A-posteriori Typing for Model-Driven Engineering: Concepts, Analysis and Applications

JUAN DE LARA, Universidad Autónoma de Madrid (Spain)
ESTHER GUERRA, Universidad Autónoma de Madrid (Spain)

Model-Driven Engineering (MDE) is founded on the ability to create and process models conforming to a meta-model. In this context, classes in a meta-model are used in two ways: as templates to create objects, and as (static) classifiers for them. These two aspects are inherently tied in most meta-modelling approaches, which results in unnecessarily rigid systems and hinders reusability of MDE artefacts.

In this work, we discuss the benefits of decoupling object creation from typing in MDE. Thus, we rely on standard mechanisms for object creation, and propose *a-posteriori* typing as a means to retype objects and enable multiple, partial, dynamic typings. This approach enhances flexibility, permits unanticipated reuse as model management operations defined for a meta-model can be reused with other models once they get reclassified, and enables bidirectional model transformation by reclassification. In particular, we propose two mechanisms to realise model retyping, and show their underlying theory and analysis methods. We show the feasibility of the approach by an implementation atop our meta-modelling tool METADEPTH, and present several applications of retypings (transformations, reuse and dynamicity).

Categories and Subject Descriptors: [**Software and its engineering**]: Model-driven software engineering; Domain specific languages; System modeling languages; Reusability

Additional Key Words and Phrases: A-posteriori model typing, Partial typing, Dynamic typing, METADEPTH

ACM Reference Format:
Juan de Lara and Esther Guerra, 2016. A-posteriori Typing for Model-Driven Engineering: Concepts, Analysis and Applications. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 59 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Model-Driven Engineering (MDE) advocates the use of models as the principal assets of software projects. In MDE, models are not passive documentation, but they are actively used to specify, simulate, test, and generate code for the application to be built, among other activities [Brambilla et al. 2012]. Frequently, models in MDE are built with Domain-Specific Modelling Languages (DSMLs) instead of using general-purpose ones like the UML. DSMLs are highly customized languages with powerful primitives for a particular application domain, like mobile and web development, controllers for embedded systems, or the description of questionnaires [Kelly and Tolvanen 2008]. The abstract syntax of DSMLs is typically described using a meta-model that defines the relevant entities and relations within the domain.

MDE has traditionally promoted a "top-down" approach, where classes in meta-models are used as templates to create objects in models, which then become classified by those classes. This kind of typing for model elements by meta-model elements

Author's addresses: J. de Lara, Computer Science Department, Universidad Autónoma de Madrid (Spain); E. Guerra, Computer Science Department, Universidad Autónoma de Madrid (Spain).

DOI: http://dx.doi.org/10.1145/0000000.0000000

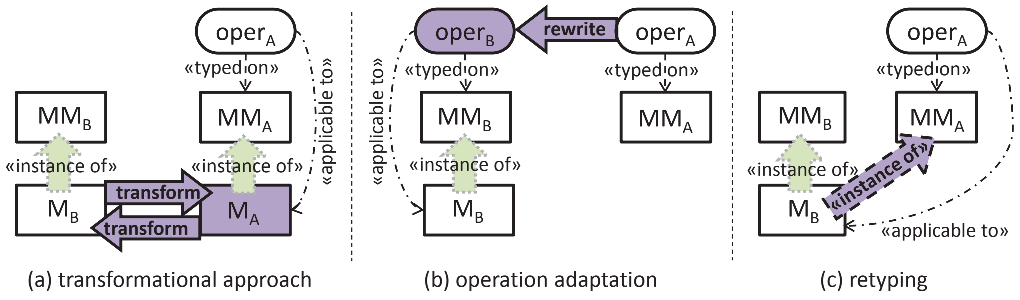


Fig. 1: Different approaches to reusing model management operations in MDE

is called *constructive* [Atkinson et al. 2011], because classes are used both to create and classify objects, objects can only be constructed by instantiating classes, and both aspects (creation and classification) cannot be separated.

Constructive typing is mainstream in MDE but lacks flexibility. For example, the MOF Support for Semantic Structures (SMOF) [OMG 2013a] standard discusses the rigidity of MOF [OMG 2014] 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 in case of having authored and reviewed articles).

The unmodifiability of the typing relation also hinders reuse. This is so as, in order to reuse an operation (e.g., a model transformation) defined over a meta-model MM_A for another meta-model MM_B, the typical solution is to transform the instance models of MM_B into instance models of MM_A, as shown in Figure 1(a). However, this solution is heavyweight, it complicates traceability with respect to the original model (M_B), and it may be necessary to transform back the operation results in terms of MM_B. Another possibility is to (manually) rewrite the operation in terms of MM_B, as depicted in Figure 1(b). However, this alternative is costly and error-prone, and moreover, any operation over MM_A to be reused with MM_B needs to be rewritten. Instead, the solution in Figure 1(c) proposes retyping the instance models of MM_B as if they were instances of MM_A. This solution is simpler and cleaner, but its realisation requires a more flexible model typing concept, able to overcome the heterogeneities between MM_A and (the instances of) MM_B.

Decoupling typing from instantiation is a well-known technique to promote reuse and ease the adaptation of existing code in object-oriented programming [Canning et al. 1989]. For example, in languages like Java, objects are created by constructors and become classified by the classes used to create them. However, additional typing mechanisms, like interfaces, allow focussing on a subset of properties that objects require in order to qualify for a certain operation. 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 [Drossopoulou et al. 2002] to allow changing the class membership of objects at runtime, which decouples even further classification from object creation.

In contrast, most MDE approaches use static constructive typing, which results in more restricted possibilities for modelling and reuse. Some more flexible proposals have emerged recently, especially in connection with reusability of MDE artefacts [de Lara and Guerra 2013; Sánchez Cuadrado et al. 2014; Zschaler 2014; Guy et al. 2012; Steel and Jézéquel 2007; Salay et al. 2015]. However, they often lack desirable features like dynamicity or multiple classifiers, while some allow reusing just some specific kinds of MDE artefacts or technologies (e.g., ATL model transformations [Sánchez Cuadrado et al. 2014]) and cannot be applied to other artefacts (e.g., code generators). Instead, we claim that a flexible notion of a-posteriori typing would enable a more general and powerful mechanism for reusability of any kind of MDE artefact. Our goal is to provide such a mechanism.

In this paper, our aim is to provide a more flexible typing in MDE, which then 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. Inspired by works on role-based modelling [Steimann 2000], we call them "role" meta-models.

We provide two ways for specifying 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 classes and interfaces in languages like Java, and permits forward and backwards (i.e., bidirectional) retyping. 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 slots in objects, and therefore their type may change whenever such values evolve. We show that the first kind of typing is just a special case of the second. Moreover, we present a set of techniques to analyse correctness of type- and instance-level specifications, as well as the properties of forward/backward reclassification implied by type-level specifications. As a proof-of-concept, we show an implementation in our METADEPTH tool [de Lara and Guerra 2010], and discuss several applications that show the benefits that our approach brings.

This paper completes our previous work [de Lara et al. 2015] as follows. We extend our analysis of the typing space to consider further possibilities (see Section 2). We include criteria for well-formedness of reclassifications, and procedures for analysing several properties of interest. We also identify some restriction possibilities for retyping specifications which are useful in practice and mimic other existing approaches. We have extended our implementation to support annotations for these restrictions, and to support richer representations of multiple typings. We compare the expressivity and efficiency of our retypings with other approaches. A repository of a-posteriori typing specifications has also been created and is available at: http://miso.es/aposteriori/.

The rest of the paper is organized as follows. First, Section 2 identifies typing alternatives in MDE, and Section 3 introduces the notation used throughout the paper, as well as our notion of type correctness. Next, Section 4 presents a-posteriori typing specifications at the type-level, while Section 5 introduces instance-level specifications. Section 6 describes some analysis possibilities for typing specifications. While our formulations are intentionally as general as possible, in Section 7 we provide some useful restrictions. Section 8 describes tool support, and Section 9 shows examples and applications of a-posteriori typing, commenting on benefits and limitations. Section 10 discusses related work, including a comparison of the expressivity with other approaches. Finally, Section 11 finishes the paper with the conclusions. An appendix provides details of the main theoretical results. A supporting Alloy [Jackson 2002] formalization is available at http://miso.es/aposteriori/alloy.html.

2. THE TYPING SPACE FOR MODEL-DRIVEN ENGINEERING

In this section, we analyse the possibilities for different aspects of typing in MDE. The feature model [Kang et al. 1990] in Figure 2 (split in two for readability) summarizes the alternatives that meta-modelling approaches may adopt, which we detail next.

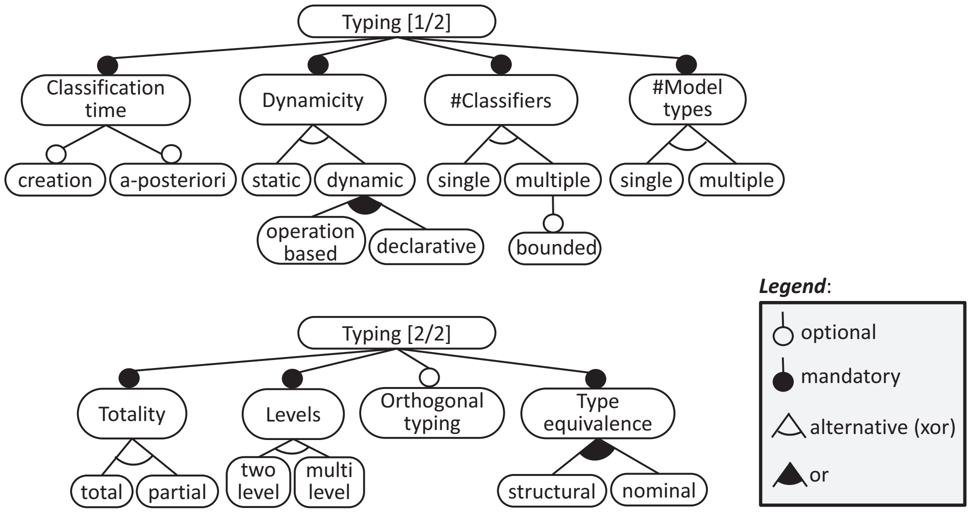


Fig. 2: Alternatives for typing in MDE

— *Classification time*. Object classification can be determined either when the object is created (constructive typing), or new classifiers can be added later (a-posteriori typing). In constructive typing, classes are used as templates to create instances, instances cannot be created in a different way, and the type of an instance is the class used to create it. Hence, creation and classification are inseparable, and the latter cannot change a-posteriori. This is the usual approach in MDE. Figure 3(a) shows an example of constructive typing where class Task is used as a template to create object review, becoming its only classifier[1]. In this and the following figures, we show the constructive type for objects in standard UML object notation, and the a-posteriori typing using stereotypes.
Instead, in a-posteriori typing, object creation and classification are separated, and objects may have other type(s) besides their constructive type. A-posteriori types do not need to be assigned statically when the creation class is defined, as is the case with interfaces in programming languages like Java, but they can be added on demand. Figure 3(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 with respect to the Planning meta-model, and two a-posteriori typings with respect to the Scheduling and Measuring meta-models. A-posteriori typings are proper typings; therefore, any model management operation defined over Scheduling or Measuring becomes applicable on the retyped model.
Note that a-posteriori typing may also be possible without constructive typing, just like in classless or prototype-based programming approaches [Noble et al. 1999; Ungar and Smith 1987]. As an example, in Figure 3(c), the review object has been created without the use of a template class, nonetheless, this untyped object receives two a-posteriori typings with respect to the Scheduling and Measuring meta-models.
— *Dynamicity*. The type of an object may be unmodifiable (feature static in Figure 2), or it may change over time as the object evolves (feature dynamic). In the latter case, the retyping might be caused by an explicit operation call (feature operation-based)

[1]In the rest of the paper, we use type/classifier and retyping/reclassification interchangeably.

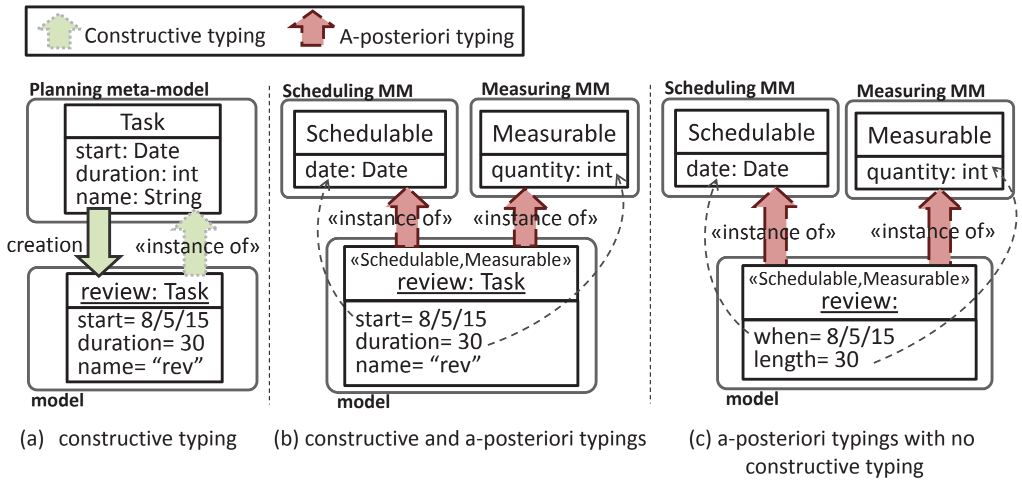


Fig. 3: Examples depicting classification time alternatives for typing

or due to the fulfilment of some conditions over properties of the object (feature declarative). Both approaches are not exclusive but can be mixed (i.e., some types may change due to operations, while others because of conditions). While operation-based dynamicity is typical of programming languages [Drossopoulou et al. 2002; Tamai et al. 2005], declarative dynamicity might be more appropriate for MDE. In particular, declarative specifications of type changes are useful to classify objects according to their properties, but keeping the specifications independent of any model management operation. The retyping conditions would be given in terms of the creation type, and the model management operations would be defined over the a-posteriori types.

While constructive typing is inherently static, a-posteriori typing can be dynamic. As an example, the bottom-left model in Figure 4 has the Planning meta-model as constructive type, and is typed a-posteriori with respect to the upper right meta-model for conference reviewer assignment. The a-posteriori typing classifies each Person object as Author if he owns 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 p2 gets the additional type Reviewer.

— *Number of classifiers*. Some type systems allow several classifiers, none subtype of the others, to share common instances. Hence, some objects may receive multiple classifiers from the same meta-model. For example, in the bottom-right model of Figure 4, p2 is typed a-posteriori as Author and Reviewer. Constructive typing does not support multiple classifiers, but a-posteriori typing may enable this feature.

Some systems like SMOF permit declaring the set of potential classifiers that instances of a given type can adopt. The UML [OMG 2013b] also supports overlapping instances through the overlapping annotation on generalization sets. Hence, the type systems of both SMOF and UML have the feature bounded selected in the feature model in Figure 2.

While some indication of allowed overlappings between classes is useful in practice, this option lies between two extremes. On the more constrained side, MOF does not allow distinct classes to share instances (unless one class is a subtype of the other). On the more flexible side, one could permit any class to share instances with

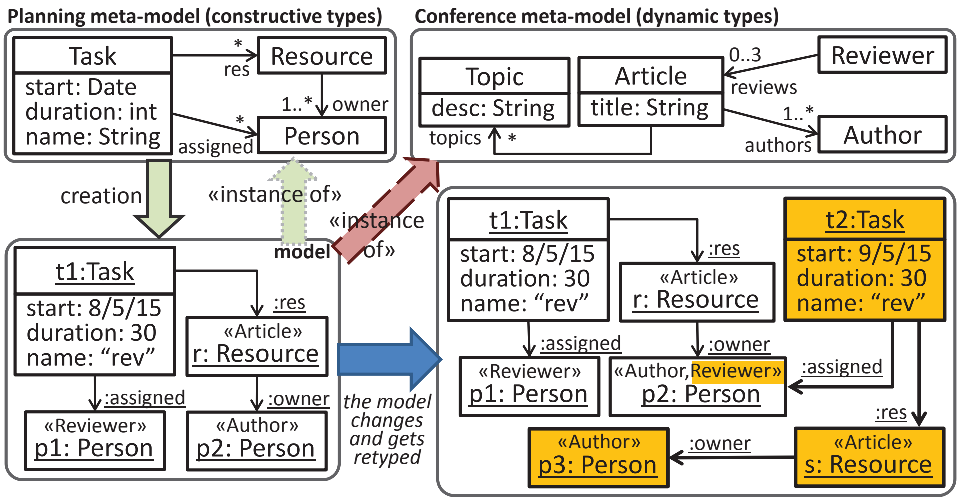


Fig. 4: Declarative dynamic typing, and multiple classifiers for objects

any other. In Figure 4, neither Reviewer nor Author have an indication of disallowed overlappings, and hence, p2 can be typed by both.

— *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 Figure 3(b) is typed a-posteriori by meta-models Scheduling and Measuring. Each one of these a-posteriori typings is a real typing, and hence, the operations defined over meta-models Scheduling and Measuring can be applied on the retyped model.

— *Totality*. Standard constructive typing is total as objects always receive a type from the instantiated meta-model (the class used for their creation). In contrast, a-posteriori typing can be partial if it is allowed to have model elements (objects, links or slots) without an a-posteriori type. For example, Task instances t1 and t2 in Figure 4 lack an a-posteriori type. Similarly, the model in Figure 3(b) is partially typed with respect to the Scheduling meta-model because slots duration and name are not typed by this meta-model. Thus, viewing this model through the typing to Scheduling ignores the untyped slots.

— *Levels of typing*. Standard frameworks – like the EMF [Steinberg et al. 2008] – only manage two meta-levels at a time (meta-models and models). We call them two-level approaches. In contrast, multi-level approaches [Atkinson and Kühne 2003; de Lara et al. 2014] 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 created at the next meta-level.

— *Orthogonal typing*. Some meta-modelling approaches distinguish between linguistic and ontological typings [Atkinson and Kühne 2003]. The linguistic typing refers to the meta-modelling primitive used to create an element. For example, if we assume we are using the UML for creating the meta-model and model in Figure 3(a), we have that the linguistic type of Task is UML Class, and the one of review is UML Object. Instead, ontological typing refers to instantiations within a domain. Both constructive and a-posteriori typings are ontological. In this way, in Figure 3(a), the ontological type of review is Task.

The availability of orthogonal typing permits having elements without ontological type (as the ontological typing may be partial) but only linguistic type. Elements with no ontological type are called linguistic extensions [de Lara and Guerra 2010].

The distinction between ontological and linguistic typings has a prominent role in multi-level modelling approaches, where it is also known as the *Orthogonal Classification Architecture* (OCA) [Atkinson and Kühne 2003].

— *Type equivalence.* This feature is related to the mechanism by which a type is assigned to an element [Pierce 2002]. In *nominal* typing, two elements have the same type if their declarations name the same classifier. In constructive typing approaches (e.g., MOF), this means that two objects have equal type if their creation class is the same, whereas they have compatible type if their creation classes are different but share a common ancestor. Instead, in *structural* typing, type conformance is checked by looking at the structure of elements. Hence, two elements have the same type if they have the same structure.

We can classify existing meta-modelling approaches based on the previous features. For example, the typing in MOF and EMF is constructive, static, total, two-level, nominal, and it allows a single classifier for objects, and a single meta-model to type a model. 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 supports multi-level modelling and enables a-posteriori, dynamic, partial typings, while objects can have multiple a-posteriori types, and models can be typed by several meta-models. Moreover, object creation does not necessarily need to rely on existing classes, but untyped objects as the one in Figure 3(c) are allowed through linguistic extensions.

3. PRELIMINARIES: NOTATION AND TYPING

In this section, we introduce the basic notation and definitions necessary for building a-posteriori typings, which will be presented later in Sections 4 and 5. In particular, in the rest of the paper, we use the following notation:

— **Meta-models.** We use MM_C to refer to the "creation" meta-model containing constructive types, and MM_R to refer to the "role" meta-model containing a-posteriori types.

— **Classes.** We use capital letters $A, B, C, ...$ and write $A \in MM_C$ for a class A belonging to the creation meta-model MM_C. Predicate $abs(A)$ indicates that class A is abstract. We use names $A, B, C, ...$ for classes in MM_C, and quoted names $A', B', C', ...$ for classes in the role meta-model MM_R.

— **Features.** We use lower-case letters $a, b, c, ...$ for features (attributes or references). $A.a$ means that a is a feature defined in A or a superclass. We sometimes refer to features by a (without a prefix class name). We use $atts(A)$ and $refs(A)$ for the sets of attributes and references of A, both owned and inherited, and write $a \in MM_C$ to denote a feature a in any class of MM_C (and similar for MM_R). We define $feats(A) = atts(A) \cup refs(A)$, and for an attribute a, predicate $default(a)$ indicates that a has a default value.

— **Feature cardinality.** Given a feature a, functions $min(a)$ and $max(a)$ yield its minimum and maximum cardinality, respectively. Function min yields a positive number or zero, while max returns an element of $\mathbb{N} \cup \{\infty\}$. For this set, we extend the usual order relations on the natural numbers ($\leq_{\mathbb{N}}$, $\geq_{\mathbb{N}}$) to consider ∞. Hence, we define

$$a \leq b \triangleq \begin{cases} true,\ if\ b = \infty \\ false,\ if\ a = \infty \wedge b \in \mathbb{N} \\ a \leq_{\mathbb{N}} b,\ if\ \{a, b\} \subseteq \mathbb{N} \end{cases}$$

and similar for $\geq$. Predicate $mand(a) \triangleq min(a) > 0$ holds if feature a is mandatory.

— **References.** Given a reference $A.r$, $tar(A.r)$ is the class r points to. Predicate $comp(A.r)$ holds if r is a composition.
— **Inheritance.** The set $sub(A)$ contains the direct and indirect subclasses of A, and $sub^*(A) \triangleq \{A\} \cup sub(A)$. Conversely, $anc(A)$ is the set of direct and indirect superclasses of A, while $anc^*(A) \triangleq \{A\} \cup anc(A)$. We admit multiple inheritance, but assume cycle-free inheritance hierarchies.
— **Feature owners.** $owner(a)$ is the class that defines feature a, while $owner^*(a) \triangleq sub^*(owner(a))$ is the set of classes that define or inherit a.
— **Objects.** Given a model M, we use $o \in M$ to mean that object o belongs to model M. Given an object o, we use $links(o)$ for the set of links defined in o, and $slots(o)$ for the set of links and attribute values defined in o, with $links(o) \subseteq slots(o)$.
— **Typing.** $type(s)$ and $type(o)$ return the type of a slot s and an object o respectively. As in general we admit multiple classifiers for objects, $type(o)$ actually returns the set of valid classifiers for o. Given an object o, $type^*(o) \triangleq \bigcup_{A \in type(o)} anc^*(A)$ is the set of its compatible classifiers.

As we will see in the following sections, an a-posteriori typing specification induces a retyping of models so that they can be seen as instances of the meta-model that defines the a-posteriori types. In order to be able to ascertain whether such a retyping is correct, first we need to introduce the notion of type correctness, where we admit untyped objects as well as multiple types for objects.

