A-posteriori Typing for Model-Driven Engineering

Juan de Lara, Esther Guerra, Jesús Sánchez Cuadrado
Universidad Autónoma de Madrid

***Abstract*—Model-Driven Engineering is founded on the ability to create and process models conformant to a meta-model. Hence, meta-model classes are used in two ways: as templates to create objects, and as classifiers for them. While these two aspects are inherently tied in most meta-modelling approaches, in this paper, we discuss the benefits of their decoupling. Thus, we rely on standard mechanisms for object creation and propose *a-posteriori* typing as a means to reclassify objects and enable multiple, partial, dynamic typings. This approach enhances flexibility, permitting unanticipated reutilization (as existing model management operations defined for a meta-model can be reused with other models once they get reclassified), as well as model transformation by reclassification. We show the underlying theory behind the introduced concepts, and illustrate its applicability using our METADEPTH meta-modelling tool.**

***Index Terms*—A-posteriori typing, Model typing, Partial typing, Dynamic typing, Flexible MDE**

I. INTRODUCTION

Model-Driven Engineering (MDE) has traditionally promoted a "top-down" approach, where classes are used to create instances, which in their turn are classified by those classes. This kind of typing is called *constructive* [1], because classes are used both to create and classify instances, and both aspects (creation and classification) cannot be separated.

Constructive typing is mainstream but lacks flexibility. For example, the SMOF standard [19] discusses the rigidity of MOF to model objects that need to change their type dynamically without losing their identity (e.g., a conference system where a Student becomes Professor), or to represent objects holding several classifiers (e.g., a person that is classified as both Author and Reviewer when he has authored and reviewed articles). The unmodifiability of classifiers also hinders reuse. This is so as, to reuse an operation (e.g., a model transformation) defined over a meta-model MM_A for another meta-model MM_B, the usual solution is to transform the instances of MM_B into MM_A. Alternatively, the operation could be rewritten in terms of MM_B. However, neither alternative is fully satisfactory. The first one is heavyweight, difficulting traceability w.r.t. the original model. The second is costly and error-prone. Instead, being able to reclassify instances of MM_B as instances of MM_A would simplify the problem.

Decoupling typing from instantiation is a well-known technique to promote reuse and ease the adaptation of existing code in object-oriented programming. For example, in Java, objects are created by constructors and get classified by the classes used to create them. However, there are additional mechanisms (like interfaces) which allow focussing on a subset of properties that objects require in order to achieve certain functionality. Hence, interfaces decouple classification from the creation type, permit several classifiers for an object, and enable reusability. Dynamic reclassification has also been realized in some object-oriented languages [9] to allow objects to change their class membership at runtime, which decouples even further classification from creation. In contrast, most MDE approaches use static constructive typing, which results in more restricted possibilities for modelling and reuse.

In this paper, we enable reclassification by separating the creation and classification types of objects, where already created instances can be assigned additional types for classification, and the type may change at runtime. Our aim is to provide a more flexible typing in MDE, which becomes multiple, partial, and dynamic. For this purpose, we define an *a-posteriori* typing that permits classifying objects by classes different from the ones used to create the objects. A consequence of this approach is that model management operations become highly reusable as, similar to Java interfaces, we can design meta-models whose primary goal is not object creation, but to serve as a type for model management operations.

We provide two ways to specify a-posteriori typings: at the type and at the instance level. The former induces a static relation between two meta-models, so that instances of one can be seen as instances of the other. This is similar to the implements relation between Java classes and interfaces. The second possibility allows classifying particular objects by defining queries assigning a given type to the result of the query. This typing is dynamic because classification may depend on the runtime values of objects, and therefore change whenever such values evolve. The first kind of typing is just a special case of the second. Moreover, we present a set of techniques to analyse dynamic type safety and type-level reclassification. As a proof-of-concept, we show an implementation in our METADEPTH [5] tool, and discuss several applications.

Organization. Sec. II overviews typing alternatives in MDE. Sec. III presents our two ways to specify a-posteriori typings. Sec. IV describes some analysis possibilities for typing specifications. Sec. V describes tool support, and Sec. VI shows examples and applications of a-posteriori typing. Sec. VII discusses related work, and Sec. VIII concludes.

II. A PANORAMA OF TYPINGS FOR MDE

Fig. 1 summarizes several alternatives regarding typing that meta-modelling approaches may adopt, namely:

- *Classification time*. Object classification can be determined when the object is created (constructive typing), or new classifiers can be added later (a-posteriori typing). In constructive typing, types are used to create instances, and the type of an instance is only the type that was used to create it. Hence,

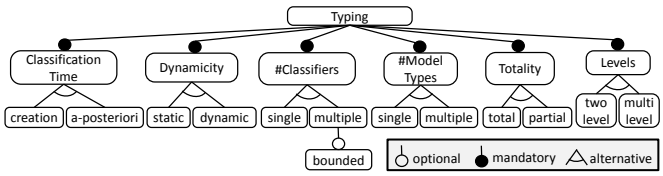

Fig. 1: Alternatives for typing

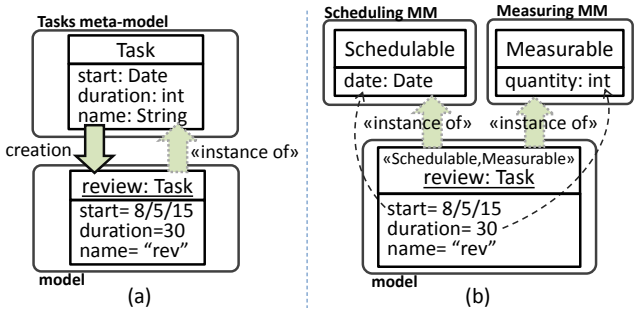

Fig. 2: (a) Constructive typing. (b) A-posteriori typing

creation and classification are inseparable. This is the usual approach in MDE. Fig. 2(a) shows a constructive typing example where class Task is used to create object review, becoming its only classifier.

Instead, in a-posteriori typing, object creation and classification are separated, and objects may have other types besides the constructive types. The a-posteriori types do not need to be assigned statically when the creation class is defined, as is the case with interfaces, but they can be added on demand. Fig. 2(b) shows an example, where the previously created object review is assigned the types Schedulable and Measurable (shown as stereotypes) from two different meta-models. In this way, the *model* has a constructive typing w.r.t. the Tasks meta-model, and two a-posteriori typings w.r.t. the Scheduling and Measuring meta-models. A-posteriori typings are proper typings; therefore, any model management operation defined over Scheduling or Measuring gets applicable on the model.

- *Dynamicity*. The type of an object may be unmodifiable (static), or it may change over time as the object evolves (dynamic). The latter is useful if we need to classify an object according to its properties. For instance, the bottom-left model in Fig. 3 has the Tasks meta-model as constructive type, and is typed a-posteriori w.r.t. the upper right meta-model for conference reviewer assignment. The a-posteriori typing classifies each Person object as Author if he is owner of some article resource, or as Reviewer if he is assigned a task with name "rev". When the model evolves (bottom-right), the new review task t2 is assigned to person p2, and hence, he gets classified as Reviewer. While constructive typing is inherently static, a-posteriori typing can be dynamic.
- *Number of classifiers*. Some type systems may allow several classifiers, none subtype of the others, to share common instances. Hence, some objects may receive multiple classifiers from the same meta-model. Constructive typing does not support this but a-posteriori typing may enable this feature.

