McSCert TECHNICAL REPORT

# Mechanics of Megamodeling: Design Patterns and Laws

Zinovy Diskin, Sahar Kokaly, Tom Maibaum

McSCert–TR 2013–03–14 March 2013




McMaster Centre for Software Certification
Information Technology Building 101
1280 Main Street West
Hamilton, Ontario, Canada L8S 4K1

**WWW page: https://www.mcscert.ca/**

# Mechanics of Megamodeling: Design Patterns and Laws

Zinovy Diskin[1,2], Sahar Kokaly[1], Tom Maibaum[1]

[1] NECSIS, McMaster University, Canada
{diskinz|kokalys|maibaum}@mcmaster.ca
[2] University of Waterloo, Canada
zdiskin@gsd.uwaterloo.ca

**Abstract.** A megamodel is a model, whose instances are systems of models and intermodel relationships. To be independent of a particular modeling language, typical megamodels reduce models and their relationships to unstructured nodes and edges, thus creating a significant gap between abstract megamodels and their concrete instances. We address this gap problem by mathematical means borrowed from category theory, and propose *mapping-aware* megamodels, which provide additional internal structure to nodes and edges, yet remain independent of modeling languages.

## 1 Introduction

A *multimodel* is a system of models and the relationships between them. A *megamodel* is a model whose instances are multimodels. Phrased differently, a megamodel is a schema for a class of multimodels, e.g., the one in Fig. 1. Names $A_i$, $A$, $B$ refer to models, and $M_i$, $M$, $N$ to metamodels; colon denotes the 'conformsTo' relationship. Edges refer to different types of intermodel relationships: arc $o$ denotes an overlap between models $A_1$ and $A_2$, arrows $e_i$ are model embeddings, and arrow $t$ specifies model $B$ as the result of transformation $t$ applied to model $A$. (Relationships conform to their respective metamodels, the $R$'s, to be discussed later.) For example, one may think of $A_1$ and $A_2$ as a class and a sequence diagram, $A$ as an UML model including them, and $B$ as Java code generated from $A$. In practical applications, this megamodel would also contain other models $A_i$ and their overlaps in the direction $A_1, A_2$: think of other class, sequence, statechart and other types of diagrams. Moreover, a typical megamodel would also contain models preceding and succeeding those $A_i$ via transformation/refinement chains along the direction $AB$. The need for building and maintaining megamodels within MDE was put forward by Bezivin et al [3,4] and others [17]; more recent applications and considerations can be found in [14].

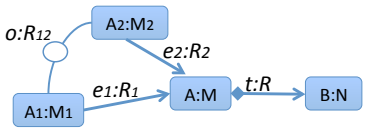


**Fig. 1.** A sample megamodel

Multimodeling is ubiquitous in MDE, and megamodeling should also be. However, it has gained much less attention in theory and practice than one would expect. A possible reason for this is that megamodeling is inherently very abstract; the megamodel designer abstracts away all specialities of models involved, and reduces models and intermodel relations to nodes and edges lacking structural details. Not too much could be said about properties of relations between models and laws of operations over them in such an abstract setting. We will refer to this problem as the megamodeling *abstraction gap*.

A special challenge is to understand and precisely specify model relationships. In the typical metamodels ($R$'s in Fig. 1) one can find in the literature, relationships are defined as objects having a source and a target model, and a type (conformsTo, transformsTo, etc.). That is, all relationships are just edges, whereas their semantics are hidden in the type name and are not presented in the megamodel. We come to the abstraction gap problem again.

In this paper, we address the gap problem with proper mathematical tools borrowed from category theory. We propose a new type of megamodel, *mapping-aware (MA-)megamodel*, which provides additional internal structure to a megamodel's nodes and edges, yet remains abstract and independent of particular modeling languages. We respect the two (diverging!) requirements by zooming into 'conformsTo' statements, i.e., predicate declarations $(A, M) \in \mathsf{conformsTo} \subset \mathsf{Models}\times\mathsf{Metamodels}$, and replacing them by typing mappings $t_A\colon A \to M$, which sends elements of $A$ to their types in $M$. The declaration above does not possess an internal structure: it is nothing more than a labeled pair (we will say *link*) $(A, M)$. In contrast, a typing mapping is a set of labeled links $(a, m) \in t_A \subset A\times M$, which has an internal structure similar to $A$ and $B$ (e.g., if the latter are graphs, then $t_A$ is also a graph consisting of (node,node) and (edge, edge) pairs). Similarly, we replace a labeled link $(A, B) \in \mathsf{transformsTo}_\mathsf{t} \subset M^\bullet\times N^\bullet$ (where $M^\bullet, N^\bullet$ denote classes of models defined by metamodels $M, N$ resp., and $\mathsf{t}$ is the name of the transformations) by a traceability mapping $m_\mathsf{t}\colon A \to B$ consisting of links $(a, b) \in A\times B$ which again has a structure similar to $A$ and $B$. Similarly, we replace a labeled link $(A, A') \in \mathsf{updatesTo} \subset M^\bullet\times M^\bullet$ with a delta specifying the update, and so on. Expanding "flat" links into "deep" mappings moves megamodeling into a richer structural space, in which many important properties of, and operations on megamodels can be accurately specified.

Referring to mechanics in the title of the paper is not happenstance. When we zoom into a megamodel's nodes and edges, we can disassemble them into elementary building blocks. We specify a compact library of such blocks, and show how they can be combined to reconstruct classical megamodeling constructs: conformance, overlapping, consistency, and transformation relationships. Moreover, by combining the same blocks we can build new usable constructs, e.g., bidirectional transformations and heterogeneous merge. In fact, we provide a library of design patterns for multimodel engineering, and moreover, we give our patterns a formal semantics, and identify useful mathematical laws. We also

propose a notational discipline for specifying ma-metamodels and drawing them on the back of the envelope.

In Sections 2 and 3 we introduce mapping-aware megamodels via a series of simple examples, specify our library of elementary building blocks, and demonstrate how they can be combined to produce more complex megamodeling constructs. Section 2 is devoted to basic intermodel relations (megamodeling "statics"), and Section 3 is about basic operations over models and mappings crucial for model transformations and synchronization ("dynamics"). In Section 4 we discuss how to adapt the framework for more general situations. We show that richer mappings and constraints can be packaged nicely into the simple syntax developed in Sections 2 and 3. Section 5 concludes.

## 2 MA-Statics: Models, their Overlap and Consistency

We begin with the two most basic blocks of the framework: a model and a model mapping. Then we build the overlap relation between models using these blocks.

### 2.1 Elementary blocks: Models

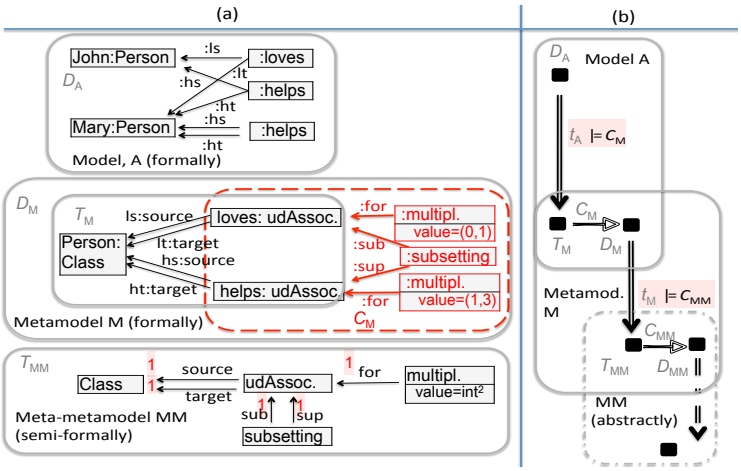


**Fig. 2.** A sample model and its formalization via graphs and graph mappings

Fig. 2(a) presents a simple object model A describing John, Mary and their happy relations. Normally, OIDs would be anonymous, and names would be attributes, but for simplicity, we use names as OIDs. (Some details are discussed in footnotes 3,4 on pages 6,7.) It is easy to see that the model is a legal instance of the metamodel M, in which the triangle-head arrow denotes subsetting.

