Intermodeling, Queries, and Kleisli Categories

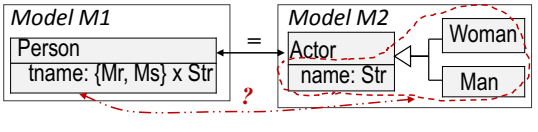
Zinovy Diskin[1,2], Tom Maibaum[1], and Krzysztof Czarnecki[2]

[1] Software Quality Research Lab,
McMaster University, Canada
[2] Generative Software Development Lab,
University of Waterloo, Canada
{zdiskin,kczarnec}@gsd.uwaterloo.ca, tom@maibaum.org

Abstract. Specification and maintenance of relationships between models are vital for MDE. We show that a wide class of such relationships can be specified in a compact and precise manner, if intermodel mappings are allowed to link derived model elements computed by corresponding queries. Composition of such mappings is not straightforward and requires specialized algebraic machinery. We present a formal framework, in which such machinery can be defined generically for a wide class of metamodel definitions. This enables algebraic specification of practical intermodeling scenarios, e.g., model merge.

1 Introduction

Model-driven engineering (MDE) is a prominent approach to software development, in which models of the domain and the software system are primary assets of the development process. Normally models are inter-related, perhaps in a very complex way, and to keep them consistent and use them coherently, relationships between models must be accurately specified and maintained. As noted in [1], "development of well-founded techniques and tools for the creation and maintenance of intermodel relations is at the core of MDE."



A major problem for intermodel specifications is that different models may structure the same information differently. The inset figure shows an example: model (class diagram) M1 considers Persons and their names with titles (attribute 'tname'), whereas M2 considers Actors and uses subclassing rather than titles. Suppose that classes Person in model M1 and Actor in M2 refer to the same class of entities but name them differently. We may encode this knowledge by linking the two classes with an "equality" link. In contrast, specifying "sameness" of tnames and subclassing is not straightforward and seems to be a difficult problem.

In the literature, such indirect relationships are usually specified by *correspondence rules* [2] or *expressions* [3] attached to the respective links (think of

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 163–177, 2012.

expressions replacing the question mark above). When such-annotated links are composed, it is not clear how to compose the rules; hence, it is difficult to manage scenarios that involve composition of intermodel mappings. The importance and difficulty of the mapping composition problem is well recognized in the database literature [3]; we think it will also become increasingly important in software engineering with the advancement and maturation of MDE methods.

The main goal of the paper is to demonstrate that the mapping composition problem can be solved by applying standard methods of categorical algebra, namely, the *Kleisli construction*, but applied in a non-standard way. In more detail, we present a specification framework, in which indirect links are replaced by direct links between derived rather than basic model elements. Here "derived" means that the element is computed by some operation over basic elements. We call such operations *queries*, in analogy with databases; the reader may think of some predefined query language that determines a class of legal operations and the respective derived elements. We will call links and mappings involving queries *q-links* and *q-mappings*.

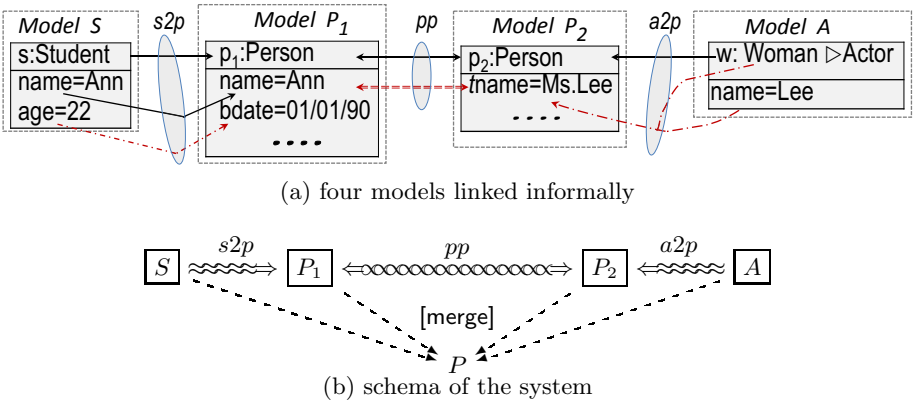
As q-mappings are sequentially composable, the universe of models and q-mappings between them can be seen as a category (in precise terms, the Kleisli category of the monad modeling the query language). Hence, intermodeling scenarios become amenable to algebraic treatment developed in category theory. We consider connection to categorical machinery to be fruitful not only theoretically, but also practically as a source of useful design patterns. In particular, we will show that q-mappings are instrumental for specifying and guiding model merge.

The paper is structured as follows. Sections 2 and 3 introduce our running example and show how q-links and q-mappings work for the problem of model merge. Section 4 explains the main points of the formalization: models' conformance to metamodels, retyping, the query mechanism and q-mappings. Section 5 briefly describes related work and Section 6 concludes.

2 Running Example

To illustrate the issues we need to address, let us consider a simple example of model integration in Fig. 1. Subfigure (a) presents four object models. The expression o:Name declares an object o of class Name; the lower compartment shows o's attribute values, and ellipses in models P_1, P_2 refer to other attributes not shown. In model A, class Woman extends class Actor. When we refer to an element e (an object or an attribute) of model X, we write $e@X$. Arrows between models denote intermodel relationships explained below.

Suppose that models P_1 and P_2 are developed by two different teams charged with specifying different aspects of the same domain—different attributes of the same person in our case. The bidirectional arrow between objects $p_1@P_1$ and $p_2@P_2$ means that these objects are different representations of the same person. Model P_1 gives the first name; P_2 provides the last name and the title of the person ('tname'). We thus have a complex relationship between the attributes,



(a) four models linked informally

(b) schema of the system

Fig. 1. Running example: four models and their relationships, informally

shown by a dashed link (brown with a color display): both attributes talk about names but are complementary. Together, the two links form an informal mapping *pp* between the models.