A model M is well-typed with respect to a meta-model MM via a typing $type$ iff:

— *Object typing correctness*. Objects must be well-typed, i.e., their types must belong to the meta-model.

$$\forall o \in M \bullet type(o) \subseteq MM \tag{1}$$

If we need in addition to constrain objects to have at least one type ($\forall o \in M \bullet type(o) \neq \emptyset$), then the typing becomes total on objects.
— *Subclassing implies instance subsetting*. The set of instances whose type is compatible with a class A must be a superset of the set of instances whose type is compatible with every subclass of A.

$$\forall A, B \in MM \bullet B \in sub(A) \implies \{o \in M \mid B \in type^*(o)\} \subseteq \{o \in M \mid A \in type^*(o)\} \tag{2}$$

— *Abstract classes do not have instances*. Please note that Equation (3) uses $type$, as instances may still have a type compatible with an abstract one (i.e., using $type^*$).

$$\forall A \in MM \bullet abs(A) \implies \nexists o \in M \bullet A \in type(o) \tag{3}$$

— *Slot correctness*. If a slot is typed, so must be the owner object. Moreover, the type of the slot must be a feature of some of the types of the object.

$$\forall o \in M \bullet A.a \in type(o.s) \implies A \in type^*(o) \tag{4}$$

— *Slot completeness*. Objects must have a slot for every mandatory feature defined by each of their types. Alternatively, the mandatory feature may have a default value.

$$\begin{aligned} &\forall o \in M,\ \forall A \in MM \bullet A \in type(o) \implies \\ &\qquad \forall a \in feats(A) \bullet mand(a) \implies \\ &\qquad default(a) \ \vee \ \exists s \in slots(o) \bullet A.a \in type(o.s) \end{aligned} \tag{5}$$

— *Link correctness*. The target object of a typed link cannot be untyped. Moreover, as the target object of a link may have multiple types, at least one such type should be compatible with the target class of each type of the link.

$$\forall o \in M, \forall l \in links(o), \forall A \in MM, \forall r \in refs(A) \bullet \\ A.r \in type(o.l) \implies type(tar(o.l)) \cap sub^*(tar(A.r)) \neq \emptyset \tag{6}$$

— *Composition correctness*. An object cannot be pointed to by two links whose type is a composition (Equation (7)). Moreover, there cannot be cycles of links whose type is a composition (Equation (8)). As links may have several types, Equation (7) requires that if an object is pointed to by several links, then at most one of the types of those links can be a composition. Equation (8) uses operation $closureComp(o)$ for the transitive closure of any composition stemming from o. This is the bag of objects that can be reached from o using links with type any composition, where no link is traversed twice.

$$\begin{aligned} &\forall o_i, o_j \in M, \forall l \in links(o_i), \forall r \in type(o_i.l) \bullet tar(o_i.l) = o_j \wedge comp(r) \\ &\implies \forall o_k \in M, \forall l_k \in links(o_k) \bullet l \neq l_k \wedge tar(o_k.l_k) = o_j \\ &\implies r' \in type(o_k.l_k) \implies \neg comp(r') \end{aligned} \tag{7}$$

$$\forall o \in M \bullet o \notin closureComp(o) \tag{8}$$

— *Cardinality correctness*. In every object, the number of slots with type a is within the cardinality interval of a, or feature a has default value and is not instantiated in the object. Given a set S, we write $|S|$ to denote its cardinality.

$$\forall o \in M, \forall a \in MM \bullet min(a) \leq |\{o.s_i \mid a \in type(o.s_i)\}| \leq max(a) \vee \\ (default(a) \wedge \{o.s_i \mid a \in type(o.s_i)\} = \emptyset) \tag{9}$$

Once the notion of type correctness is defined, the next two sections examine two ways to specify a-posteriori typings: at the type level and at the instance level. The former is a particular case of the latter, but it facilitates analysis, permits forward and backward reclassification, and is more concise. The latter enables dynamic typing as well as the possibility of having objects without a constructive type.

4. SPECIFYING A-POSTERIORI TYPING AT THE TYPE-LEVEL

An a-posteriori typing specification 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. This approach is similar to the binding specification of [Sánchez Cuadrado et al. 2014], the model subtyping relation of [Guy et al. 2012], and the I-reflective morphisms of [Hermann et al. 2009]. However, it is more flexible and expressive, as it allows overlapping classes (i.e., objects with multiple types) and derived features. Moreover, in [Sánchez Cuadrado et al. 2014], the binding is defined from the concept (similar to our role meta-model) to the creation meta-model, while in this work it goes the other way round, which is more natural to express retypings.

Figure 5 shows a type-level specification example. It maps class Task to Schedulable, attribute start to date, and it conceptually defines a derived attribute months (which is defined in the specification itself) that gets bound to span. This way, Schedulable, date and span become a-posteriori types for Task instances and their slots.

In the remainder of this section, we describe the features of this specification mode. When several alternatives are possible (e.g., allowing a single or multiple a-posteriori classifiers for instances), we opt for the most general one, though this could be constrained in particular implementations. We provide a collection of useful restrictions and their implications in Section 7.

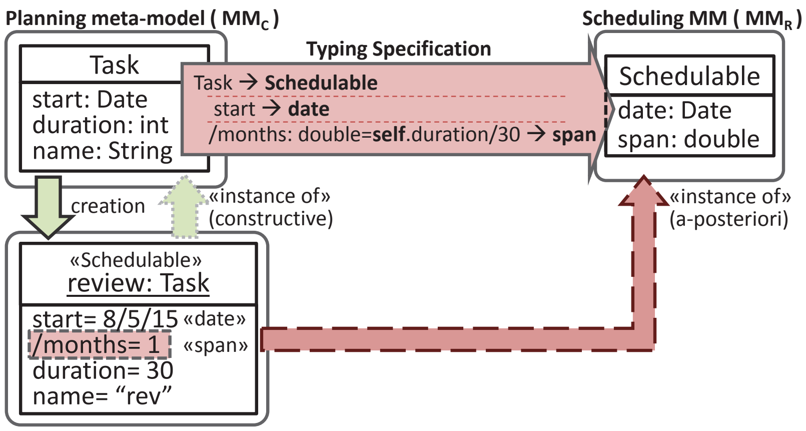


Fig. 5: Specification of a-posteriori typing at the type-level

4.1. Type-level specifications and well-formedness rules

Our type-level specifications are collections of partial functions $TS = \{ts_i\}_{i \in I}$ from elements of MM_C (classes, attributes, references) to elements of MM_R. We use partial functions 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, thus enabling 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).

Next, we enumerate the laws that a type-level a-posteriori typing specification $TS = \{ts_i\}_{i \in I}$ must obey, in order to ensure well-formed retypings. We provide examples after each rule to enhance understanding. For the moment, we state the conditions assuming that the model, once retyped, is read-only. Section 7.2 will introduce additional constraints in case the retyped model is to be modified.

— *Non-abstract mappings*. Classes of MM_C cannot be mapped to abstract classes of MM_R:

$$\forall A \in MM_C, \forall A' \in MM_R \bullet ts_i(A) = A' \implies \neg abs(A') \tag{10}$$

Example. Figure 6 shows two reasons for this rule. Part (a) shows that, should we allow mapping A to A', we would break the assumption that abstract classes do not have instances. Hence, the bottom OCL query, defined over MM_R, would yield an unexpected result when evaluated on the retyped model M_C. Part (b) shows a situation where an abstract class A (with a concrete subclass B) is mapped to A'. In this case, a similar problem arises, as object b with type B would be reclassified as A'. Equation (10) is sufficient to obtain correctness in this sense, but it is not a necessary condition. We might still obtain correct retypings if any concrete class $A \in MM_C$ that is mapped (directly or indirectly as in Figure 6(b)) to an abstract class $A' \in MM_R$, is also mapped to a concrete subclass of A'; however, in such a case, the semantics without the mapping to A' would be the same. We do allow mapping abstract to concrete classes, as for now we assume that retyped models are read-only.

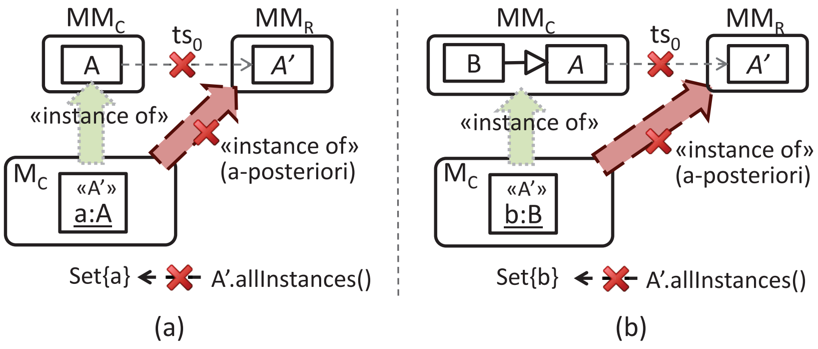


Fig. 6: Incorrect specifications: (a) concrete to abstract class, (b) abstract to abstract class

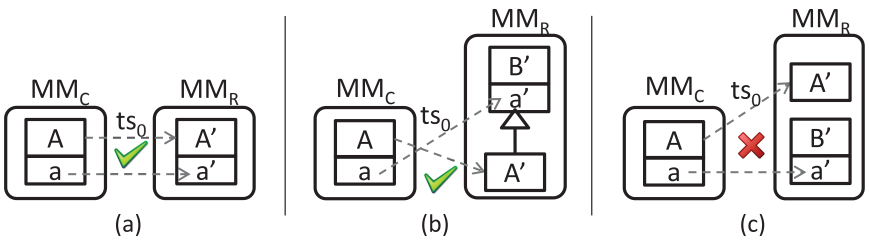


Fig. 7: Correct and incorrect feature mappings for type-level specifications.

— *Correct context for feature mappings*. The features of any class $A \in MM_C$ can only be mapped to features of the class A is mapped to:

$$\forall A \in MM_C, \forall a \in feats(A), \forall a' \in MM_R \bullet \\ ts_i(A.a) = a' \implies ts_i(A) \; is \; defined \; \wedge \; a' \in feats(ts_i(A)) \tag{11}$$

Example. Figure 7 shows some correct and incorrect mappings according to this rule. We allow direct mappings like (a), but also structural ones like (b). Equation (11) permits the latter because $feats$(A') returns the features of A', both owned and inherited. Case (c) is disallowed because a' is not a feature of A'.

— *Compatibility of feature cardinality*. A feature in MM_C can only be mapped to a feature in MM_R with the same or wider cardinality interval.

$$\forall a' \in MM_R, \forall a \in MM_C \; \bullet \; ts_i(a) = a' \implies min(a') \leq min(a) \wedge max(a') \geq max(a) \tag{12}$$

Example. The instances of feature a can only be seen as correct instances of feature a' if they obey the cardinality of a'. For example, the cardinality interval of feature r' in Figure 8(a) is wider than the one for r, and hence, any instance of MM_C will be correctly typed by MM_R. As a consequence, any query over collection r' (like the one at the bottom of the figure) will yield a consistent result. While this is a necessary condition, it is not sufficient, as Figure 8(b) shows. This is so as we allow non-injective mappings, and so, some ts_j mappings may relate several features of a class in MM_C to the same feature in MM_R. The next rule handles this case[2].

— *Compatibility of feature cardinality (2)*. The previous rule is a particular case of a more general rule: if several features are mapped to the same feature $a' \in MM_R$,

[2]Although Equation (13) subsumes Equation (12), we include both for clarity.

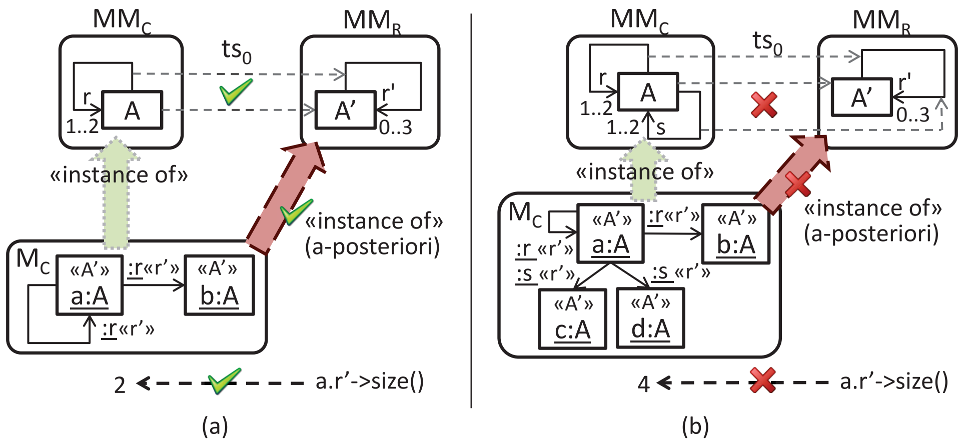


Fig. 8: Type-level specification examples. (a) Correct retyping: mapping to a wider cardinality interval. (b) Incorrect retyping: the interval in MM_R is not equal or wider than the sum of intervals in MM_C.

then, the cardinality interval of a' needs to be the same or wider than the interval made of the sum of the intervals of the features mapped to a'.

$$\forall a' \in MM_R \bullet min(a') \leq \sum_{\substack{a \in MM_C \\ ts_i(a)=a'}} min(a) \ \wedge \ max(a') \geq \sum_{\substack{a \in MM_C \\ ts_i(a)=a'}} max(a) \tag{13}$$

— *Compatibility of reference type*. The target of a reference $A.r$ has to be mapped to the target class of the reference r is mapped to, or to a subclass:

$$\forall A \in MM_C, \forall r \in refs(A), \forall r' \in MM_R \bullet ts_i(A.r) = r' \implies \\ \exists\, C \in anc^*(tar(A.r)) \bullet ts_i(C) \in sub^*(tar(r')) \tag{14}$$

The equation ensures that if $A.r$ is mapped, then there is at least one mapping for $tar(A.r)$ (or an ancestor of it) compatible with $tar(r')$.
Example. Figure 9(a) illustrates this rule. We have $tar(\mathsf{A.r}) = \mathsf{B}$, $tar(ts_0(\mathsf{r})) = \mathsf{D'}$, and as Equation (14) demands, $ts_0(\mathsf{B}) = \mathsf{B'} \in sub^*(\mathsf{D'})$. Equation (11) allows mapping a reference in MM_C to an inherited one (e.g., mapping A.r to A'.r', where A owns r and A' inherits r'). In Equation (14), $tar(A.r)$ does not need to be directly mapped, but it is enough if any of its ancestors C is mapped. Additionally, although the equation demands the mapping i of some $C \in anc^*(tar(A.r))$ to be compatible with $tar(r')$, $tar(A.r)$ can have other mappings (e.g., $ts_1(tar(\mathsf{A.r})) = \mathsf{E'}$ in the figure).

— *Compatibility of composition*. A non-composition reference in MM_C cannot be mapped to a composition in MM_R.

$$\forall r' \in MM_R, \forall r \in MM_C \bullet (ts_i(r) = r' \wedge comp(r')) \implies comp(r) \tag{15}$$

Example. As Figure 9(b) shows, should we allow mapping a non-composition into a composition, we may obtain ill-retyped models that do not satisfy the tree-shape restrictions that compositions impose, like the absence of cycles. Assuming retyped models are read-only, we do allow mapping compositions to non-compositions.

— *Compatibility of composition (2)*. A reference in MM_C cannot be mapped to two composition references in MM_R. This is so as objects cannot be pointed to (i.e., be

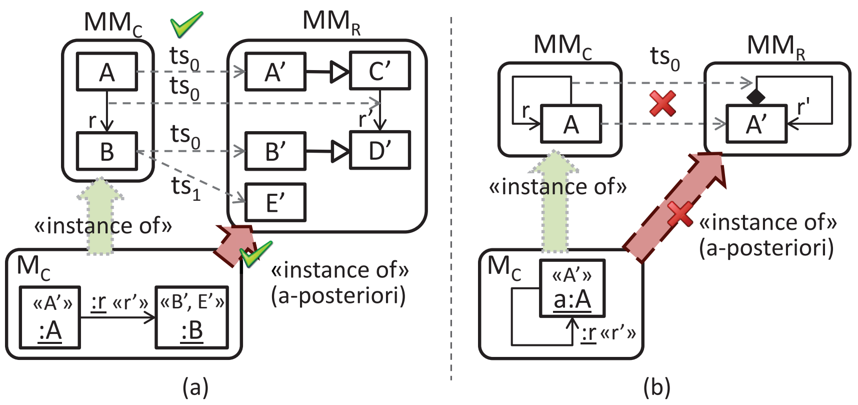


Fig. 9: Type-level specification examples. (a) Correct retyping: reference to inherited reference. (b) Incorrect retyping: non-composition to composition.

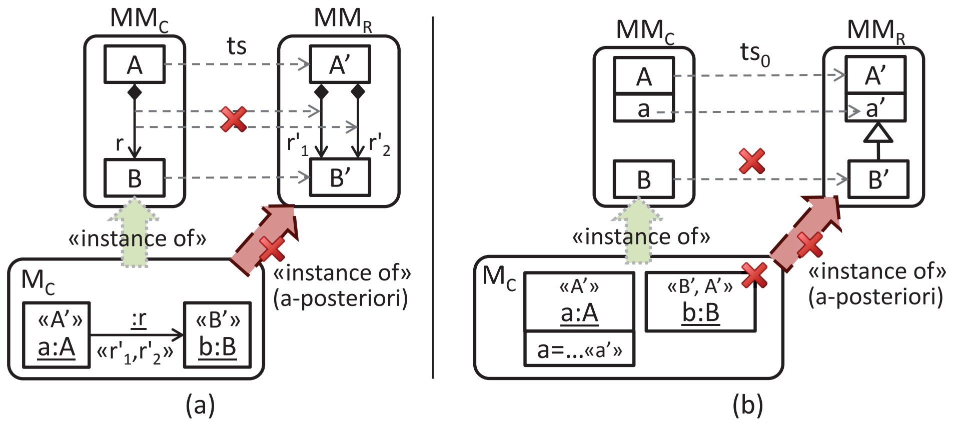


Fig. 10: Incorrect type-level specifications. (a) Composition r is mapped twice. (b) Missing binding for mandatory feature B'.a'.

contained) by two composition references.

$$\forall A \in MM_C, \forall r \in refs(A), \forall r', r'' \in MM_R, \forall i, j \in I \bullet \\ i \neq j \wedge ts_i(A.r) = r' \ \wedge ts_j(A.r) = r'' \wedge r' \neq r'' \implies \\ \neg\, comp(r') \ \vee \ \neg\, comp(r'') \tag{16}$$

Example. Figure 10(a) shows an example of incorrect specification, where r has been mapped twice. This makes object b to belong to two composition references when the model is retyped, which is not correct.

— *Complete instantiations*. If a class $A' \in MM_R$ is mapped from a class of MM_C, all mandatory features of A' should be mapped as well, or have a default value. This rule is needed to emulate correct instantiations which instantiate all mandatory features of classes. Moreover, we do not allow mapping features in MM_C with a default value to mandatory features in MM_R, as default values implicitly make

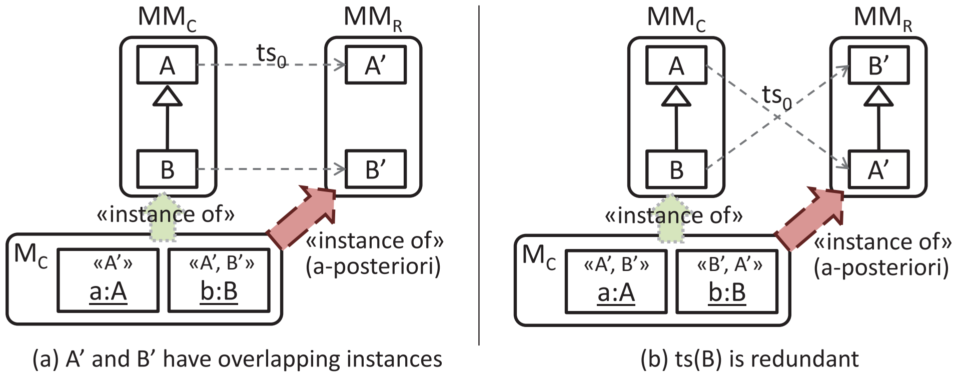


Fig. 11: Type-level specification examples. The stereotypes in each object o show $type'^*(o)$.

mandatory features optional.

$$\forall A \in MM_C, \forall A' \in MM_R, \forall a' \in feats(A') \bullet ts_i(A) = A' \wedge mand(a') \implies \\ default(a') \vee \exists a \in feats(A) \bullet \neg default(a) \wedge ts_i(A.a) = A'.a' \tag{17}$$

Example. Figure 10(b) 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 Figure 5 illustrates this possibility, as it defines a derived attribute months, its calculation procedure, and a mapping to attribute Schedulable.span. A derived feature $A.da$ is defined by an expression that gets evaluated in the context of objects of type A or a subtype. The expression of da is typed over a class in MM_C (for derived references), or has a primitive type (for derived attributes). We write $A.da(o)$ to refer to the value of feature da in object o. 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.

It is remarkable that our typing specification does not define conditions to preserve the compatibility of inheritance hierarchies in MM_C and MM_R, in contrast with other approaches to relate meta-models [Hermann et al. 2009]. This is so as we enable more flexible typings, where a-posteriori types may have overlapping instances. For example, in Figure 11(a), two classes A and B related by inheritance are mapped to two independent classes A' and B', respectively. In this way, 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. Another scenario that yields overlapping instances is a class A mapped to two different classes A' and B' (i.e., $ts_i(A) = A'$ and $ts_j(A) = B'$). As Figure 11(b) shows, "reversing" inheritance relations in MM_C in MM_R is not problematic for our approach either. In this case, the mapping $ts_0(B) = B'$ is redundant because it assigns to B the same a-posteriori type that B already inherits from A (i.e., A' and B').

4.2. Computation of retyped model views

Given an instance model M of MM_C, its retyping with respect to MM_R according to a type-level specification $TS = \{ts_i\}_{i \in I}$ is performed by composing the types from MM_C

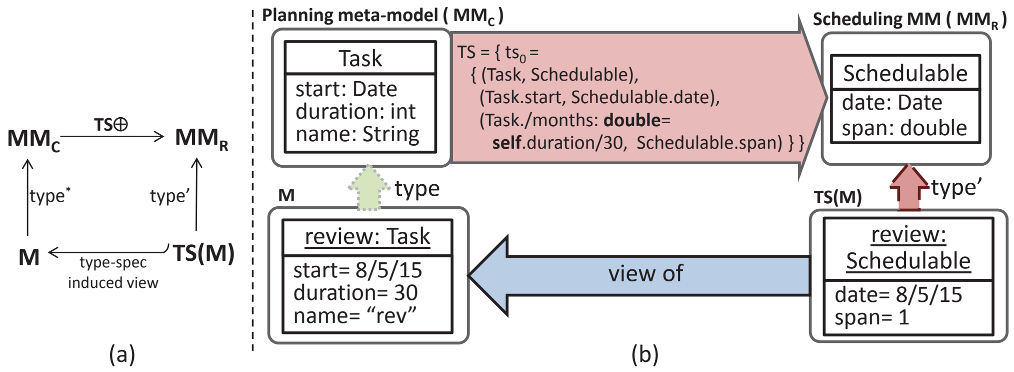


Fig. 12: (a) Retyping M with respect to MM_R. (b) Example of retyping.

and each ts_i function. The retyping induces a view of M, written $TS(M)$, which is typed by MM_R by a typing $type'(TS(M))$ (see Figure 12(a)). The view and the typing are calculated as follows:

(1) Each object $o \in M$ is retyped to:

$$type'(o) = \{ts_k(A) \mid k \in I \ \wedge \ ts_k(A) \ is \ defined \ \wedge \ A \in type^*(o)\} \tag{18}$$

(2) Any object o such that $type'(o)$ is not empty belongs to $TS(M)$; otherwise it does not.
(3) For each derived feature $A.da$, a new virtual slot $o.da$ is added to each object o such that $A \in type^*(o)$. Slot $o.da$ receives the value $A.da(o)$. Then, the virtual slot is retyped as a regular slot, as explained in the next step.
(4) Each slot $o.s \in M$ is retyped to:

$$type'(o.s) = \{ts_k(A.a) \mid k \in I \wedge ts_k(A.a) \ is \ defined \wedge A \in type^*(o) \wedge a \in type(s)\} \tag{19}$$

(5) Any slot $o.s$ such that $type'(o.s)$ is not empty belongs to $TS(M)$; otherwise it does not.

Equation (18) assigns to an object o the a-posteriori types mapped from the direct and indirect types of o. Just like a normal typing, o is also indirectly typed by the supertypes of each class in $type'$; hence, given an object o, $type'^*(o) = \bigcup_{A \in type'(o)} anc^*(A)$. We assume that the original typing $type$ allows multiple classifiers for an object, and it can be the creation typing or an a-posteriori typing. In Equation (19), $o.s$ receives the a-posteriori types of $A.a$, where A is the creation type of o or one of its superclasses.