A typical megamodeling abstraction of this is shown in (c). The diagram specifies a relationship $(A, M) \in \mathsf{conformsTo} \subset \mathsf{Models} \times \mathsf{Metamodels}$ (which we call a *link*) between models. The problem with this specification is an essential

gap between the real model and its abstract description. The compact syntax of UML diagrams hides a multitude of structural connections not shown in (c). Our plan is to zoom into objects in column (a), reveal their structures and relations, and build their ma-megamodel, whose further abstraction would be (c).

We begin with an accurate formalization of the metamodel. The latter consists of types (one class and three unidirectional associations), represented by the *type graph* $T_M$ in column (b1) of Fig. 2 (found in the lower stadium). The three constraints declared in the metamodel are not types but are important elements of the metamodel: they are represented by blank nodes (red with a color display) connected to the types they constrain by dashed (red) arrows. Constraints thus form the *constraint graph*, $C_M$ (dashed-frame in Fig. 2). The intersection graph $T_M \cap C_M \subset T_M$ consists of those types that are in the scope of at least one constraint. Types and constraints together form the data graph $D_M$ of the metamodel (the outer stadium).

Note that every element in $D_M$ should be typed by the respective element of the meta-metamodel, say, $MM$, which is not shown (but omnipresent in $M$): Person is a Class, three loops are (unidirectional) Associations, nodes (1..3) and (0..1) are multiplicities, and the intermediate constraint node is of type Subsetting. We omitted all nodes and arrow types but for Subsetting and its two reference arrows 'sup' and 'sub', which we left for illustration: they are separated from the anonymous element names by colons.

Model $A$ is represented by its data graph $D_A$ (the upper stadium in (b1) and (b2) of Fig. 2), which is typed over the metamodel. Types are shown with curved links (orange with a color display), and form the model's *typing* mapping $t_A\colon D_A \to T_M$: $t_A(1) = \mathsf{likes}$, $t_A(2) = t_A(4) = \mathsf{help}$, etc. Of course, $t_A(\mathsf{Mary}) = t_A(\mathsf{John}) = \mathsf{Person}$, but we omitted these links to avoid cluttering the figure. Note that the pair $(D_A, t_A)$ satisfies all constraints declared in $M$, and we write $t_A \models C_M$. It is an important statement about the model; in fact, it is a part of the model as shown in Fig. 2, where the model frame encompasses its data graph, the metamodel, and the constraint satisfaction statement. Now connections between model $A$ and its metamodel $M$ are accurately specified. Similarly, an accurate specification of metamodel $M$ should include its typing mapping $t_M\colon D_M \to T_{MM}$ and a statement $t_m \models C_{MM}$. We should repeat the same for $MM$ and so on until we get the most basic reference model saying that we live in the world of graphs having nodes and edges (TR provides some details. We refer to the tower in column (a1) by dots. If model $A$ were instantiable, the tower could be extended upwards. The pattern recurrently repeated in the tower is a graph mapping $t_x\colon D_x \to T_y$ from data graphs $D_x$ ($x = A, M, MM$) to the respective type graphs $T_y$ ($y = M, MM, \mathbb{M}$) one level down, such that the constraints are satisfied: $t_x \models C_y$.

Our work in column (b1) is abstractly specified in column (b2). Black circles denote graphs, and arrows refer to mappings between graphs. Note that these mappings are ***correct graph morphisms***: they send nodes to nodes, and arrows to arrows, such that their incidence is preserved. We will say that typing mappings are *structure preserving*. Finally, with one more abstraction step, we

can denote all data embodied into models $A$ and $B$ by nodes $A$ and $B$, and the relationship between them described above by an arrow as shown in Fig. 2(c). In fact, diagram/schema (c) says that the structure referred to by $A$ contains the structure referred to by $M$. The meaning of the relationship is hidden in the label 'confTo' (and described in column (b2)), but what is exactly specified is just a link $(A, M)$ labeled by a type. We denote links by thin arrows with bullet-tails. In contrast, mappings between structures, which themselves are sets of links, are denoted by thick double-body arrows as shown in column (b2).

Thus, a metamodel is a pair of graphs, $M = (T, C)$. It is assigned with two classes of instances: i) those that are legally typed (called *premodels*), $\mathsf{Inst}^{\circ}(M) = \{A \mid t_A : D_A \to T \text{ is a correct graph mapping}\}$ but perhaps do not satisfy the constraints, and ii) those that satisfy all constraints (actual models) $\mathsf{Inst}^{\bullet}(M) = \{A \mid t_A \models C_M\} \subset \mathsf{Inst}^{\circ}(M)$. This distinction is important in formalization of multimodel's consistency via merge (discussed later).

**Pattern 1** *A model is a (total)* typing *mapping from a model's* (instance) data *to a model's* metadata *(types).*
**The Laws.** *Typing must be a correct graph morphism, and all constraints declared in the metamodel are to be satisfied.*

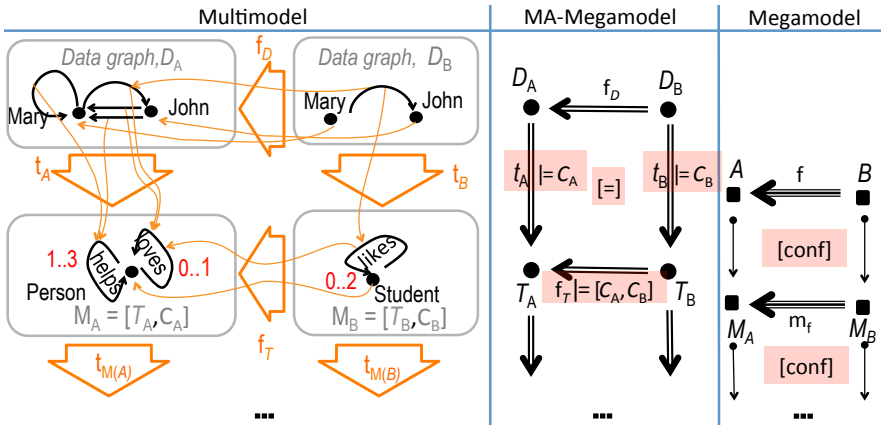


**Fig. 3.** Model mapping

## 2.2 Elementary blocks: Model mappings

Suppose that two modelers built their own models $A$ and $B$ of the same domain, shown in Fig. 3(left cell). Model $A$ is the next step of our story: now Mary and John love each other. Model $B$ describes a particular view of the case, and uses its own names. Since the referent domain is the same, it may happen that elements in $A$ and $B$ refer to the same objects in the domain, and we want to make this correspondence explicit. Suppose we know that John and Mary of model $B$ refer to the same real world persons as John and Mary of model $A$. To specify these facts, we relate the 'same' elements by inter-model links shown curved in the

figure (orange with a color display). We will call them *correspondence* links, or just *corr-links*. However, these links do not respect typing and map Students to Persons. To make corr-linking type-safe, we need to match the respective types and map class Student in $B$ to class Person in $A$. Similarly, if we know that "Mary likes John" is model $B$'s restatement of $A$'s "Mary loves John", we need to match the respective inter-object links, but first we need to match their types and map association 'likes' in $M_B$ to association 'loves' in $M_A$ (where $M_X$ denotes the metamodel of model $X = A, B$).[3]

The result (shown in the middle column) is that our correspondence mapping between models $f\colon A \leftarrow B$ consists of two parts: an instance-data mapping, $f_D\colon D_A \leftarrow D_B$, and a meta-data mapping $f_T\colon T_A \leftarrow T_B$, where we write $T_A$ for $T_{M(A)}$ and similarly for $T_B$. Type-safety can now be stated as *commutativity* of the square diagram: $b.f_D.t_A = b.t_B.f_T$ for any element $b$ of datagraph $D_B$.