We also assume that model P_1 is supplied with a secondary model S, representing a specific view of P_1 to be used and maintained locally at its own site (in the database jargon, S is a materialized view of P_1). Mapping $s2p$, consisting of three links, defines the view informally. Two solid-line links declare "sameness" of the respective elements. The dash-dotted link shows relatedness of the two attributes but says nothing more. Similarly, mapping $a2p$ is assumed to define model A as a view to model P_2: the solid link declares "sameness" of the two objects, and the dash-dotted link shows relatedness of their attributes and types. Mappings $s2p$, pp and $a2p$ bind all models together, so that a virtual integrated (or merged) model, say P, should say that Ms. Ann Lee is a 22 year old student and female actor born on Jan 1, 1990. Diagram Fig. 1(b) presents the merge informally: horizontal fancy arrows denote intermodel mappings, and dashed inclined arrows show mappings that embed the models into the merge.

Building model management tools capable of performing integration like above for industrial models (normally containing thousands of elements) requires clear and precise specifications of intermodel relationships. Hence, we need a framework in which intermodel mappings could be specified formally; then, operations on models and model mappings could be described in precise algebraic terms. For example, merging would appear as an instance of a formal operation that takes a diagram of models and mappings and produces an integrated model together with embeddings as shown in Fig. 1(b). We want such descriptions to be generic and applicable to a wide class of scenarios over different metamodels. Category theory does provide a suitable methodological framework (cf. [4,5,6]), e.g., homogeneous merge can be defined as the colimit of the corresponding diagram [7,8], and heterogeneity can be treated as shown in [9]. However, the basic prerequisite for applying categorical methods is that mappings and their

composition must be precisely defined. It is not straightforward even in our simple example, and we will briefly review the problems to be resolved.

Thinking in terms of elements, a mapping should be a set of links between models' elements as shown by ovals in Fig. 1(a). We can consider a link formally as a pair of elements, and it works for those links in Fig. 1(a), which are shown with solid lines. Semantically, such a link means that two elements represent the same entity in the real world. However, we cannot declare attributes 'age' in model S (we write 'age'@S) and 'bdate'@P_1 to be "the same" because, although related, they are different. Even more complex is the relationship between attribute 'tname' in base model P_2 and the view model A: it involves attributes and types (the Woman-Actor subclassing) and is shown informally by a two-to-one dash-dotted link. Finally, the dashed link between elements 'name'@P_1 and 'tname'@P_2 encodes a great deal of semantic information described above.

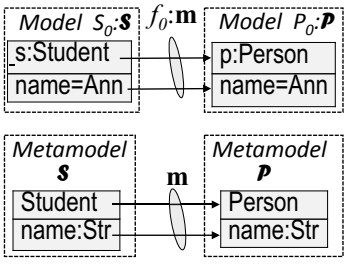
As stated in the Introduction, managing indirect links via their annotation by correspondence rules or expressions leads to difficult problems in mapping composition. In contrast, the Kleisli construction developed in categorical algebra provides a clear and concise specification framework, in which indirect relationships are modeled by q-mappings; the latter are associatively composable and constitute a category. The next section explains the basic points of the approach.

3 Intermodeling and Kleisli Mappings

We consider our running example and incrementally introduce main features of our specification framework.

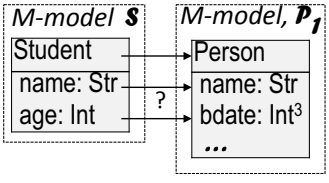
3.1 From Informal to Formal Mappings

Type Discipline. Before matching models, we need to match their metamodels. Suppose that we need to match models S_0 and P_0 over corresponding metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$, resp. (see the inset figure on the right), linking objects s@S_0 and p@P_0 as being "the same". These objects have different types ('Student' and 'Person', resp.), however, and, with a strict type discipline, they cannot be matched. Indeed, the two objects can only be "equated" if we know that their types actually refer to the same, or, at least, overlapping, classes of real world objects. For simplicity, we assume that classes Student@$\boldsymbol{\mathcal{S}}$ and Person@$\boldsymbol{\mathcal{P}}$ refer to the same class of real world entities but are named differently; and their attributes 'name' also mean the same. To make this knowledge explicit, we match the metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$ via mapping $\boldsymbol{m}$ as shown in the inset figure. After the metamodels are matched, we can match type-safely objects s



and p, and their attributes as well. The notation f_0:$\boldsymbol{m}$ means that each link in mapping f_0 is typed by a corresponding link in mapping $\boldsymbol{m}$. Below we will often omit metamodel postfixes next to models and model mappings if they are clear from the context.

Indirect Linking, Queries and Q-mappings. As argued above, to specify relationships between models S and P_1 in Fig. 1, we first need to relate their metamodels (the inset figure on the right). We cannot "equate" attributes 'age' and 'bdate', however. The cornerstone of our approach to intermodeling is to specify indirect relationships by *direct* links to derived elements computed with suitable queries. For example, attribute 'age' can be derived from 'bdate' with an obvious query $Q1$:



$$/\text{age} = Q1(\text{bdate}) = 2012 - \text{bdate.byear},$$

Our notation follows UML by prefixing the names of derived elements by slash; $Q1$ is the name of the query; 2012 − bdate.byear is its definition; and 'byear' denotes the year-field of the bdate-records. Now the relation between metamodels $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}_1$ is specified by three directed links, i.e., pairs, (Student, Person), (name, name) and (age, /age) as shown in the bottom of Fig. 2(a) (basic elements are shaded; the derived attribute '/age' is blank). The three links form a *direct* mapping $\boldsymbol{m_1}\colon \boldsymbol{\mathcal{S}} \to \boldsymbol{\mathcal{P}}_1^+$, where $\boldsymbol{\mathcal{P}}_1^+$ denotes metamodel $\boldsymbol{\mathcal{P}}_1$ augmented with derived attribute /age. Since mapping $\boldsymbol{m_1}$ is total, it indeed defines metamodel $\boldsymbol{\mathcal{S}}$ as a view of $\boldsymbol{\mathcal{P}}_1$. Query $Q1$ can be executed for any model over metamodel $\boldsymbol{\mathcal{P}}_1$, in particular, P_1 (Fig. 2(a) top), which results in augmenting model P_1 with the corresponding derived element; we denote the augmented model by P_1^+. Now model S can be directly mapped to model P_1^+ as shown in Fig. 2(a), and each link in mapping f_1 is typed by a corresponding link in mapping $\boldsymbol{m_1}$.