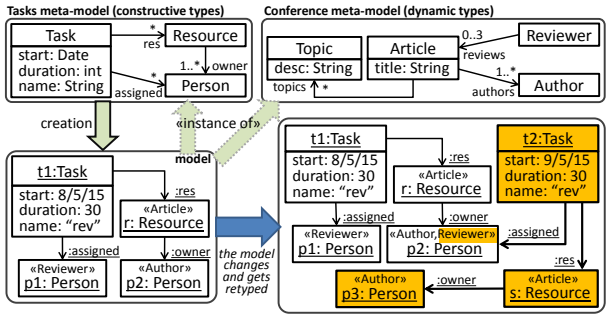

Fig. 3: Dynamic typing, multiple classifiers for objects

For example, in the bottom-right model of Fig. 3, p2 is typed a-posteriori as Author and Reviewer. Some systems like SMOF permit declaring the set of potential classifiers that instances of a given type can adopt (feature bounded in Fig. 1). The UML [18] also supports overlapping instances through annotation overlapping on generalization sets.

- *Number of model types*. Besides the constructive type, a model can be typed a-posteriori by 0 or more meta-models. For example, the model in Fig. 2(b) is typed a-posteriori by both meta-models Scheduling and Measuring.
- *Totality*. Constructive typing is total, as instances always receive a type from the instantiated meta-model. In contrast, a-posteriori typing can be partial if it is allowed to have model elements (objects, links or fields) without an a-posteriori type. For example, Task instances in Fig. 3 lack an a-posteriori type. Similarly, the model in Fig. 2(b) is partially typed w.r.t. the Scheduling meta-model because fields duration and name aren not typed by this meta-model.
- *Levels of typing*. In standard frameworks, like EMF, the workspace only manages two meta-levels at a time (meta-models and models). We call them two-level. Instead, multi-level approaches [7] permit working with models at any number of meta-levels simultaneously, and the types defined in a meta-level can influence the instances several meta-levels below (instead of just the ones at the next level).

We can classify existing meta-modelling approaches based on the previous features. For example, the typing in MOF and EMF is constructive, static, total, allows a single classifier for objects, a single meta-model to type a model, and is two-level. SMOF is more flexible, as it supports a-posteriori, dynamic, total typings, as well as assigning (bounded) multiple classifiers to objects, and typing models by a single meta-model. As we will see later, our modelling tool METADEPTH enables a-posteriori, dynamic, partial typings, while objects can have multiple a-posteriori classifiers, models can be typed by several meta-models, and supports multi-level modelling.

III. A-POSTERIORI TYPING

In this section, we examine two ways to specify a-posteriori typings: at the type and at the instance levels. The former is a particular case of the latter, but it provides a concise specification mechanism and facilitates analysis (see Sec. IV-B).

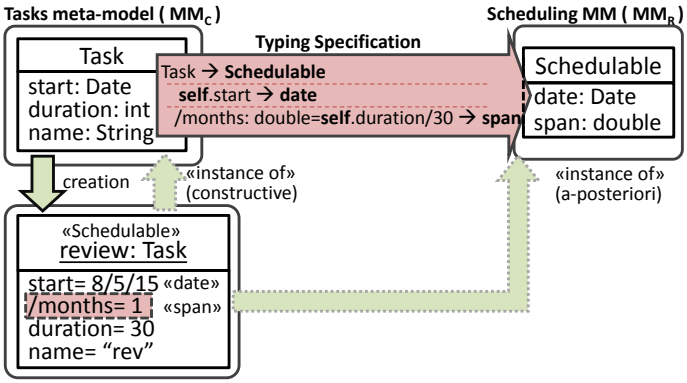

Fig. 4: Specification of a-posteriori typing at the type-level

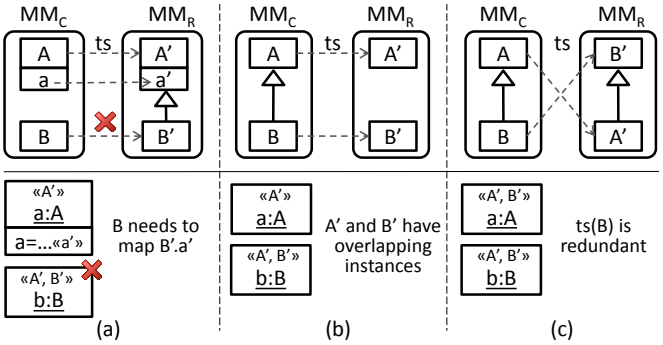

Fig. 5: Type-level specification examples. (a) Incorrect typing: missing binding for mandatory feature B'.a'. (b) A-posteriori types with overlapping instances. (c) Redundant specification.

A. Type-level a-posteriori typing

The specification of a-posteriori typings at the type-level is given by a (static) relation between two meta-models: a "creation" meta-model MM_C containing the constructive types, and a "role" meta-model MM_R containing the a-posteriori types. This relation maps classes, attributes and references from MM_C to those of MM_R. Fig. 4 shows an example. The typing specification maps class Task to Schedulable, attribute start to date, and it conceptually defines a derived attribute months that gets bound to span. This way, Schedulable, date and span become a-posteriori types of Task instances.

This relation is similar to the binding specification of [4], and to the model subtyping relation of [12] (but allowing overlapping classes). In the remaining of this section, we describe the features of this specification mode using an informal style to enhance comprehension.

A type-level specification is a collection of partial functions $TS = \{ts_i\}_{i \in I}$ from elements of MM_C (classes, attributes, references) to elements of MM_R. Functions are partial because not every element of MM_C needs to be mapped to an element of MM_R. It is a collection to permit elements in MM_C to be mapped to several elements in MM_R, and hence enable multiple simultaneous classifiers. The functions in the collection do not need to be jointly surjective, because some elements of MM_R might be unmapped (just like a class in a meta-model may lack instances).

Notation. We use $A \in MM_C$ for a class A belonging to MM_C. $A.a$ means that a is a feature (attribute or reference) defined in A or a superclass. We sometimes refer to features by a (without a prefix class name). $sub(A)$ is the set of subclasses of A, and $sub^*(A) = \{A\} \cup sub(A)$. $atts(A)$ and $refs(A)$ are the sets of attributes and references of A, both owned and inherited. $feats(A) = atts(A) \cup refs(A)$. $abs(A)$ is a function stating whether class A is abstract, while $mand(a)$ denotes that feature a is mandatory. Given a reference $A.r$, $tar(A.r)$ is the class r points to. Finally, $type(s)$ and $type(o)$ return the type of a slot s and an object o.