Note that all mappings involved are correct graph morphisms: we match classes and objects to classes and objects, associations/links to associations/links, and the incidence of nodes and arrows is preserved. To simplify notation, below we will often only show corr-links relating arrows, links between nodes can be inferred from them. Note also that both mappings $f_{D,T}$ are totally defined but not surjective: the 'help'-side of the domain is modeled in $A$ but is ignored by $B$. We say that $A$ has its *private* part wrt. $B$, whereas $B$ does not have such a part and everything it says can be found in $A$. How to specify intermodel relations when each of the models has its own private part is discussed below in Sect. 2.3.

As constraints are an integral part of models, we need to discuss compatibility of mapping $f_T$ with constraints declared in the metamodels. Speaking syntactically, constraints are a part of the structure, and hence should be preserved as well. This means that if metamodel $M_B$ declares a constraint $\mathsf{c}$ for an arrow $b$ (we write $\mathsf{c}[b]$), then metamodel $M_A$ should declare the same constraint for arrow $a = f_T(b) \in M_A$, i.e., we must have declaration $\mathsf{c}[a]$ in graph $C_A \stackrel{\text{def}}{=} C_{M(A)}$. Alternatively, we must at least have other $C_A$-constraints logically imply $\mathsf{c}[a]$, and hence the latter is implicitly present in $M_A$. This is indeed the case for mapping $f_T$ in Fig. 3 because for multiplicities we have $(0..1)[a] \models (0..2)[a]$ for any arrow $a$. In general, mapping $f_T$ translates any constraint $\mathsf{c}$ over type graph $T_B$ into a constraint $f_T(\mathsf{c})$ over graph $T_A$. Hence, set $C_B$ is translated into set $f_T(C_B)$ of constraints over $T_A$. If $f_T(C_B) \subset C_A$, or at least, $C_A \models f_T(C_B)$, we call mapping $f_T$ *compatible* with the constraints, and write $f_T \models [C_A, C_B]$; we will also say $f_T$ is a metamodel *morphism*, or *legal* metamodel mapping. Compatibility has an important semantic interpretation discussed in Sect. 3.1.

[3] Recall that Student/likes and Person/loves are OIDs rather than attributes, and we map OIDs to OIDs. If they were values of attribute name, we could simply exclude them from the domain of the mapping, as we would exclude other auxiliary (wrt. modeling as such) attributes, e.g., timestamps. On the other hand, if we do want to pay attention to names, then we have a conflict between the models, and their correspondence must be specified by a span of mappings, rather than by a mapping, as will be explained in Sect. 2.3.

The rightmost diagram shows a further abstraction of what we did. All data embodied by models are referred to by nodes $A$, $B$, $M_A$ and $M_B$ with two vertical arrows showing conformance as above. Horizontal model mappings are denoted by triple arrows because they embody data and metadata mappings (the latter also includes a meta-metadata mapping). In addition, the metadata mapping is assumed to be compatible with constraints. In other words, the diagram (c) encapsulates all data specified in (b); this is denoted by label [conf].

**Pattern 2 (Model Mappings)** *A model mapping is a pair of (total)* correspondence *mappings between the respective data and metadata parts of the models. The result is a square of mappings.*
**The Laws.** *To ensure type-safety, the model mapping square is required to be commutative. Moreover, translations of constraints declared for the source of a mapping are to be implied by the constraints in the target: the target is to be at least as constrained, perhaps more constrained, than the source.*

### 2.3 Model overlap and consistency

**2.3.1 Simple overlaps.** Models $A$ and $B$ in Fig. 4 again present two views of the same domain. The views overlap as Mary and John in model $A$ and Mary and Jo in model $B$ correspond, and the love and like links between them do too.

However, we cannot specify this overlap by a totally defined mapping from one model to another because each of them has its own private information: attribute 'age' in $A$ and attribute 'gpa' in $B$. In addition, correspondence links constitute an important part of the megamodel, and we may want to annotate them with auxiliary metadata (e.g., timestamp, authorship). Both issues (totality and annotation) can be managed by *reifying* the correspondence links with a new model $O$(verlap) as shown in the figure. Elements of $O$ could be thought of as pairs of elements $(a, b) \in A \times B$), and total mappings, $f\colon A \leftarrow O$ and $g\colon O \rightarrow B$, as projections identifying the corresponding parts of the components.

A pair of mappings with a common source is called a (binary) *span*, model $O$ is its *head*, projection mappings $f, g$ are *legs*, and their targets $A, B$ are *feet* of the span. There are also $m$-ary spans with $m$ legs and feet.

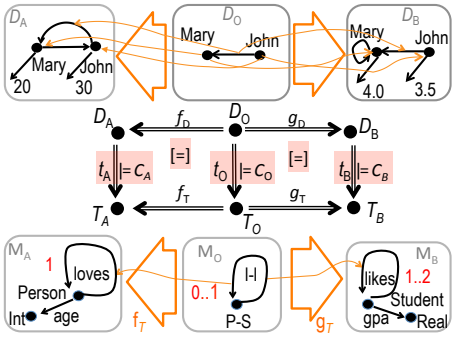


**Fig. 4.** Overlapping models

Note that model $O$ satisfies neither constraints $C_A$ nor $C_B$. Indeed, as model $A$ misses an important fact that Mary loves herself while model $B$ misses that she likes John, neither of these two links occurs in the overlap model $O$. Hence, we need to relax multiplicity in the metamodel $M_O$ to a (0..1) value.[4]This is a general rule: the head

[4] In addition, suppose that both metamodels $M_{A,B}$ specify a mandatory attribute name for Person and Student, and Mary, John, Jo are attribute values rather than

of the overlap span is always less constrained than its feet, and projections are correct model mappings compatible with the constraints.

**2.3.2 Complex overlap via constraints.** In the example above, the overlap model $M_O$ declares only a set of "equations" (John=John, Mary=Mary, etc.) specifying a correspondence between models $A$ and $B$. However, models can interact in a more complex way. An example is shown in Fig. 5, in which associations in the models ('loves' and 'helps', resp.) are different, but are logically related by a constraint $\mathsf{c}_{\mathrm{hl}}$: "if X helps Y, then X loves Y".

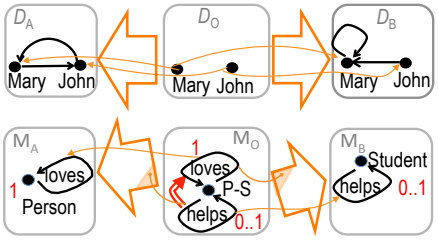


**Fig. 5.** Overlap via constraints

This constraint is declared in the new metamodel $M_O$ and denoted there by a double-body (red) arrow between the associations. Note that constraint $\mathsf{c}_{\mathrm{hl}}$ is an essentially new piece of data, it belongs to neither $M_A$ nor $M_B$ and cannot be declared in either of them. Respectively, projection mappings are partially defined (note links that go into projection arrows and vanish there). We call spans *partial*, and overlaps *complex*. Finally, we show in [6] that specifying overlap of $n$-models may need several $m$-ary (total and partial) spans ($2 \le m \le n$).

**Pattern 3 (Model Overlap)** *Overlap of two models is a span of model mappings. The latter are either total, if overlapping amounts to correspondence equations between elements, or partial, if new constraints are introduced. Overlap of $n$-models is a set of $m$-ary (total and partial) spans with $2 \le m \le n$.*

**2.3.3 Consistency and merge.** The upper part of Fig. 5 shows models $A, B$ and their overlap model $O$. All three models conform to their metamodels, but together they are inconsistent. Indeed, the intermodel constraint $\mathsf{c}_{\mathrm{lh}}$ (subsetting) together with model $B$ imply that Mary loves herself, which is missing from model $A$. Moreover, this fact cannot be added to model $A$ as it would violate the multiplicity 1 in the metamodel. Thus, the models are inconsistent: if model $B$ is faithful (to reality), then Mary does not love John, if model $A$ is faithful, then Mary does not help herself (indeed, John's help should be sufficient).