Alternatively, we can see TS as a function from elements of MM_C to sets of elements of MM_R, by defining $TS^{\oplus} = \bigcup_{i \in I} ts_i$, so that $TS^{\oplus}(A) = \bigcup_{i \in I} ts_i(A)$ (and $TS^{\oplus}(A) = \emptyset$ if all $ts_i(A)$ are undefined). Therefore, a type-level retyping from M to MM_R is calculated by composing $TS^{\oplus} \circ type^*$ (i.e., "following the arrows" of Figure 12(a))[3]. Derived features are calculated and then retyped in the same way. $TS(M)$ is then calculated by removing the untyped objects and slots. As an example, Figure 12(b) shows the retyping of M with respect to MM_R. Since $TS(M)$ is a view of M, the objects and slots in $TS(M)$ are not created new, but they are simply views on top of M's elements.

The following theorem states that the procedure for type-level specifications yields correct retypings, according to the well-formedness criteria defined in Section 3.

[3]While $TS^{\oplus}$ is more convenient for type composition, our original formulation of TS is more direct for defining retypings as sets of mappings.

THEOREM 4.1 (TYPE-LEVEL RETYPING CORRECTNESS).
Given meta-models MM_C and MM_R, a type-level specification TS and a model M typed by MM_C, then $TS(M)$ is well-typed with respect to MM_R using $type'$ according to the well-typed criteria of Section 3. The typing $type'(TS(M))$ is total on objects.

PROOF. (Idea) By checking each of the 9 well-typing conditions in Equations (1–9):

— Equation (1) *"objects are well-typed"* holds by item (1) of the construction of $TS(M)$.
— Equation (2) *"subclassing implies instance subsetting"* holds due to definition of $type^*$.
— Equation (3) *"abstract classes do not have instances"* holds due to Equation (10) *"non-abstract mappings"*.
— Equation (4) *"slot correctness"* holds due to Equation (11) *"correct context for feature mappings"*.
— Equation (5) *"slot completeness"* holds due to Equations (12) *"compatibility of feature cardinality"* and (17) *"complete instantiations"*.
— Equation (6) *"link correctness"* follows from (14) *"compatibility of reference type"*.
— Equations (7) and (8) *"composition correctness"* follow from Equations (16) and (15) *"compatibility of composition"*, respectively.
— Equation (9) *"cardinality correctness"* follows from Equations (12) and (13) *"compatibility of feature cardinality"*.

See proof details in appendix. □

4.3. Features of type-level retyping specifications

Next, we analyse the features that type-level a-posteriori specifications yield, according to the feature model shown in Figure 2.

— *Classification time*. Objects can be classified a-posteriori by a type-level retyping specification TS.
— *Dynamicity*. Type-level a-posteriori typing is not dynamic as the a-posteriori types of objects do not change when the model evolves. Although it is possible to define other a-posteriori types, the existing ones do not change because the types assigned by TS to every object depend statically on the source class from MM_C, not on the object properties.
— *Number of classifiers*. Objects may have several a-posteriori types, i.e., types in the role meta-model can have overlapping instances. This happens if ts_i, ts_j map a class $A \in MM_C$ to several classes in MM_R, or if the classes that participate in the typing specification are related by inheritance (see Figure 11(a)). Constraining the specification TS to be a single function instead of a collection, and adding an extra 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. An implementation may restrict the overlapping to occur only between selected annotated classes of MM_R, as UML and SMOF do. These restrictions will be analysed in detail in Section 7.
— *Number of model types*. A model M may be typed a-posteriori with respect to several role meta-models. It is enough to define several retyping specifications.
— *Totality*. The model elements (objects and slots) whose creation type and its supertypes are not mapped to any element in the role meta-model, will lack an a-posteriori type. This is possible because each ts_i can be partial. Moreover, such unmapped model elements do not belong to the view $TS(M)$, and are "invisible" when using operations defined over MM_R.

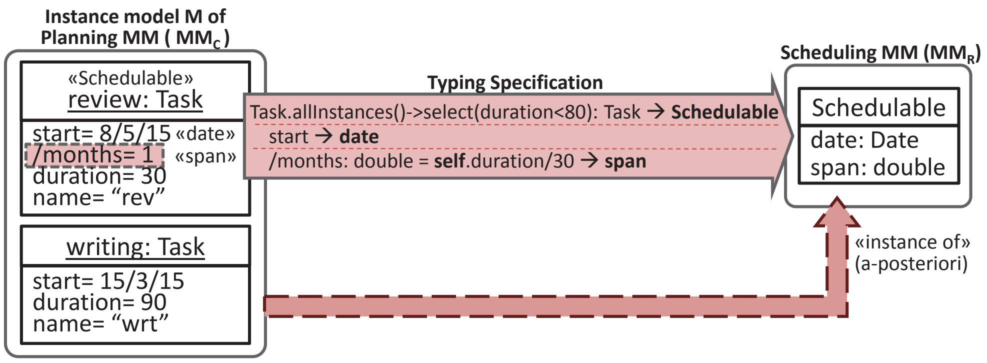


Fig. 13: Specification of a-posteriori typing at the instance-level

— *Levels of typing*. Type-level a-posteriori typing does not introduce additional meta-levels. However, $TS^{\oplus}$ has the same structure as $type$, and hence, it could be interpreted as a typing relation between MM_C and MM_R.

— *Orthogonal typing*. Type-level a-posteriori typing introduces additional typings for a model, which are orthogonal to the creation types. However, it does not consider linguistic typing but only ontological typing.

— *Type equivalence*. Type-level a-posteriori typing does not introduce native mechanisms for structural typing, but can emulate it. This is so as a specification TS may assign the same a-posteriori type to objects with different creation type but similar structure. This can be done by mapping their classes to the same class in MM_R. However, the converse (assigning different a-posteriori types to objects with same creation type) cannot be achieved because each ts_i maps classes to classes and is not able to select a subset of the instances of a class.

Later, we will analyse other properties that type-level specifications may have, like bidirectionality (Section 6) or the possibility to modify retyped models using operations typed on MM_R (Section 7). Before, the next section introduces a more expressive approach to specify a-posteriori typings at the instance level, and shows how to translate type-level specifications into them.

5. SPECIFYING A-POSTERIORI TYPING AT THE INSTANCE-LEVEL

The specification of a-posteriori typings at the instance-level consists of queries that are evaluated over the model to be retyped, and their results are assigned types from the role meta-model. Figure 13 shows an example of instance-level specification. The first line assigns the a-posteriori type Schedulable to all tasks with duration less than 80, and therefore, object review receives this a-posteriori type, but object writing does not.

In this section, we first describe instance-level specifications and their well-formedness rules (Section 5.1). Some of these rules require constructing OCL invariants, and the use of constraint solving. Hence, in contrast to type-level specifications, they are treated semi-formally only. Then, we explain how to calculate retyping views (Section 5.2), next how to translate type-level specifications into instance-level ones (Section 5.3), and finally, we analyse the features of instance-level specifications (Section 5.4).

5.1. Instance-level specifications and well-formedness rules

An instance-level a-posteriori typing specification from a creation meta-model MM_C to a role meta-model MM_R is (conceptually) a collection of partial functions $IS = \{is_i\}_{i \in I}$

from instances of MM_C (objects and their properties) to elements of MM_R (classes, attributes and references). The is functions are specified by means of queries. Hence, we have functions $is_i(exp : A) \mapsto A'$, where exp is a query that returns a set of objects of type $A \in MM_C$, and each object in this set is assigned type $A' \in MM_R$.

We do not assume a particular query language in retyping expressions, but one could use for example OCL or graph constraints [Ehrig et al. 2006]. In our implementation and for illustration we use OCL. The only assumption about the query language is that a query $exp : A$ should be evaluable on a model M, written $exp(M)$, and the resulting objects of the query should be of type A.

As an example, Figure 14 shows an ill-typed instance-level specification IS made of a function is_0 with three mappings. The queries in the mappings include the type of the returned objects (e.g., A), but not the cardinality (e.g., A[1..*]) or kind (e.g., Bag(A)) of the returned collection. The query in the first mapping is well-typed, because the objects returned by B.allInstances() are compatible with the declared type A. The query in the second mapping is incorrectly typed because A.allInstances() is not compatible with a collection of B. The query in the third mapping is well-typed. Actually, the type defined by the queries is not needed when *using* the retyping specification, as the queries are just evaluated on the models; however, this type is needed to *type-check* the specification for correctness, as we will see later.

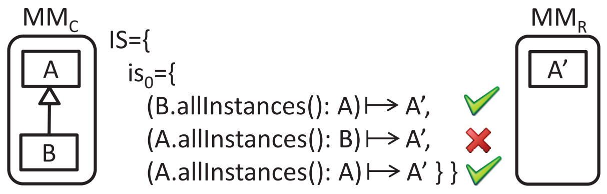

MM_C
A
B
IS={
is_0={
(B.allInstances(): A)$\mapsto$ A',
(A.allInstances(): B)$\mapsto$ A',
(A.allInstances(): A)$\mapsto$ A' } }
MM_R
A'

Fig. 14: Ill-typed instance-level specification

Object slots can also be assigned a-posteriori types from MM_R using functions $is_i(A.a) \mapsto A'.a'$. Such functions require the existence of an object mapping $is_i(exp : A) \mapsto A'$, meaning that the slots with type a of the objects in $exp(M)$ are retyped to $A'.a'$. Similar to type-level specifications, instance-level specifications can include the definition of derived features to be mapped to features in MM_R. Derived features are defined by a name, an expression, a type and a cardinality (see e.g., the definition of the derived feature months in Figure 13). Similar to the type-level case, derived features are not defined in MM_C, but directly in the specification.

Dually, one can see instance-level specifications as type-level specifications where class mappings include a filter $exp : A$ that selects only those A objects in $exp(M)$. While this may look a small change, it enhances expressivity at the cost of complicating the analysis of the well-formedness rules as, should we use analogous rules to the type-level specification ones, we would obtain too strict conditions. This is so as some of these rules define sufficient but not necessary conditions for instance-level specifications (in particular, the rules for compatibility of feature cardinality and composition), while others are necessary but not sufficient (the rule for compatibility of reference type). Our approach is to encode some correctness aspects of a specification as OCL expressions, and check the satisfiability of the expressions using a model finder [Jackson 2002; 2006; Kuhlmann and Gogolla 2012]. Model finders take as input a meta-model and a number of OCL invariants, and produce a model conformant to the meta-model and satisfying the invariants, if such a model exists within the search bounds.

Next, we revise the different well-formedness rules for instance-level specifications.

— *Non-abstract mappings*. Objects cannot be assigned an abstract type from MM_R.

$$\forall A \in MM_C, \forall A' \in MM_R \bullet is_i(exp : A) = A' \implies \neg abs(A') \qquad (20)$$

Remark. This rule is equivalent to Equation (10) for type-level specifications.

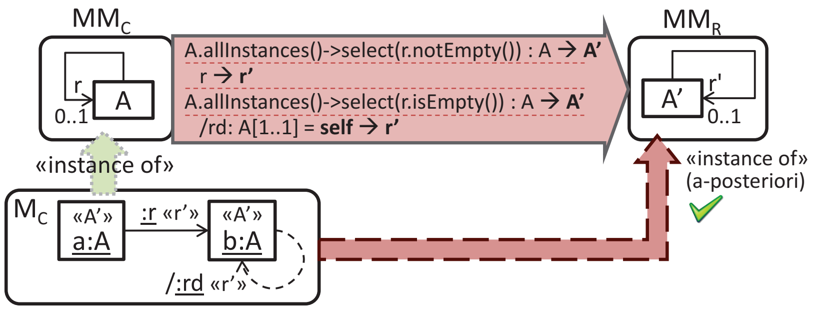

MM_C
r
A
0..1
A.allInstances()->select(r.notEmpty()) : A → A'
r → r'
A.allInstances()->select(r.isEmpty()) : A → A'
/rd: A[1..1] = self → r'
MM_R
A'
r'
0..1
«instance of»
«instance of» (a-posteriori)
M_C
«A'» a:A
:r «r'»
«A'» b:A
/:rd «r'»

Fig. 15: Correct instance-level specification

— *Correct context for slot mappings*. The slots of any object can only be mapped to owned or inherited features of the class the object is mapped to.

$$\begin{aligned}\forall A \in MM_C, \forall a \in feats(A), \forall a' \in MM_R \bullet\ & is_i(A.a) = a' \\ \implies\ & is_i(exp : A)\ is\ defined \land a' \in feats(is_i(exp : A))\end{aligned} \tag{21}$$

Remark. This rule is similar to Equation (11) for type-level specifications.

— *Compatibility of feature cardinality*. If an instance-level specification contains some mapping for a feature $a' \in MM_R$, then, the cardinality of a' must be the same or wider than the sum of the cardinality intervals of the slots mapped to a'.
Remark. This rule is similar to Equations (12) and (13) for type-level specifications. However, in this case, these equations are sufficient but not necessary conditions. This is so as fulfilling both equations ensures the retyped model will conform to MM_R; however, there may be instance-level specifications that do not fulfil the equations but always produce correct retyped models nonetheless. To detect such cases and widen the number of valid instance-level specifications we resort to constraint solving.
Example. Figure 15 illustrates this situation. The first two lines in the instance-level specification map A objects with non-empty r to A', and their link r to r'. The last two lines in the specification map A objects with empty r to A' as well, and a derived attribute rd which contains the object self to r'. This specification violates Equation (13) as the cardinality of r' ([0..1]) is not wider than the sum of the cardinalities of the two mapped links ([1..2]); however, because rd is only defined if r is empty, this specification always yields correct retyped models, one of which is shown in the figure (the dotted link is the derived feature rd as a result of the retyping). Hence, to analyse more precisely the compatibility of cardinalities in this example, we need to ensure that the invariant in Listing 1 is not satisfiable in retyped models. The invariant checks the upper cardinality of r', but not the lower one as it is zero.

```
A'.allInstances()→exists(a | a.r'→size() > 1)
```

Listing 1: Checking cardinality constraints on retyped models (in MM_R)

However, this invariant needs to be analysed in the context of MM_C, taking into account the retyping specification. Hence, we translate the invariant in terms of MM_C by using the retyping specification IS. This yields the invariant in Listing 2. The translation wraps objects into sets when necessary, to allow querying their size.

```
−− First mapping
A.allInstances()→select(r.notEmpty()) −− objects in this set are mapped to A'
   →exists(a | a.r→size() > 1) −− link r is mapped to A'.r'
```

```
or
-- Second mapping
A.allInstances()→select(r.isEmpty()) -- objects in this set are mapped to A'
   →exists(a | Set{a}→size() > 1) -- derived link /rd = self is mapped to A'.r'
or
-- Overlapping instances of first and second mapping
A.allInstances()→select(r.notEmpty())→intersection( A.allInstances()→select(r.isEmpty()) )
   →exists(a | (a.r→size() + Set{a}→size()) > 1)
```

Listing 2: Checking cardinality constraints for the specification of Figure 15 (in MM_C)

The model finder does not find any instance of MM_C that satisfies the invariant in Listing 2. This means that the retyping specification preserves the cardinalities in MM_C.

Invariant construction. In general, the invariant needed to check cardinality compatibility is built as follows. Given a feature $A'.r'$ in MM_R; a set $J = \{j1, ..., jn\}$ of indices with $is_j(exp_{j1} : A) = A'$; and $is_j(A.r_{11}) = A'.r'$, ..., $is_j(A.r_{1m}) = A'.r'$; we use the template in Listing 3 to build the invariant. We assume the general case in which $A.r_{11}$ ... $A.r_{1m}$ are derived features defined by OCL expressions exp_{s11} ... exp_{s1m}. If they are regular feature mappings, then the expressions have the form $exp_{s11} = self.r_{11}$ ... $exp_{s1m} = self.r_{1m}$. In the template, $exp_{sij}[a/self]$ is the expression that results from substituting $self$ (either implicit or explicit) by a in exp_{sij}[4]. Moreover, the checks for the maximum (resp. minimum) cardinality of $A'.r'$ are not needed if its upper (resp. lower) cardinality is $*$ (resp. 0).

```
1  exp_j1→exists(a | let size : Integer = exp_s11[a/self] →size() +...+ exp_s1m[a/self] →size()
2                      in size > max(A'.r') or size < min(A'.r'))
3  or ...
4  exp_jn→exists(a | let size : Integer = exp_sn1[a/self] →size() +...+ exp_snm[a/self] →size()
5                      in size > max(A'.r') or size < min(A'.r'))
6  or
7  -- all twofold combinations of expressions in J with compatible type
8  exp_j1→intersection(exp_j2)→exists (a |
9     let size : Integer = exp_s11[a/self] →size() +...+ exp_s1m[a/self] →size() +
10                          exp_s21[a/self] →size() +...+ exp_s2m[a/self] →size()
11    in size > max(A'.r') or size < min(A'.r'))
12 or ... -- similar for 3-fold,..., n-fold combinations, with J = {j1, ..., jn}
```

Listing 3: Template to generate OCL invariants for cardinality checks

The resulting invariant checks the minimum and maximum cardinalities of the individual mappings to feature $A'.r'$ (lines 1–6), and then, it considers objects selected by several mappings (lines 8–12). If this expression is satisfiable, then, there is some model of MM_C that violates the cardinality of r' when it is retyped using IS.

— *Compatibility of reference type*. Checking the compatibility of the target type of references has two parts, one static and the other one using model finding. The static part amounts to check a similar condition to Equation (14), namely:

$$\begin{aligned}\forall A \in MM_C, \forall r \in refs(A), \forall r' \in MM_R \bullet is_i(A.r) = r' \implies \\ \exists\, C \in anc^*(tar(A.r)) \bullet is_i(exp_C : C) \in sub^*(tar(r'))\end{aligned} \tag{22}$$

Remark. This equation ensures that if the feature $A.r$ is mapped, then there is at least one mapping from objects of type $tar(A.r)$ that is compatible with $tar(r')$. Even though this condition is sufficient for type-level specifications, it is not for instance-level ones.

[4]In this and the following templates, we assume that the variable a does not occur free in the replaced expression (exp_{sij} in this case).

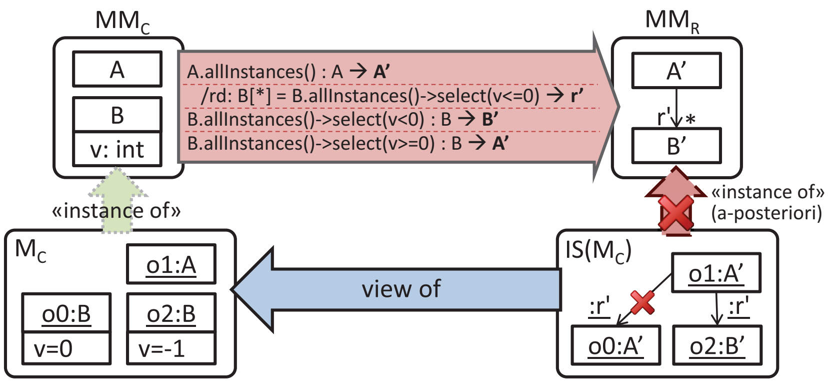

MM_C
A
B
v: int
A.allInstances() : A → A'
/rd: B[*] = B.allInstances()->select(v<=0) → r'
B.allInstances()->select(v<0) : B → B'
B.allInstances()->select(v>=0) : B → A'
MM_R
A'
r'
*
B'
«instance of»
«instance of» (a-posteriori)
M_C
o1:A
o0:B
v=0
o2:B
v=-1
view of
IS(M_C)
o1:A'
:r'
:r'
o0:A'
o2:B'

Fig. 16: Incorrect instance-level specification leading to an ill-typed model

Example. Figure 16 illustrates why Equation (22) is not sufficient. The instance-level specification in the figure maps all A objects to A', and their derived feature rd containing all B objects with zero or negative value for v is mapped to r'. Moreover, B objects with negative v are mapped to B', and those with zero or positive v are mapped to A'. This specification fulfils Equation (22), as rd: B[*] is mapped to A'.r': B'[*], and there is a mapping that retypes B objects into B'. However, object o0 in M_C is retyped as A', and therefore, the link from o1 to o0 is incorrectly typed because r' should only contain B' objects.
We could strength Equation (22) as follows, to demand *every* mapping from objects of type $tar(A.r)$ and its ancestors to be compatible with $tar(r')$:

$$\begin{aligned}\forall A \in MM_C, \forall r \in refs(A), \forall r' \in MM_R \bullet\ is_i(A.r) = r' \implies \\ \exists\, C \in anc^*(tar(A.r)) \bullet\ is_i(exp_C : C) \in sub^*(tar(r')) \wedge \\ \forall\, k \in I, \forall\, D \in anc^*(tar(A.r)) \bullet \\ is_k(exp_k : D)\ is\ defined \implies is_k(exp_k : D) \in sub^*(tar(r'))\end{aligned} \tag{23}$$

This would render the specification of Figure 16 as invalid because there is a mapping of B objects to A', which is not compatible with the type of r'. However, this equation is too strict as it can invalidate specifications which do not yield incorrect retypings, like for instance, changing the $v <= 0$ condition to $v < 0$ in the example. Therefore, we use model finding to analyse correctness more precisely. The idea is to derive an OCL invariant which, if satisfiable, implies that the specification is incorrect. Listing 4 shows such an invariant for the specification of Figure 16, which checks if r' can contain objects with type different from B.

```
A'.allInstances()→exists(a | −− There is an object a with type A'
    a.r'→exists(b | −− whose link r' contains an object b
        B'.allInstances()→excludes(b))) −− such that the type of b is not B'
```

Listing 4: Checking correctness of typings to $A'.r'$ (in MM_R)

Model $IS(M_C)$ in Figure 16 satisfies this invariant as object o0 is in r' but its type is not B'. Hence, the specification is incorrect. However, as in the previous rule, our goal is to evaluate the invariant in the context of MM_C to assert there is no model of MM_C satisfying it. For this purpose, we translate the previous invariant using the specification IS, obtaining the invariant in Listing 5.

```
A.allInstances()→exists(a | −− First mapping to A'
    B.allInstances()→select(v<=0)→exists(b |      −− mapping to A'.r'
        B.allInstances()→select(v<0)→excludes(b)))  −− b does not have type B'
```

Listing 5: Checking correctness of typings to $A'.r'$ for specification in Figure 16 (in MM_C)

This invariant detects when the a-posteriori type of an object b is not valid for the reference A'.r', either because the type is incompatible with B' (as is the case of object o0 in the example), or because b is selected for its inclusion in A'.r' but it does not receive an a-posteriori type at all. Model M_C in Figure 16 satisfies this invariant as object o0 fulfils the condition in line 3.

Invariant construction. In the general case, the invariant is constructed as follows. Given a reference $A'.r'$ in MM_R with $tar(A'.r') = B'$; and given an instance-level specification with $is_i(exp_A : A) = A'$, $is_i(A.r) = A'.r'$, $is_{i_1}(exp_{B_1} : B_1) = B'_1$, ..., $is_{i_n}(exp_{B_n} : B_n) = B'_n$ with $B'_i \in sub^*(B')$ and $B_i \in anc^*(tar(A.r))$, we build the invariant of Listing 6. As in Listing 3, we consider that $A.r$ could be derived and it is defined by an OCL expression exp_s. If $A.r$ is not derived, then exp_s has the form $self.r$.

```
exp_A→exists(a |
    exp_s[a/self]→exists(b |
        exp_B1→...→union(exp_Bn)→excludes(b)))
```

Listing 6: Template to generate OCL invariants for checking reference correctness

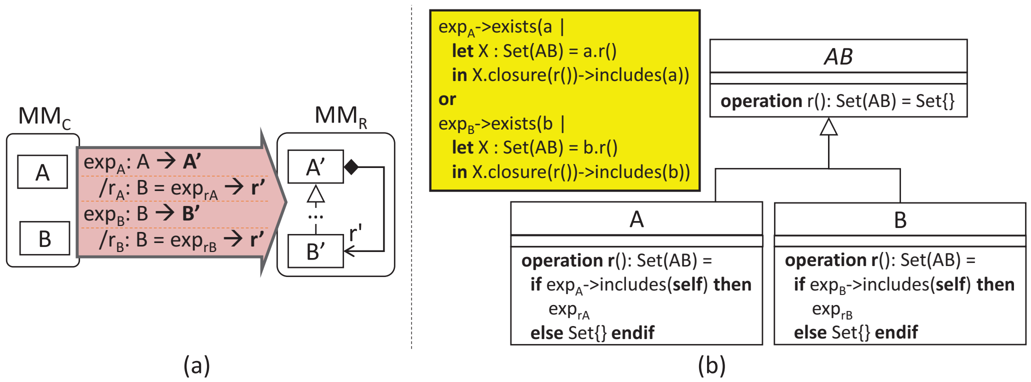
If there are several mappings $is_{i_1}(exp_{A_1} : A_1)$, ..., $is_{i_m}(exp_{A_m} : A_m)$ to some class $D' \in sub^*(A')$, then we build the invariant in Listing 3 for each mapping and or-concatenate all of them (as D' inherits r'). If the resulting invariant is satisfiable, the typing is not safe.