Type-level a-posteriori specifications must obey the following well-formedness rules:

1) Classes of MM_C cannot be mapped to abstract classes of MM_R: $\forall A' \in MM_R \bullet ts_i(A) = A' \implies \neg abs(A')$
2) The features of a class $A \in MM_C$ should be mapped to features of the class A is mapped to: $\forall a \in feats(A) \bullet ts_i(A.a) = a' \implies a' \in feats(ts_i(A))$
3) A feature in MM_C can only be mapped to a feature in MM_R with the same or ampler cardinality interval.
4) The target of a reference $A.r$ should be mapped to the target of the reference r is mapped to, or to a subclass: $\forall r \in refs(A) \bullet ts_i(A.r) = r' \implies ts_i(tar(A.r)) \in sub^*(tar(ts_i(A.r)))$
5) A non-composition reference in MM_C cannot be mapped to a composition reference in MM_R.
6) If a class $A' \in MM_R$ is mapped by a class of MM_C, all mandatory features of A' should be mapped as well:

$$\begin{aligned} &\forall A' \in MM_R \bullet ts_i(A) = A' \implies \\ &\quad \forall a' \in feats(A') \bullet mand(a') \implies \\ &\quad \exists a \in feats(A) \bullet ts_i(A.a) = A'.a' \end{aligned} \tag{1}$$

This condition is needed to emulate correct instantiations. Fig. 5(a) shows an example of incomplete specification, as a binding from some feature of B to B'.a' is missing.

To enhance flexibility, type-level specifications are allowed to define virtual derived features (attributes or references), to be mapped to features of MM_R. This is useful when the mapping between some aspect of MM_C and MM_R is not direct, but requires adaptation. The type-level specification in Fig. 4 illustrates this possibility, as it defines a derived attribute months, mapped to attribute span of Schedulable. A derived feature $A.da$ is defined by an expression that gets evaluated in the context of objects of type A. Derived features are mapped following the same rules as non-derived features, from a virtual meta-model extension of MM_C that incorporates all derived features defined by the a-posteriori specification.

Given a model M instance of MM_C, it is retyped according to $TS = \{ts_i\}$ (written $TS(M)$) by composing the types from MM_C and each ts_i as follows:

- An object o of type $A \in MM_C$ is retyped to $ts_i(A)$, and indirectly, to the superclasses of $ts_i(A)$. Moreover,

it inherits the a-posteriori types of the superclasses of A (see Fig. 5(b)).
- A slot or link $o.s$ of type $A.a \in MM_C$ is retyped to $ts_i(A.a)$.

It is remarkable that our typing specification does not define any condition to preserve the compatibility of inheritance hierarchies in MM_C and MM_R, in contrast with other approaches to relate meta-models [13]. This happens because we enable more flexible typings, where a-posteriori types may have overlapping instances. For example, in Fig. 5(b), two classes A and B related by subtyping get mapped to two independent classes A' and B' respectively. As a consequence, the instances of B will be typed a-posteriori by both A' (because B inherits the a-posteriori typing from A) and B'. Hence, A' and B' may have common instances. "Reversing" inheritance relations in MM_C in MM_R is not problematic either, as Fig. 5(c) shows. In this case, the mapping $ts(B) \mapsto B'$ becomes redundant because it assigns to B the same a-posteriori types that B already inherits from A (i.e., A' and B').

It is interesting to analyse the features that type-level a-posteriori specifications yield:

- Objects with different creation type may have the same a-posteriori type if their classes are mapped to the same class in MM_R. That is, if given two classes A and B of MM_C, we have that $ts_i(A) = ts_j(B)$. In a weaker version, A and B become related if $ts_i(A)$ and $ts_j(B)$, or A and B, share a common ancestor class.
- Objects with same creation type cannot have different a-posteriori types. This is not possible because each ts_i maps classes to classes, and is not able to select a subset of the instances of a class.
- Objects whose creation type and its supertypes are not mapped to a role meta-model class, lack an a-posteriori type. This is possible because each ts_i can be partial.
- Objects may have several a-posteriori types (i.e., types in the role meta-model can have overlapping instances). This is so if ts_i, ts_j map a class $A \in MM_C$ to several classes in MM_R. As above mentioned, another source of overlapping instances is the existence of inheritance relationships between the classes that participate in the typing specification (see Fig. 5(b)). Constraining the specification TS to be a single function instead of a collection, and adding an additional rule to preserve the compatibility of inheritance hierarchies in MM_C and MM_R, would yield type-level specifications where a-posteriori types do not have overlapping instances. Restricting our typing specification to be non-overlapping and not multiple, yields the meta-model relations defined in [13]. Implementations may restrict overlapping to occur only between selected annotated classes of MM_R, as UML and SMOF do.
- The defined a-posteriori typing is not dynamic. This means that the assigned a-posteriori types do not change when the model evolves. Although one could define additional a-posteriori types, the existing ones do not change. This is so because the type assigned by TS depends statically on a class from MM_C.

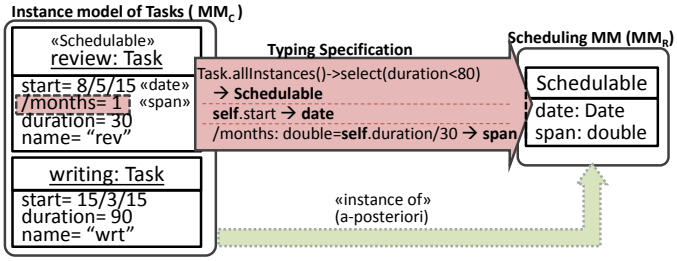

Fig. 6: Specification of a-posteriori typing at the instance-level

Further analysis of retypings will be given in Section IV.

Once an instance model M of MM_C is retyped by TS to MM_R, we can manipulate M via operations defined over MM_R. Retyping M by TS produces a virtual view of M, which is seen as an instance of MM_R. Every read operation defined over MM_R is safe over $TS(M)$. Writing a feature $A'.a' \in MM_R$ is only possible if the source feature $A.a \in MM_C$ such that $ts(A.a) = A'.a'$ is not derived. For multi-valued features (i.e., with upper cardinality bigger than 1), adding an element is safe if the source feature in MM_C has the same cardinality. For writing, compositions in MM_C can only be mapped to compositions in MM_R. New objects of type $A' \in MM_R$ can be created by an operation defined over MM_R if A' is not mapped from two different classes of MM_C. If it receives more than one mapping, it is not possible to know which class in MM_C should be instantiated. Moreover, if the class $A \in MM_C$ such that $ts(A) = A'$ has unmapped features, the operation defined over MM_R will not be able to initialize them when creating objects of type A'.

Next, we present a more expressive approach to specify a-posteriori typings at the instance level, and show how to translate type-level specifications into them.

B. Instance-level a-posteriori typing

The specification of a-posteriori typings at the instance-level consists of queries that are evaluated over the model to be typed, and their results are assigned types from the role meta-model. Fig. 6 shows an example of instance-level specification, where the first two lines assign the a-posteriori type Schedulable to all tasks with duration less than 80, and therefore, object review gets this type a-posteriori, but not object writing.