The same idea works for specifying mapping $a2p$ in Fig. 1. The only difference is that now derived elements are computed by a more complex query (with two *select-from-where* clauses, 'title=Ms' and 'title=Mr') as shown in Fig. 2(b): mapping $\boldsymbol{m_2}$ provides a view definition, which is executed for model P_2 and results in view model A and traceability mapping f_2. Thus, we formalize arrows $s2p$, $a2p$ in Fig. 1 as *q-mappings*, that is, mappings into models and metamodels augmented with derived elements. Ordinary mappings can be seen as degenerate q-mappings that do not use derived elements.

Links-with-New-Data via Spans. In Section 2, relationships between models P_1 and P_2 in Fig. 1 were explained informally. Fig. 3 gives a more precise description. We first introduce a new metamodel $\boldsymbol{\mathcal{P}}_{12}$ (the shaded part of metamodel $\boldsymbol{\mathcal{P}}_{12}^+$), which specifies new concepts assumed by the semantics. Then we relate these new concepts to the original ones via mappings $\boldsymbol{r}_1, \boldsymbol{r}_2$; the latter one uses derived elements. Queries $Q4_{1,2}$ are projection operations, and query Q_3 is the pairing operation. In particular, mapping $\boldsymbol{r}_2$ says that attribute 'fname'@$\boldsymbol{\mathcal{P}}_{12}^+$ does not match any attribute in model $\boldsymbol{\mathcal{P}}_2^+$, 'lname'@$\boldsymbol{\mathcal{P}}_{12}^+$ is the

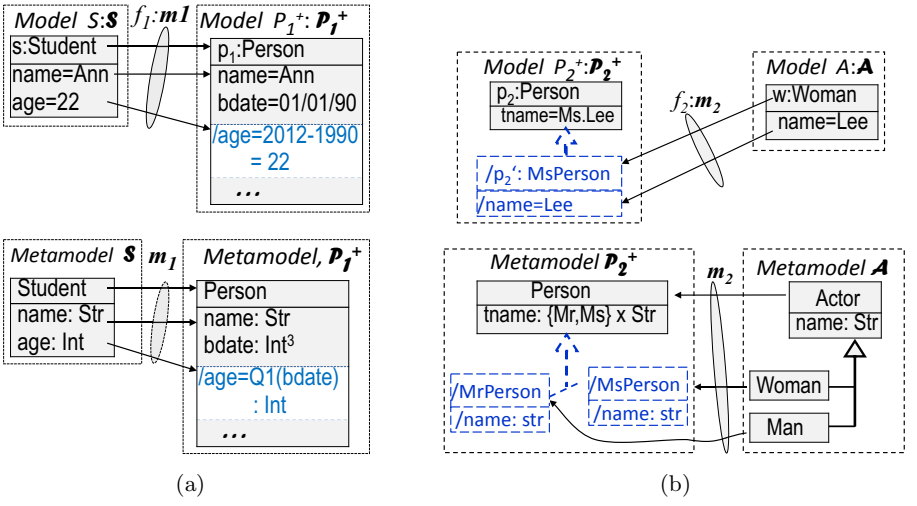


Fig. 2. Indirect matching via queries and direct mappings

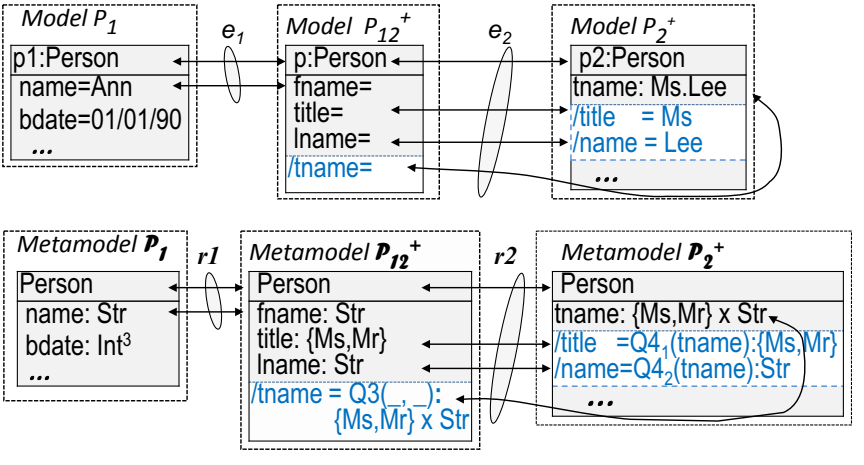


Fig. 3. Matching via spans and queries

same as '/name'@$\boldsymbol{\mathcal{P}}_2^+$ (i.e., the second component of 'tname'), and 'tname'@$\boldsymbol{\mathcal{P}}_2^+$ "equals" the pair of attributes (title, lname) in $\boldsymbol{\mathcal{P}}_{12}^+$.

On the level of models, we introduce a new model P_{12} to declare sameness of objects $p_1@P_1$ and $p_2@P_2$, and to relate their attribute slots. The new attribute slots are kept empty—they will be filled-in with the corresponding local values during the merge.

