$ is given as

$$[\![\mathcal{M}^{\rightarrow}]\!] \triangleq \Uparrow^{<}_{\rightarrow}(\sigma(\mathcal{M}^{\rightarrow})).$$

And, consequently, there must be a syntax transformation $decl : \mathcal{M}^{\rightarrow} \mapsto \mathcal{M}^{<}$ so that

$$\Uparrow^{<}_{\rightarrow}(\sigma(\mathcal{M}^{\rightarrow})) = [\![decl(\mathcal{M}^{\rightarrow})]\!].$$

Note that, in this setting, we are not considering a metamodel $\mathcal{M}^{\rightarrow}$ as mere abstract syntax for a model type expression $\mathcal{M}^{<}$, as both syntactic representations, which have different semantics, are only related by *decl*. This gives us the sufficient equipment for reflecting the subtyping relation from model type expressions to metamodels:

Definition 5.1 (Metamodel subtyping)**.**

$$\mathcal{M}_1^{\rightarrow} \leq: \mathcal{M}_2^{\rightarrow} \iff decl(\mathcal{M}_1^{\rightarrow}) \leq: decl(\mathcal{M}_2^{\rightarrow})$$

When $\mathcal{M}_1^{\rightarrow} \leq: \mathcal{M}_2^{\rightarrow}$, we have that $[\![\mathcal{M}_1^{\rightarrow}]\!] \subseteq [\![\mathcal{M}_2^{\rightarrow}]\!]$.

5.3 Semantics of Metamodel Specifications

In this subsection, we present how metamodel specifications $(\mathcal{M}^{\rightarrow}, \Omega)$ declare model types by extending metamodels $\mathcal{M}^{\rightarrow}$ with OCL expressions Ω. The semantics of OCL expressions is reused from [27, 28]. It is worth noting that Clark et al. provided a formal semantics for OCL [10] that is more amenable to our model type expressions from a semantic point of view, if we consider the MML calculus [11]. However, the USE semantics comes equipped with the tool support that we need in subsequent sections.

We first define the semantics of a metamodel specification $(\mathcal{M}^{\rightarrow}, \Omega)$ using the original USE semantics $I[\![\Omega]\!](\mathcal{M}^{\rightarrow}, \Gamma)$, where Ω is a set of OCL invariants and Γ is the environment of variables for evaluating the OCL invariants Ω, and then we define the corresponding model type.

The original OO semantics of a metamodel specification $(\mathcal{M}^{\rightarrow}, \Omega)$ (representing an object model[4]) can be extended as follows:

$$\sigma((\mathcal{M}^{\rightarrow}, \Omega)) = \{G \in \sigma(\mathcal{M}^{\rightarrow}) \mid I[\![\Omega]\!](\mathcal{M}^{\rightarrow}, \emptyset) = true\}.$$

We define the satisfaction of OCL constraints Ω in a metamodel $\mathcal{M}^{\rightarrow}$ as follows

$$\models_{\mathcal{M}^{\rightarrow}}(\Omega) \triangleq \sigma((\mathcal{M}^{\rightarrow}, \Omega)).$$

We write $\not\models_{\mathcal{M}^{\rightarrow}}(\Omega)$ whenever $\models_{\mathcal{M}^{\rightarrow}}(\Omega) = \emptyset$. We write $G \models_{\mathcal{M}^{\rightarrow}}(\Omega)$ whenever a model G conforms to a metamodel specification $(\mathcal{M}^{\rightarrow}, \Omega)$, that is $G \in \sigma((\mathcal{M}^{\rightarrow}, \Omega))$.

The model type of a metamodel specification $(\mathcal{M}^{\rightarrow}, \Omega)$ is defined as follows:

Definition 5.2 (Model Type of $(\mathcal{M}^{\rightarrow}, \Omega)$)**.**

$$[\![(\mathcal{M}^{\rightarrow}, \Omega)]\!] \triangleq \Uparrow^{<}_{\rightarrow}(\sigma((\mathcal{M}^{\rightarrow}, \Omega)))$$

5.4 Metamodel Specification Subtyping

To check if there are inconsistencies due to the co-existence of OCL constraints of the model subtype and of the model supertype, an extension metamodel $\mathcal{M}^{\rightarrow}_{sub,\,super}$ is synthesized from metamodels $\mathcal{M}^{\rightarrow}_{sub}$ and $\mathcal{M}^{\rightarrow}_{super}$. The main purpose of $\mathcal{M}^{\rightarrow}_{sub,\,super}$ is twofold: its semantics $[\![\mathcal{M}^{\rightarrow}_{sub,\,super}]\!]$ subsumes the semantics of $[\![\mathcal{M}^{\rightarrow}_{sub}]\!]$ while it is subsumed by the semantics of $[\![\mathcal{M}^{\rightarrow}_{super}]\!]$, that is

$$[\![\mathcal{M}^{\rightarrow}_{sub}]\!] \subseteq [\![\mathcal{M}^{\rightarrow}_{sub,\,super}]\!] \subseteq [\![\mathcal{M}^{\rightarrow}_{super}]\!]; \qquad (1)$$

and it reifies $\mathcal{M}^{\rightarrow}_{sub} \leq: \mathcal{M}^{\rightarrow}_{super}$ as an explicitly declared (static) subclassing relation $\prec$ between classes of $\mathcal{M}^{\rightarrow}_{sub}$ and classes of $\mathcal{M}^{\rightarrow}_{super}$ – that is, for any $c_{sub} \in \textsc{Class}_{sub}$ and $c_{super} \in \textsc{Class}_{super}$,

$$c_{sub} \prec c_{super} \iff \texttt{type}(c_{sub}) \leq: \texttt{type}(c_{super}). \qquad (2)$$