An instance-level a-posteriori typing specification from a creation meta-model MM_C to a role meta-model MM_R is a collection of partial functions $IS = \{is_i\}_{i \in I}$ from instances of MM_C (objects, slots and links) to elements of MM_R (classes, attributes and references). It must fulfil the same well-formedness rules as type-level specifications, though at the instance-level, with some particularities we discuss next.

In particular, function $is_i(oset_j) \mapsto C_j$ maps sets of objects from a model M instance of MM_C, to classes in MM_R. The set of objects can be obtained in any way; in this paper, we assume they are gathered using OCL expressions. We write

$exp_i(M) \mapsto oset_i$ to denote a function taking a model M of MM_C and returning a set $oset_i$ of its objects. Like in type-level specifications, if $is_i(oset_j) \mapsto C_j$, then objects in $oset_j$ are assigned type C_j and every supertype of C_j.

Slots have to be mapped in the context of objects. Therefore, to map slots, is_i takes two arguments $is_i(oset_j, s) \mapsto C_j.f_k$. This means that slot s in every object of $oset_j$ is mapped to attribute $C_j.f_k$. The mapped slot s should be valid in every object of $oset_j$: $\forall o \in oset_j \bullet type(o.s) \in atts(type(o))$. Such slots should be compatible with the class $oset_j$ is mapped to. That is, if $is_i(oset_j) = C_j$, then $is_i(oset_j, s) = C_j.f_k$, where $f_k \in atts(C_j)$. This corresponds to well-formedness rule 2 of type-level specifications.

Links are mapped similarly. If we have $is_i(oset_j, l) \mapsto C_j.r_k$, then we require l to be valid in every object in $oset_j$ and $is_i(oset_j) = C_j$. Moreover, the objects that result from evaluating l in every object of $oset_j$ should be compatible with $tar(C_j.r_k)$. More formally: $oset_j \rightarrow collect(o|o.l) \subseteq is_i^{-1*}(tar(C_j.r_k))$, where $is_i^{-1*}(A)$ is the set $\bigcup_{is_i(oset_j) \in sub^*(A)} is_i(oset_j)$. The equivalent rule at the type level is stronger (rule 4), as it demands compatibility between the type of the link and the mapped reference $tar(C_j.r_k)$. At the instance level, the condition is relaxed to consider the actual type of the objects in set $oset_j$.

Similar to the type-level case, mandatory features in MM_R should get a mapping from some slot in IS:

$$\begin{aligned} &\forall A' \in MM_R \bullet is_i(oset_j) = A' \implies \\ &\quad \forall a' \in feats(A') \bullet mand(a') \implies \\ &\quad \exists\ s \bullet is_i(oset_j, s) = A'.a' \end{aligned} \tag{2}$$

Finally, instance-level specifications allow the specification and mapping of derived features, which should obey the same well-formedness rules as non-derived features.

Instance-level specifications are more expressive than the type-level ones, leading to typings with the following features:

- Objects with different creation type may have the same a-posteriori type, if these objects are selected by queries mapped to the same class in the role meta-model.
- Objects with same creation type may have different a-posteriori types. This is possible if several queries select objects with same creation type, but they are mapped to different classes in the role meta-model.
- Objects that are not selected by any query lack an a-posteriori type. This is possible because the defined a-posteriori typing can be partial.
- Objects may have several a-posteriori types, if selected by various queries and mapped to different MM_R classes.
- The defined a-posteriori typing can be dynamic. This is possible if an object changes its attribute values in such a way that it becomes selected or deselected by different queries. For example, in Fig. 6, modifying writing.duration to 30 makes writing Schedulable, while setting review.duration to 81 makes review to drop its Schedulable type.

Any type-level specification $TS = \{ts_i\}_{i \in I}$ can be translated to an instance-level specification $IS = \{is_i\}_{i \in I}$, building the mappings $is_i \in IS$ from every $ts_i \in TS$ as follows:

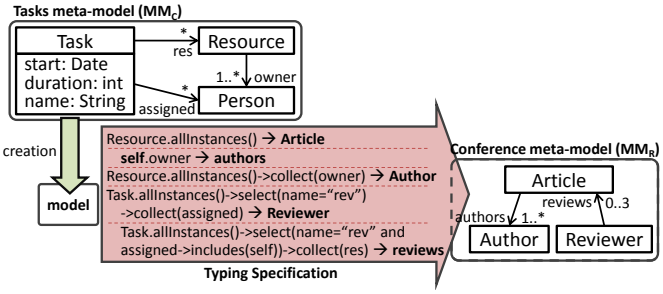

Fig. 7: Specification of a-posteriori typing at the instance-level

- $\forall ts_i(A) = A'$, set $is_i(A.allInstances()) \mapsto A$.
- $\forall ts_i(A.f) = A'.f'$, set $is_i(A.allInstances(), f) \mapsto A'.a'$.

A well-formed type-level specification TS yields a well-formed instance-level specification IS. Equation 2 follows almost directly from equation 1. This is so as, if $A' \in MM_R$ and $mand(A'.a')$, then, by equation 1, we have that $\exists a \in feats(A) \bullet ts_i(A.a) = A'.a'$ and $ts_i(A) = A'$. This means that we have created the mappings $is_i(A.allInstances()) \mapsto A'$ and $is(A.allInstances(), a) \mapsto A'.a'$, satisfying equation 2.

IV. Analysis of a-posteriori typing

In this section, we present a number of analysis techniques for a-posteriori typing specifications which are based on the use of constraint solving and model finders [15].

A. Analysing dynamic type safety

Dynamic typing may lead to undesired model views, where the a-posteriori typing is unsafe. For example, if a reference should contain objects with a certain a-posteriori type, and an object in the reference changes its type but the a-posteriori type of the reference is not updated, then, the reference will contain an object with an unwanted a-posteriori type. As a rule of thumb, reclassification may be unsafe for objects pointed by references. Some programming languages bound the classes an object may be retyped to, and forbid references to such classes [9]. Instead, we allow more flexibility, and provide an analysis mechanism to detect unsafe typings.

As an example, assume the instance-level a-posteriori typing specification in Fig. 7. The specification classifies all resources in a model as Articles, their owners as Authors, and all Persons assigned a task named "rev" as Reviewers.

Here, we would like to analyse whether there might be Tasks models with incoherent a-posteriori type. This would be the case if references authors or reviews can contain objects with wrong type. For the case of authors, we need to check that there cannot be articles that contain in its reference authors an object which is not typed as Author. This can be proved by checking if the following OCL constraint is unsatisfiable:

```
1 Article.allInstances()->exists(a |
2     a.authors->exist(r |
3         Author.allInstances()->excludes(r)))
```

However, our interest is on analysing tasks models. Hence, the previous constraint is translated in terms of constructive types. For this, we use the typing specification as it defines

the query over constructive types that corresponds to each a-posteriori type. This way, analysing the dynamic safety of authors amounts to checking the following constraint is not satisfiable on MM_C:

```
1 Resource.allInstances()->exists(a | -- Resource is mapped to Article
2   a.owner->exists (r |                -- owner is mapped to authors
3     Resource.allInstances()           -- Resource.owner is mapped to Author
4       ->collect(owner)->excludes(r)))
```

Listing 1: Checking type safety for authors.