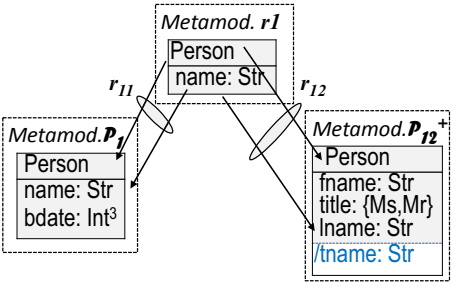
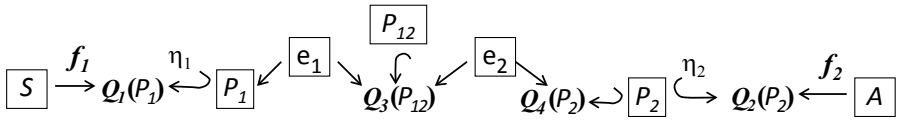


Fig. 4. Partial mappings via spans

It is well-known that algebra of totally defined functions is much simpler than that of partially defined ones. Neither of the mappings $\boldsymbol{r}_k$, e_k $(k = 1, 2)$ is total (recall that $\boldsymbol{\mathcal{P}}_2$ and P_2 may contain other attributes not shown in our diagrams). To replace these partial mappings with total ones, we apply a standard categorical construction called a *span*, as shown in Fig. 4 for mapping $\boldsymbol{r}_1$. We reify $\boldsymbol{r}_1$ as a new model $\boldsymbol{r}1$ equipped with two total projection mappings $\boldsymbol{r}_{11}$, $\boldsymbol{r}_{12}$.

Thus, we have specified all our data via models and functional q-mappings as shown in the diagram below; arrows with hooked tails denote inclusions of models into their augmentations with derived elements computed with queries Qi.



3.2 Model Merging: A Sample Multi-mapping Scenario

We want to integrate data specified by the diagram above. We focus first on merging models P_1, P_2 and P_{12} without data loss and duplication. The type discipline prescribes merging their metamodels first. To merge metamodels $\boldsymbol{\mathcal{P}}_1^+$, $\boldsymbol{\mathcal{P}}_2^+$, and $\boldsymbol{\mathcal{P}}_{12}^+$ (see Fig. 3), we take their disjoint union (no loss), and then glue together elements related by mappings $\boldsymbol{r}_{1,2}$ (to avoid duplication). The result is shown in Fig. 5(a). There is a redundancy in the merge since attribute 'tname' and pair (title, lname) are mutually derivable. We need to choose either of them as a basic structure, then the other will be derived (see Fig. 5(b1,b2)) and could be omitted from the model. We call this process *normalization*. Thus, there are two normalized merged metamodels. Amongst the three metamodels to be merged, we favor metamodel $\boldsymbol{\mathcal{P}}_{12}$ in which attribute 'tname' is considered derived from 'title' and 'lname', and hence choose metamodel $\boldsymbol{\mathcal{P}}_{n1}^+$ as the merge result (below we omit the subindex).

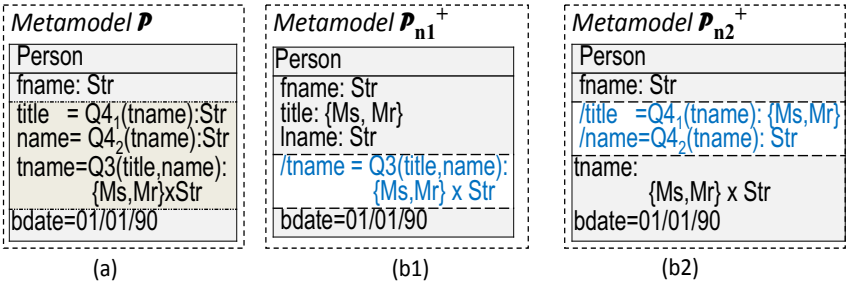


Fig. 5. Normalizing the merge

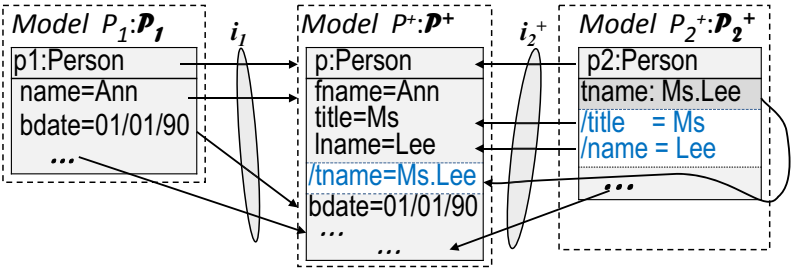


Fig. 6. Result of the merge modulo match in Fig. 3

Now take the disjoint union of models P_1^+, P_2^+, P_{12}^+ (Fig. 3), and glue together elements linked by mappings $e_{1,2}$. Note that we merge attribute slots rather than values; naming conflicts are resolved in favor of names used in metamodel $\mathcal{P}_{12}^+$. The merged model is in Fig. 6. Note how important is the interplay between basic-derived elements in mapping e_2 in Fig. 3: without these links, the merge would contain redundancies. All three component models are embedded into the merge by injective mappings $i_{1,2,3}$ (mapping i_3 is not shown).

Merge and Integration, Abstractly. The hexagon area in Fig. 7 presents the merge described above, now in an abstract way. Nodes in the diagram denote models; arrows are functional mappings, and hooked-tail arrows are inclusions. Computed mappings are shown with dashed arrows (blue if colored), and computed model P^+ is not framed.