In the following, the synthesis process is discussed and illustrated with the running example. We assume that the set of classifier names for $\mathcal{M}^{\rightarrow}_{sub}$ and $\mathcal{M}^{\rightarrow}_{super}$ are disjoint. If that is not the case, a renaming has to be applied to the classifiers of the subtype metamodel first so as to ensure that precondition. Given the metamodels $\mathcal{M}^{\rightarrow}_{sub}$ and $\mathcal{M}^{\rightarrow}_{super}$, and a subclassing relation $\prec$ between their classes, the operation $synth(\mathcal{M}^{\rightarrow}_{sub}, \mathcal{M}^{\rightarrow}_{super}, \prec)$ produces a metamodel $\mathcal{M}^{\rightarrow}_{sub,\,super}$ that is defined as follows:

Initialization. Initially, the set $\textsc{Class}_{sub,\,super}$ of classifiers of $\mathcal{M}^{\rightarrow}_{sub,\,super}$ is formed by those of $\mathcal{M}^{\rightarrow}_{sub}$ and of $\mathcal{M}^{\rightarrow}_{super}$. Then, the subclassing relation $\prec_{sub,\,super}$ for $\mathcal{M}^{\rightarrow}_{sub,\,super}$ is obtained as follows $\prec_{sub,\,super} = \prec_{sub} \cup \prec_{super} \cup \prec$. Classes of the supertype are made abstract by adding constraints

$$\bigcup_{c \in \textsc{Class}_{super}} isAbstract(c) \qquad (3)$$

to $\textsc{Cons}_{sub,\,super}$. These constraints are added to narrow down the search for inconsistencies to $[\![\mathcal{M}^{\rightarrow}_{sub}]\!]$.

[3] This proposition relies on the following observation: a system state for a metamodel $\mathcal{M}^{\rightarrow}$ is defined as a structure $\sigma(\mathcal{M}^{\rightarrow}) = (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}, \sigma_{\textsc{Assoc}})$ in [27], where $\sigma_{\textsc{Class}}$ resembles N_G, $\sigma_{\textsc{Att}}$ resembles E_{NA}, and $\sigma_{\textsc{Assoc}}$ resembles E_G. The direction of the set inclusion is justified by the fact that E-graphs in **AG** are not necessarily typed.

[4] Noting: that the syntax we introduced for object models only considers binary associations (without an explicit name for the association); that attribute operation signatures $p : \textsc{Class} \rightarrow \mathcal{B} \sqcup \textsc{Enum}$ can be extracted from propositions of the form $p(c)$ and $type(p) = t$ where $t \in \mathcal{B} \sqcup \textsc{Enum}$; and that compositions are immaterial in USE format.

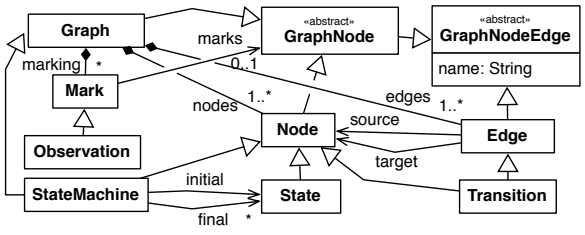

Figure 2. Synthesized metamodel $\mathcal{M}^{\rightarrow}_{sm,g}$

Virtual superclass extraction. $\prec_{sub,super}$ is analyzed and for each class that inherits the same feature f declared in two different superclasses sc_1 and sc_2, we extract an abstract virtual superclass $sc_{1,2}$, where the order in the subscript 1, 2 is inessential, of the two superclasses sc_1 and sc_2 in order to ensure the diamond property. The feature f is pulled up to the virtual superclass $sc_{1,2}$ (and removed from the superclasses sc_1 and sc_2). $\textsc{Class}_{sub,super}$ is augmented with $sc_{1,2}$ and $\prec_{sub,super}$ is augmented with $(sc_1, sc_{1,2})$ and $(sc_2, sc_{1,2})$. This analysis is repeated recursively until all virtual superclasses and the corresponding specialization links are added.

Preserving the semantics of subclassing. The next step consists in pulling up common features from classes in the subtype metamodel $\mathcal{M}^{\rightarrow}_{sub}$ to classes in the supertype metamodel $\mathcal{M}^{\rightarrow}_{super}$. For each class $c_{sub} \in \textsc{Class}_{sub}$ that declares a feature $p(c_{sub}) \in \textsc{Cons}_{sub}$ such that there is an ancestor class $c_{super} \in \textsc{Class}_{super}$ declaring a feature $p(c_{super}) \in \textsc{Cons}_{super}$, the structural feature $p(c_{super})$ is added to $\textsc{Cons}_{sub,super}$ and removed from $\textsc{Cons}_{super}$ together with all the constraints defined for p. In addition, the constraints of feature p for c_{sub} in $\textsc{Cons}_{sub}$ are redefined for c_{super} and added to $\textsc{Cons}_{sub,super}$ while they are removed from $\textsc{Cons}_{sub}$ together with $p(c_{sub})$. This means that the feature p is pulled up from c_{sub} to the ancestor c_{super} (and removed from c_{sub}). The rest of constraints from $\textsc{Cons}_{sub}$ and $\textsc{Cons}_{super}$ are added to $\textsc{Cons}_{sub,super}$. Hence, $[\![\mathcal{M}^{\rightarrow}_{sub,super}]\!] \subseteq [\![\mathcal{M}^{\rightarrow}_{super}]\!]$.