To make the arguments above, and those in section 2.3.1, more precise, we need some formal details.

A metamodel is a pair of graphs, $M = (T, C)$. It is assigned two classes of instances: i) those that are legally typed (called *premodels*), $\mathsf{Inst}^{\circ}(M) =$

OIDs. In order to specify overlap accurately, we need to include attribute name into the overlap metamodel, and corrs $f(\mathsf{Mary}) = \mathsf{Mary}$, $g(\mathsf{Mary}) = \mathsf{Mary}$ into projection mappings. However, the name of object J-J must be set to an unknown value '?' (a labeled null in the database jargon) with corrs $f(?) = \mathsf{John}$ and $g(?) = \mathsf{Jo}$, which points to a conflict between views. Hence, metamodel $O$ must allow unknown values for shared attributes.

$\{A \mid t_A : D_A \to T$ is a correct graph mapping$\}$, but perhaps do not satisfy the constraints, and ii) those that satisfy all constraints (actual models), $\mathsf{Inst}^\bullet(M) = \{A \mid t_A \models C_M\} \subset \mathsf{Inst}^\circ(M)$. Suppose we have a heterogeneous collection $\mathcal{A}$ of models $\{A_i \mid A_i \in \mathsf{Inst}^\bullet(M_i), i = 1, 2, ..n\}$ with a set of their overlap spans. Let $\biguplus D_\mathcal{A}$ denote the merge of all model datagraphs modulo their overlap, and similarly $\biguplus T_\mathcal{A}$ is the merge of all type graphs. Constraints can also be merged logically: we first translate them to the type graph $\biguplus T_\mathcal{A}$, and then integrate them as logical theories over $\biguplus T_\mathcal{A}$ [15]. We thus obtain a merged set of constraints $\biguplus C_\mathcal{A}$, and a merged metamodel $M_\mathcal{A} = (\biguplus T_\mathcal{A}, \biguplus C_\mathcal{A})$. It can be proven that the merged data can be correctly typed over the merged metadata, and we thus have a premodel $t_\mathcal{A} : \biguplus D_\mathcal{A} \to \biguplus T_\mathcal{A} \in \mathsf{Inst}^\circ(M_\mathcal{A})$. However, as our example shows, this premodel can violate constraints $\biguplus C_\mathcal{A}$, and thus fall outside the set $\mathsf{Inst}^\bullet(M_\mathcal{A})$. Examples and details can be found in [23] for the homogeneous case, and in [6] for the heterogeneous case. We see that consistency of a multimodel is a property of the span specifying their overlap!

**Pattern 3 completed: The Laws.** *The merge of a system of models modulo their overlap span is a correct premodel. However, it can violate inter-model constraints. This is what we call* inconsistency*.*

## 3 MA-Dynamics: Model Transformations

There are two fundamental operations over models: computing a view of a given source, and generating a source from a given view. The roles played by the view models in these scenarios are entirely different: the view is *descriptive* for the former, and *prescriptive* for the latter (cf. analytical vs. synthetic views in [20]). We will consider these operations in Sections 3.1 and 3.2 resp. In Section 3.3 we show that complex model transformations can be seen as combinations of the two operations.

To differentiate between given (*basic*) objects, and those computed with an operation (*derived*), we will use the following formatting (different from the static figures above). Basic models and mappings are shaded, and their nodes and links are solid. Derived models and mappings are blank, and their nodes and links are blank and dashed (and additionally blue with a color display).

### 3.1 Descriptive views

We return to the case described in Fig. 3, but now consider it in a different context (see Fig. 6). We have two metamodels, $M = (T_M, C_M)$ and $N = (T_N, C_N)$, and a mapping $v : T_M \leftarrow T_N$ that describes $N$ as a view of $M$. We want to consider this mapping as a *view definition* in the technical sense, i.e., as a declarative specification that can be executed for any instance of $M$, e.g., model $A$ shown in the figure. The result should be an instance of $N$, model $V = \mathsf{get}_v(A)$ (read "get the view $v$ of $A$"). In contrast to Fig. 3 where model $B$ and mapping $f$ are given, model $V$ and traceability mapping $trace_V$ are to be computed as a database view would be.

The computing procedure works as follows. We take an element $n \in T_N$, find all elements $a$ in $D_A$ whose type is $v(n)$ and copy them to $V$ with type $n$. In detail, if $t_A(a) = v(n)$, we create a copy $a^* \in D_V$ and set $t_V(a^*) = n$ (the figure shows how it works). Thus, all elements in graph $D_A$, whose types belong to the image $v(T_N)$, are copied into graph $D_V$ and respectively retyped. It is easy to see that the graph structure (incidence of nodes and arrows) is preserved as soon as both mappings, $v$ and $t_A$, are structure preserving. This fact is formally proven in category theory, where the operation we have just described is called a *pull-back* (arrow $t_A$ is *pulled-back along* arrow $v$) [1]. We have thus specified a function $\mathsf{get}_v\colon \mathsf{Inst}^\circ(M) \to \mathsf{Inst}^\circ(N)$.

Note that pulling back also produces a traceability mapping $trace_V\colon D_A \leftarrow D_V$ such that the entire square diagram commutes. This means that the pair $(v, trace_V)$ is a (pre)model mapping $\overline{v_A}\colon A \leftarrow V$.

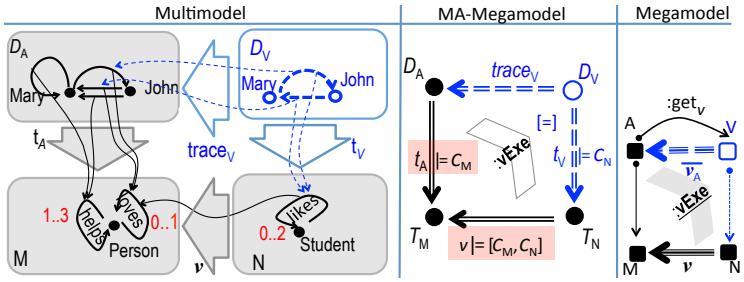


**Fig. 6.** Operation of view computation

Does view $\mathsf{get}_v(A)$ satisfy the constraints $C_N$ declared in metamodel $N$? Suppose $c[x] \in C_N$ is a multiplicity constraint for arrow $x \in T_N$, which is translated into a constraint $v(c) = c[v(x)]$ for arrow $v(x) \in T_M$. If $A \models v(c)$, then $\mathsf{get}_v(A) \models c$ as view computation amounts to copying and retyping of the corresponding part of $D_A$. But, a legal $M$-instance $A \models C_M$, and so if $C_M \models v(c)$ (the case in our example), then $\mathsf{get}_v(A) \models c$ as well. In other words, if the view definition mapping is compatible with the constraints, then pulling-back a legal model $A$ produces a legal view model $V = \mathsf{get}_v(A)$, and we have a total function $\mathsf{get}_v\colon \mathsf{Inst}^\bullet(M) \to \mathsf{Inst}^\bullet(N)$. It gives a semantics for metamodel morphisms, which we discussed in Sect. 2.1.2 purely syntactically.

The middle column in Fig. 6 presents our considerations in an abstract way as the diagram operation of *view execution* (note the chevron labeled :vExe): it takes two solid (black) arrows as its input, and produces two dashed (blue) arrows as its output. The colon in the chevron's label says that we specify an *application instance* of the operation: for another source $A'$ and another view definition $v'$, we would have another instance of vExe and another computed view $V' = \mathsf{get}_{v'}(A')$. Note also that constraint satisfaction pre-conditions for vExe are shaded (with red) while derived post-conditions are not shaded (and blue). Also, not shown in the diagram, but important, is the following fact: if mapping $v$ is injective (a precondition that we normally assume by default), then mapping $trace_V$ is injective too (because pullbacks preserve injectivity [1]).