Building model P^+ does not complete integration, however. Our system of models also has two view models, S and A, and to complete integration, we need to show how views S and A are mapped into the merge P. For this goal, we need to translate queries Q_1 and Q_2 to, resp., models P_1 and P_2 from their original models to the merge model P^+ using mappings i_1, i_2. We achieve the translation by replacing each element $x@P_k$ occurring in the expression defining query Q_k $(k = 1, 2)$ by the respective element $i_k(x)@P^+$. Then we execute the queries and augment model P^+ with the respective derived elements, as shown by inclusion mappings $\eta_k^\sharp$ $(k = 1, 2)$ within the lane (a-b) in the figure: we add to model P^+ derived attribute /age (on the left) and two derived subclasses,

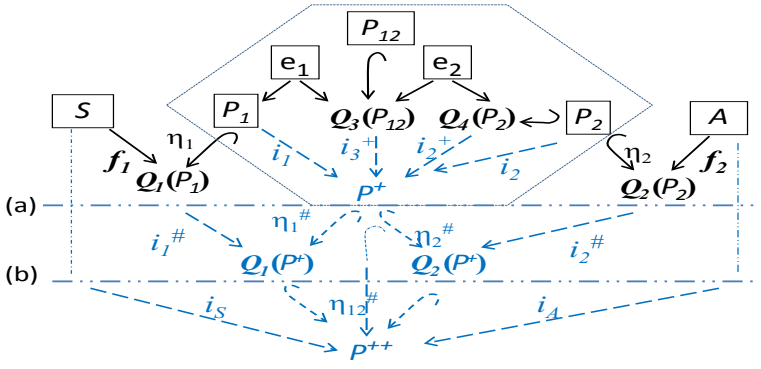


Fig. 7. The merge example, abstractly

/MrPerson and /MsPerson (on the right). Since model P^+ is embedded into its augmentations $Q_k(P^+)$ $(k = 1, 2)$, and queries Q_k preserve data embedding (are *monotonic* in database jargon), the result of executing Q_k against model P_k can be embedded into the result of executing Q_k against P^+. So, we have mappings $i_k^\sharp$ making squares $[P_k\ P^+\ Q_k(P^+)\ Q_k(P_k)]$ $(k = 1, 2)$ commutative.

Finally, we merge queries Q_1 and Q_2 to model P^+ into query Q_{12}, whose execution adds to model P^+ both derived attribute /age and the derived subclasses. We denote the resulting model by P^{++} and η_{12}: $P^+ \hookrightarrow P^{++}$ is the corresponding inclusion (see the lower diamond in Fig. 7). Now we can complete integration by building mappings i_S: $S \to P^{++}$ and i_A: $A \to P^{++}$ by sequential composition of the respective components. These mappings say that Ms. Ann Lee is a student and an actor—information that neither P^+ nor P^{++} provide.

3.3 The Kleisli Construction

The diagram in Fig. 7 is precise but looks too detailed in comparison with the informal diagram Fig. 1(b). We want to design a more compact yet still precise notation for this diagram.

Note that the diagram uses frequently the following mapping pattern

$$X \xrightarrow{\ f\ } Q(Y) \xleftarrow{\ \eta\ } Y,$$

where X, Y are, resp., the source and the target models; $Q(Y)$ is augmentation of Y with elements computed by a query Q to Y; and η is the corresponding inclusion. The key idea of the Kleisli construction developed in category theory is to view this pattern as an arrow $K\colon X \Rightarrow Y$ comprising two components: a query Q_K to the target Y and a functional mapping $f_K\colon X \to Q_K(Y)$ into the corresponding augmentation of the target. Thus, the query becomes a part of the mapping rather than of model Y, and we come to the notion of q-mapping mentioned above. We will often denote q-mappings by double-body arrows to recall that they encode both a query and a functional mapping. By a typical

abuse of notation, a q-mapping and its second component (the functional mapping) will be often denoted by the same letter; we write, say, $f\colon X \Rightarrow Y$ and $f\colon X \to Q(Y)$ using letter f for both. With this notation, the input data for integration (framed nodes and solid arrows in diagram Fig. 7) are encoded by the following diagram

$$\boxed{S} \xRightarrow{f1} \boxed{P_1} \xLeftarrow{e_1} \boxed{P_{12}} \xLeftarrow{e_2} \boxed{P_2} \xLeftarrow{f_2} \boxed{A}$$

where spans e_1, e_2 from Fig. 7 are encoded by arrows with bullets in the middle. Note a nice similarity between this and our original diagram Fig. 1(b)(its upper row of arrows); however, in contrast to the latter, the arrows in the diagram above have the precise meaning of q-mappings.

Finally, we want to formalize the integration procedure as an instance of the colimit operation: as well-known, the latter is a quite general pattern for "putting things together" [4]; see also [7,10,8] for concrete examples related to MDE. To realize the merge-as-colimit idea, we need to organize the universe of models and q-mappings into a category, that is, define identity q-mappings and composition of q-mappings. The former task is easy: given a model X, its identity q-mapping $\mathbb{1}_X\colon X \Rightarrow X$ comprises the empty query $Q_\varnothing$, so that $Q_\varnothing(X) = X$, and the mapping $1_X\colon X \to Q_\varnothing(X)$, which is the identity mapping of X to itself.

Composition of q-mappings is, however, non-trivial. Given two composable q-mappings $f\colon X \Rightarrow Y$ and $g\colon Y \Rightarrow Z$, defining their composition $f;g\colon X \Rightarrow Z$ is not straightforward, as shown by the diagram in Fig. 8 (ignore the two dashed arrows and their target for a moment): indeed, after unraveling, mappings f and g are simply not composable. To manage the problem, we need to apply query Q_f to model $Q_g(Z)$ and correspondingly extend mapping g as shown in the diagram. Composition of two queries is again a query, and thus pair $(f; g^{\#}, Q_f \circ Q_g)$ determines a new q-mapping from X to Z.

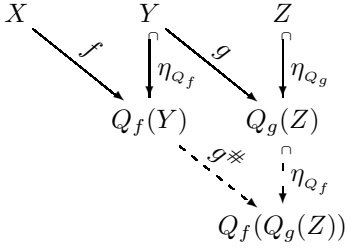


Fig. 8. Q-mapping composition

The passage from g to $g^{\#}$—the *Kleisli extension operation*—is crucial for the construction. (Note that we have used this operation in Fig. 7 too). On the level of metamodels and query definitions (syntax only), Kleisli extension is simple and amounts to term substitution. However, queries are executed for models, and an accurate formal definition of the Kleisli extension needs non-trivial work to be done. We outline the main points in the next section.