The algorithm to build these constraints is as follows. Given a reference $A.r$ with $tar(r) = B$ in MM_R; and given an instance-level specification with $is_i(exp_A) = A$, $is_i(exp_A, exp_r) = A.r$ and $is_{i_1}(exp_{B_1}) = B$, ..., $is_{i_n}(exp_{B_n}) = B$; we build the constraint:

exp_A->exists(a | exp_r[a/self] -> exists(r |
exp_{B_1}->union(exp_{B_2})...->union(exp_{B_n})->excludes(r)))

where $exp_r[a/self]$ is the expression exp_r substituting $self$ by a. If there are several mappings $is_{i_1}(exp_{A_1})$,..., $is_{i_m}(exp_{A_m})$ assigned to A, we build the previous constraint for each such mapping. If any such constraint is satisfiable, the typing is not safe.

Correctness of cardinalities is analysed similarly, by checking the unsatisfiability of constraints expressed over MM_R (e.g., Reviewer.allInstances()->exists(r | r.reviews->size() > 3)), once they have been translated into constructive types, like:

```
1 Task.allInstances()->select(name="rev")->collect(assigned)->exists(r |
2     Task.allInstances()->select(name="rev" and
3     assigned->includes(r))->collect(res)->size() > 3)
```

Listing 2: Checking cardinality constraints.

Hence, given a reference $A.r$ in MM_R with cardinality [$\min_r$, $\max_r$]; and given a specification with $is_i(exp_A) = A$ and $is_i(exp_A, exp_r) = A.r$; we build the constraint:

exp_A->exists(a | exp_r[a/self] -> size() > $\max_r$) or
exp_A->exists(a | exp_r[a/self] -> size() < $\min_r$)

with the first term only needed if $\max_r \neq *$ and the second if $\min_r > 0$. If this expression is satisfiable, the cardinality of r might be violated. If A receives multiple mappings, then the constraint has to be checked for all of them.

Please note that we only need to do this analysis once, when the a-posteriori typing is specified.

B. Analysing type-level reclassification

Meta-models may include OCL constraints. While our typing specifications have well-formedness rules ensuring correct retyping, these OCL constraints may impose additional restrictions. Hence, it is interesting to analyse whether, given a meta-model MM_C and a type-level specification w.r.t. a meta-model MM_R, some/every valid instance of MM_C becomes a valid instance of MM_R when the retyping is performed.

As Fig. 8(a) shows, retyping a model M_C w.r.t. MM_R creates a virtual model view M_R of M_C. This view discards the elements of M_C which are not typed by MM_R and includes the derived features. Thus, the goal of the analysis is to check whether: (a) a valid view model M_R for some M_C exists (*reclassification executability*); (b) a valid view model

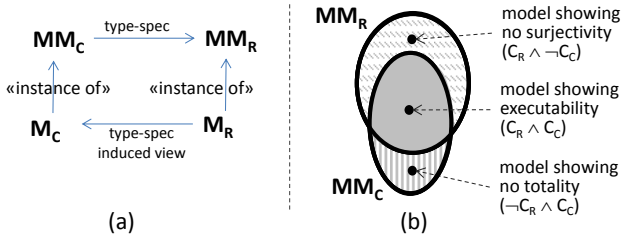

Fig. 8: Totality and surjectivity of model reclassification

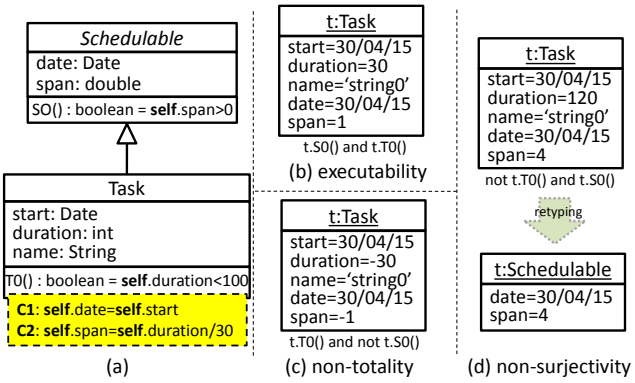

Fig. 9: (a) Merged meta-model for analysis. (b,c,d) Models showing executability, non-totality and non-surjectivity.

M_R for every model M_C exists (*reclassification totality*); (c) every possible instance of MM_R is view of some instance of MM_C (*reclassification surjectivity*). Fig. 8(b) illustrates these possibilities using sets depicting the model instances of MM_C and MM_R, dots representing models, and C_C and C_R being the sets of constraints in MM_C and MM_R respectively.

These properties are interesting for both type- and instance-level specifications, but we present the method for type-level specifications, leaving instance specification as future work.

Our analysis is based on the use of model finders (able to find model instances of a given meta-model) and is inspired by the techniques in [11]. The idea is to merge MM_C and MM_R, substituting every mapping from classes in MM_C to classes in MM_R by inheritance, making every class in MM_R abstract, removing every reference of MM_R, encoding constraints in both meta-models as boolean operations, and encoding derived attributes and mappings of features as constraints.

Fig. 9(a) shows the encoding of the type-level specification in Fig. 4. We assume that class Schedulable defines the constraint span>0, and Task defines the constraint duration<100. These constraints get encoded as operations S0 and T0. Constraint C1 in Task is generated due to the binding start → date, while C2 is generated due to the binding months → span.

Executability is proved by finding an instantiation of the merged meta-model where all Task instances yield S0()=true and T0()=true, meaning that the constraints in MM_C and MM_R hold. Thus, we have to find a model such that Task.allInstances()→forAll(t | t.S0()=true and t.T0()=true). Non-totality is proved by finding a model where all tasks fulfil $T0$, and some violates $S0$. This emulates the satisfaction of all

constraints in MM_C, while some constraint in MM_R is violated. Thus, we require: Task.allInstances()→exists(t | t.S0()=false) and Task.allInstances()→forAll(t | t.T0()=true). Conversely, non-surjectivity is proved by finding a model where all constraints of MM_R hold, and some of MM_C is violated. If such a model exists, it must be retyped w.r.t. MM_R. Figs. 9(b,c,d) show three witness models showing executability, non-totality and non-surjectivity of the analysed specification.

V. Tool support

In this section, we show an implementation of the previous concepts in MetaDepth [5]. MetaDepth supports multi-level modelling and integrates the Epsilon languages [21] for defining constraints, transformations and code generators. For this work, we have extended the tool with the possibility to specify a-posteriori typings and perform analysis.

A. Models and meta-models in MetaDepth

Models and meta-models are specified textually. As MetaDepth supports multi-level modelling, elements may be decorated with a potency (written after the '@' symbol) stating at how many meta-levels the element can be instantiated. In two-level modelling, meta-models have potency 1 and models have potency 0. Listing 3 shows part of a Tasks meta-model in lines 1–8, where Task is extended by a subclass DocTask. Lines 10–21 show a simple model.

```
1  Model Tasks {
2    Node Task {
3      start : Date;
4      duration : int;
5      name : String;
6    }
7    Node DocTask : Task{}
8  }
9
10 Tasks someTasks {
11   Task t0 {
12     start = "30/04/2015";
13     duration = 30;
14     name = "coding";
15   }
16   DocTask t1 {
17     start = "30/05/2015";
18     duration = 90;
19     name = "write manual";
20   }
21 }
```

Listing 3: Meta-model and example model in MetaDepth.