— *Compatibility of composition*. Retyped models cannot contain cycles of containment links, and objects cannot be pointed to by two containment links.
Remark. This can be checked with similar conditions to Equations (15) and (16) (i.e., compositions in MM_R can only be mapped by compositions, and one composition in MM_C cannot be mapped to two compositions in MM_R). However, instance-level specifications make heavy use of derived references which do not indicate whether they are considered "compositions". Hence, we resort to constraint solving to check the compatibility of compositions.
Invariant construction. Given a containment reference $A'.r'$ with $tar(A'.r') \in sub^*(A')$, we can use the invariant in Listing 7 to detect cycles in instances of MM_R.

```
A'.allInstances()→exists(a | a.r'→closure(r')→includes(a))
```

Listing 7: Checking cycles of reference $A'.r'$ (in MM_R)

We need to translate this invariant in terms of MM_C to check if it is unsatisfiable on its instances. Hence, given a reference $A'.r'$ with $comp(A'.r')$ and $tar(A'.r') \in sub^*(A')$ (see Figure 17(a)), and an instance-level specification with $is_i(exp_A : A) = A'$, $is_i(A.r_A) = A'.r'$, $is_i(exp_B : B) = B'$, $is_i(B.r_B) = B'.r'$ and $B' \in sub^*(A')$, we build the invariant shown in Figure 17(b). As before, we assume that $A.r_A$ and $B.r_B$ are defined by OCL expressions exp_{rA} and exp_{rB}. To be able to perform the transitive closure, we need to introduce a new abstract class $AB \in MM_C$, parent of both A and B. This class defines an operation $r()$, redefined in A and B, which emulates the mappings to r'. If the resulting invariant is satisfiable in MM_C, then the specification may produce retyped models that contain cycles of $A'.r'$. Should A' defined more compositions in addition to r', mapped from references in A or B, then operation $r()$ should return the content of all of them.



```
expA->exists(a |
  let X : Set(AB) = a.r()
  in X.closure(r())->includes(a))
or
expB->exists(b |
  let X : Set(AB) = b.r()
  in X.closure(r())->includes(b))
```

```
operation r(): Set(AB) = Set{}
```

```
operation r(): Set(AB) =
  if expA->includes(self) then
    exprA
  else Set{} endif
```

```
operation r(): Set(AB) =
  if expB->includes(self) then
    exprB
  else Set{} endif
```

Fig. 17: Checking composition cycles. (a) Schema of specification. (b) Template to generate OCL invariants for checking cycles of composition, and meta-model modification.

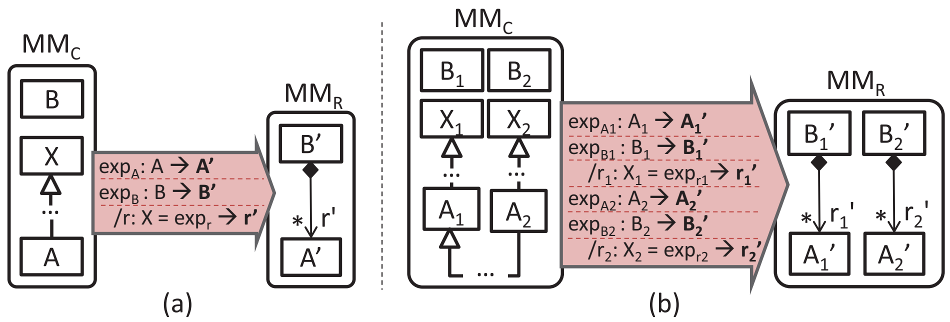

MM$_C$
B
X
A
exp$_A$: A → **A'**
exp$_B$: B → **B'**
/r: X = exp$_r$ → **r'**
MM$_R$
B'
A'
*
r'
(a)
MM$_C$
B$_1$
B$_2$
X$_1$
X$_2$
A$_1$
A$_2$
exp$_{A1}$: A$_1$ → **A$_1$'**
exp$_{B1}$: B$_1$ → **B$_1$'**
/r$_1$: X$_1$ = exp$_{r1}$ → **r$_1$'**
exp$_{A2}$: A$_2$ → **A$_2$'**
exp$_{B2}$: B$_2$ → **B$_2$'**
/r$_2$: X$_2$ = exp$_{r2}$ → **r$_2$'**
MM$_R$
B$_1$'
B$_2$'
A$_1$'
A$_2$'
r$_1$'
r$_2$'
(b)

Fig. 18: Schema of instance-level specifications considered in the compatibility of compositions. (a) Mapping to one composition. (b) Mapping to two compositions.

In addition to acyclicity, we need to check that no object is contained in two links whose type is a composition. We proceed in two steps: first, we check that no object can be contained in two links with the same composition type, and second, we check that no object is contained in two links with different composition type.
Figure 18(a) shows a schema of the instance-level specification for the first case. In detail, for each composition reference $B'.r' \in MM_R$, we generate the invariant in Listing 8. Line 1 checks whether some object selected by exp_B and mapped to B' contains the same object duplicated in r (with r calculated by exp_r). Lines 3–6 check whether some object selected by exp_A and mapped to A', is included in the collection r of two B objects. If the invariant is satisfiable for some composition reference, then the specification may produce retypings with incorrect compositions.

```
1 expB→exists(b | expr[self/b]→asSet()→size() < expr[self/b]→size())
2 or
3 expA→exists(a |
4    expB→exists(b1, b2 |
5        b1 <> b2 and
6        expr[self/b1]→includes(a) and expr[self/b2]→includes(a)))
```

Listing 8: Template to generate OCL invariants for detecting objects contained in two links with the same composition type (in MM_C)

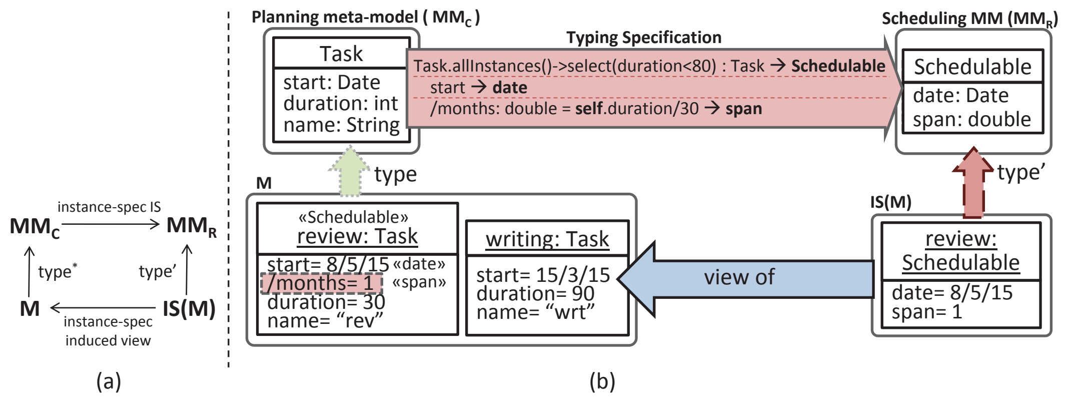


Fig. 19: (a) Retyping M with respect to MM_R. (b) Example of retyping.

To check that no object can be included in two different types of composition, we use the invariant shown in Listing 9 for each pair of composition references $r'_1, r'_2 \in MM_R$. This invariant is created only if there are two expressions exp_{A1} and exp_{A2} with compatible type (and hence overlapping instances), both mapped to two classes $A'_1, A'_2 \in MM_R$ target of different composition references (see Figure 18(b)). If some of the created invariants are satisfiable, then the specification is incorrect.

```
exp_A1→intersection(exp_A2)→exists(a |
    exp_B1→exists(b1 | exp_r1[self/b1]→includes(a))
    and
    exp_B2→exists(b2 | exp_r2[self/b2]→includes(a)))
```

Listing 9: Template to generate OCL invariants for detecting objects contained in two links with different composition type (in MM_C)

— *Complete instantiations*. If an instance-level specification contains some mapping for a class $A' \in MM_R$, then all mandatory features of A' should be mapped as well or have a default value.

$$\forall A \in MM_C, \forall A' \in MM_R, \forall a' \in feats(A') \bullet is_i(exp : A) = A' \wedge mand(a') \implies default(a') \vee \exists a \in feats(A) \bullet \neg default(a) \wedge is_i(A.a) = A'.a' \quad (24)$$

Remark. This rule is equivalent to Equation (17) for type-level specifications, and can be checked statically.

5.2. Computation of retyped model views

As Figure 19(a) shows, retyping a model M conforming to MM_C via an instance-level specification IS is similar to the case of type-level specifications (see Section 4.2). The only difference is that, in this case, given a model M and an object $o \in M$, o gets retyped by all mappings containing queries that select o. Hence:

$$type'(o) = \{is_k(exp_A : A) \mid k \in I \wedge is_k(exp_A : A)\, is\, defined \wedge A \in type^*(o) \wedge o \in exp_A(M)\}$$

We write $IS(M)$ for the retyped view of M using IS. The retyping yields a typing $type'$ from $IS(M)$ to MM_R. Figure 19(b) shows the retyped view $IS(M)$ of Figure 13. The view only contains one object, and ignores slots duration and name.

The following theorem states that instance-level specifications yield correct typings that fulfil the well-formedness criteria defined in Section 3.

THEOREM 5.1 (INSTANCE-LEVEL RETYPING CORRECTNESS).

Given meta-models MM_C and MM_R, an instance-level specification IS and a model M typed by MM_C, then $IS(M)$ is well-typed with respect to MM_R using $type'$ according to the well-typed criteria of Section 3. The typing $type'(IS(M))$ is total on objects.

PROOF. (Idea) Similar to Theorem 4.1, we check each condition for well-typing:

— Equation (1) *"objects are well-typed"* holds by the construction of $type'(o)$ in $IS(M)$.
— Equation (2) *"subclassing implies instance subsetting"* holds due to the definition of $type^*$.
— Equation (3) *"abstract classes do not have instances"* holds due to Equation (20) *"non-abstract mappings"*.
— Equation (4) *"slot correctness"* follows from (21) *"correct context for feature mappings"*.
— Equation (5) *"slot completeness"* follows from (24) *"complete instantiations"*.
— Equation (6) *"link correctness"* holds if the invariant in Listing 6 is unsatisfiable.
— *"Composition correctness"* has two parts: Equation (7) holds if the invariants in Listings 8 and 9 are unsatisfiable, and Equation (8) holds if the invariant in Figure 17(b) is unsatisfiable.
— Equation (9) *"cardinality correctness"* holds if the invariant in Listing 3 is unsatisfiable.

See details in appendix. □

5.3. From type-level to instance-level specifications

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

$$\begin{aligned} is_i = &\{(A.allInstances() : A) \mapsto A' \mid ts_i(A) = A'\} \cup \\ &\{(A.f : A) \mapsto A'.f' \mid ts_i(A.f) = A'.f' \ with\ f \in feats(A)\} \cup \\ &\{(A./da : T, exp_{da}) \mapsto A'.f' \mid ts_i(A./da : T, exp_{da}) = A'.f' \ with\ da\ a\ derived \\ &\ feature\ with\ type\ T\} \end{aligned} \tag{25}$$

The construction in Equation (25) translates each type-level class mapping $ts_i(A) = A'$ to an instance-level mapping that maps all instances of A to A'. The translation of feature mappings is straightforward.

A well-formed type-level specification yields a well-formed instance-level specification, as the next theorem shows.

THEOREM 5.2 (TYPE-TO-INSTANCE TRANSLATION CORRECTNESS).
Given a type-level specification $TS = \{ts_i\}_{i\in I}$, its translation into an instance-level specification using the construction in Equation (25) yields a correct instance-level specification $IS = \{is_i\}_{i\in I}$.

PROOF. (Idea). By checking each well-typing condition. See details in appendix. □

In general, instance-level specifications cannot be translated into equivalent type-level ones. This is so as, for example, an instance-level specification can map two different sets of objects of type $A \in MM_C$ to two different classes $A', B' \in MM_R$. This is not possible in type-level specifications.

5.4. Features of instance-level retyping specifications

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

— *Classification time*. Just like type-level specifications, instance-level specifications enable a-posteriori typings.
— *Dynamicity*. The defined a-posteriori typing can be dynamic if the specification contains mappings that select or deselect objects according to their features. For example, in Figure 19(b), setting writing.duration to 30 makes writing Schedulable, while setting review.duration to 80 makes review drop its Schedulable a-posteriori type.
— *Number of classifiers*. Objects may have several a-posteriori types from the same role meta-model, if they are selected by several mappings to different MM_R classes.
— *Number of model types*. As in the case of type-level specifications, a model M can be typed a-posteriori with respect to several role meta-models.
— *Totality*. Objects not selected by any query lack an a-posteriori type, in which case, they are excluded from the view $IS(M)$. This is allowed because the defined a-posteriori typing can be partial.
— *Levels of typing*. Instance-level specifications do not introduce new meta-levels.
— *Orthogonal typing*. As in the case of type-level specifications, instance-level specifications introduce an additional typing of ontological nature.
— *Type equivalence*. Similar to type-level specifications, objects with different creation type can receive the same a-posteriori type, if they are selected by the same or different queries and mapped to the same type in MM_R. However, differently from type-level specifications, instance-level specifications also allow mapping objects with the same creation type to different types in MM_R. This is possible if the specification contains queries that distinguish the objects and map them to different types.

6. ANALYSIS OF A-POSTERIORI TYPING

The well-formedness rules of type-level and instance-level specifications ensure correct retypings, but neglect the OCL integrity constraints that the involved meta-models MM_C and MM_R may have. Therefore, we are interested in analysing whether given a retyping specification from a creation meta-model MM_C to a role meta-model MM_R, some/every valid instance of MM_C becomes a valid instance of MM_R when the retyping is performed and the integrity constraints of both meta-models are considered.

As we have seen, retyping a model M_C with respect to MM_R creates a virtual model view of M_C, named $TS(M_C)$ for type-level specifications and $IS(M_C)$ for instance-level specifications. In both cases, the view discards the elements of M_C which are not typed by MM_R, and includes the derived features defined by the specification. Thus, the goal of the proposed analysis is to check whether:

(a) a valid view for some valid model M_C exists (*retyping executability*),
(b) a valid view for every valid model M_C exists (*retyping totality*),
(c) every instance of MM_R is a view of some instance of MM_C (*retyping surjectivity*),
(d) a retyping specification from MM_C to MM_R can be reversed to obtain a valid specification from MM_R to MM_C (*retyping bidirectionality*).

Executability is a basic property of specifications, which indicates that at least one model can be correctly retyped taking into account the integrity constraints of both meta-models. A specification is *total* if it can be applied to every instance of MM_C, and yields a correct view. A *surjective* specification is able to produce every instance of MM_R by retyping some source model. A *bidirectional* specification can be used to retype instance models of MM_C into MM_R and the other way round. In a way, retypings act as (simple) model-to-model transformations where no objects are produced in the target model, but the target model is calculated as a view of the source.

Figure 20 illustrates the conditions for the first three retyping properties. The figure uses sets to depict the instances of MM_C and MM_R, dots to represent a model or a view, and C_C and C_R to represent the sets of integrity constraints in MM_C and MM_R

respectively. Hence, a model showing executability satisfies the constraints C_C of the creation meta-model, and its retyped view satisfies those of the role meta-model. A model showing no totality satisfies the constraints C_C of the creation meta-model, but its retyping does not satisfy the constraints C_R of the role meta-model. Finally, a model showing no surjectivity is one that, once retyped, satisfies the constraints of the role meta-model, but the model itself does not satisfy the constraints of the creation meta-model.

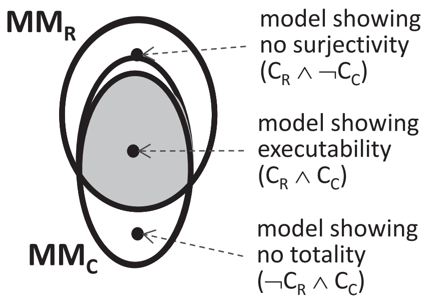

MM_R
model showing no surjectivity ($C_R \wedge \neg C_C$)
model showing executability ($C_R \wedge C_C$)
MM_C
model showing no totality ($\neg C_R \wedge C_C$)

Fig. 20: Conditions for executability, totality and surjectivity of retypings.

These properties are interesting for both type-level and instance-level specifications, though in this paper, we focus on their analysis for type-level ones. For instance-level specifications, we need to translate the constraints in MM_R to the context of MM_C using the expressions in the specification mappings. While this is technically possible, e.g., rewriting the abstract syntax tree of the OCL constraints in the style of [Clarisó et al. 2016], we leave the details to future work.

In Section 6.1 we present analysis techniques for executability, totality and surjectivity that consider the general case in which MM_C and MM_R may have OCL integrity constraints. The techniques rely on OCL manipulation and model finding, and hence, they are presented semi-formally only. In Section 6.2 we introduce conditions for bidirectionality that can be checked statically.

6.1. Executability, totality and surjectivity

Our analysis for type-level specifications is inspired by [Guerra and de Lara 2017]. The idea is to construct an analysis meta-model MM_A, merging MM_C and MM_R according to the typing specification TS. MM_A contains all integrity constraints of MM_C and MM_R. In this way, retyping executability can be analysed by checking the instantiability of MM_A, while totality and surjectivity can be analysed by negating the constraints of MM_R or MM_C in MM_A respectively.

The analysis meta-model MM_A contains all classes, features and inheritance relations of MM_C and MM_R. However, the classes of MM_R are made abstract in MM_A to avoid their instantiation, as conceptually, we want to obtain instances of MM_C. Moreover, each integrity constraint in MM_C and MM_R is encoded as an operation in the context class of the constraint. This is convenient to be able to negate the constraints when analysing totality and surjectivity.

Then, we extend MM_A to take into account the specification TS. In particular, every class mapping is converted into an inheritance relation (see Figure 21(a)). The rationale is that the retyping TS makes every instance of A to be seen as an instance of A'. Every attribute mapping is encoded as a constraint demanding the same value for the mapped attributes (see Figure 21(b)). This is a way to emulate redefinition, but keeping both attributes to maintain the validity of any integrity constraint from MM_C and MM_R in MM_A. Similarly, mappings for derived features are encoded as constraints (see Figure 21(c)). Finally, reference mappings are also encoded as constraints demanding equal reference values (see Figure 21(d)). We consider that two references a and a' have the same value if a contains all elements in a', a' contains all elements in a, and the size of both collections is the same[5]. Moreover, if several references from the same class in MM_C are mapped to the same reference in MM_R, then we consider the union of all references in MM_C when checking for equality (see Figure 21(d)).

[5]For simplicity, we consider references as sets, i.e., we neglect sequences and bags.

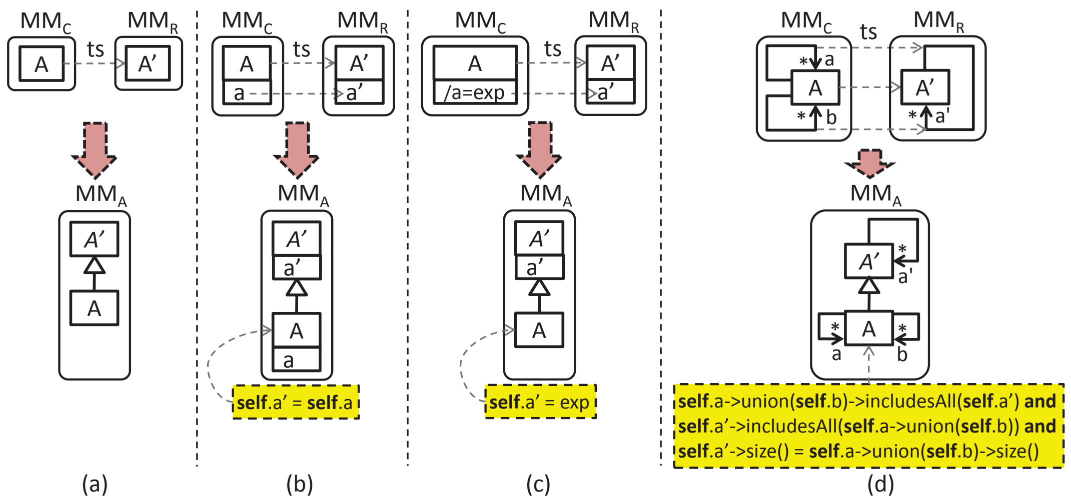

MMC
MMR
ts
A
A'
a
a'
/a=exp
MMA
self.a' = self.a
self.a' = exp
self.a->union(self.b)->includesAll(self.a') and
self.a'->includesAll(self.a->union(self.b)) and
self.a'->size() = self.a->union(self.b)->size()
(a)
(b)
(c)
(d)

Fig. 21: Translation of retyping specification into analysis meta-model: (a) Class mappings become inheritance relations; (b) Attribute mappings become constraints; (c) Derived features become constraints; (d) Reference mappings become constraints.

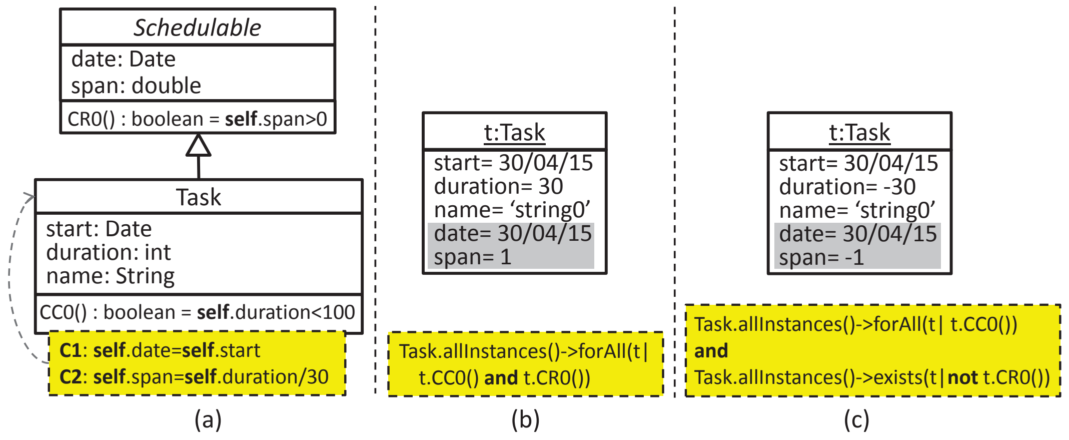


Fig. 22: (a) Analysis meta-model MM_A. (b) Extra invariant to analyse executability, and model showing executability. (c) Extra invariant to analyse totality, and model showing non-totality.

As an example, Figure 22(a) shows the analysis meta-model MM_A for the type-level specification in Figure 5. For illustration, we assume that class Schedulable declares the constraint span>0, and Task declares the constraint duration<100. These constraints are encoded as operations CR0 and CC0 respectively. The inheritance relation is created due to the class mapping Task → Schedulable, and Schedulable is made abstract. Constraint C1 in Task is created due to the mapping start → date, while C2 is created due to the mapping \months=self.duration/30 → span.