4 A Sketch of the Formal Framework

Due to space limitations, we describe very briefly the main points of the formal framework. All the details, including basic mathematical definitions we use, can be found in the accompanying technical report [11] (the TR).

4.1 Model Translation, Traceability and Fibrations

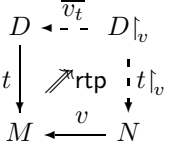
The Carrier Structure. We fix a category $\mathbb{G}$ with pullbacks, whose objects are to be thought of as (directed) graphs, or many-sorted (colored) graphs, or attributed graphs [12]. The key point is that they are definable by a metametamodel itself being a graph with, perhaps, a set of *equational* constraints. In precise categorical terms, we require $\mathbb{G}$ to be a presheaf topos [13], and hence a $\mathbb{G}$-object can be thought of as a system of sets and functions between them (e.g., a graph consists of two sets, Nd and Arr, and two functions from Arr to Nd—think of the source and the target of an arrow). It allows us to talk about elements of $\mathbb{G}$-objects, and ensures that $\mathbb{G}$ has limits, colimits, and other good properties. We will call $\mathbb{G}$-objects *'graphs'* (and as a rule skip the quotes), and write $e \in G$ to say that e is an element of graph G.

For a graph M thought of as a metamodel, an *M-model* is a pair $A = (D_A, t_A)$ with D_A a graph and $t_A\colon D_A \to M$ a mapping (arrow in category $\mathbb{G}$) to be thought of as *typing*. In a heterogeneous environment with models over different metamodels, we may say that a model A is merely an arrow $t_A\colon D_A \to M_A$ in $\mathbb{G}$, whose target M_A is called *the metamodel* of A (or the *type graph*, and the source D_A is the *data carrier* (the *data graph*). In our examples, a typing mapping for OIDs was set by colons: writing p:Person for a model A means that $p \in D_A$, Person $\in M_A$ and $t_A(p) =$ Person. For attributes, our notation covers even more, e.g., writing 'name=Ann' (nested in class Person) refers to some arrow $x\colon y \to$ Ann in graph D_A, which is mapped by t_A to arrow *value*: name $\to$ *String* in graph M_A, but names of elements x, y are not essential for us. Details can be found in [10, Sect.3].

A *model mapping* $f\colon A \to B$ is a pair of $\mathbb{G}$-mappings, $f_{\text{meta}}\colon M_A \to M_B$ and $f_{\text{data}}\colon D_A \to D_B$, commuting with typing: $f_{\text{data}}; t_B = t_A; f_{\text{meta}}$. Below we will also write f_M for f_{meta} and f_D for f_{data}. Thus, a model mapping is a commutative diagram; we usually draw typing mappings vertically and mappings f_M, f_D horizontally. We assume the latter to be monic (or injective) in $\mathbb{G}$ like in all our examples. This defines category ***Mod*** of models and model mappings.

As each model A is assigned with its metamodel M_A, and each model mapping $f\colon A \to B$ with its metamodel component $f_M\colon M_A \to M_B$, we have a projection mapping $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$, where we write ***MMod*** for either entire category $\mathbb{G}$ or for its special subcategory of 'graphs' that can serve as metamodels (e.g., all finite 'graphs'). It is easy to see that $\boldsymbol{p}$ preserves mapping composition and identities, and hence is a functor.

To take into account constraints, we need to consider metamodels as pairs $M = (G_M, C_M)$ with G_M a carrier graph and C_M a set of constraints. Then not any typing $t_A\colon D_A \to G_M$ is a model: a legal t_A must also satisfy all constraints in C_M. Correspondingly, a legal mapping $f\colon M \to N$ must be a 'graph' mapping $G_M \to G_N$ compatible with constraints in a certain sense (see [10] or [8] for details). We do not formalize constraints in this paper, but in our abstract definitions below, objects of category ***MMod*** may be understood as pairs $M = (G_M, C_M)$ as above, and ***MMod***-arrows as legal metamodel mappings.



Retyping. Any metamodel mapping $v\colon M \leftarrow N$ generates retyping of models over M into models over N as shown by the diagram on the right. If an element $e \in N$ is mapped to $v(e) \in M$, then any element in 'graph' D typed by $v(e)$, is retyped by e. Graph $D\restriction_v$ consists of such retyped elements of D, and mapping $\overline{v_t}$ traces their origin. Overall, we have an operation that takes two arrows, v and t, and produces two arrows, $\overline{v_t}$ and $t\restriction_v$, together making a commutative square as shown above.

Formally, elements of $D\restriction_v$ can be identified with pairs $(e, d) \in N \times D$ such that $v(e) = t(d)$, and mappings $t\restriction_v$ and $\overline{v_t}$ are the respective projections. The operation just described is well-known in category theory by the name *pullback* (PB) : typing arrow $t\restriction_v\colon D\restriction_v \to N$ is obtained by *pulling back* arrow t along arrow v. If we want to emphasize the vertical dimension of the operation, we will say that traceability arrow $\overline{v_t}$ is obtained by *lifting* arrow v along t.

Abstract Formulation via Fibrations. Retyping can be specified as a special property of functor $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$. That is: for an arrow $v\colon M \leftarrow N$ in $\boldsymbol{MMod}$, and an object A over M (i.e., such that $\boldsymbol{p}(A) = M$), there is an arrow $\overline{v_A}\colon A \leftarrow A\restriction_v$ over v (i.e., a commutative diagram as above), which is maximal in a certain sense amongst all arrows (commutative squares) over v. Such an arrow is called the *(weak)* $\boldsymbol{p}$*-Cartesian lifting* of arrow v, and is defined up to canonical isomorphism. Functor $\boldsymbol{p}$ with a chosen Cartesian lifting for any arrow v, which is compatible with arrow composition, is called a *split fibration* (see [14, Exercise 1.1.6]). Thus, existence of model retyping can be abstractly described by saying that we have a split fibration $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$.

We will call such a fibration an *(abstract) metamodeling framework*.