B. Specification of a-posteriori typings

MetaDepth permits specifying both type-level and instance-level typings. Instance-level specifications are given by mapping queries written in the Epsilon Object Language (EOL, a variant of OCL) [14] to types. As an example, Listing 4 shows the specification in Fig. 6. Line 2 maps the instances of Task with duration less than 80, to type Schedulable (all is an abbreviation for allInstances). The keyword with sets the context of the following mappings to the objects selected by the previous query. The computations of derived attributes (like months in line 3) are also expressed in EOL. Every EOL expression is enclosed between '$'. Fig. 10(a) shows a schema of the different relations between the models involved in instance-level specifications, with '@' denoting potency.

```
1 type Tasks Scheduling inst {
2     $Task.all.select(x | x.duration < 80)$ > Schedulable
3         with { /months : double = $self.duration/30$ > span,
4                start > date}}
```

Listing 4: Instance-level a-posteriori typing in MetaDepth

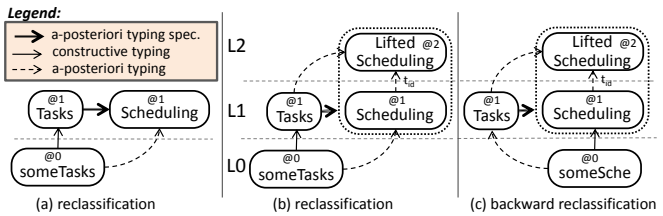

Fig. 10: A-posteriori typing specifications in MetaDepth: (a) at the instance level; (b,c) at the type level

A-posteriori typings induced by instance-level specifications are dynamic. Once the specification is applied to model someTasks, evaluating the query Schedulable.all yields Set{t0}, but upon changing t1.duration to 3, the query yields Set{t0, t1}. Thus, accessing instances through a-posteriori types involves a transparent evaluation of their associated queries. To improve efficiency, we profit from Epsilon's caching mechanism.

If type dynamicity is not needed, using type-level specifications is more efficient as no queries need to be evaluated, and they can be easily defined by simple mappings. However, our current implementation is more restricted than the one presented in Sec. III-A, as a class in MM_C cannot be mapped to two classes in MM_R. Our implementation of instance-level specifications allows multiple typing though.

In MetaDepth, type-level specifications are actually implemented as a typing between the creation and role meta-models. As they reside in the same meta-level, we promote the role meta-model to a higher meta-level, and then use it to retype the creation meta-model and its instances. In the example, we lift the Scheduling meta-model using the command lift Scheduling, which outputs the model LiftedScheduling with potency 2. This approach has two advantages: retyping instances of Tasks to Scheduling amounts to type composition, natively supported by the tool. Second, it enables bidirectional retypings, as instances of Scheduling can also be retyped w.r.t. Tasks.

Figs. 10(b,c) show a schema of type-level specifications. Note that backward reclassification may not yield a unique typing if an object can be classified by several types. For example, reclassifying back the model in Listing 6 to the Tasks meta-model yields four possible a-posteriori typings, which result from all possible combinations of assigning Task and DocTask as types for t0 and t1 (as MetaDepth does not support multiple classifiers in type-level specifications). Moreover, we ignore derived attributes in backward reclassification, as it would require calculating the inverse of arbitrary expressions.

Listing 5 shows a type-level specification. It only needs to define the mapping of features, as the mapping of classes can be automatically induced.

```
1 type Tasks LiftedScheduling {
2   Task::start> Schedulable::date,
3   Task::/months : $double = self.duration/30$ > Schedulable::span
4 }
```

Listing 5: Type-level specification in MetaDepth.

Given an a-posteriori typing, retyping models is automatic. The tool permits displaying a model using the a-posteriori types

using the command dump ⟨model⟩ as ⟨role-meta-model⟩. If we type dump someTasks as Scheduling, the tool displays:

```
Scheduling someTasks {
  Schedulable t0 {
    date = "30/04/2015";
    span = 1.0;
  }

  Schedulable t1 {
    start = "30/05/2015";
    span = 3.0;
  }
}
```

Listing 6: Reclassification in METADEPTH.

Internally, this model is just another view of someTasks using the a-posteriori typing.

C. Analysis of typing specifications

METADEPTH can check executability, non-totality, and non-surjectivity of retypings, as explained in Sec. IV-B. In particular, the command refinement Tasks Scheduling tries to find a counterexample witness model that satisfies the constraints of Tasks and violates some of Scheduling. Similarly, command refinement Tasks Scheduling strict checks if there is a model fulfilling all constraints in Tasks and violating some of Scheduling.

METADEPTH relies on the USE Validator [15] to perform the analysis. Given a UML class diagram with OCL constraints, USE finds an object model satisfying the constraints, provided some exists within the search bounds. The found model is parsed back into METADEPTH, and retyped to either the creation or the role meta-model. In all our tests, USE had good searching times, finding witnesses in less than one second.

VI. APPLICATIONS OF A-POSTERIORI TYPING

A-posteriori typing has multiple applications in MDE. First, it is a mechanism to obtain views of models w.r.t. other meta-models, which allows its use to specify simple model transformations (see Section VI-A). Second, it enables the reuse of model management operations defined over a role meta-model (see Section VI-B). Finally, dynamic typing can be valuable in Models@run.time applications, where models evolve and objects can get reclassified dynamically (see Section VI-C).

A. (Bi-directional) model transformation by reclassification

Consider the DSL to describe factories in Fig. 11. It declares three kinds of machines: generators introduce parts in the factory, terminators remove parts from it, and assemblers transform parts. Machines are connected by conveyors, which transport any number of parts. Parts have a boolean flag indicating whether they passed a quality test.

Assume we want to transform Factory models into PetriNets. Petri nets are made of places and transitions. Places may hold tokens, and can be connected to transitions (and vice versa). Tokens must be in exactly one place, as required by constraint cont. Transitions can *fire* if all incoming places (relation ins) hold some token. If a transition fires, a token is deducted from every input place and added to its output places (outs). The envisioned transformation translates any kind of machine into a transition, conveyors into places, and parts into tokens.