Executability is proved by finding an instance of the analysis meta-model that satisfies all constraints from MM_C and MM_R. Thus, we have to find a model where the invariant Task.allInstances()→forAll(t | t.CC0() and t.CR0()) holds. Technically, to achieve this, we add this invariant to a dummy class Context in MM_A, and require one instance of this class. Then, we use a model finder to check whether such a model exists. In this

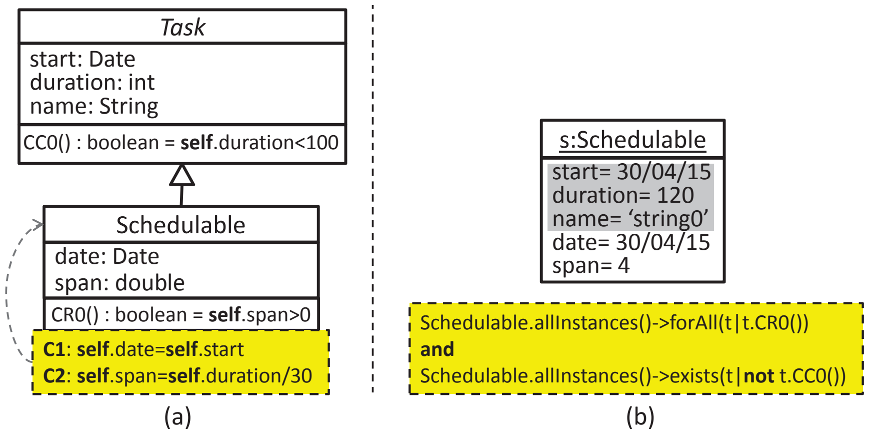

Task
start: Date
duration: int
name: String
CC0() : boolean = self.duration<100
Schedulable
date: Date
span: double
CR0() : boolean = self.span>0
C1: self.date=self.start
C2: self.span=self.duration/30
(a)
s:Schedulable
start= 30/04/15
duration= 120
name= 'string0'
date= 30/04/15
span= 4
Schedulable.allInstances()->forAll(t|t.CR0())
and
Schedulable.allInstances()->exists(t|not t.CC0())
(b)

Fig. 23: Additional constraint for MM_A showing non-surjectivity and retyped model w.r.t MM_R

case, the search produces the model in Figure 22(b), where we have to remove the auxiliary Context object, as well as the shaded features which belong to MM_R, if we want to obtain a valid instance of MM_C.

Non-totality is proved by finding an instance of MM_A that satisfies all constraints from MM_C but violates some from MM_R. Thus, in the example, all tasks should fulfil CC0, and some should violate CR0. That is, we have to find a model where the invariant Task.allInstances()→forAll(t | t.CC0()) and Task.allInstances()→exists(t | not t.CR0()) holds. Figure 22(c) shows one such model, as the duration of t is smaller than 100 (i.e., CC0 is true) but its span is negative (i.e., CR0 is false). Thus, the specification is not total.

For surjectivity, we build the analysis meta-model following the given procedure, but changing the roles of MM_C and MM_R. Thus, the classes in the creation meta-model become abstract in MM_A, and the classes in the role meta-model inherit from the creation classes to which they are mapped. This is so as, in this case, we want to obtain instances of MM_R. Figure 23(a) shows the new analysis meta-model for the example. Then, non-surjectivity is analysed by finding an instance of MM_A that satisfies all constraints from MM_R but violates some from MM_C. Figure 23(b) shows the invariant to check, as well as a model that satisfies the invariant (duration is bigger than 100, and hence, CC0 is false). However, differently from the previous properties, finding a model does not prove the specification is non-surjective, as there may be other instances of MM_C which also get retyped to the same model returned by the finder. In this case, we can manually assess that this is not the case, therefore, we conclude the specification is non-surjective.

The presented analysis relies on model finders that perform a bounded search of models up to a certain, predefined bound. If no model is found to demonstrate a property, one can increase the search scope and repeat the search. Choosing the right bound is a trade-off between the verification time (shorter for smaller bounds) and confidence (higher for bigger bounds). This choice is normally done by hand, although some recent efforts are being spent on its automation [Clarisó et al. 2015]. Nonetheless, according to the "small scope" hypothesis [Jackson 2006], a large proportion of errors in a system (or meta-model) can be identified by considering only instances within a small scope.

6.2. Bidirectionality

A specification is bidirectional if it can be used backwards to reclassify models of MM_R with respect to MM_C. In particular, TS is bidirectional if the specification TS^{-1} that results from reversing all mappings is well-formed, i.e., it satisfies all well-formedness conditions of Section 4.1. Bidirectionality is an important property when retypings are used as a mechanism to specify model transformations (see Section 9.1).

Bidirectionality can be checked statically with no need for model finding. It generally amounts to ask for the converse of the well-formedness rules of Section 4.1, as follows:

— *Non-abstract mappings*. Equation (10) forbids mappings *to* abstract classes of MM_R. Hence, to be bidirectional, TS cannot have mappings *from* abstract classes of MM_C.

$$\forall A \in MM_C, \forall A' \in MM_R \bullet ts_i(A) = A' \implies \neg abs(A) \tag{26}$$

— *Correct context for feature mappings*. Equation (11) demands features of a class to be mapped to features of the mapped class. Conversely, we require that if $A'.a'$ is mapped from a feature a, a class owning or inheriting a should be mapped to A'.

$$\begin{aligned}&\forall A' \in MM_R, \forall a' \in feats(A'), \forall a \in MM_C \bullet \\ &\quad ts_i(a) = A'.a' \implies \exists A \in MM_C \bullet ts_i(A) = A' \wedge a \in feats(A)\end{aligned} \tag{27}$$

— *Compatibility of feature cardinality*. Equation (12) demands equal or wider cardinality intervals for the features mapped in MM_R. Hence, to be bidirectional, TS can only map features with the same cardinality, see Equation (28). Equation (13) handles the general case in which several features of MM_C are mapped into a single feature of MM_R, demanding the interval of the latter to be equal or wider than the sum of the intervals of the former. Hence, to be bidirectional, TS requires that if a feature $a \in MM_C$ is mapped into several ones, then the aggregated cardinality intervals of the latter features should be equal to the cardinality of a. This is only possible if the intervals have bounds [0..*], as demanded by Equation (29).

$$\forall a \in MM_C, \forall a' \in MM_R \bullet ts_i(a) = a' \implies min(a) = min(a') \wedge max(a) = max(a') \tag{28}$$

$$\begin{aligned}&\forall a \in feats(A) \bullet ts_i(a)\ is\ defined \wedge ts_j(a)\ is\ defined \wedge ts_i(a) \neq ts_j(a) \implies \\ &\quad min(a) = 0 = min(ts_i(a)) \wedge max(a) = \infty = max(ts_i(a))\end{aligned} \tag{29}$$

— *Compatibility of reference type*. Equation (14) demands compatibility of the target of mapped references in MM_R. For bidirectionality, Equation (29) demands compatibility of the target of mapping references in MM_C.

$$\begin{aligned}&\forall A' \in MM_R, \forall r' \in refs(A'), \forall r \in MM_C \bullet ts_i(r) = A'.r' \implies \\ &\quad \exists\, C \in sub^*(tar(r)) \bullet ts_i(C) \in anc^*(tar(A'.r'))\end{aligned} \tag{30}$$

— *Compatibility of composition*. Equation (15) forbids mapping non-compositions in MM_C to compositions in MM_R. Hence, to be bidirectional, TS cannot map compositions in MM_C to non-compositions in MM_R, see Equation (31). In addition, Equation (16) forbids mapping a composition in MM_C to two compositions in MM_R. Therefore, to be bidirectional, TS cannot map two compositions in MM_C to a composition in MM_R, see Equation (32).

$$\forall r \in MM_C, \forall r' \in MM_R \bullet (ts_i(r) = r' \wedge comp(r)) \implies comp(r') \tag{31}$$

$$\begin{aligned}&\forall A' \in MM_R, \forall r' \in refs(A'), \forall r_1, r_2 \in MM_C, \forall i, j \in I \bullet \\ &\quad i \neq j \wedge ts_i(r_1) = A'.r' \wedge ts_j(r_2) = A'.r' \wedge r_1 \neq r_2 \implies \\ &\quad \neg\, comp(r_1) \vee \neg\, comp(r_2)\end{aligned} \tag{32}$$

— *Complete instantiations*. Equation (17) demands all mandatory features of mapped classes in MM_R to receive a mapping or have a default value. Hence, to be bidirectional, TS should map all mandatory features in MM_C without a default value, if

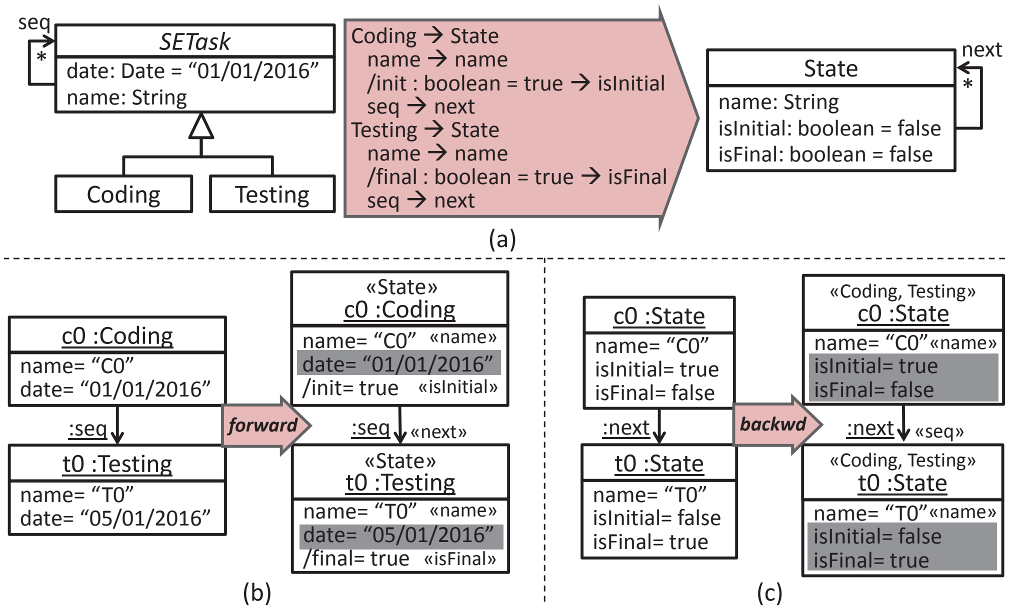


Fig. 24: (a) Bidirectional retyping specification. (b) Forward retyping. (c) Backward retyping.

their owner class (or a subclass) is mapped, see Equation (33).

$$\forall A \in MM_C, \forall A' \in MM_R, \forall a \in feats(A) \bullet ts_i(A) = A' \wedge mand(a) \implies default(a) \vee \exists a' \in feats(A') \bullet \neg default(a') \wedge ts_i(A.a) = A'.a' \tag{33}$$

— *Derived features*. Derived features defined in MM_C to calculate the value of features in MM_R are ignored in TS^{-1}. In practice, since there might be features in MM_C with no equivalent in MM_R, TS^{-1} may need to be manually extended with derived features over MM_R.

THEOREM 6.1 (BIDIRECTONALITY). *Given a well-formed type-level specification TS, TS^{-1} is well-formed if TS satisfies Equations (26)–(33).*

PROOF. (Idea) Equations (26)–(33) are the converse of the rules in Equations (10)–(17) for well-formedness of type-level specifications, or can be deduced from them. See details in the appendix. □

Figure 24(a) shows an example retyping specification TS from a simple software process meta-model to a meta-model for automata. The specification fulfils the conditions for bidirectionality, as it does not map the abstract class SETask, it maps references with the same cardinality, it does not map compositions, and all mandatory features are either mapped (name) or have an initial value (date). Figure 24(b) uses the specification TS for forward retyping a process model, and Figure 24(c) uses TS^{-1} for backward retyping an automaton. The initial automaton in (c) is the result of the retyping in (b); however, the process model that results from the retyping in (c) is not the one in (b), but instead, its objects receive multiple a-posterior types (Coding and Testing). This can be interpreted as the overlapping of all possible valid retypings for the model (4 in total, as each object can be either Coding or Testing). Further analysis of bidirectional properties and issues of type-level specifications, in the style of [Stevens 2010; Fos-

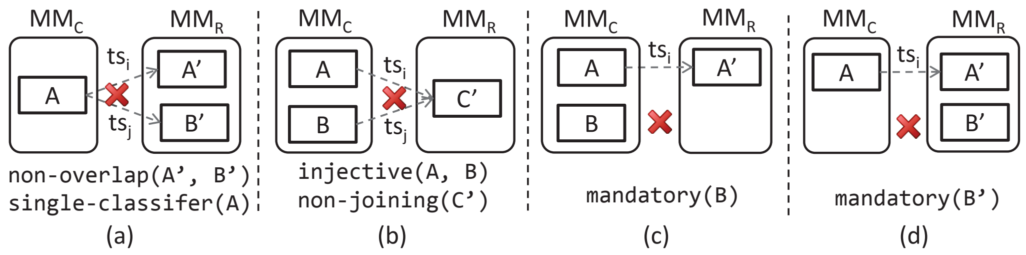


Fig. 25: Structural restrictions and induced forbidden mappings.

ter et al. 2007; Fischer et al. 2015], are left for future work. More discussions on the interpretation of multiple typings are given in Section 8.

7. RESTRICTING A-POSTERIORI TYPINGS

In this section, we analyse some useful restrictions for retyping specifications. As our formalization was purposely made as general as possible, these restrictions help the designer to express the expectations on possible typings, and to preserve the intent of the reusable operations defined over the role meta-model. We distinguish between *structural restrictions* which constrain the allowed mappings between the creation and role meta-models, and *behavioural restrictions* derived from the expected usage of the retyped models. While the well-formedness rules in Sections 4.1 and 5.1 assumed read-only operations for the retyped models, we will see that update, creation and deletion operations require specifications to fulfil further constraints.

7.1. Structural restrictions

Structural restrictions impose certain constraints on type/instance-level specifications to ensure retypings with certain properties, like non-overlapping instances or type coverage. We have kept our retyping specifications as general as possible, however, in practice, it becomes useful to have some more control of the allowed mappings. For example, we can use these restrictions to mark combinations of classes that are allowed to share instances (e.g., Author and Reviewer in Figure 4), or to mark mandatory/optional classes in a role meta-model. We use predicates to express restrictions either over classes of MM_C or MM_R (e.g., non-overlap(A, B), mandatory(C)) or over whole meta-models (e.g., total(MM$_C$)). Structural restrictions are also useful to model other more restrictive meta-model mapping approaches, like the I_Graph approach in [Hermann et al. 2009]. In that approach, restricted kinds of mappings between meta-models are defined (injective, subtype-preserving), in order to ensure correct graph-based transformation of meta-models.

Figure 25 illustrates the different structural restrictions and the forbidden specification mappings they induce. Some restrictions for the role meta-model have a dual restriction for the creation meta-model. For example, restriction non-overlap(A', B') forbids overlapping instances of two classes $A', B' \in MM_R$, while conversely, restriction single-classifier(A) constrains class $A \in MM_C$ to have at most one a-posteriori type. Hence, both restrictions forbid multiple mappings from classes in MM_C to classes in MM_R, as depicted in Figure 25(a).

In the following subsections, we analyse the different structural restrictions.

7.1.1. Non-overlapping instances. This restriction is used to control which classes in MM_R are allowed to share instances (see Figure 25(a)). This is useful as some model management operations defined over MM_R might not expect two classes, not related by inheritance, to share instances. Additionally, we might want to mark classes in

MM_R whose combination makes sense (e.g., a Person can be retyped both as Author and Reviewer in Figure 4), but discard other combinations.

Two classes A' and B' of MM_R not related by inheritance overlap if they share instances: $\{o \mid A' \in type(o)\} \cap \{o \mid B' \in type(o)\} \neq \emptyset$. We write non-overlap(A', B') to denote that classes A' and B' (one not a subclass of the other) cannot overlap. This restriction can be used to ensure that two unrelated classes in MM_R will not overlap under a retyping specification, or to constrain objects to receive at most one a-posteriori type from MM_R.

Equation (34) formulates this non-overlapping condition for type-level specifications. It requires that every pair of classes in MM_R not related through inheritance do not share instances.

$$\begin{aligned}\forall A', B' \in MM_R \bullet\ & A' \notin sub^*(B') \wedge non-overlap(A', B') \implies \\ & ts_i(A) = A' \wedge ts_j(B) = B' \implies A \notin sub^*(B)\end{aligned} \tag{34}$$

The condition for instance-level specifications is similar, using the typing information associated to the mapping expressions. In this case, the condition is sufficient but not necessary. This is so as one may permit $A \in sub^*(B)$ if exp_A and exp_B select disjoint sets of objects. While this analysis could be done via constraint solving, we skip the details for brevity.

Implementation-wise, one could use the converse predicate overlapping(A', B', ...) (present, e.g., in the UML) to indicate the allowed overlappings by setting the default to non-overlapping and allowing any number of parameters.

7.1.2. Injectivity. This restriction ensures that two objects that do not share a classifier in MM_C, do not share a classifier under a retyping. As we will see later, this is useful if the retyped model view is modified by creating objects of some class C' in MM_R. For this to be possible, such class cannot receive mappings from two classes in MM_C. Injectivity can be stated as a predicate over individual classes of MM_C (injective(A, B), with $A, B \in MM_C$) or over the meta-model (injective(MM$_C$)). The condition for type-level specifications is as follows (see also Figure 25(b)):

$$\begin{aligned}\forall A, B \in MM_C \bullet\ & injective(A, B)\ \wedge\ A \notin sub^*(B) \implies \\ & ts_i(A) \notin sub^*(ts_j(B))\end{aligned} \tag{35}$$

The condition for instance-level specifications is similar, using the typing information associated to the mapping expressions. Predicate injective(MM$_C$) is a shortcut for injective(A, B) $\forall A, B \in MM_C$. Moreover, injectivity and non-overlapping are dual concepts for retyping specifications and their inverses.

7.1.3. Subtype preservation and reflection. Subtype preservation ensures that if two classes are related through inheritance in MM_C, so are their mapped classes in MM_R. This condition is called I-compatibility in [Hermann et al. 2009]. For type-level specifications, this condition is formulated as follows:

$$\begin{aligned}subt-preserve(MM_C) \implies \forall A, B \in MM_C \bullet\ & A \in sub^*(B) \implies \\ & ts_i(A) \in sub^*(ts_j(B))\end{aligned} \tag{36}$$

The condition for instance-level specifications is similar, using the typing information associated to the mapping expressions. *Subtype reflection* is the dual of subtype preservation. Hence, it applies to MM_R and ensures that if two classes of MM_R are related through inheritance, so are the classes in MM_C from which they are mapped.

7.1.4. Type totality. Sometimes, one is interested in building total typings where every object typed by MM_C receives an a-posteriori type from MM_R. For example, in case we retype to a meta-model to obtain model metrics, we might want to ensure that every object of MM_C is mapped. For this purpose, we first define the predicate mandatory(A), which requires mandatorily a typing for class $A \in MM_C$ (see Figure 25(c)). Then, totality for MM_C can be expressed as total(MM_C) $\equiv$ mandatory(A) $\forall A \in MM_C$. We formulate this condition for type-level specifications as:

$$\forall A \in MM_C \bullet mandatory(A) \implies \exists i \bullet ts_i(B) = A' \wedge A \in sub^*(B) \tag{37}$$

While the totality analysis presented in Section 6.1 analysed whether some instance of MM_C could yield an ill-retyped model, this restriction ensures that all objects within an instance of MM_C will have an a-posteriori type.

For instance-level specifications, the condition is not sufficient. Analysing predicate mandatory(A) (for $A \in MM_C$) requires using constraint solving to check if there can be objects of type A that are not selected by any mapping whose type is compatible with A. Hence, from this predicate, we generate an invariant using the template in Listing 10, where $is_i(exp_{A_i} : A_i) \mapsto A'_i$, and each $A_i \in anc^*(A)$. If the invariant is satisfiable, then the induced retyping is not total.

```
not A.allInstances()→includesAll(exp_A1→union(exp_A2)→...→union(exp_An))
```

Listing 10: OCL invariant template for checking totality of instance-level specifications

7.1.5. Type coverage. This restriction is the dual of totality. In this case, we are interested in ensuring that certain classes in MM_R are not "optional" for a given retyping specification, but they should mandatorily receive a mapping. For example, the conference meta-model in Figure 4 might consider Topic as optional, while Article, Reviewer and Author could be mandatory. This might be needed for an operation that assigns articles to reviewers, for which it is acceptable if no article has topics, but would not make sense to have empty sets of articles, reviewers or authors.

Using the mandatory predicate for classes of MM_R, we use predicate coverage(MM_R) to denote mandatory(A') $\forall A' \in MM_R$ (see Figure 25(d)). We formulate this condition for type-level specifications as follows:

$$\forall A' \in MM_R \bullet mandatory(A') \implies \exists i \bullet ts_i(A) = A' \tag{38}$$

For instance-level specifications, we use the types of the mapping expressions.

7.2. Behavioural restrictions

Behavioural restrictions are derived from the expected use of the retyped models. While the correctness rules studied so far assume read-only retyped models, we might be interested in creating, deleting or modifying objects from the models. Other useful restrictions include forbidding type dynamicity upon model changes.

7.2.1. Non-dynamicity. Type-level specifications are static, and instance-level specifications can be dynamic. Forbidding dynamicity is needed in case model management operations defined over MM_R make assumptions on object type staticness. For example, an operation may keep a list of objects of a given type, and may not expect the type of such objects to change.

In order to restrict instance-level specifications to be static, we use the predicate static(MM_R). Staticness can be statically enforced by demanding expressions with compatible type to be mapped to the same class in MM_R. This ensures non-dynamicity because, even if objects with the same type can be selected by different expressions, they all get mapped to the same role class.

$$static(MM_R) \implies \big(\forall i \in I, \forall A \in MM_C \bullet is_i(exp_A : A) = A' \\ \implies \forall B \in anc^*(A), \forall j \in I \bullet is_j(exp_B : B) = A'\big) \tag{39}$$

Another possibility to enforce staticness is to permit compatible types in MM_C to be mapped to different types in the role meta-model as long as the features used in the mapping expressions are not mapped to the role meta-model, and model updates are only performed through the role meta-model.

7.2.2. Write feature. Our notions of retyping correctness assume read-compatibility. Thus, when applying a retyping, all retyped elements in an instance model M of MM_C can be "read" as instances of MM_R. However, after the retyping, we might want to modify $TS(M)$ using operations defined over MM_R. This writing-compatibility requires further constraints, which we detail next. We assume the following predicates: add(r) for references whose instances are being modified by adding objects, delete(r) for references whose instances are being removed objects, and write(r) for references whose instances are going to be added and removed objects (creation and deletion of objects are handled in the next subsections). The predicates can be added manually to MM_R, or be automatically derived from the analysis of some model management operation.

If a reference r' in MM_R is to be added objects (i.e., we have add(r')), then the upper cardinality of r' needs to be the same as the upper cardinality of the reference r in MM_C from which r' is mapped. In case of removal of objects (delete(r')), the lower cardinalities of r' and r need to be the same. If we need to both add and remove objects from r' (write(r')), then the cardinality intervals of r' and r must be equal.

Moreover, if r' is being written either for add or delete, then it needs to be mapped uniquely. A reference is uniquely mapped if: (a) it is mapped from exactly one reference from MM_C; or (b) it is mapped from several references with different owner classes that are not in the same inheritance hierarchy. The reason for this is to be able to distinguish in which reference to write to. Figure 26 shows a forbidden case where both r$_1$ and r$_2$ are mapped to r', and their owner classes inherit one from the other. Thus, if we try to modify r' over an object with creation type B, there would be an indetermination as we would not know whether to write on r$_1$ or r$_2$.

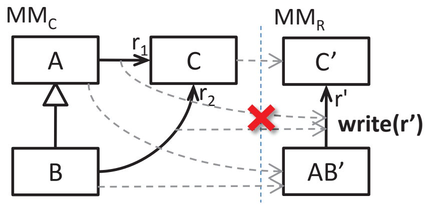


Fig. 26: Forbidden type-level mapping.