The resulting extension metamodel $\mathcal{M}^{\rightarrow}_{sm,g}$ for the running example is depicted in Fig. 2, where the multiplicities for the references `nodes` and `edges` from $\mathcal{M}^{\rightarrow}_{sm}$ have been preserved. In $\mathcal{M}^{\rightarrow}_{sm,g}$, two virtual classes, `GraphNode` and `GraphNodeEdge`, have been extracted in two consecutive iterations.

For extending model subtyping to metamodel specifications using the construction presented above, we are going to use the extension metamodel $\mathcal{M}^{\rightarrow}_{sub,super}$ synthesized by means of $synth(\mathcal{M}^{\rightarrow}_{sub}, \mathcal{M}^{\rightarrow}_{super}, \prec_{sub,super})$ where

$$\prec_{sub,super} = \mathcal{M}^{\rightarrow}_{sub} \leq: \mathcal{M}^{\rightarrow}_{super} \cap \textsc{Class}_{sub} \times \textsc{Class}_{super}.$$

Note that property (1) is satisfied as $[\![\mathcal{M}^{\rightarrow}_{sub}]\!] \subset [\![\mathcal{M}^{\rightarrow}_{sub,super}]\!]$ by definition. In the example, we can have models in $\mathcal{M}^{\rightarrow}_{sm,g}$, such an object `StateMachine` pointing to an object `Transition` through the reference `nodes`, that are not valid models in $\mathcal{M}^{\rightarrow}_{sm}$. In addition, the preservation of the constraints of structural features in $\mathcal{M}^{\rightarrow}_{sub}$ when extracting virtual classes forces $[\![\mathcal{M}^{\rightarrow}_{sub,super}]\!] \subseteq [\![\mathcal{M}^{\rightarrow}_{super}]\!]$. For instance, in the example, objects of type `State` are of type `Node` as the properties of `type(Node)` are included in `type(State)`.

On the other hand, the property (2) is ensured by constructing $\prec$ as explained above, since the extracted virtual classes have a counterpart in the semantic domain as the meet types of the corresponding object types.

Definition 5.3 (Syntactic Model Subtyping with Metamodel Specifications)**.**

$$(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub}) \leq: (\mathcal{M}^{\rightarrow}_{super}, \Omega_{super}) \triangleq \mathcal{M}^{\rightarrow}_{sub} \leq: \mathcal{M}^{\rightarrow}_{super} \wedge \models_{\mathcal{M}^{\rightarrow}_{sub,super}} (\Omega_{sub} \Rightarrow \Omega_{super})$$

Theorem 5.4 (Semantic Model Subtyping with Metamodel Specifications)**.**

$$(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub}) \leq: (\mathcal{M}^{\rightarrow}_{super}, \Omega_{super}) \Rightarrow [\![(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})]\!] \subseteq [\![(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})]\!]$$

Proof. As $\mathcal{M}^{\rightarrow}_{sub} \leq: \mathcal{M}^{\rightarrow}_{super}$, there is a synthesized metamodel $\mathcal{M}^{\rightarrow}_{sub,super}$ in which both Ω_{sub} and Ω_{super} can be satisfied.

By the construction of $\mathcal{M}^{\rightarrow}_{sub,super}$, we have that

$$[\![(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})]\!] \subseteq [\![(\mathcal{M}^{\rightarrow}_{sub,super}, \Omega_{sub})]\!].$$

Since $\models_{\mathcal{M}^{\rightarrow}_{sub,super}} (\Omega_{sub} \Rightarrow \Omega_{super})$, then

$$[\![(\mathcal{M}^{\rightarrow}_{sub,super}, \Omega_{sub})]\!] \subseteq [\![(\mathcal{M}^{\rightarrow}_{sub,super}, \Omega_{super})]\!].$$

By the construction of $\mathcal{M}^{\rightarrow}_{sub,super}$, we have that

$$[\![(\mathcal{M}^{\rightarrow}_{sub,super}, \Omega_{super})]\!] \subseteq [\![(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})]\!].$$

Hence, $[\![(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})]\!] \subseteq [\![(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})]\!]$. □