Instead of using a transformation language, which would create a separate target model conformant to the PetriNet meta-model,

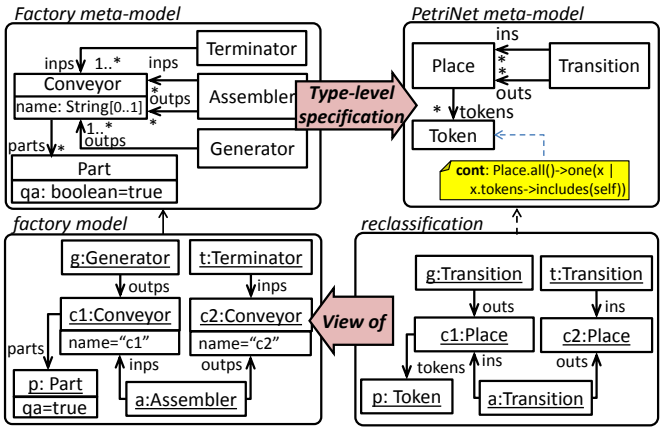

Fig. 11: Reclassifying Factory models into Petri-nets

we can use a type-level a-posteriori typing specification. This way, we can reclassify Factory models as PetriNets, producing a virtual view of the factories without the need to explicitly create a Petri net model (see Fig. 11).

Listing 7 shows the type-level specification. Conveyors are retyped as Places, Parts as Tokens, and Generators, Assemblers and Terminators as Transitions. Note that Generators are retyped as Transitions without inputs, and Terminators are retyped as Transitions without outputs.

```
type Factory LiftedPetriNet {
    Conveyor::parts > Place::tokens,
    Generator::outps > Transition::outs, Terminator::inps > Transition::ins,
    Assembler::inps > Transition::ins, Assembler::outps > Transition::outs,
}
```

Listing 7: Type-level specification.

This specification allows retyping Factory models as PetriNet, and vice versa. As an example, Listing 8 shows a PetriNet model and its two possible a-posteriori typings: the first one types the transition as Terminator, and the second as Assembler.

```
PetriNet example {
  Place p {}
  Transition t { ins = [p]; }
}
Factory example { // 1st typing
  Conveyor p {}
  Terminator t { inps= [p]; }
}
Factory example { // 2nd typing
  Conveyor p {}
  Assembler t { inps= [p]; }
}
```

Listing 8: Retyping a Petri net into a Factory.

The specification can be analysed to detect whether it is executable, total and surjective. For totality, the tool finds the witness model in Listing 9, which proves the transformation is not total because this factory cannot be retyped as a Petri net (i.e., this model "cannot be transformed"). The reason is that the model violates the constraint cont in the PetriNet meta-model, since part2 is outside any Conveyor. Moreover, the same model shows the backward transformation is not surjective, as this model cannot be produced from any Petri net.

```
Factory noRefinementWitness { // Model witness with no Petri net equivalent
  Assembler assembler2 { outps= [conveyor2, conveyor1]; }
  Conveyor conveyor1 { name= "string1"; }
  Conveyor conveyor2 { name= "string1"; }
  Generator generator2 { outps= [conveyor1]; }
```

```
6     Part part2 { qa = true; }
7 }
```

Listing 9: Witness model showing no totality.

Reclassification is not a substitute for traditional transformations. But to have an intuition of the potential applicability of this approach, we analysed the zoo of ATL transformations to see how many of them can be specified as reclassifications (see http://miso.es/dsets/atlzoo). Interestingly, 19% (23/119) of the transformations are refinements or 1-to-1 mappings that can be reformulated as retypings.

B. Reuse of model management operations

We have developed a simulator for Petri nets using EOL, a very small excerpt of it is shown in Listing 10. The simulator uses the types of the PetriNet meta-model, defining operations (like enabled) on its types. Operation step is the main simulation method, which performs one simulation step if some transition can be fired.

```
1 operation Transition enabled() : Boolean {
2   return self.ins.forAll(p| p.tokens.size()>0);
3 }
4 operation step() : Boolean {
5     var enabled : Set(Transition) := Transition.all.select( t | t.enabled());
6     ... // fire one random Transition from enabled
7 }
```

Listing 10: Small excerpt of the EOL Petri net simulator

Once the specification of the a-posteriori typing is defined, the simulator becomes applicable "as is" to the instances of the Factory meta-model. This is possible because our tool handles a-posteriori types as if they were constructive types, hence achieving reuse of the simulator in a straightforward way. Therefore, an expression like Transition.all returns all instances of all classes mapped to Transition.

In general, object creation in reused model management operations may result in non-deterministic behaviour, if the created object (tokens in the case of the simulator) is mapped to several constructive types. In this example, the behaviour is deterministic because Token was only mapped by Part. Hence, whenever the simulator creates a Token, a Part gets created instead, with its attributes initialized to the default values.

C. Dynamic typing

Let us consider factories in which parts that do not pass a quality check are not processed. Interestingly, we can still use the *same* simulator as in Section VI-B, if we include this condition in the a-posteriori typing of Parts. This way, Parts whose qa is false are not mapped to Tokens and will not be considered by the simulator. As the typing becomes dynamic, we need to use an instance-level a-posteriori typing specification, a fragment of which is shown in Listing 11. Line 3 selects in collection tokens only those parts whose qa is true.

```
1 type Factory PetriNet inst {
2     $Conveyor.all$ > Place with {
3         /sp : Token[*] = $self.parts.select(p|p.qa=true)$ > tokens
4     }
5     $Part.all.select( p | p.qa = true )$ > Token
6 }
```

Listing 11: Excerpt of instance-level typing to PetriNet

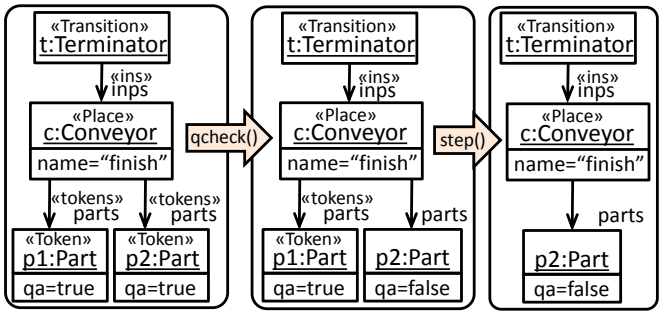

Fig. 12: Dynamic typing for the simulation of a factory.

Fig. 12 shows two simulation steps. Method qcheck is an operation over Factory that uses constructive types. It emulates a quality check, setting the qa attribute of some Parts to false according to a probability distribution. In step 2, p2.qa becomes false, hence p2 drops the classifier Token and leaves the tokens collection. Method step belongs to the original simulator, and it consumes one Token of the incoming Place to Transition t. Another advantage of this approach is that it permits having a simple simulator, unaware of possible conditions for activation or deactivation of Tokens, and permitting dynamicity of Places and Transitions by means of other dynamic typings.

VII. Related work

Next, we compare our typing approach with others proposed in the literature. Table I summarizes this comparison using most features in Fig. 1 as well as the extra criteria *Style*, which refers to the specification style (type/instance level).

TABLE I: Comparison of model typing approaches.