Finally, the rightmost column presents an even more abstract setting: models are encapsulated as nodes, and model mappings (= commutative squares of graph mappings) as arrows, from which metamodels and their mappings can be projected out. As before, vertical arrows are just links. The shaded chevron denotes an operation abstracted from the blank chevron in the middle column: the latter works with graphs, whereas the former works with models and metamodels. Note that the direction of the operation is diagonally-opposite to the direction of the view definition mapping; for the function $\mathsf{get}_v\colon \mathsf{Inst}^\bullet(M) \to \mathsf{Inst}^\bullet(N)$, this opposition is somewhat striking: the directions of $\mathsf{get}_v$ and $v$ are opposite. Our fine-tuned work with constraints is also embodied in the diagram: if $v$ is a metamodel morphism and $A$ a (legal) model, then $V$ is also a model, and $\overline{V_A}$ is a legal model mapping.

The view $V = \mathsf{get}_v(A)$ possesses a remarkable property: it is a maximal model amongst models that can be mapped to $A$ over $v$, e.g., model $B$ in Fig. 3 is mapped to model $V$ in an evident (and uniquely determined!) way. Some reflection on how the pullback works shows that it is a general property: for any model $B$ and mapping $f\colon A \leftarrow B$ such that $f_T = v$ (think of node $B$ placed to the north-east of node $V$), there is a unique mapping $!_f\colon D_V \leftarrow D_B$ such that both triangles commute: $!_f; t_V = t_B$ and $!_f; \mathit{trace}_V = f_D$. In other words, any mapping $f\colon A \leftarrow B$ factors through $\mathsf{get}_v(A)$ and we have $f = !_f; \overline{v_A}$.

**Pattern 4 (Descriptive views)** *A view definition is a metamodel mapping $v\colon M \leftarrow N$. Its execution goes in the opposite direction: it maps pre-instances of the target metamodel to pre-instances of the source metamodel, and is specified by a function $\mathsf{get}_v\colon \mathsf{Inst}^\circ(M) \to \mathsf{Inst}^\circ(N)$.*
**The Laws.** *(a) Legal instances are mapped to legal instances as soon as the view definition mapping is compatible with constraints declared in the metamodels: $C_M \models v(C_N)$. Then $\mathsf{get}_v$ is a total function $\mathsf{Inst}^\bullet(M) \to \mathsf{Inst}^\bullet(N)$. (b) For any view definition $v$ and model $A$, the view $\mathsf{get}_v(A)$ is maximal amongst models mappable to $A$ over $v$.*

### 3.2 Prescriptive views

In the example above, model $A$ was given and model $V = \mathsf{get}_v(A)$ served a purely descriptive function: to present a view of model $A$, in which 'helps' relations are ignored, and other elements are retyped. In other words, the source $A$ was primary while the view $V$ was secondary. A typical MDE example is when a model is reverse engineered from code (the challenge of this task is to find a proper view definition mapping).

Now consider the opposite situation of code generation: the view model $V$ is given and primary, while the source (code) $A$ is to be built. For example, suppose that Mary wants to achieve (implement) the situation in which John helps her as specified by the "platform-independent" model $V$ (see Fig. 7 left column). For this goal, she is going to use the "platform" of personal relations (specified by metamodel $M$), which satisfies the implementation law "If X loves Y, then X helps Y". This law is specified by a view definition mapping $v\colon M \leftarrow N$ shown

in the figure: If "X loves Y" in some instance $A$ of $M$, then "X helps Y" in the view $\mathsf{get}_v(A)$ according to the algorithm of view execution specified above. Thus, Mary should build a model $A$ over $M$ such that $V = \mathsf{get}_v(A)$. Of course, Mary would be interested in building a minimal $A$ satisfying the requirement, and it is enough to place in $A$ two objects, John and Mary, and 'John-loves-Mary' link between them. This link would implement 'John-helps-Mary' link as shown by mapping $trace_A$ in Fig. 7 (ignore the second link in graph $D_A$ and objects inside the outer square $D_V N M D_A$ for a while). Thus, $A$ can be considered as a (platform-dependent) model generated by $V$ over implementation definition $v$, and we write $A = \mathsf{gen}_v(V)$.

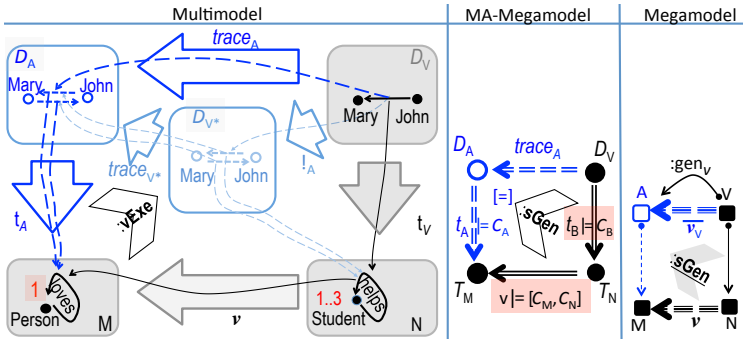


**Fig. 7.** Operation of source generation

This would be the end of the story except for the multiplicity constraint in $M$ requiring every person to love somebody. To satisfy this constraint, Mary must add to model $A$ either a link from herself to John, or a self-loop (or both, but this would violate both the multiplicity 1 and minimality of $A$). Figure 7 shows the case in which Mary chooses 'Mary-loves-John'. (In the model synchronization jargon, such a choice is called a *(synchronization) policy*.) However, now an extra help-link appears in the view $V^* = \mathsf{get}_v(A)$ (note the chevron :vExe, which "computes" view $V^*$), so that Mary needs to help John, which is not supposed by the original view $V$. Thus, the platform of personal relations with its constraint is not suitable for implementing given view $V$ exactly; $V^* \neq V$. Nevertheless, implementation works in a weaker sense: view $V^*$ properly includes $V$ via embedding $!_{trace_A} : V^* \leftarrow V$ ensured by $V^*$'s maximality (it is easy to prove that if $trace_A$ is injective, then $!_{trace_A}$ is injective too). We will refer to this inclusion as the GenGet law, as it specifies a common case: to implement all the necessary requirements, we may need to implement something extra. Importantly, this extra should appear in our computation only once: if we implement $V^*$ and build $A^* = \mathsf{gen}_v(V^*)$, then a reasonable implementation must ensure $A^* = A$ because all implementation details are already reflected in $V^*$. On the other hand, given a source $A$ and its view $V = \mathsf{get}_v(A)$, we should have $V = \mathsf{get}_v(\mathsf{gen}_v(V))$ so that the source and the view are synchronized after, at most, two synchronization steps. We call these conditions the GenGetGen and the GetGenGet laws (see [9]).

The middle column in the figure presents our considerations in an abstract way as a diagram operation of *source generation* (note the :sGen-chevron in the middle): it takes two solid (black) mappings and produces two dashed (blue). The rightmost column is analogous to the rightmost column in Fig. 6, but works in the opposite direction from the view to the source.