5.5 Model Management Operation Specifications

In this section, we extend function types to metamodel specifications in order to specify model management operations

$$(\mathcal{M}^{\rightarrow}_{pre}, \Omega_{pre}) \rightarrow (\mathcal{M}^{\rightarrow}_{post}, \Omega_{post}),$$

where $(\mathcal{M}^{\rightarrow}_{pre}, \Omega_{pre})$ specifies its preconditions and $(\mathcal{M}^{\rightarrow}_{post}, \Omega_{post})$ specifies its postconditions. An operation specified by $(\mathcal{M}^{\rightarrow}_{pre}, \Omega_{pre}) \rightarrow (\mathcal{M}^{\rightarrow}_{post}, \Omega_{post})$ can be applied to models conforming to a metamodel specification $(\mathcal{M}^{\rightarrow}_{ac}, \Omega_{ac})$ if $(\mathcal{M}^{\rightarrow}_{ac}, \Omega_{ac}) \leq: (\mathcal{M}^{\rightarrow}_{pre}, \Omega_{pre})$. On the other hand, we can reuse the results of this operation as models conforming to a (possibly different) metamodel specification $(\mathcal{M}^{\rightarrow}_{ac}, \Omega_{ac})$ if $(\mathcal{M}^{\rightarrow}_{post}, \Omega_{post}) \leq: (\mathcal{M}^{\rightarrow}_{ac}, \Omega_{ac})$. The first condition ensures that the operation can rely on values of $(\mathcal{M}^{\rightarrow}_{ac}, \Omega_{ac})$ and the second condition guarantees that its outputs will not create any problem in the expecting context.

In our running example, $(\mathcal{M}^{\rightarrow}_{g}, \Omega_{g}) \rightarrow (\mathcal{M}^{\rightarrow}_{g}, \Omega_{g})$ can be applied to a model in $(\mathcal{M}^{\rightarrow}_{sm}, \Omega_{sm})$ as is for simulating a deterministic state machine. This will result in the creation of objects `Mark`, which can be automatically coerced to the type `Observation` as explained in section 6.3.

6 Tool Support

The theory described in previous sections is implemented as a Java library that facilitates reuse in MDE. Specifically, given two metamodel specifications $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})$ and $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$, where $\mathcal{M}^{\rightarrow}_{sub}$ and $\mathcal{M}^{\rightarrow}_{super}$ are EMF metamodels and Ω_{sub} and Ω_{super} are sets of OCL constraints in USE format (whose contextual types are defined in $\mathcal{M}^{\rightarrow}_{sub}$ and $\mathcal{M}^{\rightarrow}_{super}$ resp.), the tool will determine whether $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})$ denotes a model subtype of the model type denoted by $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$. Note that any of the sets of OCL constraints may be empty.

If the check fails, there are two main sources of incompatibilities: the model types denoted by the metamodels, and the OCL constraints. In the first case, the tool points at the source of the problem by showing the classes of the supertype metamodel $\mathcal{M}^{\rightarrow}_{super}$ that are not extended by classes of $\mathcal{M}^{\rightarrow}_{sub}$. That information is useful to assess the advantage of, for example, prunning the supertype metamodel by computing the effective metamodel [29] w.r.t. a specific model management operation. In the second case, the tool will provide evidence that contradicts the compatibility property in Def. 5.3 of $\mathcal{M}^{\rightarrow}_{sub}$ w.r.t. $\mathcal{M}^{\rightarrow}_{super}$ in the form of a model $G \in [\![\mathcal{M}^{\rightarrow}_{sub,\,super}]\!]$, represented in EMF notation (that is in XMI format), that invalidates a constraint in Ω_{super}.

If the check succeeds, the tool guarantees that $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})$ is a structural refinement of $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$. Hence, any model management operation that is defined for $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$ can be safely applied to models of $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})$. Going one step further, the tool also facilitates the reuse of such operation, when it is based on EMF, by automatically synthesizing an extension metamodel $\mathcal{M}^{\rightarrow}_{sub,\,super}$ that can be substituted for $\mathcal{M}^{\rightarrow}_{super}$ in the signature of the operation ensuring its application to models conforming to $[\![(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})]\!]$ without any further change.

In the following subsections, we explain how the tool reuses a third-party bounded model finder for verifying the compatibility property of Def. 5.3 and we discuss how our subtyping relation $\leq:$ can be used to deal with multiple, dynamic and partial model typing.

6.1 Analysing OCL Constraints

In our tool we have adapted excerpts of TOTEM-MDE [15] in order to integrate the USE Validator [22] with EMF. For computing whether $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub}) \leq: (\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$ holds, the property $\models_{\mathcal{M}^{\rightarrow}_{sub,\,super}} (\Omega_{sub} \Rightarrow \Omega_{super})$ needs to be verified. Assuming Ω_{super} is of the form $\bigwedge_i$ `context` c_i `inv:` ω_i, we negate the property to be checked, obtaining

$$\models_{\mathcal{M}^{\rightarrow}_{sub,\,super}} (\Omega_{sub} \wedge \texttt{context } c \texttt{ inv: } \bigvee_i c_i\texttt{.allInstances()->exists(}\neg\omega_i\texttt{)})$$

where c is any class name from $\textsc{Class}_{super}$.

If the negated property turns out to be satisfiable, a model G conforming to $\mathcal{M}^{\rightarrow}_{sub,\,super}$, that is $G \in [\![\mathcal{M}^{\rightarrow}_{sub}]\!]$ is returned, showing that there is an inconsistency that needs to be resolved. Subject to the appropriateness of the bounds provided for the analysis, if the negated property turns out to be unsatisfiable, the original property holds and $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})$ defines a subtype of the model type denoted by $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$. In the running example, we have that $(\mathcal{M}^{\rightarrow}_{sm}, \Omega_{sm}) \leq: (\mathcal{M}^{\rightarrow}_{g}, \Omega_{g})$ but $(\mathcal{M}^{\rightarrow}_{sm}, \emptyset) \not\leq: (\mathcal{M}^{\rightarrow}_{g}, \Omega_{g})$, where $(\mathcal{M}^{\rightarrow}_{sm}, \emptyset)$ denotes a model type of non-deterministic state machines.