Approach	Class. time	Style	Dyn.	# class.	#model types	total.
Concepts [6]	post.	type	no	many	many	total
Adapters [4]	post.	type	no	many	many	partial
Zschaler [23]	post.	type	no	one	many	partial
Steel et al. [12], [22]	post.	type	no	one	many	partial
SMOF [19]	post.	-	yes	many	many	total
UML [18]	post.	-	yes	many	one	total
MOF [20]	creat.	-	no	one	one	total
This paper	post.	both	yes	many	many	partial

In previous works, we proposed *concepts* [6] as a mechanism to express requirements for model management operations. Concepts can be mapped to meta-models, and as a result, operations developed over concepts get adapted for the meta-models. Concepts allow the use of *adapters* [4] that enable more flexible mappings between concepts and meta-models via OCL expressions. This approach has a type-level specification style, lacks support for dynamicity, and has only been applied to adapt model transformations.

Zschaler [23] proposes constraint-based model types, a constraint-based specification of requirements for meta-models to qualify for operations. These types are automatically extracted from existing operations. Dynamicity, multiple classification or flexible mappings via queries are not considered.

In [12], [22], a *matching* relation is defined between two meta-models, to permit instances of the first to be accepted by the latter. Originally, matching classes required same name [22], but this is more flexible in [12]. To achieve compatibility, derived features are frequently added to the source meta-model. Still, no dynamicity or overlapping classes are considered.

Regarding standards, SMOF recognises the need for multiple and dynamic classification, and proposes annotating the classifiers that may have instances in common [19]. UML allows multiple classifiers through generalization sets [17]. While UML supports dynamic classification, neither UML nor SMOF provide support for defining a-posteriori typing specifications, and hence unanticipated reuse of operations become difficult. Finally, MOF does not support a-posteriori typing, dynamicity, or overlapping classes, and models have exactly one type.

Exploratory modelling [1] has been proposed as a way to provide a type to existing instances, where types are created on-demand based on instance features. Instead, in *a-posteriori* typing, types already exist, and the typing creates a new classification relation between those types and existing instances. Some works have analysed the compatibility of meta-models regarding acceptance of instances [12], [16]. Interestingly, our condition for totality of type-level specifications is forward compatibility in [16], and surjectivity is back compatibility.

Notably, all previous works implement a type-level style for specifications, which neglects dynamicity of typing. The work in [8] is closer to our instance-level specifications, proposing the use of queries to relate (possibly derived) elements of two models. While such relations are not retypings, they might be used to encode our instance-level specifications.

Dynamic reclassification has been more studied in object-oriented languages. In [9], objects can change its type among several *state* classes, subtypes of a given *root* class. To ensure type-safety, *state* classes are not allowed to receive references. Similar to our role meta-models, role classes [10] model the different roles individual objects can acquire or drop. While in these approaches, the change of role or classifier is done via method invokations, our instance-level specifications express these changes declaratively, which facilitates analysis.

Also for programming languages, pluggable type systems [2] allow plugging-in additional typings for a program, which is similar to our a-posteriori typing. There are few attempts to increase dynamic typings in MDE. One exception is [3], proposing the extraction of the dynamic aspect of objects, so that it can be changed dynamically, similar to the state pattern. However, it does not fully support dynamic classification.

Hence, our proposal improves existing works by more flexible, dynamic reclassification, which enables multiple classifiers.

VIII. Conclusions and future work

In this paper we have proposed a-posteriori typing as a mechanism to decouple object creation from classification. We have shown two specification styles for such typing and shown a practical implementation in MetaDepth. We have presented examples and applications showing the flexibility of the proposal. Altogether, our approach leads to a more flexible retyping than existing proposals permitting "as is" reuse of model management operations, while type dynamicity enables flexible adaptation of those operations.

Most MDE approaches, including ours, are based on nominal typing; in the future, we could use structural typing to classify untyped objects (e.g., extracted as raw data from documents) according to the features they exhibit, and as a heuristic for re-typing specifications. Currently, we can only specify non-overlapping conditions in role meta-models using OCL constraints. It would be interesting to develop annotations to signal these constraints more concisely. We would also like to improve tooling, e.g., increasing the efficiency of instance-level a-posteriori typings by smarter cache policies.

Acknowledgements. Work supported by the Spanish MINECO (TIN2011-24139 and TIN2014-52129-R), and the R&D programme of the Madrid Region (S2013/ICE-3006).

References

[1] C. Atkinson, B. Kennel, and B. Goß. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *SWESE*, pages 1–15, 2011.
[2] G. Bracha. Pluggable type systems. In *Revival of Dyn. Langs.*, 2004.
[3] M. Conrad, M. Huchard, and T. Preuss. Integrating shadows in model driven engineering for agile software development. In *CISIS*, pages 549–554, 2008.
[4] J. S. Cuadrado, E. Guerra, and J. de Lara. A component model for model transformations. *IEEE TSE*, 40(11):1042–1060, 2014.
[5] J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010.
[6] J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MODELS*, volume 6394 of *LNCS*, pages 16–30. Springer, 2010.
[7] J. de Lara, E. Guerra, and J. S. Cuadrado. When and how to use multilevel modelling. *ACM TOSEM*, 24(2):12:1–12:46, 2014.
[8] Z. Diskin, T. Maibaum, and K. Czarnecki. Intermodeling, queries, and kleisli categories. In *FASE*, volume 7212 of *LNCS*, pages 163–177. Springer, 2012.
[9] S. Drossopoulou, F. Damiani, M. Dezani, and P. Giannini. More dynamic object reclassification: Fickle$_{||}$. *ACM ToPLaS.*, 24(2):153–191, 2002.
[10] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.*, 14(3):268–296, 1996.
[11] E. Guerra and J. de Lara. Towards automating the analysis of integrity constraints in multi-level models. In *MULTI*, volume 1286 of *CEUR*, pages 63–72, 2014.
[12] C. Guy, B. Combemale, S. Derrien, J. Steel, and J. Jézéquel. On model subtyping. In *ECMFA*, volume 7349 of *LNCS*, pages 400–415. Springer, 2012.
[13] F. Hermann, H. Ehrig, and C. Ermel. Transformation of type graphs with inheritance for ensuring security in e-government networks. In *FASE*, volume 5503 of *LNCS*, pages 325–339. Springer, 2009.
[14] D. S. Kolovos, R. F. Paige, and F. Polack. The epsilon object language. In *ECMFA*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
[15] M. Kuhlmann and M. Gogolla. From UML and OCL to relational logic and back. In *MODELS*, volume 7590 of *LNCS*, pages 415–431. Springer, 2012.
[16] T. Kühne. On model compatibility with referees and contexts. *SoSyM*, 12(3):475–488, 2013.
[17] A. Olivé. *Conceptual modeling of information systems*. Springer, 2007.
[18] OMG. UML 2.5. http://www.omg.org/spec/UML/2.5/Beta2/.
[19] OMG. SMOF 1.0. http://www.omg.org/spec/SMOF/1.0/, 2013.
[20] OMG. MOF 2.4.2. http://www.omg.org/spec/MOF/, 2014.
[21] R. Paige, D. Kolovos, L. Rose, N. Drivalos, and F. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS*, pages 162–171, 2009.
[22] J. Steel and J. Jézéquel. On model typing. *SoSyM*, 6(4):401–413, 2007.
[23] S. Zschaler. Towards constraint-based model types: A generalised formal foundation for model genericity. In *Proc. VAO*, 2014.