**Pattern 5 (Prescriptive views)** *Implementation of an instance of metamodel $N$ within a platform specified by $M$ is an operation opposite to view execution, and unrolls over a view definition mapping $v\colon M \leftarrow N$. Constraints in $M$ may prevent the existence of a unique minimal implementation; then a policy is required to choose one implementation amongst all possible.*
**The Laws.** *Implementation is specified by a function $\mathit{gen}_v\colon \mathsf{Inst}^\bullet(M) \leftarrow \mathsf{Inst}^\bullet(N)$ satisfying GenGet, GenGetGen, and GetGenGet laws.*

### 3.3 Model transformations

Abstractly, a model transformation is a function $\mathsf{t}\colon \mathsf{Inst}^\bullet(M) \to \mathsf{Inst}^\bullet(N)$ sending instances of metamodel $M$ to instances of metamodel $N$. As it should work for all instances, there should be some underlying correspondence between metamodels, $r\colon M \to N$, specifying how types are related. We want to treat $r$ as a declarative definition of $\mathsf{t}$, and computation of $\mathsf{t}(A)$ as the execution of $r$ for the instance $A \in \mathsf{Inst}^\bullet(M)$, in analogy with how we considered view execution and source generation above. However, neither of the metamodels could be considered a view of the other in general. There may be types in $M$ not relevant for the transformation, which we will refer to as *private* for $M$ and its instances, and dually there are private types in $N$. Thus, a more practically applicable case is when a transformation $\mathsf{t}$ is based on a common "shared view" metamodel $S_\mathsf{t}$ in-between $M$ and $N$ (Fig. 8) with view definition mappings $v_\mathsf{t}$ and $w_\mathsf{t}$.

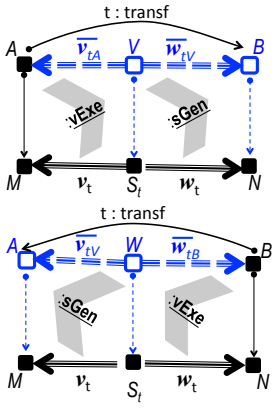


**Fig. 8.** Assembling model transformations

The span of view definitions can be executed for an arbitrary model $A \in \mathsf{Inst}^\bullet(M)$ as shown in the top row of Fig. 8. We first compute the intermediate view $V = \mathsf{get}_{v_\mathsf{t}}(A)$ by treating $v_\mathsf{t}$ descriptively. Then we generate model $B$ from this view by treating $w_\mathsf{t}$ prescriptively. Remarkably, the same span can be executed in the opposite direction as shown in the lower row: at first, mapping $w_\mathsf{t}$ is executed descriptively, then $v_\mathsf{t}$ is executed prescriptively. If both mappings are compatible with constraints, both transformations map legal instances to legal instances. Note also that traceability mappings between models are also spans. Thus, with suitable technological support, the span can be executed in either direction providing bidirectional transformation. This facility is especially effective if both models can be updated, and the changes are to be propagated to the other side in an incremental mode. This scenario is discussed in the TR.

**Pattern 6** *A model transformation definition is a span of metamodel mappings, which can be executed in both directions.*
**The Laws.** *A full set of laws is an open question. Two simple yet basic laws, identity propagation and weak invertibility, are specified in [7].*

### 3.4 Incremental update propagation

We will redefine our view and source computation operations get and gen to work incrementally: now they will take models and their updates on one side (input), and produce updates and updated models on the other side (output).

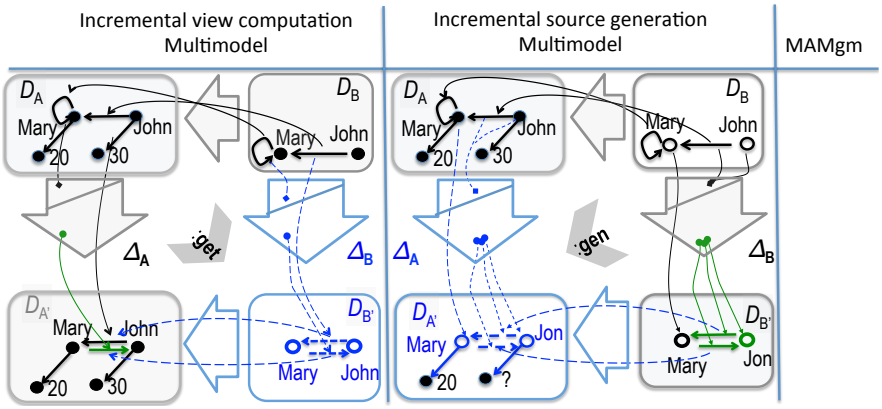


**Fig. 9.** Incremental view computation and source generation

**Incremental view computation.** We continue our love story from Fig. 6. Suppose that Mary finally loves John, so that model $A$ is updated to state $A'$ shown in Fig. 9 (we omit metamodels and typing). Updated model $B'$ could be recomputed from scratch, but it is better to compute it incrementally by propagating the $A$-update to a $B$-update. The $A$-update is specified by partial mapping $\Delta_A$, which shows that Mary's self-loop vanished (mapping is not defined on Mary), and a new link (beyond the range of the mapping) emerged. A link finished at the mapping rather than at $D'_A$, and a link started at the mapping rather than at $D_A$, are just our visualization means for the formal statements that the mapping is not defined for the former, and the mapping's image does not include the latter. These data can be directly propagated to the view side as shown in the figure: mappings make everything explicit and straightforward. Note that propagation also produces an updated traceability mapping.

**Incremental source generation.** The view update mapping $\Delta_B$ in Fig. 9 says that John has left the stage, and a new person Jon together with two new links has emerged. (Note that in both figures, matching objects before and after the update is determined by the update mapping rather than by names.) This update

again straightforwardly propagated to the source side. The only difference from the previous case is that as Jon is a new object, its attribute 'age' is set to an unknown value "?".

**Pattern 7** *Incremental update propagation is specified by tile operations in the plane Correspondences × Updates.*

## 4 Beyond simple examples

We will briefly sketch a mathematical framework, in which simple ma-megamodels introduced above can be given a precise formal meaning. The framework is based on category theory for two reasons. First, any mathematical framework taking mappings seriously is inevitably categorical. Second, although pattern composition is not our main concern in this paper, it is an important feature of the framework, and mathematics of composition is again category theory. Due to space limitations, only a brief sketch is given. Details can be found in our TR.

The presentation in this section is a trade-off between an accurate formulation of mathematical results, and readability of the text by a non-categorical audience. Achieving neither of the goals is quite possible, but we tried to do our best within the page limits. We assume the reader is familiar with basic definitions of a category and a functor; their explanation customized for the MDE audience can be found in [12] (look for the framed boxes Background). Other categorical concepts we use will be informally explained and we provide references.

In the next section we describe a general setting for premodels and typing. In Section 4.2 we argue that intermodel relations very often involve models' derived elements and queries computing them, and incorporate queries into the framework. Section 4.3 is about constraints.

**4.1 Beyond graphs but inside category theory.** MA-scenarios we discussed above have been unraveling in the universe of (directed) graphs and graph mappings. This universe is defined by the metamodel $\boxed{\mathsf{Nodes} \leftleftarrows \mathsf{Arrows}}$, which is itself a graph (by default, the arrows have multiplicity 1). Other graph-like objects can be defined similarly. For example, graphs with arrows between arrows (2-graphs) are given by metamodel $\boxed{\mathsf{Nodes} \leftleftarrows \mathsf{Arrows} \leftleftarrows \text{2-}\mathsf{Arrows}}$, edge-labeled graphs by $\boxed{\mathsf{Nodes} \leftleftarrows \mathsf{Arrows} \to \mathsf{Labels}}$, etc. Any such metamodel also fully defines mappings between its instances: nodes are mapped to nodes, arrows to arrows, labels to labels, etc., so that the incidence is preserved. Thus, if $\mathbb{M}$ is a graph treated as a meta(-meta-...) model, its class of instances $\mathsf{Inst}^\bullet(\mathbb{M})$ comes with mappings between them. We denote this universe of instances and mappings by $\mathbb{G}_\mathbb{M}$, and call its members $\mathbb{M}$-*graphs* and $\mathbb{M}$-*graph mappings*.