6.2 Multiple and Strict Typing

Given two metamodel specifications $(\mathcal{M}^{\rightarrow}_{1}, \Omega_{1})$ and $(\mathcal{M}^{\rightarrow}_{2}, \Omega_{2})$, when we compute $(\mathcal{M}^{\rightarrow}_{1}, \Omega_{1}) \leq: (\mathcal{M}^{\rightarrow}_{2}, \Omega_{2})$, we obtain multiple inheritance semantics [8], that is multiple typing, by default. However, we can force strict typing in contexts where multiple typing is not allowed by obtaining the synthesized metamodel using a strict specialization relation $\prec^{s} \subseteq \textsc{Class}_{sub} \times \textsc{Class}_{super}$ where

$$c_1 \prec^{s} c_2 \Rightarrow type(c_1) \leq:_{\emptyset} type(c_2) \wedge c_1 \prec^{s} c_3 \Rightarrow c_2 = c_3.$$

Note that there may be many such strict specialization relations in $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub}) \leq: (\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$. Our tool enumerates all of them, and for each specialization relation $\prec^{s}$, it computes: the extension metamodel $synth(\mathcal{M}^{\rightarrow}_{sub}, \mathcal{M}^{\rightarrow}_{super}, \prec^{s})$; the complement supertype metamodel $\nearrow(\mathcal{M}^{\rightarrow}_{sub}, \mathcal{M}^{\rightarrow}_{super}, \prec^{s})$ defined as the effective metamodel of those classes in $\textsc{Class}_{super}$, including their features and associated multiplicity constraints, that do not appear in the image of $\prec^{s}$; the complement subtype metamodel $\searrow(\mathcal{M}^{\rightarrow}_{sub}, \mathcal{M}^{\rightarrow}_{super}, \prec^{s})$ similarly defined for those classes in $\textsc{Class}_{sub}$ that do not appear in the preimage of $\prec^{s}$; and a rank representing the model size of the complement metamodels.

The strict specialization relation $\prec^{s}$ with the lowest rank is the solution that provides the maximal model subtype w.r.t. strict model subtyping. When computing $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub}) \leq: (\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$ using strict typing mode, the tool recommends the specialization relation $\prec^{s}$ with the lowest rank and will compute the satisfaction of OCL constraints using the synthesized metamodel $synth(\mathcal{M}^{\rightarrow}_{sub}, \mathcal{M}^{\rightarrow}_{super}, \prec^{s})$. In case of a tie, one strict specialization relation is chosen arbitrarily. However, the user can still inspect the other strict specialization relations which are enumerated by recommendation order, together with their corresponding synthesized metamodel and complement metamodels.

In the example, when forcing strict model subtyping, the extension metamodel $synth(\mathcal{M}^{\rightarrow}_{sm}, \mathcal{M}^{\rightarrow}_{g}, \prec^{s})$ that is recommended is depicted in Fig. 3, together with the complement of the subtype metamodel $\searrow(\mathcal{M}^{\rightarrow}_{sm}, \mathcal{M}^{\rightarrow}_{g}, \prec^{s})$. The complement of the supertype metamodel $\nearrow(\mathcal{M}^{\rightarrow}_{sm}, \mathcal{M}^{\rightarrow}_{g}, \prec^{s})$ is empty indicating that all object types have been covered by the model subtyping relation induced by $\prec^{s}$.

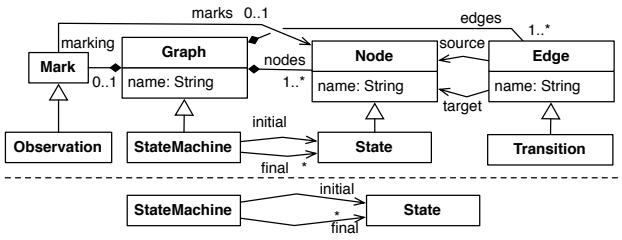

Figure 3. $synth(\mathcal{M}^{\rightarrow}_{sm}, \mathcal{M}^{\rightarrow}_{g}, \prec^{s})$ (top) and $\searrow(\mathcal{M}^{\rightarrow}_{sm}, \mathcal{M}^{\rightarrow}_{g}, \prec^{s})$ (bottom)

6.3 Dynamic Typing

Dynamic typing, understood as the property of an object to change its type while preserving its identity, can be implemented by using the construction of synthesized metamodels. A model operation $(\mathcal{M}^{\rightarrow}_{in}, \Omega_{in}) \rightarrow (\mathcal{M}^{\rightarrow}_{out}, \Omega_{out})$ can be reused for models conforming to a metamodel specification $(\mathcal{M}^{\rightarrow}_{creation}, \Omega_{creation})$ such that $(\mathcal{M}^{\rightarrow}_{creation}, \Omega_{creation}) \leq: (\mathcal{M}^{\rightarrow}_{in}, \Omega_{in})$, which entails the existence of an extension metamodel $\mathcal{M}^{\rightarrow}_{creation,in}$. That is the subtyping relation between two model types is inferred in $\mathcal{M}^{\rightarrow}_{creation,in}$, if it exists within the bounded search space. The extension metamodel is synthesized as explained in section 5.4, but *without* adding constraints of the form (*3*) in order to enable side effects of model management operations that work with supertype classes. The extension metamodel facilitates the reuse of model management operations that may be implemented using EMF-based technology. As EMF relies on a nominal type system, two cases need to be considered when applying the operation: when no new instances of classes in $\textsc{Class}_{creation,in}$ are created, that is only updates are applied; and when new instances of classes in $\textsc{Class}_{creation,in}$ are created. In the first case, the resulting model conforms to $(\mathcal{M}^{\rightarrow}_{creation}, \Omega_{creation})$ without further changes[5]. In the second case, we need to apply a coercion operation for *casting down* objects to classes defined in $(\mathcal{M}^{\rightarrow}_{creation}, \Omega_{creation})$.