Equation (40) states formally this restriction for type-level specifications.

$$\forall r' \in MM_R \bullet \; add(r') \vee delete(r') \vee write(r') \implies \\ \exists i \in I, \exists r \in MM_C \bullet ts_i(r) = r' \wedge \\ \forall i, j \in I, \exists r_1, r_2 \in MM_C \bullet ts_i(r_1) = r' = ts_j(r_2) \implies \\ r_1 = r_2 \vee owner(r_2) \notin owner^*(r_1) \tag{40}$$

Compositions cannot be mapped to writable non-compositions, as writing a non-composition has fewer constraints than writing a composition. Moreover, for simplicity, we do not allow writing derived features. While this is technically possible, it would require inverting the computations defined for derived features.

Instance-level specifications require satisfying all abovementioned restrictions when a feature needs to be modified. The only difference with respect to type-level specifications is how to check that a reference is uniquely mapped, as in this case, we need to use constraint solving to check whether an object can be selected by two expressions exp_{A_1}, exp_{A_2} in the specification, and map different references to r'. Hence, given an instance-level specification with $is_i(exp_{A_1} : A_1) = A'$, $is_i(A_1.r_1) = A'.r'$, $is_j(exp_{A_2} : A_2) = A'$, $is_j(A_2.r_2) = A'.r'$... and $A_1 \in sub^*(A_2)$, we build the invariant of Listing 11. If the invariant is not satisfiable, then $A'.r'$ is uniquely mapped.

```
exp_{A_1}→exists(a | exp_{A_2}→union(...)→includes(a)) or
exp_{A_2}→exists(a | exp_{A_1}→union(...)→includes(a)) or ...
```

Listing 11: OCL invariant template checking disjointness

7.2.3. Create object. If an operation over MM_R creates objects of type A' (denoted by create(A')), we need retyping specifications into MM_R to fulfil additional restrictions.

For type-level specifications, exactly one class A in MM_C can be mapped to a class A' in MM_R with create restrictions. Moreover, A cannot be abstract. Having several classes mapped to A' does not allow distinguishing which class to create. Having no class mapped to A' does not allow executing the creation operation. Having an abstract class mapped to A' would yield an error in the creation process. Hence:

$$\begin{aligned} &\forall A' \in MM_R \bullet create(A') \implies \\ &\qquad \big(\exists_1 A \in MM_C,\ \exists_1\, i \in I \bullet ts_i(A) = A'\big) \wedge \\ &\qquad \forall A \in MM_C \bullet ts_i(A) = A' \implies \neg abs(A) \end{aligned} \tag{41}$$

In general, it is not mandatory to bind the features of a class A in MM_C, even if A is bound to some A' in MM_R. However, if we have create(A'), we demand mapping every mandatory feature of A without a default value. Hence, for type-level specifications:

$$\begin{aligned} &\forall A' \in MM_R, \forall A \in MM_C \bullet create(A') \wedge ts_i(A) = A' \implies \\ &\qquad \forall a \in feats(A) \bullet mand(a) \implies \\ &\qquad \exists i \in I \bullet ts_i(A.a) = A'.a' \vee default(A.a) \end{aligned} \tag{42}$$

Finally, if a class $A' \in MM_R$ is created, then all mandatory references owned by A' should be marked as add.

Regarding instance-level specifications, object creation is more complex because, in general, the type to create in MM_C is not known until the object has values for its features, as the a-posteriori type of objects depends on their feature values. To avoid this indetermination, if we have a predicate create(A'), we require exactly one expression $is_i(A.allInstances() : A) \mapsto A'$. While this is not a necessary condition, it implies that objects classified as A' cannot change their type dynamically, and newly created objects with type A' get classified back as A. The previous restriction concerning the obligatory mapping of mandatory features is also required for instance-level specifications.

7.2.4. Delete object. If an operation over MM_R deletes objects of type A', then all mandatory references of the class A from which A' is mapped, as well as all mandatory references reaching A or a superclass, should be mapped to references marked with delete. This avoids potential inconsistent reference cardinalities when A' objects are deleted. Equation (43) shows this condition for type-level specifications.

$$\begin{aligned} &\forall A' \in MM_R, \forall A \in MM_C, \forall i \in I \bullet \\ &\qquad delete(A') \wedge ts_i(A) = A' \implies \\ &\qquad \big(\forall a \in refs(A) \bullet mand(a) \implies \\ &\qquad\quad \exists j \in I, \exists a' \in refs(A') \bullet ts_j(A.a) = A'.a' \wedge delete(a')\big) \wedge \\ &\qquad \big(\forall B \in MM_C, \forall r \in refs(B) \bullet mand(r) \wedge A \in sub^*(tar(B.r)) \implies \\ &\qquad\quad \exists j \in I, \exists B' \in MM_R \bullet ts_j(B.r) = B'.r' \wedge delete(r')\big) \end{aligned} \tag{43}$$

Moreover, if a class $A' \in MM_R$ is deleted, then all mandatory references reaching class A' should be marked as delete.

Instance-level specifications require the same restrictions as type-level ones.

7.2.5. Model modification vs bidirectional retyping. The specification restrictions when a retyped model is being modified are similar to the conditions for a retyping specification to be invertible. In particular, the cardinality and composition requirements to write a reference and those required by bidirectionality are the same. However, writing a reference requires uniqueness of mappings (see Figure 26), while for bidirectionality, a reference in MM_R can be mapped from several references of MM_C, in which case, the inverse retyping would add the objects to all of them. Object creation requires that all mapped classes are concrete and map their mandatory features, just like bidirectionality. Hence, marking all classes in MM_R as create, and all features as write, makes invertible every retyping to MM_R satisfying Equations (27) and (30). This is expected, since the changes in M_R should be compatible (i.e., "translatable back" in a unique way) with M_C.

8. TOOL SUPPORT

This section describes an implementation of the presented concepts in METADEPTH [de Lara and Guerra 2010]. This is a textual multi-level modelling tool that integrates the Epsilon languages [Paige et al. 2009] 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 some of the described analysis.

8.1. Models and meta-models in METADEPTH

Models and meta-models are specified textually in METADEPTH. As the tool supports multi-level modelling [de Lara et al. 2014], elements may be decorated with a potency (written after the '@' symbol) stating at how many consecutive meta-levels the element can be instantiated. In two-level modelling, meta-models have potency 1 and models have potency 0.

Listing 12 shows the Planning meta-model, where Task is extended by a subclass DocTask in line 9. Listing 13 shows a model with one instance of Task (t0) and another instance of DocTask (t1). The name of the type is used to instantiate the model or class (e.g., Planning someTasks in line 1, Task t0 in line 2).

```
 1 Model Planning {
 2   Node Task {
 3     start : Date;
 4     duration : int [0..1];
 5     name : String [0..1];
 6     res : Resource [∗];
 7     assigned : Person [∗];
 8   }
 9   Node DocTask : Task {}
10   Node Resource {
11     owner : Person [1..∗];
12   }
13   Node Person {}
14 }
```

Listing 12: Meta-model in METADEPTH

```
 1 Planning someTasks {
 2   Task t0 {
 3     start = "30/04/2015";
 4     duration = 30;
 5     name = "coding";
 6   }
 7   DocTask t1 {
 8     start = "30/05/2015";
 9     duration = 90;
10     name = "write manual";
11   }
12 }
```

Listing 13: Model in METADEPTH

8.2. Specification of a-posteriori typings

METADEPTH permits specifying both type-level and instance-level retypings. Instance-level specifications are given by mapping queries written in the Epsilon Object Language (EOL, a variant of OCL) [Kolovos et al. 2006] into types. As an example, Listing 14 shows the specification in Figure 13. Line 2 maps the instances of Task with duration less than 80, to type Schedulable. 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 4) are also expressed in EOL. A derived attribute starts by "/", followed by its name, a colon, its type, and the computation expression enclosed between '$'.

```
1 type Planning Scheduling inst {
2     $Task.allInstances()→select(x | x.duration < 80)$ > Schedulable with {
3         start > date,
4         /months : double = $self.duration/30$ > span
5     }
6 }
```

Listing 14: Instance-level a-posteriori typing in METADEPTH

A-posteriori typings induced by instance-level specifications are dynamic. Once the specification is applied to model someTasks in Listing 13, evaluating the query Schedulable.allInstances() over the model 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 have implemented a cache mechanism for the type of the model objects. This cache is invalidated when the model changes. Currently, we invalidate all typings, but we could optimize this by invalidating only the typing of objects whose mapping expressions access the changed model elements. This could be done with incremental OCL evaluation techniques [Cabot and Teniente 2009], but it is left for future work.

If type dynamicity is not needed, then using type-level specifications is more efficient as there is no need to evaluate queries, and specifications are easily defined by simple mappings. However, our current implementation is more restricted than the one presented in Section 4, as a class in MM_C cannot be mapped to two classes in MM_R. Our implementation of instance-level specifications allows this feature, though.

Listing 15 shows the type-level specification in Figure 5. It only needs to define the mapping of features, as the mapping of classes is automatically induced.

```
type Planning Scheduling {
  Task::start > Schedulable::date,
  Task::/months : double = $self.duration/30$ > Schedulable::span
}
```

Listing 15: Type-level specification in METADEPTH

Once an a-posteriori specification is applied, the retyping of models is automatic. The tool permits displaying a model using the a-posteriori types using the command dump ⟨model⟩ as ⟨role-meta-model⟩. After applying the retyping of Listing 15, if we type dump someTasks as Scheduling, the tool displays the model in Listing 16. Internally, this model is a view of someTasks using the a-posteriori typing, and so, no objects are created.

```
Scheduling someTasks {
  Schedulable t0 {
    date = "30/04/2015";
    span = 1.0;
  }
  Schedulable t1 {
    date = "30/05/2015";
    span = 3.0;
  }
}
```

Listing 16: Retyping in METADEPTH

The tool also permits using type-level specifications backwards, as explained in Section 6.2. In some cases, backward retypings may produce multiple typings. For example, Listing 17 shows a Scheduling model with one Schedulable object s0. If we use the retyping specification in Listing 15 backwards, s0 can be reclassified as both Task and DocTask. This can be interpreted as the retyping producing two "overlapped" models with the same structure but different typing. Hence, METADEPTH provides two different a-posteriori typings for the model with respect to the Planning meta-model. The first one, shown in lines 1–3 of Listing 18, retypes s0 as Task. The second one, shown in lines 4–6 of Listing 18, retypes s0 as DocTask. Retyping back the someTasks model of Listing 16 would yield four different a-posteriori typings, mapping t0 and t1 to all possible valid combinations of Task and DocTask. Interpreting objects with multiple a-posteriori types as overlapping models is especially useful when retypings are used as bidirectional transformations (see Section 9.1).

```
Scheduling anExample {
  Schedulable s0 {
    date = "30/04/2015";
    span = 1.0;
  }
}
```

Listing 17: Scheduling model

```
1 Planning anExample { // First typing
2   Task s0 { start = "30/04/2015"; }
3 }
4 Planning anExample { // Second typing
5   DocTask s0 { start = "30/04/2015"; }
6 }
```

Listing 18: Backward retypings

There is another possible interpretation for objects with multiple a-posteriori types, which is useful when an instance-level specification assigns several types to the same object, meaning that the object may simultaneously play several roles in the role meta-model. For example, in Figure 4, a retyping specification would assign the a-posteriori types Author and Reviewer to any Person having authored an article and being assigned a review task. Listings 12 and 19 show the creation and role meta-models for this example, respectively. In this case, an object with several a-posteriori types could be visualized either as a single object, or as a different object for each a-posteriori type (though all would have the same identifier, and internally, there would be a single object). As an example of the second option, Listing 20 shows a Planning model, and Listing 21 shows its retyping, where person p gets retyped as both Author (line 2) and Reviewer (lines 4–6). The visualization shows the appropriate features in each case (reviews when p plays the reviewer role, and none when p is shown as an author).

8.3. Analysis of typing specifications

METADEPTH can check executability, non-totality and non-surjectivity of type-level retypings, as explained in Section 6.1. In particular, the command refinement Planning Scheduling tries to find a counterexample witness model that satisfies the constraints of Planning and violates some constraint of Scheduling. Similarly, command refinement Planning Scheduling strict checks if there is a model fulfilling all constraints in Scheduling and violating some of Planning. For the specification in Listing 15, no witness model is found because the retyping is total and surjective.

```
Model Conference {
  Node Author {}
  Node Reviewer {
    reviews : Article [0..3];
  }
  Node Article {
    authors : Author [1..*];
  }
}
```

Listing 19: Role meta-model

```
Planning mc {
  Person p {}
  Person p2 {}
  Task t {
    start = "30/04/2015";
    assigned = [p];
    res = [r2];
  }
  Resource r {
    owner= [p];
  }
  Resource r2{
    owner= [p2];
  }
}
```

Listing 20: Planning model

```
Conference mr {
  Author p {}
  Author p2 {}
  Reviewer p {
    reviews = [r2];
  }
  Article r {
    authors= [p];
  }
  Article r2 {
    authors= [p2];
  }
}
```

Listing 21: Retyping

METADEPTH relies on the USE Validator [Kuhlmann and Gogolla 2012] 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. METADEPTH parses back the found model, 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.

8.4. Retyping restrictions as annotations

In METADEPTH, all elements (models, objects at any meta-level and features) can be decorated with annotations. Annotations can have parameters, which can be of primitive type or references to other objects. This way, we have implemented the restriction predicates of Section 7 as annotations. Internally, the annotations are reified as objects that refer to the annotated elements. This permits the retyping algorithm to access the used annotations and check if the specification fulfils the corresponding restrictions. As an example, Listing 22 shows the Scheduling meta-model with class Schedulable tagged as mandatory, hence making any specification targeting Scheduling to obey the restriction in Equation (38) (see Section 7.1.5).

```
Model Scheduling {
  @mandatory
  Node Schedulable {
    date : Date;
  }
}
```

Listing 22: Restriction predicates as annotations

9. APPLICATIONS OF A-POSTERIORI TYPING

A-posteriori typing has multiple applications in MDE. First, it is a mechanism to obtain (updateable) views of models with respect to other meta-models, which enables its use to specify simple model transformations (see Section 9.1). Second, it allows the reuse of model management operations defined over a role meta-model for other unrelated meta-models via a type-level or instance-level specification (see Section 9.2). Dynamic typing can be valuable in Models@run.time applications, where models evolve and objects can be dynamically retyped (see Section 9.3). As a summary, we end this section with a discussion of the benefits, implications and limitations of this new retyping approach (see Section 9.4).

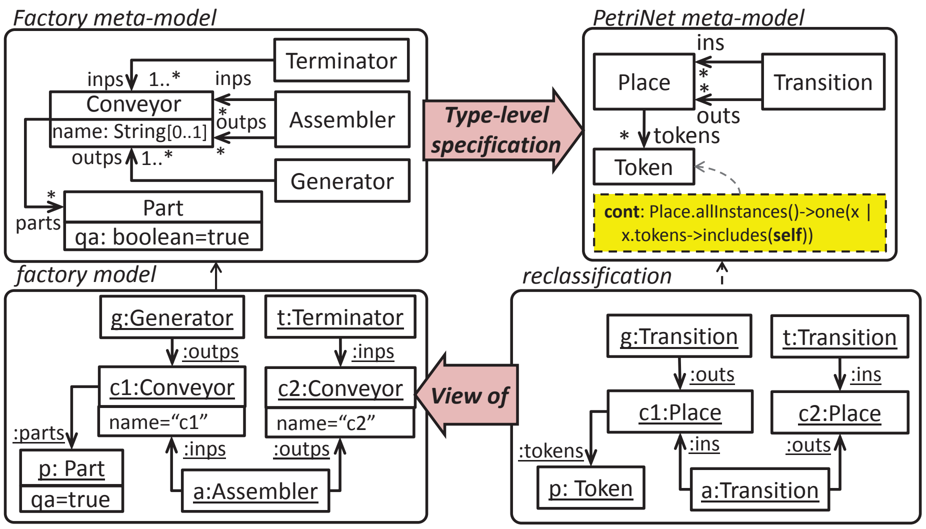


Fig. 27: Reclassifying Factory models into Petri-nets

9.1. Bidirectional model transformations by reclassification

Consider the meta-model to describe factories in Figure 27. It declares three kinds of machines: generators introduce parts in the factory, terminators remove parts from it, and assemblers transform parts. Machines of any kind can be connected by conveyors, which may transport any number of parts. Parts have a boolean flag indicating whether they passed a quality test. We will make use of this flag in Section 9.3.

Assume we want to transform Factory models into Petri nets for their analysis or simulation [Murata 1989]. Petri nets are bipartite automata made of places and transitions. Places may hold zero or more tokens, and these must belong to exactly one place, as required by constraint cont. Transitions may have input and output places (relations ins and outs). A transition can *fire* if all its input places hold some token. If a transition fires, a token is subtracted from every input place and added to all its output places. 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, 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 Figure 27).

Listing 23 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 PetriNet {
    Conveyor::parts > Place::tokens,
    Generator::outps > Transition::outs, Generator::/inps : Conveyor[∗] = $Set{}$ > Transition::ins,
    Terminator::inps > Transition::ins, Terminator::/outps : Conveyor[∗] = $Set{}$ > Transition::outs,
    Assembler::inps > Transition::ins, Assembler::outps > Transition::outs,
}
```

Listing 23: Mapping factories to Petri nets via a type-level specification

This specification allows retyping Factory models as PetriNet models, and vice versa. This is so as it fulfils the bidirectionality requirements stated in Section 6.2. As an example, Figure 28 shows a PetriNet model and its two possible a-posteriori typings: the first one types the transition as Terminator, and the second as Assembler. The figure shows the model both in METADEPTH syntax (left) and abstract syntax (right).

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

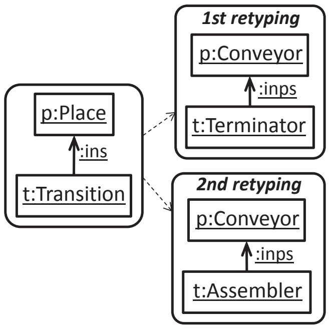


Fig. 28: Retyping a Petri net into a Factory. Left: METADEPTH syntax. Right: Abstract syntax.

The specification can be analysed to detect whether it is executable, total or surjective. For totality, the tool finds the witness model in Figure 29, which proves the transformation is not total because this factory model cannot be retyped as a Petri net (i.e., the 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 proves that the backward transformation is not surjective, as this factory model cannot be produced from any valid Petri net.

```
// Model witness with no Petri net equivalent
Factory noRefinementWitness {
  Assembler assembler2 {
     outps = [conveyor2, conveyor1];
  }
  Conveyor conveyor1 {
     name = "string1";
  }
  Conveyor conveyor2 {
     name = "string1";
  }
  Generator generator2 {
     outps = [conveyor1];
  }
  Part part2 {
     qa = true;
  }
}
```

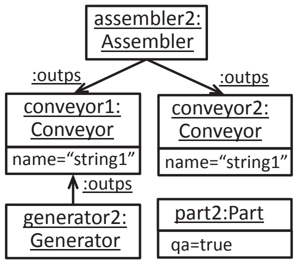


Fig. 29: Witness model showing no totality. Left: METADEPTH syntax. Right: Abstract syntax.

While retyping specifications are less expressive than full-fledged transformation languages, they have some advantages. First, as we have seen, they allow analysing

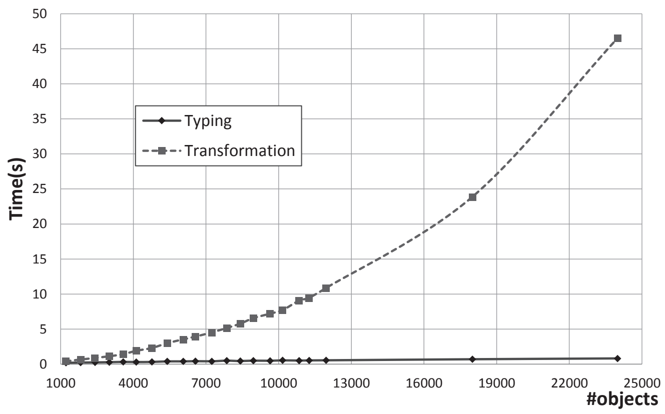


Fig. 30: Comparing the performance of a-posteriori typing and traditional transformation

properties like executability, totality or surjectivity. Second, they have the potential to be more efficient, as there is no need to create target objects.

To assess the potential performance gain, we compared the time to perform a retyping and the time to execute an equivalent transformation. More in detail, we encoded the retyping in Listing 23 as a transformation using the Epsilon Transformation Language (ETL) [Kolovos et al. 2008], which is integrated within METADEPTH. Then, we generated models ranging from 1.200 to 24.000 objects. Models were generated randomly but trying to fulfil reasonable expectations of the domain, like having the double of conveyors and assemblers than terminators and generators, conveyors with one to five parts, and machines with one to three inputs and outputs. Finally, we executed the retyping and the transformation 5 times for each model, taking the average time. More precisely, we measured the time to retype and print the model to the console, compared to the time to transform it and print the output model. We included the printing time because the cost of retyping a model is mostly the time employed to calculate the new types, which is performed lazily when the model is accessed, or printed.

Figure 30 shows the results of this experiment, where retyping clearly outperforms traditional transformation, reaching a speed up of 57$\times$ for models of size 24.000. While retyping time was dominated by printing the model (as it involves calculating the new types for the objects), transformation time was dominated by computation (creation of objects). Overall, retyping and printing a model with 24.000 objects took around 800 milliseconds, and only 1 millisecond was spent in processing the retyping specification itself. Please note that, once retyped, the model can be saved in a file using the a-posteriori typing, so that the next time it is loaded, there is no time penalty in calculating the a-posteriori types. More details of the experiment, including the models and transformation used, are available at http://miso.es/aposteriori/comparison.html. While these results are very promising, further experimental work on comparing traditional transformation and retyping is left for future work.

Finally, another benefit is that the transformations defined through retyping are incremental, as changes in the source model are reflected in the target. Actually, no propagation is needed because the source and target objects are the same. For type-level specifications, the synchronization between the source and target models is obtained for free. If the retyped (i.e., target) model changes through operations over MM_R, the changes are reflected immediately on the source model because both models are the same (though with different types). Finally, as this example showed, restricted kinds of type-level specifications yield (simple) bidirectional transformations (see Section 6.2).

To have an intuition of the applicability of retypings as model transformations, we have analysed the zoo of ATL transformations[6] to see how many of them can be specified as retypings (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. In http://miso.es/aposteriori/ we have created a repository of transformations expressed as a-posteriori retyping specifications.

9.2. Reuse of model management operations

Listing 24 shows an excerpt of a Petri net simulator, written in EOL. The simulator uses the types of the PetriNet meta-model, defining operations (like enabled and fire) on its types. Operation step (lines 14-22) 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 Transition fire() {
 5   for (p in self.outs) // add a token to every output place
 6     p.tokens.add(new Token);
 7   }
 8   for (p in self.ins) { // remove a (random) token from every input place
 9     var t := p.tokens.random();
10     p.tokens.remove(t);
11     delete t; // remove the token object from the model
12   }
13 }
14 operation step() : Boolean {
15   var enabled : Set(Transition) := Transition.allInstances().select(t | t.enabled());
16   if (enabled.size()>0) { // fire one random Transition from enabled
17     var t := enabled.random();
18     t.fire();
19     return true;
20   }
21   return false;
22 }
```

Listing 24: 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 METADEPTH 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.allInstances() returns all instances of all classes mapped to Transition.