Analysis of our arrow diagrams in Sections 2 and 3 shows that similar diagrams can be built for general graph-like structures defined by the framed metamodels above. Indeed, CT-proves that for any graph $\mathbb{M}$, the universe $\mathbb{G}_\mathbb{M}$ is a category called a *presheaf topos* [1], and any presheaf topos is similar enough

to the presheaf topos of ordinary graphs. In particular, $\mathbb{M}$-graphs can be retyped via pullbacks as described in Section 3.1, and Patterns 4,5 with their associated laws are valid for $\mathbb{M}$-graphs (so far, in the constraint-free setting).

As $\mathbb{M}$-graphs are so similar to ordinary graphs, we will often skip index and prefix $\mathbb{M}$, and call them simply graphs.

**4.2 Beyond simple constraints** We begin with a slightly more general formulation of the constraint mechanism considered in Section 2.1. Given a graph $T$ (to be thought of as a type graph), a *constraint (declaration)* over $T$ is a pair $c = (P_c, S_c)$, where $P_c$ is a predicate (like "multiplicity (0..1)" or "subsetting" in Fig. 2), and $S_c$ is a list of $T$'s elements — the scope of the constraint. In the linear notation, we normally encode a constraint by a formula $P_c(S_c)$, e.g., $P(x, y, y)$., and we will aslo often write constraints as $P_c(S_c)$. Each predicate has its predefined arity: a single arrow for any multiplicity, two arrows with a common source for subsetting, two opposite arrows between two nodes for declaring mutual invertibility of two arrows, and so on. Different constraints may use the same predicate, e.g., we may have several multiplicity and several subsetting declarations. On the other hand, the same element in $T$ may occur in the scopes of different constraints like, e.g., loops 'helps' and 'loves' in Fig. 2. A constraint's scope is actually an assignment of $T$'s elements to elements of the arity shape. For example, in the subsetting declaration reproduced in Fig. **??**, the type graph elements, node $n$ and arrows $a, b$, are assigned to the respective elements of the arity shape graph: node 'Subset' and arrows 'sup' and 'sub'. This assignment is a correct graph mapping from the arity shape graph to the type graph.

Thus, constraint declarations are specified by graph mappings and can be reproduced in any universe $\mathbb{G}_\mathbb{M}$. Moreover, it can be shown that semantics of constraints can also be defined via mappings, and hence can be defined in any $\mathbb{G}_\mathbb{M}$ (all that is needed is having pullbacks, and all presheaf toposes do have them) — details can be found in [11,22]. Paper [11] also proves that the law of Pattern 4 holds in the general $\mathbb{G}_\mathbb{M}$-setting, and hence the laws of Pattern 5 remain valid too.

In addition, as any presheaf topos is closed under colimits (CT's way to say "merge"), $\mathbb{M}$-graphs can be merged modulo spans specifying their overlap, and hence our laws for overlap and consistency can be formulated for $\mathbb{G}_\mathbb{M}$ as well.

**4.3 Queries and q-mappings** All mappings we considered so far consisted of links relating a model's element to a model's element. Such simple linking only covers a part of practically interesting cases. Often, an element in a model, say, $B$, is to be linked to a *derived* element in another model, $A$, i.e., an element that is not present in $A$ but can be computed with a relevant query against $A$. For example, in Fig. 4, it may be that class Student of model $B$ corresponds to the class of Persons whose age is less than 30. Or, in Fig. 6, Students of $B$ corresponds to those Persons of $A$ who help at least two other Persons. These derived classes in $A$ can be computed by simple queries, but we may think of correspondences involving complex queries as well. Consider our example of view execution in Fig. 6, where mapping $v$ identifies types Student and Person.

Suppose now that only *young* Persons with age less than 25 can be Students, and inter-Student relation 'likes' is the same as 'love' between young Persons. This new situation is specified by a mapping $v$ in Fig. 10. The target of the mapping is metamodel $M$ augmented with the query specification given by the arrow diamond in metamodel $M^{++}$. In detail: arrow $i$ refers to inclusion of the set of integers less than 25 into the set of all Integers, and the two remaining dash arrows and blank node specify the evident Select-From-Where query $Q$ against class Person, which results in class Person* (of young persons) subsetting Person. For any model $A$ typed over graph $T_M$, this query can be executed and produce an augmented model $A^+ = [\![ Q ]\!](A)$ typed over $M^+ = Q(M)$ as shown in the figure: metamodels $M^{++}$ (shown) and metamodel $M^+$ (not shown) have the same graph but different constraints.

Let us discuss the difference. Note that constraints and queries can interact. Even if any Person must love somebody, it should not necessarily hold for Person*, and hence the multiplicity of the loop 'love*' should be set to (0..1). Thus, metamodel $M^+ = Q(M)$ obtained from metamodel $M$ by adding the query definition would have the derived multiplicity (0..*) for the loop 'loves*. However, the figure shows a stronger multiplicity (1..*) which does not belong to $M^+ = Q(M)$. This constraint is a new datum added to metamodel $M^+$, which makes it a different metamodel $M^{++}$. In detail, $T_{M^{++}} = T_{M^+}$ but $C_{M^{++}} = C_{M^+} \setminus \{C_{0..1}\} \cup \{C_{1..*}\}$. The view definition mapping is compatible with this constraints but incompatible with $C_{0..1}$.

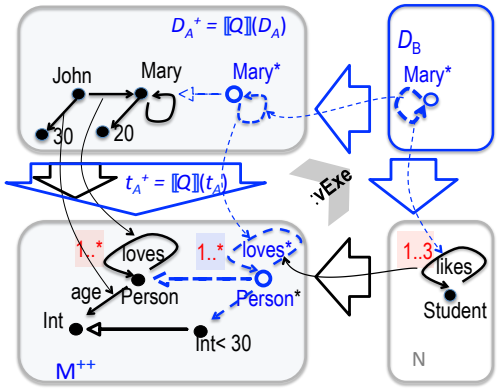


**Fig. 10.** View computation with queries

A view definition involving queries is executed in two steps presented by two diagram operations in Fig. 11(a1). First, the query is executed as explained above (the left tile in the figure). Then the result is retyped as explained in Sect. 3.1 (the right tile). We can encapsulate the two steps within one diagram operation as shown in Fig. 11(a2). An ordinary model mappings is shown in Fig. 11(b). Horizontal arrows now denote *q-mappings*, which consist of *q-links*, i.e., by definition, links relating an element in the source model with a basic or derived element in the target model. Such derived elements are, in fact, queries to the target model, hence letter 'q'. In paper [10] we show that q-mappings can be composed and have other properties making their encapsulation consistent: one can manipulate q-mappings as if they were ordinary mapping. Patterns formulated in Section 3 work for the q-mappings as well.

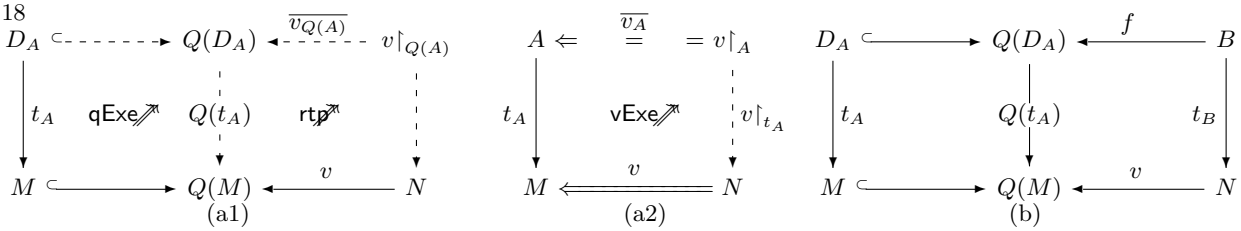


**Fig. 11.** View mechanism (a1, a2) and a general q-mapping (a)

We showed in [5,10] that queries can also be incorporated into the MA-framework. A query language is modeled by a categorical construct called a *monad* [18]. Mappings that send source model elements to queries against the target model are known as *Kleisli mappings*. Thus, view definitions are Kleisli mappings. Kleisli mappings can be composed and form a category (the Kleisli category of the monad). Moreover, since ordinary (typing) mappings can be pulled back over a Kleisli mapping (producing a Kleisli mapping for traceability), our dynamic patterns and laws defined in Section 3 hold for view definitions based on complex queries (if the latter satisfy some *monotonicity* condition [10]; fortunately, widely used Select-Project-Join queries are monotonic).