For a model $G \in [\![(\mathcal{M}^{\rightarrow}_{in}, \Omega_{in})]\!]$, the coercion operation

$$G \text{ as } \mathcal{M}^{\rightarrow}_{creation} \text{ with } \prec^{s}$$

where $\prec^{s}$ is a strict subtyping relation between classes in $\textsc{Class}_{creation}$ and $\textsc{Class}_{in}$, requires all mandatory features in $\searrow(\mathcal{M}^{\rightarrow}_{creation}, \mathcal{M}^{\rightarrow}_{in}, \prec^{s})$ to have a default value. That is, those mandatory structural features that are not initialized in the operation need to be properly defined in a model.

[5]When the precondition of disjoint classifier names is not satisfied, the subtyping method for metamodel specifications applies an automatic renaming. This forces the use of a retyping G' of models $G \in [\![\mathcal{M}^{\rightarrow}_{creation}]\!]$ so that $G' \in [\![\mathcal{M}^{\rightarrow}_{creation,in}]\!]$. The tool offers automated support for applying adaptations of this kind based on the synthesized extension metamodel, as illustrated with an example at https://mde-subtyping.github.io/web/.

The coercion operator changes the type of those objects in G, whose type is a class $c_{super} \in \textsc{Class}_{\in}$, for the greatest class c_{sub}, with respect to $\prec_{creation}$, in $\textsc{Class}_{creation}$ such that $c_{sub} \prec^{s} c_{super}$. The operator is not defined when the choice of c_{sub} is not unique – e.g. when multiple inheritance is used in $\mathcal{M}^{\rightarrow}_{in}$. In the strict subtyping relation of Fig. 3, the simulation operation $(\mathcal{M}^{\rightarrow}_{g}, \Omega_{g}) \rightarrow (\mathcal{M}^{\rightarrow}_{g}, \Omega_{g})$ creates a new object `Mark`, marking the state that has been executed. The type of the created objects is known in the extension metamodel $\mathcal{M}^{\rightarrow}_{sm,g}$ but not in $(\mathcal{M}^{\rightarrow}_{sm}, \Omega_{sm})$. The coercion operator can then be applied to cast them down to the type `Observation`, yielding a valid model in $\mathcal{M}^{\rightarrow}_{sm}$.

6.4 Partial vs Total Typing

Given two metamodel specifications $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})$ and $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$, if $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub}) \leq: (\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$ then $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$ totally types the models that conform to $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})$. In such case, $|\searrow(\mathcal{M}^{\rightarrow}_{sub}, \mathcal{M}^{\rightarrow}_{super}, \prec)| = 0$. When this equation does not hold, the complement of the subtype metamodel with respect to the inferred specialization relation $\prec$, which may or may not be strict, tells us that $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$ is only a partial type for models that conform to $(\mathcal{M}^{\rightarrow}_{sub}, \Omega_{sub})$, explaining what parts of such models are not typed by $(\mathcal{M}^{\rightarrow}_{super}, \Omega_{super})$. In the example of Fig. 3, the complement $\searrow(\mathcal{M}^{\rightarrow}_{sm}, \mathcal{M}^{\rightarrow}_{g}, \prec^{s})$ of the subtype metamodel shows that the references `initial` and `final` are not covered by the supertype metamodel in $synth(\mathcal{M}^{\rightarrow}_{sm}, \mathcal{M}^{\rightarrow}_{g}, \prec^{s})$, which is useful for selecting a specific strict subtyping.

On the other hand, when in the complement of the supertype, we have that $|\nearrow(\mathcal{M}^{\rightarrow}_{sub}, \mathcal{M}^{\rightarrow}_{super}, \prec)| \neq 0$, then $\mathcal{M}^{\rightarrow}_{sub}$ is not a subtype of $\mathcal{M}^{\rightarrow}_{super}$. To improve the reuse of model management operations in this situation, we can compute the effective metamodel of the model management operation inducing a more general model type, removing those OCL constraints that become redundant.

7 Related Work

Steel et al. [31] formalized the notion of model type as a type group, generalizing the notion of object typing. Ours is formalized as a union type instead, enabling the possibility of having heterogeneous root objects in a model. In [19], as the isomorphic subtyping relation based on model type matching is too strict, a non-isomorphic subtyping relation based on adaptation was introduced in order to facilitate more flexible reuse of model management operations, such as class renamings. Our subtyping relation approach can, however, work both for nominal and for structural type systems providing support for metamodels. This means that subtyping is not simply isomorphic as it works modulo class renaming for any possible renaming, which need not be known a priori.