The simulator performs write operations on the retyped model: it creates and deletes Token objects, and adds and deletes Token objects from reference Place.tokens. Hence, the PetriNet meta-model in Listing 25 is annotated to reflect these restrictions. While these annotations are currently performed manually, the process could be automated by a static analysis of the operation to be reused (e.g., in line with [Zschaler 2014]). We leave this for future work. Also, note that we need the annotations because the operation modifies the model in-place. Should we reuse a model-to-model transformation, we would not need annotations, as the source model would be read-only.

```
Model PetriNet {
  Node Place{
     @write
     tokens : Token[*];
```

[6]http://www.eclipse.org/atl/atlTransformations/

```
  }
  Node Transition {
    ins : Place[∗];
    outs : Place[∗];
  }
  @delete
  @create
  Node Token {
    cont : $Place.allInstances()→one (x | x.tokens→includes(self))$
  }
}
```

Listing 25: Petri-net meta-model with reclassification restrictions

As discussed in Section 7.2.3, 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, and so this kind of mapping is disallowed. In this example, creating Tokens does not produce any problem, as 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.

9.3. Dynamic typing

Now, we consider factories in which parts that do not pass a quality check are not processed. Interestingly, this can still be done using the *same* simulator as in Section 9.2, if we include this condition in the a-posteriori typing of Parts. This way, Parts whose qa slot is false are not mapped to Tokens and will not be considered by the simulator.

As in this case the typing becomes dynamic, we need an instance-level a-posteriori typing specification. This is shown in Listing 26. Line 3 selects in collection tokens only those parts whose attribute qa is true.

```
1  type Factory PetriNet inst {
2    $Conveyor.allInstances()$ > Place with {
3      /sp : Token[∗] = $self.parts→select(p | p.qa=true)$ > tokens
4    }
5    $Part.allInstances()→select(p | p.qa = true)$ > Token
6    $Terminator.allInstances()$ > Transition with { inps > ins }
7    $Generator.allInstances()$ > Transition with { outps > outs }
8    $Assembler.allInstances()$ > Transition with {
9      inps > ins,
10     outps > outs
11   }
12 }
```

Listing 26: Instance-level typing specification from Factory to PetriNet

Figure 31 shows two simulation steps. Method qcheck is an operation over the Factory meta-model 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.

An 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.

9.4. Discussion: strengths and limitations

9.4.1. Strengths. As Section 9.1 illustrated, a-posteriori typings can be used to define simple model-to-model transformations, but have the potential to be more efficient

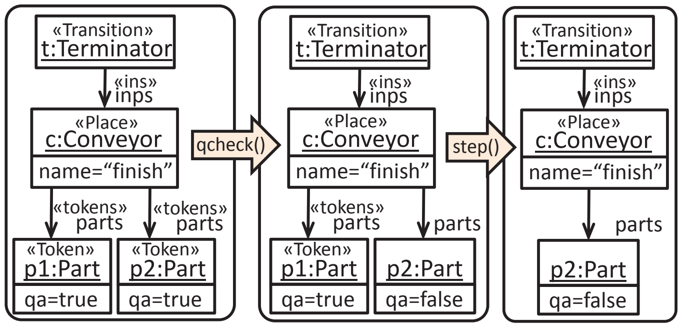


Fig. 31: Dynamic typing for the simulation of a factory.

than transformations because the overhead of creating objects is eliminated (see Figure 30). Incrementality and synchronization of source and target models come for free, while in contrast, traditional transformation approaches require implementing dedicated mechanisms for that. Finally, due to its limited expressivity, type-level a-posteriori specifications can be analysed for executability, surjectivity and bidirectionality.

As Section 9.2 showed, a-posteriori typing also enables flexible reuse, allowing model management operations defined over a role meta-model to be reused without change on a creation meta-model. As we will analyse in Section 10.2, this flexibility goes beyond the capabilities of other existing reuse approaches. Finally, as seen in Section 9.3, a-posteriori typings can be dynamic and multiple, which adds further flexibility to reuse.

9.4.2. Limitations. Our retyping approach enables dynamic and multiple typing. However, some model management operations may not be prepared to handle objects that change their type dynamically or with several types. Section 9.3 illustrated how dynamic typing can be used to neglect some objects (e.g., Parts that are no longer typed as Tokens). While the simulator in Listing 24 would work in this case, analysis methods, e.g., for reachability analysis, may not work in this setting. For this reason, Section 7 introduced different retyping restrictions that allow documenting the coarse-grained behaviour of model management operations.

Retyping can also be used to write simple transformations, even bidirectional ones. However, it is not meant to replace full-fledged model transformation languages. In particular, specifications cannot create new objects (just retype them), which limits their expressivity. Moreover, their main construct is the notion of mapping, which maps every object in a given set to a class in MM_R. Instead, transformation rules may have complex patterns as trigger. Finally, retypings lack useful transformation constructs, like helpers or operations, which allow the reuse of recurring expressions.

Regarding the reuse approaches for model management operations shown in Figure 1, our retypings are a flexible mechanism that does not need to declare forward and backward transformations, but a single retyping specification is enough. However, if the operation to be reused modifies the model, then the limitations stated in Section 7 need to be considered. Nonetheless, if a specification qualifies for writing, then properties like synchronization and incrementality come for free.

10. COMPARISON WITH RELATED WORK

Next, we compare our approach with different proposals for model typing in MDE (Section 10.1), existing mechanisms for reutilization of model management operations (Section 10.2), related works on view generation (Section 10.3), bidirectional trans-

Table I: Comparison of model typing approaches.

Approach	Classific. time	Dynamic classifiers	# Classif.	#Model types	Totality
SMOF [OMG 2013a]	a-post.	yes	many	many	total
Stereotypes [Fuentes and Vallecillo 2004]	a-post.	no	many	many	partial
UML [OMG 2013b]	a-post.	yes	many	one	total
MOF [OMG 2014]	creation	no	one	one	total
This paper	a-post.	yes	many	many	partial

formation approaches (Section 10.4), and techniques proposed by the programming languages community to achieve program reuse and dynamic typing (Section 10.5).

10.1. Typing approaches in modelling

Table I summarizes a comparison of the most prominent approaches to model typing in MDE, using some features of the typing space presented in Figure 2.

Regarding standards, SMOF recognises the need for multiple and dynamic classification, and proposes annotating the classifiers that may have instances in common [OMG 2013a]. UML allows multiple classifiers through generalization sets [Olivé 2007]. While UML supports dynamic classification, neither UML nor SMOF provide support for defining a-posteriori typing specifications, and hence, unanticipated reuse of operations is difficult. MOF does not support a-posteriori typing, dynamicity or overlapping classes, and models have exactly one type.

The UML provides an extensibility mechanism to decorate model elements with *stereotypes* [Fuentes and Vallecillo 2004]. These are defined in a so-called *profile* that declares the stereotype features and which UML elements can be tagged by the stereotypes. Stereotypes are assigned after creating the element, so the classification time is a-posteriori. However, stereotypes are rarely dynamic, but an element may have multiple stereotypes. Our a-posteriori typing could be a substitute for this profiling mechanism, as UML model elements could be typed a-posteriori with respect to a role meta-model containing the desired stereotypes.

Exploratory modelling [Atkinson et al. 2011] 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 [Guy et al. 2012; Kühne 2013]. Interestingly, our condition for totality of type-level specifications is called *forward compatibility* in [Kühne 2013], and surjectivity is called *back compatibility*.

Notably, all previous works implement a type-level style for retyping specifications, or target operation adaptation and neglect dynamicity of typing. The work in [Diskin et al. 2012] 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.

The notion of role in modelling languages is also related to our proposal. In [Steimann 2000], the authors review characteristics of roles in modelling languages. For example, roles have their own properties, objects can play several roles simultaneously, objects may acquire and abandon roles dynamically, and objects of unrelated types can play the same role. The author also analyses representation mechanisms for roles: as named places in relationships (like in UML), as separate instances joined to an object, and as generalization or specialization hierarchies which can be mixed with regular classes. The latter representation is deemed problematic, as roles are sometimes seen as generalization of classes, while others are specializations. Our a-posteriori typing approach permits most characteristics of roles listed in [Steimann

Table II: Comparison of transformation reuse approaches.

Approach	Classif. time	Style	Dyn.	# Classif.	#Model types	Total.
Concepts [de Lara and Guerra 2013]	a-post.	T	no	many	many	total
Adapters [Sánchez Cuadrado et al. 2014]	a-post.	T/I	no	many	many	partial
Zschaler [Zschaler 2014]	a-post.	T	no	one	many	partial
Steel et al. [Guy et al. 2012; Steel and Jézéquel 2007]	a-post.	T	no	one	many	partial
This paper	a-post.	T/I	yes	many	many	partial

2000], but while dynamicity of roles is normally achieved explicitly (e.g., by invoking an operation), we obtain dynamicity implicitly upon model changes. Our separation of the creation and role meta-models, and the explicit binding mechanism, avoids the representation problems of mixing hierarchies of classes and roles.

10.2. Reuse approaches in MDE

Table II compares the most prominent approaches to reuse of model management operations. We use the criteria of Table I, as well as the extra criterion *Style*, which refers to the approach to bridge the meta-models involved (Type or Instance level).

Inspired by generic programming, in [de Lara and Guerra 2013], we proposed *concepts* [Gregor et al. 2006] as a mechanism to express requirements for model management operations. Generic model management operations are defined over concepts, which can be bound to concrete meta-models. This way, the operation gets adapted for a concrete meta-model. Originally, the binding of concepts to meta-models were simple mappings similar to type-level specifications. However, the difference is that type-level specifications produce a retyped view of the model, while concept binding produces an adaptation of the particular model management operation.

Adapters [Sánchez Cuadrado et al. 2014] make concept binding more flexible by allowing the use of OCL expressions in mappings. Hence, this approach has a hybrid type-level and instance-level specification style. However again, it adapts the model management operations. In [Sánchez Cuadrado et al. 2014], we targeted ATL model transformations, while in [de Lara and Guerra 2014], a formalization of adapters was used to rewrite graph transformations. Adapters add flexibility to concepts, but still target operation adaptation. Instead, a-posteriori typing enables reuse by producing a view of the model, which permits reusing any kind of model management operation and benefits from dynamicity.

Instead of adapting the operation, other approaches target model adaptation. In [Guy et al. 2012; Steel and Jézéquel 2007], a *matching* relation defined between two meta-models permits instances of the former to be accepted by the latter. Originally, matching classes required same name [Steel and Jézéquel 2007], but this is more flexible in [Guy et al. 2012]. Hence, effects are similar to type-level specifications. To achieve compatibility, derived features are frequently added to the source meta-model. Still, dynamicity and overlapping classes are not considered.

Zschaler [Zschaler 2014] proposes constraint-based model types, a constraint-based specification of meta-model requirements to qualify for operations. These types are automatically extracted from existing operations. Dynamicity, multiple classification or flexible mappings (e.g., via queries as we propose in instance-level specifications) from concrete meta-models to the constraint-based model type are not considered. While that approach could benefit from the flexible mappings of our instance-level specifications, the derivation of a model type from a model management operation proposed in [Zschaler 2014] could be used by our derivation of annotation restrictions.

Some works aim at defining transformation intents, and checking whether an adapted transformation still meets them [Salay et al. 2015; Salay et al. 2016]. Our annotations in Section 7 permit restricting retyping specifications for reusing an operation. However, we would need a more proper way to specify intents, e.g., using OCL and constraint solving to check their satisfaction. We leave this for future work.

In [Wende et al. 2009; Wende 2012], a technique to define reusable language components is proposed. A language component has abstract, concrete syntax and semantics. The abstract syntax is given by a meta-model exposing class roles to which classes in other meta-models can be connected, hence defining the component interface. The approach includes a composition language, which in the simplest case consists of a collection of mappings, similar to our type-level specifications. However, in our case, the goal is retyping and not the composition of the creation and role meta-models. Moreover, our role meta-models can contain features, and mappings can define derived features. In both cases, the semantics of the mappings is expressed through inheritance.

10.2.1. Expressivity. Next, we compare the expressivity of our approach with those in Table II. As we have seen, instance-level specifications are more expressive than type-level ones. Therefore, we only compare a-posteriori typing w.r.t. adapters and the binding language of [Sánchez Cuadrado et al. 2014], as this is the only approach supporting an instance-level specification style. In [Sánchez Cuadrado et al. 2014], some typical heterogeneities that adapters are able to bridge were presented, such as:

— *Class merge.* Two classes in the concept are mapped to the same meta-model class.
— *Class split.* A class in the concept is mapped to two classes in the meta-model.
— *Flatten hierarchy.* A class hierarchy in the concept is flattened in the meta-model.
— *Association to class.* An association in the concept is represented as an intermediate class in the meta-model.
— *Class to association.* A class in the concept is represented as an association in the meta-model.
— *Association to navigation expression*. A reference in the concept is represented by other means in the meta-model.
— *Subclass to enumerate*. A class hierarchy in the concept is represented in the meta-model using a single class with an attribute that has an enumerated type.
— *Attribute conversion*. Conversion between primitive datatypes, and from non-primitive to primitive datatypes.

In our approach, the role meta-model (MM_R) is the equivalent of the concept. While the binding with adapters is performed from the concept to the meta-model, in our case, it is performed from the meta-model to MM_R; however, we swap the creation and role meta-models to facilitate the comparison and keep the nomenclature. Our approach is able to resolve all heterogeneities in [Sánchez Cuadrado et al. 2014]. Figure 32 shows two of the most interesting cases, while the interested reader can find all cases in http://miso.es/aposteriori/. The upper part of the figure shows the heterogeneity *Class split*, which maps one class in MM_C to two classes in MM_R. This can be solved using a type-level specification where Attribute is mapped to both Attr and Port[7].

The figure also shows the instance-level specification that solves the heterogeneity *Association to class*. It maps MClass to Class, and MClass objects with children to Generalization. Interestingly, in [Sánchez Cuadrado et al. 2014], this discrepancy (which occurs in the converse mapping *Class to Association*) is resolved by creating a virtual class in MM_C that is mapped to Generalization. Instead, we can map certain MClass objects to Generalization in addition to Class. This is possible because the cardinality of reference Generalization.class is 1, and hence, we can map the expression self to it. However, the use of virtual classes is more general as it would allow handling the case where the

[7]As our METADEPTH implementation does not support this multiple retyping for type-level specifications, we implemented it using an instance-level specification.

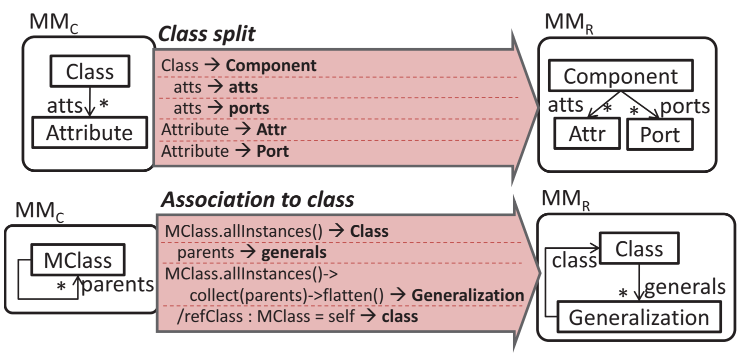


Fig. 32: Some resolvable heterogeneities

cardinality of Generalization.class is *. This is not possible in general with a-posteriori typing, as it would not be possible to find the suitable MClass objects to be typed as Generalization.

Altogether, our approach has an expressive power comparable to adapters, though adapters can define virtual classes which enhance flexibility. However, while adapters are tied to a specific transformation language (ATL), a-posteriori typing is applicable "as is" with any model management language, and we provide analysis mechanisms.

10.3. Views

Views are used in different technological spaces as a mechanism to extract relevant parts of an information source (e.g., a model or a database). They present simpler, aggregated or restructured data from an information source, or hide certain details to users. Hence, our retyping techniques could be employed to define views of models conformant to a creation meta-model MM_C.

Views are a well-known mechanism in database systems [Date 2003], where they are defined by means of queries. Views can be classified as materialized or snapshots if they are physically stored in a new table, or virtual if they are not. The former are more efficient, but they need synchronization with the source database. Our approach is virtual as the retyping does not create new objects. Another issue is related to the update of the source database when the view is modified. This problem has been researched since the 80's, and techniques to characterize updatable views [Dayal and Bernstein 1982] and computation techniques have been developed [Bancilhon and Spyratos 1981]. Similarly, in Section 7.2, we provide restrictions characterizing retypings that permit the modification of a retyped model.

Model views are common in MDE, being heavily used in view-based modelling approaches [Atkinson et al. 2010; Burger et al. 2016; Kilov et al. 2013; Guerra et al. 2009]. Views can be projections of a (possible single) underlying model, or a synthetic means to create a common model. Our approach can combine both, as retyped models are derived (projected) but can be modified under certain conditions. Most approaches use materialized model views, realized through model transformations expressed either using a general-purpose transformation language [Atkinson et al. 2010; Guerra et al. 2009] or a dedicated one [Burger et al. 2016]. More similar to our approach, in [Jakob et al. 2006], virtual model views are defined by triple graph grammars, though dynamic object reclassification is not considered.

Altogether, our approach could be used as a mechanism to define updatable virtual model views, supporting advanced features like dynamic retyping.

10.4. Bidirectional transformation approaches

In Section 6.2, we have seen that some retyping specifications can be used backwards. Moreover, in Section 7.2, we introduced retyping restrictions that permit updating the retyped models using types in MM_R. These modifications get immediately reflected in the source model, as the source and retyped models are indeed the same model.

Bidirectional approaches have been investigated in databases to solve the view-update problem [Bancilhon and Spyratos 1981; Dayal and Bernstein 1982] (described in Section 10.3); in the programming languages community with the lenses framework [Foster et al. 2007]; and in the modelling community with approaches like Triple Graph Grammars [Schürr 1994] and QVT-Relational [OMG 2016; Stevens 2010].

Several authors have proposed classifications for bidirectional synchronization approaches [Diskin et al. 2016b; Hidaka et al. 2016]. For example, in [Hidaka et al. 2016], approaches are classified according to their technical space, the way to define correspondence relations, the change propagation mechanism, and the execution semantics. Our approach lies within the MDE technical space. Regarding correspondences, specifications are defined in a unidirectional way, but they can be applied backwards for some type-level cases. It is not backward functional because, as we have illustrated in Figure 28, a model can admit several retypings when the specification is used backwards. The different solutions can be represented in a unique model, where objects may have multiple types. Some authors have proposed the representation of the possible solutions given by a synchronizer by means of models with uncertainty [Diskin et al. 2016a].

Most approaches in MDE use a tracing mechanism between the source and target models. As in our case, both models are the same, we do not need traces. Actually, our approach can be seen as based on model complements [Bancilhon and Spyratos 1981; Hidaka et al. 2016]. A model complement contains information from the source model, complementary to the target model, in order to avoid information loss. While some approaches store such complement externally [Hidaka et al. 2016], in our case, both source, target and complement are the same model. Finally, our approach propagates changes from target to source, from source to target, and it is incremental.

Compared to the expressivity of other languages, the bidirectional transformations we can express are limited to mappings between source and target types. However, these are not necessarily bijective, as we have seen in the example of Figure 28.

10.5. Reuse, genericity and dynamic typing in programming languages

Dynamic reclassification has been more studied in object-oriented languages. In [Drossopoulou et al. 2002], objects can change their 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 [Gottlob et al. 1996] model the different roles individual objects can acquire or drop. While in these approaches, the change of role or classifier is done via method invocations, our instance-level specifications express these changes declaratively, which facilitates analysis. In [Li 2004], objects can adopt or drop roles dynamically. For this purpose, classes are annotated with possible dynamic parents and children. In [Tamai et al. 2005], the authors present Epsilon, an object model with dynamic roles that can require interfaces from the objects to get bound to. In this way, objects can either implement the interface in the standard way (i.e., belong to a class that implements the interface), or a manual binding can be given. ObjectTeams/Java [Herrmann 2007] is an extension of Java with roles, which can be dynamically adopted and dropped by objects upon entering a context (a container class). In this approach, role instances are explicit, and roles should declare the allowed base class that can take the role (along with some condition) and mappings from methods of the role to methods of the class. This means that role definitions cannot be reused with different sets of base classes. In our approach, decoupling the binding from the role meta-model allows reusability of the latter for multiple creation meta-models.

Also for programming languages, pluggable type systems [Bracha 2004] 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 [Conrad et al. 2008], which proposes 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.

11. 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 a-posteriori typing, together with correctness rules and analysis mechanisms. The feasibility of our approach has been demonstrated by a practical implementation atop METADEPTH. We have also presented several examples and applications which show the flexibility and usefulness 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. Retyping specifications can also be used as a simple form of bidirectional transformation, enabling their analysis, and having the potential to be more efficient than traditional transformation approaches.

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 retyping specifications. We also plan to explore more in detail the use of retyping specifications as bidirectional model transformations. At the tool level, we are increasing the efficiency of instance-level typings by smarter cache policies able to analyse the impact of model changes.

Acknowledgements

This work was supported by the Spanish MINECO (TIN2014-52129-R), and the R&D programme of the Madrid Region (S2013/ICE-3006). We are grateful to the referees for their accurate comments.

REFERENCES

Colin Atkinson, Bastian Kennel, and Björn Goß. 2011. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *SWESE*. 1–15.

Colin Atkinson and Thomas Kühne. 2003. Model-driven development: A metamodeling foundation. *IEEE Software* 20, 5 (2003), 36–41.

Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2010. Orthographic software modeling: A practical approach to view-based development, In ENASE 2008/2009, Revised Selected Papers. *Communications in Computer and Information Science* 69 (2010), 206–219.

François Bancilhon and Nicolas Spyratos. 1981. Update semantics of relational views. *ACM Trans. Database Syst.* 6, 4 (1981), 557–575.

Gilad Bracha. 2004. Pluggable type systems. In *Revival of Dyn. Langs.*

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

Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. 2016. View-based model-driven software development with ModelJoin. *Software and System Modeling* 15, 2 (2016), 473–496.

Jordi Cabot and Ernest Teniente. 2009. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software* 82, 9 (2009), 1459–1478.

P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. 1989. Interfaces for strongly-typed object-oriented programming *(OOPSLA)*. ACM, New York, NY, USA, 457–467.

Robert Clarisó, Jordi Cabot, Esther Guerra, and Juan de Lara. 2016. Backwards reasoning for model transformations: Method and applications. *Journal of Systems and Software* 116 (2016), 113–132.

Robert Clarisó, Carlos A. González, and Jordi Cabot. 2015. Towards domain refinement for UML/OCL bounded verification. In *SEFM (LNCS)*, Vol. 9276. Springer, 108–114.

Marc Conrad, Marianne Huchard, and Thomas Preuss. 2008. Integrating shadows in model driven engineering for agile software development. In *CISIS*. 549–554.

C. J. Date. 2003. *An Introduction to Database Systems, 8th Edition*. Pearson Education.

Umeshwar Dayal and Philip A. Bernstein. 1982. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.* 7, 3 (1982), 381–416.

Juan de Lara and Esther Guerra. 2010. Deep meta-modelling with METADEPTH. In *TOOLS (LNCS)*, Vol. 6141. Springer, 1–20.

Juan de Lara and Esther Guerra. 2013. From types to type requirements: genericity for model-driven engineering. *Software and System Modeling* 12, 3 (2013), 453–474.

Juan de Lara and Esther Guerra. 2014. Towards the flexible reuse of model transformations: A formal approach based on graph transformation. *J. Log. Algebr. Meth. Program.* 83, 5-6 (2014), 427–458.

Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 12:1–12:46.

Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2015. A-posteriori typing for model-driven engineering. In *MODELS*. IEEE, 156–165.

Zinovy Diskin, Romina Eramo, Alfonso Pierantonio, and Krzysztof Czarnecki. 2016a. Incorporating uncertainty into bidirectional model transformations and their delta-lens formalization. In *Bx 2016 (CEUR Workshop Proceedings)*, Vol. 1571. CEUR-WS.org, 15–31.

Zinovy Diskin, Hamid Gholizadeh, Arif Wider, and Krzysztof Czarnecki. 2016b. A three-dimensional taxonomy for bidirectional model synchronization. *Journal of Systems and Software* 111 (2016), 298–322.

Zinovy Diskin, T. Maibaum, and Krzysztof Czarnecki. 2012. Intermodeling, queries, and Kleisli categories. In *FASE (LNCS)*, Vol. 7212. Springer, 163–177.