Also, as Kleisli categories are closed under colimits, our model overlap patterns and laws hold for cases when one or several legs of the overlap span are Kleisli mappings, i.e., involve queries. We conclude that all static and dynamic patterns work for complex mappings involving queries.

A formal framework integrating constraints and queries in the general $\mathbb{G}_{\mathbb{M}}$-setting is described in [5]. It is based on the notion of *fibration* [1], which is, basically, CT's way of saying "view execution", as specified in the rightmost diagram in Fig. 6 (see [8] for details).

**4.4 Summary** An *MA-megamodeling* framework is a triple $\mathcal{F} = [\mathbb{G}, \mathbb{Q}, (\mathbb{C}, \models)]$, where $\mathbb{G}$ is a presheaf topos determined by some graph $\mathbb{M}$— (meta-metamodel), $\mathbb{Q}$ is a cartesian monad on $\mathbb{G}^{\rightarrow}$ providing a query language, and pair $(\mathbb{C}, \models)$ is a constraint language: functor $\mathbb{C}\colon \mathbb{G} \to \mathbf{Set}$ gives its syntax, and a family of relations $\models_T$, $T \in \mathbb{G}$, provides semantics. These data defines a fibration $\boldsymbol{p^{\bullet}_{\mathbb{Q}}}\colon \boldsymbol{Mod^{\bullet}_{\mathbb{Q}}} \to \boldsymbol{MMod^{\bullet}_{\mathbb{Q}}}$, where $\boldsymbol{Mod^{\bullet}_{\mathbb{Q}}}$ and $\boldsymbol{MMod^{\bullet}_{\mathbb{Q}}}$ are two respective Kleisli categories.

## 5 Related work

In different incarnations, megamodeling (Mgm) has appeared in several domains. In databases, it is known as model management[2], focusing on operations on models and model mappings, but not considering megamodels as such. Mathematical foundations are based on the relational data model.

In Algebraic Specifications, institution theory (ITh) [16,19] can be related to Mgm. Metamodels are logical theories, whose classes of models (instances) are categories. View definitions are theory morphisms. Our Pattern 4 law is a basic

postulate in ITh (the Translation Axiom). Typing mappings between models and theories are ignored in ITh, and multilevel constructions (say, metamodel, model, instance) are not considered (see [6] for a more detailed discussion).

A framework directly using models-as-graphs without encoding them into logical theories was developed by the graph-transformation (GT) community [13]. Unlike institutions, typing is fundamental in GT, but the approach is mainly operational via graph-grammars; declarative aspects of megamodeling operations (our view and transformation definition mappings) are typically not considered in GT.

In software engineering, the importance of megamodels for MDE was emphasized by Bezivin *et al* [3,4]. They do not consider megamodels formally, but focus on megamodeling applications and use in MDE. A recent promising technological application for Object/XML mapping can be found in [14]. They focus on the linguistic architecture of software products and develop a megamodeling framework with corresponding language and tool support. Foundations of modeling and, in fact, megamodeling itself, are discussed in [20] in a very general setting, which is too abstract for dealing with patterns and mathematical laws.

Initial elaboration of the ideas presented in the paper can be found in [8] but in a different context of model synchronization. Overlap and consistency are discussed in detail in [5] (with examples related to multimodeling via UML), but compatibility of mappings with constraints is not considered there. Mappings involving queries are elaborated in [10], but model transformations are not considered.

## 6 Conclusion

In this paper we addressed the problem of an abstraction gap between megamodels and their instances (multimodels). We proposed MA-megamodels, which provide additional internal structure to the usual megamodel elements (nodes referring to models, and edges referring to intermodel relationships) yet remain independent of modeling languages.

MA-megamodeling allowed us to disassemble megamodels' nodes and edges into elementary blocks. Then we combined these blocks to restore classical megamodeling constructs: conformance, overlapping, consistency, and transformation relationships. We have also shown that new constructs can be built by combining the same blocks, e.g., bidirectional transformations. In this way, we provided a library of design patterns for megamodel engineering, and outlined a mathematical framework in which these patterns can be provided with formal semantics. Though the full details are not present in the paper, wherein we concentrate on the concepts and demonstration of their adequacy, they can be filled in using standard concepts of category theory.

Going forward, we intend to elaborate these ideas both theoretically and practically. For the latter, we are going to explore the engineering applications of the MA-megamodeling approach to multimodeling within the NECSIS research network — a collaborative project between academia, the automotive industry

(General Motors Canada) and IBM Canada, [21] which focuses on MDE-based design of embedded systems.

## References


1. Barr, M., Wells, C.: Category theory for computing science. Prentice Hall (1995)
2. Bernstein, P., Melnik, S.: Model management 2.0: manipulating richer mappings. In: SIGMOD Conference. pp. 1–12 (2007)
3. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: MDAFA. pp. 33–46 (2004)
4. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2004)
5. Diskin, Z.: Towards generic formal semantics for consistency of heterogeneous multimodels. Tech. Rep. GSDLAB 2011-02-01, University of Waterloo (2011)
6. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: MoDELS Workshops: Selected papers. LNCS, vol. 6627. Springer (2010)
7. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: The symmetric case. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS. Lecture Notes in Computer Science, vol. 6981, pp. 304–318. Springer (2011)
8. Diskin, Z.: Model synchronization: Mappings, tiles, and categories. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE. Lecture Notes in Computer Science, vol. 6491, pp. 92–165. Springer (2009)
9. Diskin, Z.: Lax lenses. Tech. Rep. GSDLab-TR 2013-03-01, University of Waterloo (2013)
10. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and kleisli categories. In: de Lara, J., Zisman, A. (eds.) FASE. Lecture Notes in Computer Science, vol. 7212, pp. 163–177. Springer (2012)
11. Diskin, Z., Wolter, U.: A diagrammatic logic for object-oriented visual modeling. Electr. Notes Theor. Comput. Sci. 203(6), 19–41 (2008)
12. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. Journal of Object Technology 10, 6: 1–25 (2011)
13. Ehrig, H., Ehrig, K., Prange, U., Taenzer, G.: Fundamentals of Algebraic Graph Transformation (2006)
14. Favre, J.M., Lämmel, R., Varanovich, A.: Modeling the linguistic architecture of software products. In: MoDELS. pp. 151–167 (2012)
15. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of ACM 39(1), 95–146 (1992)
16. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of ACM 39(1), 95–146 (1992)
17. Hebig, R., Seibel, A., Giese, H.: On the unification of megamodels. In: Proceedings of the 4th International Workshop on Multi-Paradigm Modeling (MPM 2010). Electronic Communications of the EASST, vol. 42 (2011)
18. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
19. Mossakowski, T., Tarlecki, A.: Heterogeneous logical environments for distributed specifications. In: Corradini, A., Montanari, U. (eds.) WADT. Lecture Notes in Computer Science, vol. 5486, pp. 266–289. Springer (2008)

20. Muller, P.A., Fondement, F., Baudry, B., Combemale, B.: Modeling modeling modeling. Software and System Modeling 11(3), 347–359 (2012)
21. NECSIS, https://www.necsis.ca/: Network for the Engineering of Complex Software-Intensive Systems for Automotive Systems (2011)
22. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A diagrammatic formalisation of mof-based modelling languages. In: Oriol, M., Meyer, B. (eds.) TOOLS (47). Lecture Notes in Business Information Processing, vol. 33, pp. 37–56. Springer (2009)
23. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency checking of conceptual models via model merging. In: RE. pp. 221–230. IEEE (2007)