4.2 Query Mechanism via Monads and Fibrations

Background. A *monad* (in the Kleisli form) over a category $\boldsymbol{C}$ is a triple $(\mathsf{Q}, \eta, \_^{\#})$ with $\mathsf{Q}\colon \boldsymbol{C}_0 \to \boldsymbol{C}_0$ a function on $\boldsymbol{C}$-objects, η an operation that assigns to any object $X \in \boldsymbol{C}_0$ a $\boldsymbol{C}$-arrow $\eta_X\colon X \to \mathsf{Q}(X)$, and $\_^{\#}$ an operation that assigns to any $\boldsymbol{C}$-arrow $f\colon X \to \mathsf{Q}(Y)$ its *Kleisli extension* $f^{\#}\colon \mathsf{Q}(X) \to \mathsf{Q}(Y)$ such that $\eta_X; f^{\#} = f$. Two additional laws hold: $\eta_X^{\#} = 1_{\mathsf{Q}(X)}$ for all X, and $f^{\#}; g^{\#} = (f; g^{\#})^{\#}$ for all $f\colon X \to \mathsf{Q}(Y)$, $g\colon Y \to \mathsf{Q}(Z)$. In our context, if $\boldsymbol{C}$-objects are models and a monad over $\boldsymbol{C}$ is given by a query language, object $\mathsf{Q}(X)$ is to be understood as model X augmented with all derived elements computable by all possible queries. In other words, $\mathsf{Q}(X)$ is the object of queries against model X. We will identify a monad by its first component.

Any monad Q generates its *Kleisli* category $\boldsymbol{C}_{\mathsf{Q}}$. It has the same objects as $\boldsymbol{C}$, but a $\boldsymbol{C}_{\mathsf{Q}}$-arrow $f\colon X \Rightarrow Y$ is a $\boldsymbol{C}$-arrow $f\colon X \to \mathsf{Q}(Y)$. Thus, Kleilsi arrows are a special "all-queries-together" version of our q-mappings. As we have seen in Sect. 3.3, Fig. 8, composition of $\boldsymbol{C}_{\mathsf{Q}}$-arrows, say, $f\colon X \Rightarrow Y$ and $g\colon Y \Rightarrow Z$ is not immediate since f's target and g's source do not match after unraveling their definitions. The problem is resolved with the Kleisli extension operation and, moreover, the laws ensure that $\boldsymbol{C}$-objects and $\boldsymbol{C}_{\mathsf{Q}}$-arrows form a category.

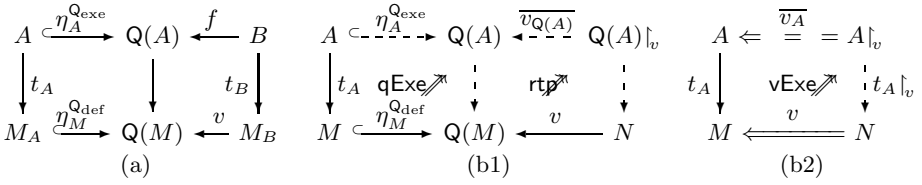


Fig. 9. Q-mappings (a) and view mechanism (b1,b2)

Lemma 1 ([15]). *If category* $\boldsymbol{C}$ *has colimits of all diagrams from a certain class* $\mathcal{D}$*, then the Kleisli category* $\boldsymbol{C}_{\mathsf{Q}}$ *has* $\mathcal{D}$*-colimits as well.*

Query Monads and Their Kleisli Categories. In the TR, we carefully motivate the following definition:

Definition 1 (main) A *monotonic query language* over an abstract metamodeling framework $\boldsymbol{p}$: $\boldsymbol{Mod} \to \boldsymbol{MMod}$ is a pair of monads $(\mathsf{Q}, \mathsf{Q}_{\mathrm{def}})$ over categories $\boldsymbol{Mod}$ and $\boldsymbol{MMod}$, resp., such that $\boldsymbol{p}$ is a monad morphism, and monad Q is $\boldsymbol{p}$-Cartesian, i.e., is compatible with the Cartesian structure of functor $\boldsymbol{p}$.

In the context of this definition, the Kleisli construction has an immediate practical interpretation. Arrows in the Kleisli category $\boldsymbol{Mod}_{\mathsf{Q}}$ are shown in Fig. 9(a). They are, in fact, our q-mappings, and we will also denote category $\boldsymbol{Mod}_{\mathsf{Q}}$ by $\boldsymbol{qMap}_{\mathsf{Q}}$(we thus switch attention from the objects of the category to its arrows). It immediately allows us to state (based on Lemma 1) that if $\mathcal{D}$-shaped configurations of models related by ordinary (not q-) model mappings are mergeable, then $\mathcal{D}$-shaped configurations of models and q-mappings are mergeable as well. For example, merge in our running example can be specified as the colimit of the diagram of Kleilsi mappings on p.10.

Metamodel-level components of q-mappings between models are arrows in $\boldsymbol{MMod}_{\mathsf{Q}_{\mathrm{def}}}$, and they are nothing but view definitions: they map elements of the source metamodel to queries against the target one Fig. 9(a). Hence, we may denote $\boldsymbol{MMod}_{\mathsf{Q}_{\mathrm{def}}}$ by $\boldsymbol{viewDef}_{\mathsf{Q}_{\mathrm{def}}}$. View definitions can be executed as shown in Fig. 9(b1): first the query is executed, and then the result is retyped along the mapping v (dashed arrows denote derived mappings).

The resulting operation of *view execution* is specified in Fig. 9(b2), where double arrows denote Kleisli mappings. Properties of the view execution mechanism are specified by Theorem 1 proved in the TR.