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani, and Paola Giannini. 2002. More dynamic object reclassification: Fickle$_{||}$. *ACM ToPLaS.* 24, 2 (2002), 153–191.

Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer.

Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. 2015. The essence of bidirectional programming. *SCIENCE CHINA Information Sciences* 58, 5 (2015), 1–21.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3, Article 17 (2007), 65 pages.

Lidia Fuentes and Antonio Vallecillo. 2004. An introduction to UML profiles. *UPGRADE* V, 2 (2004), 6–13.

Georg Gottlob, Michael Schrefl, and Brigitte Röck. 1996. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.* 14, 3 (1996), 268–296.

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel dos Reis, and Andrew Lumsdaine. 2006. Concepts: linguistic support for generic programming in C++. *SIGPLAN Not.* 41, 10 (2006), 291–310.

Esther Guerra and Juan de Lara. 2017. Automated analysis of integrity constraints in multi-level models. *Data & Knowledge Engineering* 107 (2017), 1 – 23.

Esther Guerra, Juan de Lara, Alessio Malizia, and Paloma Díaz. 2009. Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information & Software Technology* 51, 4 (2009), 769–784.

Clement Guy, Benoît Combemale, Steven Derrien, Jim Steel, and Jean-Marc Jézéquel. 2012. On model subtyping. In *ECMFA (LNCS)*, Vol. 7349. Springer, 400–415.

Frank Hermann, Hartmut Ehrig, and Claudia Ermel. 2009. Transformation of type graphs with inheritance for ensuring security in e-government networks. In *FASE (LNCS)*, Vol. 5503. Springer, 325–339.

Stephan Herrmann. 2007. A precise model for contextual roles: The programming language ObjectTeams/-Java. *Applied Ontology* 2, 2 (2007), 181–207. See also http://www.eclipse.org/objectteams/.

Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. 2016. Feature-based classification of bidirectional transformation approaches. *Software and System Modeling* 15, 3 (2016), 907–928.

Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press.

Johannes Jakob, Alexander Königs, and Andy Schürr. 2006. Non-materialized model view specification with triple graph grammars. In *ICGT (LNCS)*, Vol. 4178. Springer, 321–335.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Software Engineering Institute.

Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley. http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html

Haim Kilov, Peter F. Linington, José Raúl Romero, Akira Tanaka, and Antonio Vallecillo. 2013. The reference model of open distributed processing: Foundations, experience and applications. *Computer Standards & Interfaces* 35, 3 (2013), 247–256.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. The Epsilon Object Language. In *ECMFA (LNCS)*, Vol. 4066. Springer, 128–142.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2008. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations, First International Conference, ICMT (Lecture Notes in Computer Science)*, Vol. 5063. Springer, 46–60.

Mirco Kuhlmann and Martin Gogolla. 2012. From UML and OCL to relational logic and back. In *MODELS (LNCS)*, Vol. 7590. Springer, 415–431.

Thomas Kühne. 2013. On model compatibility with referees and contexts. *Software and System Modeling* 12, 3 (2013), 475–488.

Liwu Li. 2004. Extending the Java language with dynamic classification. *Journal of Object Technology* 3, 7 (2004), 101–120.

T. Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580.

James Noble, Antero Taivalsaari, and Ivan Moore. 1999. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag.

Antoni Olivé. 2007. *Conceptual Modeling of Information Systems*. Springer.

OMG. 2013a. SMOF 1.0. http://www.omg.org/spec/SMOF/1.0/. (2013).

OMG. 2013b. UML 2.5. http://www.omg.org/spec/UML/2.5/Beta2/. (2013).

OMG. 2014. MOF 2.4.2. http://www.omg.org/spec/MOF/. (2014).

OMG. 2016. QVT 2.0. http://www.omg.org/spec/QVT/. (2016).

Richard Paige, Dimitrios Kolovos, Louis Rose, Nicholas Drivalos, and Fiona Polack. 2009. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS*. IEEE Computer Society, Washington, DC, USA, 162–171.

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

Rick Salay, Steffen Zschaler, and Marsha Chechick. 2015. Transformation reuse: What is the intent?. In *AMT (CEUR Workshop Proceedings)*, Vol. 1500. CEUR-WS.org, 7–15.

Rick Salay, Steffen Zschaler, and Marsha Chechik. 2016. Correct reuse of transformations is hard to guarantee. In *ICMT (LNCS)*, Vol. 9765. Springer, 107–122.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2014. A component model for model transformations. *IEEE Trans. Software Eng.* 40, 11 (2014), 1042–1060.

Andy Schürr. 1994. Specification of graph translators with triple graph grammars. In *WG '94 (LNCS)*, Vol. 903. Springer, 151–163.

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

Friedrich Steimann. 2000. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.* 35, 1 (2000), 83–106.

Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley Professional, Upper Saddle River, NJ.

Perdita Stevens. 2010. Bidirectional model transformations in QVT: Semantic issues and open questions. *Software and System Modeling* 9, 1 (2010), 7–20.

Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. 2005. An adaptive object model with dynamic role binding. In *ICSE*. ACM, New York, NY, USA, 166–175.

David Ungar and Randall B. Smith. 1987. Self: The power of simplicity. In *OOPSLA*. ACM, New York, NY, USA, 227–242.

Christian Wende. 2012. *Language Family Engineering with Features and Role-Based Composition*. Ph.D. Dissertation. Technischen Universität Dresden.

Christian Wende, Nils Thieme, and Steffen Zschaler. 2009. A role-based approach towards modular language engineering. In *SLE (LNCS)*, Vol. 5969. Springer, 254–273.

Steffen Zschaler. 2014. Towards constraint-based model types: A generalised formal foundation for model genericity. In *VAO*. ACM, New York, NY, USA, Article 11, 8 pages.

Appendix

This appendix presents the details of some of the results of the paper. A supporting Alloy [Jackson 2002] formalization of typings and type-level retypings is available at http://miso.es/aposteriori/alloy.html, together with assertions helping in proving Theorems 4.1 and 6.1.

Proof of Theorem 4.1 (Type-level retyping correctness)

PROOF. Given an arbitrary model M, we check each well-formedness criterion enunciated in Section 3 for the view $TS(M)$ w.r.t. MM_R.

— *Object typing correctness*. Equation (1) holds because each object in $TS(M)$ has a type in MM_R, given by item (1) of the construction of $TS(M)$. Moreover, the typing in $TS(M)$ is total on objects, because there are no untyped objects in $TS(M)$. This is so as by item (2) in the construction of $TS(M)$, the objects whose creation type (or its supertypes) is not mapped to MM_R are not included in $TS(M)$.
— *Subclassing implies instance subsetting*. Equation (2) would fail if given two classes A', B' in MM_R, with $B' \in sub(A')$, there is an object o with $B' \in type^*(o)$ and $A' \notin type^*(o)$. However, this cannot occur by the very definition of $type^*$, with $type^*(o) = type(o) \bigcup_{A \in type(o)} anc(A)$.
— *Abstract classes do not have instances*. Equation (3) holds because, according to Equation (10), classes of MM_C cannot be mapped to abstract classes of MM_R. Assuming that $type$ is correct, we have that $\nexists A \in type(o)$ s.t. $abs(A)$. Hence, $\nexists A' \in type'(o)$ s.t. $abs(A')$ because no class in $ts_i(A)$ can be abstract.
— *Slot correctness*. Equation (4) would fail for an object o owning a slot whose type is not a feature of any class in $type^*(o)$. However, this is not permitted by Equation (11), which demands for any mapping $ts_i(A.a)$ that $ts_i(A)$ is defined and $ts_i(A.a) \in feats(ts_i(A))$.
— *Slot completeness*. Given an object o with $type(o) = A$, Equation (5) would fail if o lacks a slot for a mandatory feature of $ts_i(A)$ or its ancestors, and the feature has no default value. This might occur if TS allowed mapping non-mandatory to mandatory features. However, this is disallowed by Equation (12), which demands the cardinality interval of every feature $A'.a'$ to be equal or wider than the cardinality interval of any feature mapped to it. This means that, if $min(A'.a') > 0$, then, for any $A.a$ with $ts_i(A.a) = A'.a'$, we have $min(A'.a') \leq min(A.a)$, and hence $mand(A'.a') \implies mand(A.a)$. Additionally, by Equation (17), we have that if a mandatory feature in MM_R has not been mapped, but its owning class is mapped, then the feature has a default value.
— *Link correctness*. Equation (6) follows from Equation (14). Equation (6) would fail if there is a typed link $o.l$ such that: (a) the link points to an untyped object, or (b) the link points to an object with no type compatible with $tar(type(o.l))$. Possibility (a) cannot happen because Equation (14) requires that for any reference $A.r$ that is mapped, $tar(A.r)$ is mapped as well. Option (b) cannot happen either because if a reference $A.r$ is mapped, then the type $ts_i(tar(A.r))$ will be compatible with $tar(ts_i(A.r))$, since Equation (14) requires that $ts_i(tar(A.r)) \in sub^*(tar(ts_i(A.r)))$.

Hence, we can conclude that a typed link does point to objects whose type is compatible with the class pointed by the link type.

— *Composition correctness*. The property has two parts. First, Equation (7) requires that no object is pointed by two links whose type is a composition; this is guaranteed by Equation (16) which forbids mapping a composition reference twice. Second, Equation (8) requires acyclicity of containment links; this is guaranteed by Equation (15) which constrains composition references to be mapped only to compositions. Hence, if M satisfies this property, so does $TS(M)$.

— *Cardinality correctness*. Equation (9) would fail if the links stemming from an object break the cardinality interval defined by their types. However, assuming an initially correct $type$, this cannot happen because Equation (12) only allows mapping $A.r$ to a reference with same or wider cardinality, while Equation (13) takes care of not breaking the cardinality interval of any reference $A'.r'$ that receives several mappings. For attributes the reasoning is similar, but an attribute with default value may not be instantiated in an object. Equation (17) does not allow mapping default attributes to mandatory attributes with no default. This means that objects with no instantiation for a default attribute can only be retyped to default attributes, or to optional ones.

□

Proof of Theorem 5.1 (Instance-level retyping correctness)

PROOF. We follow the same strategy as with Theorem 4.1. Given an arbitrary model M, we check each well-formedness rule in Section 3 for the view $IS(M)$ w.r.t. MM_R.

— *Object typing correctness.* Similar to the type-level case, Equation (1) holds because any object in $IS(M)$ is typed by MM_R, and the typing from $IS(M)$ is total on objects.

— *Subclassing implies instance subsetting*. Assume two arbitrary mappings $is_i(exp_i : A) \mapsto A'$, $is_j(exp_j : B) \mapsto B'$ with $B' \in sub(A')$; and let $S_1 = exp_i(M)$, $S_2 = exp_j(M)$ be the object sets resulting from evaluating the expressions on model M. Hence, $\forall o \in S_1 \bullet A' \in type'(o)$, and $\forall o \in S_2 \bullet B' \in type'(o)$. Moreover, $\forall o \in S_1 \bullet \{A'\} \subseteq type'^*(o)$, and $\forall o \in S_2 \bullet \{A', B'\} \subseteq type'^*(o)$. Then, $S_2 = \{o \mid B' \in type'^*(o)\} \subseteq S_1 \cup S_2 = \{o \mid A' \in type'^*(o)\}$, as required by Equation (2).

— *Abstract classes do not have instances*. Equation (3) holds because, according to Equation (20), objects of M cannot be mapped to abstract classes of MM_R.

— *Slot correctness*. Equation (4) follows from Equation (21), which requires that if a slot of an object is mapped, then the mapping is to some feature of the class the object is mapped to.

— *Slot completeness*. Similar to the type-level case, Equation (5) holds due to Equation (24), which demands complete instantiations.

— *Link correctness*. Equation (6) holds if the invariant in Listing 6 is unsatisfiable. This is so as the invariant ensures that every object mapped to A' cannot contain (through reference r') an object that does not have any type compatible with B'. For this purpose, the invariant takes all expressions exp_{B_i} which yield types B'_1,..., B'_n compatible with B', and checks whether there is an object mapped to A' that may include in its mapping to r' an object b that does not belong to the union of those expressions. In such a case, the invariant would be satisfiable and Equation (6) would not hold, as $\forall T \in type(b) \bullet T \notin sub^*(tar(A.r))$, and hence, $type(b) \cap sub^*(tar(A.r)) = \emptyset$, contradicting Equation (6). Similarly, if there is an object mapped to A' which may contain in its mapping to r' an object b without a-posteriori types, the invariant would be satisfied and Equation (6) would fail as well. This is so as $type(b) = \emptyset$, and hence, $type(b) \cap sub^*(tar(A.r)) = \emptyset$. Conversely, if the invariant of Listing 6 is not satisfiable, then the expressions mapped to $A'.r'$ do not select objects which are not compatible with B', hence making Equation (6) hold.

— *Composition correctness*. The property has two parts. First, Equation (8) requires acyclicity of compositions. This is checked by the invariant template in Figure 17(b). If this invariant is satisfiable, then there is an object $o \in M$ selected by (say) expression exp_B such that exp_{rB} has a cycle. This would imply that $IS(M)$ contains a cycle of $A'.r'$ links, and therefore, Equation (8) would not hold. If the invariant is not satisfiable, then there is no object in exp_B (resp. exp_A) s.t. exp_{rB} (resp. exp_{rA}) has a cycle, and therefore, Equation (8) holds.
Second, Equation (7) requires no object is contained in two links with composition type. This is checked by the invariant templates in Listings 8 and 9. If the invariant in Listing 8 is satisfiable, it is because either the expression in line 1 or the expression in lines 3–6 is true. In the former case, there is a b object (with $B' \in type'(b)$) which contains a duplicate object in its link mapped to $B'.r'$, with $comp(B'.r')$. This makes Equation (7) fail because, setting $o_i = b = o_k$, we have that there are l, l_k both pointing to the same object, with $type(l) = type(l_k) = A'.r'$ and $comp(A'.r')$. Similarly, if the expression in lines 3–6 is true, then there is an object a contained in two links of objects b_1 and b_2, where the links are typed by the composition $A'.r'$. This makes Equation (7) fail because, setting $o_i = b_1$ and $o_k = b_2$, we have that there are l, l_k both pointing to the same object, with $type(l) = type(l_k) = A'.r'$ and $comp(A'.r')$. Finally, if the invariant in Listing 9 is satisfiable, then there is an object a contained in two links of objects b_1 and b_2 whose type is a composition. A similar reasoning to the previous cases makes Equation (7) fail.
Conversely, if the invariants in Listings 8 and 9 are not satisfiable, then no object typed by A' is contained in two links whose type are compositions, and hence, Equation (7) holds.

— *Cardinality correctness*. If the invariant in Listing 3 is satisfiable, then some of the expressions in lines 1–2, 4–5, or 8–11 (or expressions for some other n-fold combination) are true. If the expressions in lines 1–2 or 4–5 are true, then there is some object o selected by (say) exp_{j1} and making $A' \in type'(o)$, such that the sum of the size of the object sets selected by mappings $is_j(A.r_{11}) = A'.r'$,..., $is_j(A.r_{1m}) = A'.r'$ violates the cardinality of $A'.r'$. This makes Equation (9) false. If the expression in lines 8–11 is true, then there is some object o selected by two expressions exp_{j1} and exp_{j2}, such that the sum of the object sets selected by mappings $is_j(A.r_{11}) = A'.r'$,..., $is_j(A.r_{1m}) = A'.r'$, $is_j(A.r_{21}) = A'.r'$,..., $is_j(A.r_{2m}) = A'.r'$ violates the cardinality of $A'.r'$, making Equation (9) false. This reasoning is applicable to arbitrary n-fold combinations of expressions. Conversely, if the invariant in Listing 3 is not satisfiable, then there is no object o such that $A' \in type'(o)$ and the links of o typed by $A'.r'$ violate the cardinality of $A'.r'$. Therefore, in this case, Equation (9) holds. Similar to the type-level case, Equation (24) does not allow mapping default attributes to mandatory attributes with no default. Hence, objects with no instantiation for a default attribute can only be retyped to default attributes, or to optional ones.

□

Proof of Theorem 5.2 (Type-to-instance translation correctness)

PROOF. It is easy to see that the instance-level specification IS resulting from TS satisfies all well-formedness rules for instance-level specifications:

— *Non-abstract mappings*. By Equation (10), in every class mapping $ts_i(A) = A' \in TS$, A' is concrete. Hence, for every object mapping $is_i(A.allInstances() : A) = A' \in IS$, we have that A' is concrete.

— *Correct context for slot mappings*. By Equation (11), for every feature mapping $ts_i(A.a) = a'$, we have that $ts_i(A)$ is defined and $a' \in feats(ts_i(A))$. Since every slot mapping in IS has the form $is_i(A.a) = ts_i(A.a)$, then $is_i(A.allInstances() : A)$ is defined because $ts_i(A)$ is defined, and $a' \in feats(is_i(A.allInstances() : A))$ because $a' \in feats(ts_i(A))$.

— *Compatibility of feature cardinality*. By Equation (13), the cardinality interval of any feature a' of MM_R is wider than the sum of the cardinality intervals of all features mapped to a' by TS. Therefore, the cardinality checkings in Listing 3 will be always false, making the invariant unsatisfiable, as required for correctness.

— *Compatibility of reference type*. By Equation (14), for every reference mapping $ts_i(A.r) = r' \in TS$, we have that $ts_i(tar(A.r)) \in sub^*(tar(r'))$. Since every link mapping in IS has the form $is_i(A.r) = r'$, then there is a mapping $is_i(B.allInstances() : B) = B'$ with $B = tar(A.r)$ and $B' \in sub^*(tar(r'))$, as required by Equation (22). Moreover, the produced instance-level mappings make the invariant template in Listing 6 unsatisfiable. This is so, as we would have invariants of the form:

```
A.allInstances()→exists(a |
   a.r →exists(b |
      B₁.allInstances()→...→union(Bₙ.allInstances())→excludes(b)))
```

where $ts_{i_1}(B_1)$,..., $ts_{i_n}(B_n) \in sub^*(tar(r'))$. According to Equation (14), there is a mapping $ts_i(tar(A.r)) \in sub^*(tar(r'))$, and hence, one such $B_i = tar(A.r)$. As a consequence, the invariant cannot be satisfiable because all elements in collection r of every a s.t. $type^*(a) = A$ will be in the set $B_i.allInstances()$.

— *Compatibility of composition*. By Equation (15), TS cannot map non-compositions in MM_C to compositions in MM_R, i.e., only compositions can be mapped to compositions. Since there cannot be cycles of compositions, the invariant in Figure 17(b) cannot be satisfiable (expressions exp_{rA}, exp_{rB} will have the form $self.r$ with $comp(r)$). Because TS forbids mapping non-compositions to compositions, Listings 8 and 9 cannot be satisfiable. This is so as all exp_r have the form $self.r$ with $comp(r)$, but no object can be included in two composition links.

— *Complete instantiations*. By Equation (17), for every class mapping $ts_i(A) = A' \in TS$, we have that every mandatory feature of A' is mapped from some feature of A, or it has a default value. Hence, for the produced instance-level mappings $is_i(A.allInstances() : A) = A'$, we also have mappings $is_i(A.a)$ for every mandatory feature of A' that has no default value, making Equation (24) hold.

□

Proof of Theorem 6.1 (Bidirectionality)

PROOF. We check that Equations (26) to (33) are the converse or follow from the converse of Equations (10) to (17) for well-formedness of type-level specifications. For this purpose, we build TS^{-1} – the inverse of TS – from MM_R to MM_C by using the backward mappings ts_i^{-1}. Since the mappings $ts_i \in TS$ may map several classes (resp. features) of MM_C to a same class (resp. feature) of MM_R, in order to build TS^{-1}, we require a normalized specification TS^N with at most one class mapping in each ts_i and non-injective feature mappings. The normalized TS^N is built by placing each mapping from a class in MM_C, together with the mapping of its features, in a different ts_i. If there are two mappings from features f_1 and f_2 of a class A to the same feature in MM_R, then two functions ts_i, ts_j are created with one of these mappings each, and being equal in the rest of elements.

Then, we check the well-formedness conditions on TS^{-1}, and compare with the conditions required for bidirectionality, which are expressed over TS:

— Equation (10) forbids mappings to abstract classes. Applied to TS^{-1}, it yields: $\forall A' \in MM_R, \forall A \in MM_C \bullet ts_i^{-1}(A') = A \implies \neg abs(A)$. But $ts_i^{-1}(A') = A \iff ts_i(A) = A'$, which is precisely what Equation (26) requires.

— Equation (11) demands features of a class to be mapped to features of the mapped class. Applied to TS^{-1}, it yields:

$$\forall A' \in MM_R, \forall a' \in feats(A'), \forall a \in MM_C \bullet \\ ts_i^{-1}(A'.a') = a \implies ts_i^{-1}(A')\ is\ defined \wedge a \in feats(ts_i^{-1}(A'))$$

But we have that $ts_i^{-1}(A'.a') = a \iff ts_i(a) = A'.a'$, and $ts_i^{-1}(A')\ is\ defined \wedge a \in feats(ts_i^{-1}(A')) \iff \exists A \in MM_C \bullet ts_i(A) = A' \wedge a \in feats(A)$, which is required by Equation (27).

— Equation (12) demands features in MM_C mapped to a features in MM_R to have the same or wider cardinality interval. Applied to TS^{-1}, it yields:

$$\forall a \in MM_C, \forall a' \in MM_R \bullet ts_i^{-1}(a') = a \implies min(a) \leq min(a') \wedge max(a) \geq max(a')$$

We know that $ts_i^{-1}(a') = a \iff ts_i(a) = a'$. By Equation (12), we have $min(a') \leq min(a) \wedge max(a') \geq max(a)$. Then, it follows that $min(a) = min(a') \wedge max(a) = max(a')$, as Equation (28) demands.

— Equation (13) handles the general case for cardinality. Applied to TS^{-1}, it yields:

$$\forall a \in MM_C \bullet min(a) \leq \sum_{\substack{a' \in MM_R \\ ts_i^{-1}(a') = a}} min(a') \wedge max(a) \geq \sum_{\substack{a' \in MM_R \\ ts_i^{-1}(a') = a}} max(a') \tag{44}$$

Equation (28) requires $min(a) = min(a')$ when $ts_i(a) = a'$. Hence, Equation (44) can only hold for features $a \in MM_C$ mapped to several $a'_1, ..., a'_n \in MM_R$ if $min(a) = min(a'_1) = ... = min(a'_n) = 0$. A similar reasoning requires $max(a) = \infty$ if a is mapped to several $a'_1, ..., a'_n \in MM_R$. Equation (29) demands these conditions.

— Equation (14) demands compatibility of reference type. Applied to TS^{-1}, it yields:

$$\forall A' \in MM_R, \forall r' \in refs(A'), \forall r \in MM_C \bullet ts_i^{-1}(A'.r') = r \implies \\ \exists\, C' \in anc^*(tar(A'.r')) \bullet ts_i^{-1}(C') \in sub^*(tar(r))$$

We have $ts_i^{-1}(A'.r') = r \iff ts(r) = A'.r'$. Let be $C = ts^{-1}(C')$, with $C' \in anc^*(tar(A'.r'))$, and so, $C' = ts_i(C)$. Then, we can reformulate the condition as $\exists\, C \in sub^*(tar(r)) \bullet ts_i(C) \in anc^*(tar(A'.r'))$, as demanded by Equation (30).

— Equation (15) forbids a non-composition reference in MM_C to be mapped to a composition. Applied to TS^{-1} yields Equation (31), where we have used $ts_i^{-1}(r') = r \iff ts_i(r) = r'$.

— Equation (16) forbids mapping a reference in MM_C to two composition references in MM_R. Applied to TS^{-1} yields Equation (32), where we have used $ts_i^{-1}(A'.r') = r_1 \iff ts_i(r_1) = A'.r'$ and similarly for r_2.

— Equation (17) demands that, if a class A' is mapped, all its mandatory features should be mapped or have a default value. Applied to TS^{-1}, it yields Equation (33), where we have used $ts_i^{-1}(A') = A \iff ts_i(A) = A'$, and $ts_i^{-1}(A'.a') = A.a \iff ts_i(A.a) = A'.a'$.

□