Based on the notion of model subtyping in [31], Sen et al. [30] presented a mechanism to reuse model transformations by means of their effective metamodel $\mathcal{M}^{\rightarrow}_{in}$. A model transformation is then made reusable by *manually* weaving aspects of the effective metamodel $\mathcal{M}^{\rightarrow}_{in}$ with the metamodel $\mathcal{M}^{\rightarrow}_{actual}$ for which the transformation is to be applied so that $\mathcal{M}^{\rightarrow}_{actual}$ becomes a subtype of $\mathcal{M}^{\rightarrow}_{in}$. In our approach, the extension metamodel $\mathcal{M}^{\rightarrow}_{actual,in}$ that facilitates the satisfiability of the model subtyping relation between two metamodel specifications corresponds to the weaved metamodel that enables the reuse of the transformation. However, in our case, the extension metamodel is synthesized *automatically* and OCL constraints are considered.

Regarding as-is reuse of model transformations, Lara et al. proposed a-posteriori typing specifications [14] for decoupling the creation interpretation of a metamodel, when regarded as a *template* for new models, from their role interpretation, when regarded as a *classifier* for existing models. These specifications can be applied both to the type level and to the instance level, providing support for multiple, partial and dynamic typing. Bidirectional typing specifications are formalized in [13], providing support for backward compatibility to the original metamodel when reusing another operation. Our synthesis of the extension metamodel resembles the construction of the analysis metamodel for reasoning about OCL constraints in [13]. An important difference relies in the inference of inheritance relationships. Moreover, when there are name clashes in features from the resulting superclasses and subclasses, they rename the features in the superclass and add an additional OCL constraint to enforce that their values are the same. We solve this problem, by applying a refactoring that preserves the type of the feature in the subtype, which may be more restrictive, both avoiding duplicate code and skipping the use of auxiliary OCL constraints. Their approach cannot solve the problem described in the introduction as typing specifications are defined explicitly and require knowledge about the involved metamodel specifications a priori. The motivation behind our approach focusses on checking subtyping automatically without up-front manual intervention, leaving out explicit adaptation. In our case, re-typing is implicitly handled through the model subtyping relation and explicit re-typings (coercion) are only required when reusing EMF model management operations, as explained in section 6.3. On the other hand, our approach is framed at the type level.

Varró et al. also use ontological typing of objects in VPM [33] for defining metamodel conformant models. More interestingly, they provide a refinement calculus that handles (multiple) inheritance and instantiation, which provides a subsumption relationship between metamodels. Although they consider refinement of behavioural specifications, they can only reason with a subset of the metamodels expressible in EMF as they do not consider well-formedness constraints.

Graph constraints, introduced by Heckel et al. in [21] for plain graphs, and later extended by Taentzer et al. to graphs with type node inheritance in [32], capture a notion of metamodel specification that is similar to the one that we are dealing with. Propagation of constraints along model transformations, which can be regarded as adaptations, specified by triple graph grammars is considered by Harmut in [17]. Constraint propagation determines when a target metamodel, which can have its own constraints, is compatible with respect to the constraints of the source metamodel. Our proposal does not require an explicitly defined up-front mapping and the underlying theory is available in a tool.

Zschaler [35] introduced an abstract notion of model type with constraints as a graph, with class names as nodes and associations as edges, coupled with a conjunction of first-order constraints over the graph signature. A model matching relation is used to match a metamodel to its model type and a type system is provided to infer the minimal model types of a model management operation. The constraints that can be inferred refer to multiplicity bounds and to whether a class is abstract or not. When a model management operation is invoked over a model, the type system checks that the model satisfies the constraints of the model type associated with the parameter. Thus, model subtyping is implicitly taken care of by constraint satisfaction allowing for reuse. The inferred minimal model type is similar, in its intent, to the model type of an effective metamodel, which is assumed to be inferred from a model transformation, in our approach. On the other hand, the notion of model type has not been discussed independently of type inference for a model management operation and is, then, tied to the language used for defining model management operations, which is, at present, less expressive than those used in practice.

None of the approaches researched above consider reasoning on metamodels, which are not related a priori, enriched with OCL constraints in model subtyping.

8 Concluding Remarks

The notion of model subtyping has been revisited from a subsumption perspective. Structural model subtyping has been extended to metamodel specifications as the problem of whether two groups of OCL constraints are compatible when their metamodels (or class diagrams) are not related a priori. We have implemented a procedure for solving this problem that, in addition, gives us: an extension metamodel for reusing model management operations whose signature involves the supertype metamodel when the metamodel specifications are compatible; and the reason why their metamodels are not subtype of each other, or else a metamodel-conformant model that satisfies the constraints of the subtype but not some of the supertype, when they are not compatible. The procedure is decidable when appropriate bounds are used for finding the witness model.

Structural model subtyping can be used for developing metamodel specification refinements without losing expressive power with respect to other approaches to subtyping. We have shown that it is powerful enough so as to provide some adaptations for free without having to provide any explicit information relating metamodel specifications a priori, becoming both isomorphic and non-isomorphic [19]. We have also discussed that structural model subtyping provides a convenient theoretical framework to deal with dynamic, partial and multiple model typing. Nevertheless, structural model subtyping suffers from being too liberal in contexts where all the object subtypings need to be analyzed. To mitigate this threat to usability, our tool provides a recommendation algorithm that suggests an optimal strict model subtyping. The inclusion of a more prescriptive approach that limits the amount of valid object subtypings or that dictates how subtypings should be performed with user information provided up front is a potential extension.