Theorem 1. *Let* $(\mathsf{Q}, \mathsf{Q}_{\mathrm{def}})$ *be a monotonic query language over an abstract metamodeling framework* $\boldsymbol{p}$: $\boldsymbol{Mod} \to \boldsymbol{MMod}$*. It gives rise to a split fibration* $\boldsymbol{p}_{\mathsf{Q}}$: $\boldsymbol{qMap}_{\mathsf{Q}} \to \boldsymbol{viewDef}_{\mathsf{Q}_{\mathrm{def}}}$ *between the corresponding Kleisli categories.*

Theorem 1 says that implementing view computation via querying followed by retyping is compositional. More precisely, views implemented via querying followed by retyping can be composed sequentially, and execution of the resulting composite view amounts to sequential composition of executions of its component views. Such compositionality is an evident requirement for any reasonable implementation of views, and views implemented according to our framework satisfy this requirement.

5 Related Work

Modeling inductively generated syntactic structures (term and formula algebras) by monads and Kleisli categories is well known, e.g., [16,17]. Semantic structures (algebras) then appear as Eilenberg-Moore algebras of the monad. In our approach, carriers of algebraic operations stay within the Kleilsi category. It only works for monotonic query languages, but the latter form a large, practically interesting class. (E.g, it is known that Select-Project-Join queries are monotonic.) We are not aware of a similar treatment of query languages in the literature.

Our notion of metamodeling framework is close to *specification frames* in institution theory [18]. Indeed, inverting the projection functor gives us a functor $\boldsymbol{p}_{\mathbf{Q}}^{-1}$: $\boldsymbol{viewDef}^{op}_{\mathbf{Q}_{\mathrm{def}}} \to \mathbf{Cat}$, which may be interpreted in institutional terms as mapping theories into their categories of models, and theory mappings into translation functors. The picture still lacks constraints, but adding them is not too difficult and can be found in [19]. Conversely, there are attempts to add query facilities to institutions via so called *parchments* [20]. Semantics in these attempts is modeled in a far more complex way than in our approach.

In several papers, Guerra et al. developed a systematic approach to intermodeling based on TGG (Triple Graph Grammars), see [1] for references. The query mechanism is somehow encoded in TGG-production rules, but precise relationships between this and our approach remain to be elucidated.

Our paper [9] heavily uses view definitions and views in the context of defining consistency for heterogeneous multimodels, and is actually based on constructs similar to our metamodeling framework. However, the examples therein go one step "down" in the MOF-metamodeling hierarchy in comparison with our examples here, and formalization is not provided. The combination of those structures with structures in our paper makes a two-level metamodeling framework (a fibration over a fibration); studying this structure is left for future work.

6 Conclusion

The central notion of the paper is that of a q-mapping, which maps elements in the source model to queries applied to the target model. We have shown that q-mappings provide a concise and clear specification framework for intermodeling scenarios, in particular, model merge. Composition of q-mappings is not straightforward: it requires free term substitution on the level of query definition (syntax), and actual operation composition on the level of query execution (semantics). To manage the problem, we model both syntax and semantics of a monotonic query language by a Cartesian monad over the fibration of models over their metamodels. Then q-mappings become Kleilsi mappings of the monad, and can be composed. In this way the universe of models and q-mappings gives rise to a category (the Kleisli category of the monad), providing manageable algebraic foundations for specifying intermodeling scenarios.

Acknowledgement. We are grateful for anonymous referees for valuable comments. Financial support was provided with the NECSIS project funded by Automotive Partnership Canada.

References

1. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-modelling: From Theory to Practice. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 376–391. Springer, Heidelberg (2010)
2. Romero, J., Jaen, J., Vallecillo, A.: Realizing correspondences in multi-viewpoint specifications. In: EDOC, pp. 163–172. IEEE Computer Society (2009)
3. Bernstein, P.: Applying model management to classical metadata problems. In: Proc. CIDR 2003, pp. 209–220 (2003)
4. Goguen, J.: A categorical manifesto. Mathematical Structures in Computer Science 1(1), 49–67 (1991)
5. Fiadeiro, J.: Categories for Software Engineering. Springer, Heidelberg (2004)
6. Batory, D.S., Azanza, M., Saraiva, J.: The Objects and Arrows of Computational Design. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 1–20. Springer, Heidelberg (2008)
7. Sabetzadeh, M., Easterbrook, S.M.: View merging in the presence of incompleteness and inconsistency. Requir. Eng. 11(3), 174–193 (2006)
8. Rossini, A., Rutle, A., Lamo, Y., Wolter, U.: A formalisation of the copy-modify-merge approach to version control in mde. J. Log. Algebr. Program. 79(7), 636–658 (2010)
9. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 165–179. Springer, Heidelberg (2011)
10. Diskin, Z.: Model Synchronization: Mappings, Tiles, and Categories. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 92–165. Springer, Heidelberg (2011)
11. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and Kleisli categories. Technical Report GSDLab-TR 2011-10-01, University of Waterloo (2011), `http://gsd.uwaterloo.ca/QMapTR`
12. Ehrig, H., Ehrig, K., Prange, U., Taenzer, G.: Fundamentals of Algebraic Graph Transformation (2006)
13. Barr, M., Wells, C.: Category theory for computing science. PrenticeHall (1995)
14. Jacobs, B.: Categorical logic and type theory. Elsevier Science Publishers (1999)
15. Manes, E.: Algebraic Theories. Springer, Heidelberg (1976)
16. Jüllig, R., Srinivas, Y.V., Liu, J.: Specware: An Advanced Evironment for the Formal Development of Complex Software Systems. In: Nivat, M., Wirsing, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 551–554. Springer, Heidelberg (1996)
17. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
18. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of ACM 39(1), 95–146 (1992)
19. Diskin, Z.: Towards generic formal semantics for consistency of heterogeneous multimodels. Technical Report GSDLAB 2011-02-01, University of Waterloo (2011)
20. Goguen, J., Burstall, R.: A Study in the Foundations of Programming Methodology: Specifications, Institutions, Charters and Parchments. In: Poigné, A., Pitt, D.H., Rydeheard, D.E., Abramsky, S. (eds.) Category Theory and Computer Programming. LNCS, vol. 240, pp. 313–333. Springer, Heidelberg (1986)