Acknowledgments

The author thanks the anonymous reviewers for their insightful and helpful feedback. This work was partially supported by InnovateUK KTP 10567.

References

[1] 2010. Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (Dec 2010), 1–418. https://doi.org/10.1109/IEEESTD.2010.5733835

[2] Martín Abadi and Luca Cardelli. 1996. *A Theory of Objects.* Springer.

[3] Artur Boronat. 2007. *MOMENT: a formal framework for MOdel manageMENT.* PhD thesis. Universitat Politècnica de València (UPV), Spain.

[4] Artur Boronat and José Meseguer. 2009. Algebraic Semantics of OCL-constrained Metamodel Specifications. In *TOOLS (47).* LNBIP 33, 96–115.

[5] Artur Boronat and José Meseguer. 2010. An Algebraic Semantics for MOF. *Formal Aspects of Computing* 22 (2010), 269–296. Issue 3.

[6] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers.

[7] Kim B. Bruce and Joseph C. Vanderwaart. 1999. Semantics-Driven Language Design:. *ENTCS* 20 (1999), 50 – 75.

[8] Luca Cardelli. 1988. A Semantics of Multiple Inheritance. *Inf. Comput.* 76, 2/3 (1988), 138–164.

[9] Marsha Chechik, Michalis Famelis, Rick Salay, and Daniel Strüber. 2016. Perspectives of Model Transformation Reuse. 9681 (2016), 28–44.

[10] Tony Clark. 1999. Type Checking UML Static Diagrams. 1723 (1999), 503–517.

[11] Tony Clark, Andy Evans, and Stuart Kent. 2001. The Metamodelling Language Calculus: Foundation Semantics for UML. 2029 (2001), 17–31.

[12] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2007. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.* 376, 3 (2007), 139–163.

[13] Juan de Lara and Esther Guerra. 2017. *A Posteriori* Typing for Model-Driven Engineering: Concepts, Analysis, and Applications. *ACM Trans. Softw. Eng. Methodol.* 25, 4 (2017), 31:1–31:60.

[14] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2015. A-posteriori typing for Model-Driven Engineering. (2015), 156–165.

[15] Juan de Lara, Juri Di Rocco, Davide Di Ruscio, Esther Guerra, Ludovico Iovino, Alfonso Pierantonio, and Jesús Sánchez Cuadrado. 2017. Reusing Model Transformations Through Typing Requirements Models. 10202 (2017), 264–282.

[16] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation.* Springer.

[17] Hartmut Ehrig, Frank Hermann, Hanna Schölzel, and Christoph Brandt. 2011. Propagation of Constraints along Model Transformations Based on Triple Graph Grammars. *ECEASST* 41 (2011).

[18] Martin Gogolla, Jørn Bohling, and Mark Richters. 2005. Validating UML and OCL models in USE by automatic snapshot generation. *Software & Systems Modeling* 4, 4 (2005), 386–398.

[19] Clement Guy, Benoît Combemale, Steven Derrien, Jim Steel, and Jean-Marc Jézéquel. 2012. On Model Subtyping. In *ECMFA 2012.* 400–415.

[20] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer* 37, 10 (2004), 64–72.

[21] Reiko Heckel and Annika Wagner. 1995. Ensuring Consistency of Conditional Graph Grammars - A Constructive Approach -. *ENTCS* 2 (1995), 118 – 126.

[22] Mirco Kuhlmann and Martin Gogolla. 2012. From UML and OCL to Relational Logic and Back. In *MODELS*, Vol. 7590. LNCS Springer, 415–431.

[23] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. 2011. Extensive Validation of OCL Models by Integrating SAT Solving into USE. 6705 (2011), 290–306.

[24] Thomas Kühne. 2006. Matters of (Meta-)Modeling. *Software and System Modeling* 5, 4 (2006), 369–385.

[25] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. 2015. Reuse in model-to-model transformation languages: are we there yet? *Software and System Modeling* 14, 2 (2015), 537–572.

[26] Iman Poernomo. 2006. The Meta-Object Facility Typed. In *SAC.* ACM, 1845–1849.

[27] Mark Richters. 2002. *A precise approach to validating UML models and OCL constraints.* Ph.D. Dissertation. University of Bremen, Germany.

[28] Mark Richters and Martin Gogolla. 2002. OCL: Syntax, Semantics, and Tools. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language.* Vol. 2263. LNCS, 42–68.

[29] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. 2009. Meta-model Pruning. In *MoDELS 2009.* 32–46.

[30] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jézéquel. 2012. Reusable model transformations. *Software and System Modeling* 11, 1 (2012), 111–125.

[31] Jim Steel and Jean-Marc Jézéquel. 2007. On model typing. *Software and System Modeling* 6, 4 (2007), 401–413.

[32] Gabriele Taentzer and Arend Rensink. 2005. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In *FASE*, Vol. 3442. LNCS, 64–79.

[33] Dániel Varró and András Pataricza. 2004. Generic and Meta-transformations for Model Transformation Engineering. 3273 (2004), 290–304.

[34] Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière. 2013. Typing artifacts in megamodeling. *Software and System Modeling* 12, 1 (2013), 105–119.

[35] Steffen Zschaler. 2014. Towards Constraint-Based Model Types: A Generalised Formal Foundation for Model Genericity. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO '14).* Article 11, 8 pages.