GSDLAB TECHNICAL REPORT

Towards generic formal semantics for consistency of heterogeneous multimodels

Zinovy Diskin

GSDLAB–TR 2011–02–01 February 2011

Generative Software Development Lab

University of Waterloo

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

Towards generic formal semantics for consistency of heterogeneous multimodels

Zinovy Diskin

Generative Software Development Lab.,
Department of Electrical and Computer Engineering,
University of Waterloo, Canada
`zdiskin@gsd.uwaterloo.ca`

Abstract. Modeling is almost always *multimodeling*: a system is represented by a set of interrelated models, each one capturing a specific *local* view or aspect of the system. Since local models are models of the same system, they implicitly overlap and hence may be *globally* consistent or inconsistent.
Paper [1] pesents a framework for specifying overlaps between local models and defining their global consistency. Its essential feature is that overlap specifications involve models' derived elements defined by respective queries. An advantage of the framework is that heterogeneous consistency checking is reduced to the homogeneous case yet merging partial metamodels into one global metamodel is not needed.
The goal of the present report is to support the framework with a generic formal semantics. Genericness means independency of the framework from specific details of the constraint and query languages employed for model definition and manipulation. It inevitably leads to a formal framework based on category theory; the notions of a monad and fibration will be essential.

1 Introduction

Modeling is almost always *multimodeling*: a system is represented by a set of interrelated models, each one capturing a specific *local* view or aspect of the system. Different views require different modeling means (languages, tools, and intuitions), and their models are often built by different teams that possess the necessary experience and background. This makes modeling of complex systems heterogeneous, collaborative, and distributed.

A fundamental problem of multimodeling is ensuring *global* consistency of the set of local models. Indeed, since local models are models of the same system, they implicitly overlap and hence may be consistent or inconsistent wrt. a set of *global* constraints. Specifying overlaps of heterogeneous models is a crucial issue of multimodeling. This issue is addressed in paper [1].

The key message of [1] is that a multimodel is not just a set of models. A multimodel is a set of *base* models *and* a structure of auxiliary models and model mappings specifying *correspondences* between base models. As examples in [1, Section 3] show, models may overlap in several different ways, and the correspondence structure may thus be a complex network over which models interact ("communicate").

In a nutshell, a heterogeneous multimodel is a pair $(\mathcal{A}, \mathcal{C})$ with $\mathcal{A} = \{A_1{:}M_1..$ $..A_k{:}M_k\}$ a family of base models A_i over their metamodels M_i, and $\mathcal{C} = \{C_1{:}O_1..$ $..C_l{:}O_l\}$ a system of model correspondence spans C_j over a system of (heads of) spans O_j specifying metamodel overlap. In other words, the correspondence part of a multimodel is a network of auxiliary models and mappings in-between models A_i, which resides over the respective network of auxiliary metamodels and mappings in-between metamodels M_i. The two-level structure of the overlap specification is essential: models may overlap only via paths declared in the metamodel schema.

This Technical Report is a mathematical companion to paper [1]. Its goal is to show how the brief description above can be made precise. Section 2 prersents an example of model translation and its semi-formal discussion to guide intuition for more formal subsequent development. Section 3 specifies a very abstract formal framework for model translation, which takes into account neither constraints nor queries and derived elements. Nevertheless, having the translation mechianism established, Section 4 defines global consistency of a heterogeneous multimodel and shows that consistency checking can be indeed realized in the framework of Section 3. Section 5 investigates how the abstract framework of Section 3 can be implemented with constructs close to modeling practice: typed structures, query and constraint languages. This is the most technically demanding part of the report, which needs some more advanced category theory (fibrations and monads). Appendix presents basic definitions about spans, multispans and their colimits.

2 Model translation via arrows and diagrams

This section (taken from [2]) shows that model translation (MT) can be treated as a view computation, whose view definition is given by a corresponding metamodel mapping. An algebraic model of the view mechanism is also discussed.

2.1 MT-semantics and metamodel mappings

The MT-task is formulated as follows. Given two metamodels, $\boldsymbol{\mathcal{S}}$ (the source) and $\boldsymbol{\mathcal{T}}$ (the target), we need to design a procedure translating $\boldsymbol{\mathcal{S}}$-models into $\boldsymbol{\mathcal{T}}$-models. It can be formally specified as a function $\mathbf{f}\colon \mathbf{S} \to \mathbf{T}$ between the spaces of models (instances of the corresponding metamodels). The only role of metamodels in this specification is to define the source and the target spaces, and metamodels are indeed often identified with their model spaces [3–5]. However, a reasonable model translation $\mathbf{f}\colon \mathbf{S} \to \mathbf{T}$ should be compatible with model semantics. The latter is encoded in metamodels, and hence a meaningful translation should be somehow related to a corresponding relationship between the metamodels. A simple case of such a relationship is when we have a mapping

$f\colon \boldsymbol{\mathcal{T}} \to \boldsymbol{\mathcal{S}}$ between the metamodels. Indeed, if we want to translate $\boldsymbol{\mathcal{S}}$-model into $\boldsymbol{\mathcal{T}}$-models, the concepts specified in $\boldsymbol{\mathcal{T}}$ should be somewhere in $\boldsymbol{\mathcal{S}}$. The following example explains how it works.

Suppose that our source models consist of Person objects with attributes qName and phone: the former is complex and composed of a qualifier (Mr or Ms) and a string. The metamodel, $\boldsymbol{\mathcal{S}}$, is specified in the lower left quadrant of Fig. 1. Oval nodes refer to value types. The domain of the attribute 'qName' is a Cartesian product (note the label $\otimes$) with two projections 'name' and 'qual'. The target of the latter is a two-element enumeration modeled as the disjoint union of two singletons. Ignore dashed (blue with a color display) arrow and nodes for a while.

A simple instance of metamodel $\boldsymbol{\mathcal{S}}$ is specified in the upper left quadrant. It shows two Person-objects with names Mr.Lee and Ms.Lee (ignore blue elements again). Types (taken from the metamodel) are specified after colons and give rise to a mapping $t_A\colon A \to \boldsymbol{\mathcal{S}}$.

Another metamodel is specified in the lower right quadrant. Note labels disj and cov "hung" on the inheritance tree: they are diagram predicates (constraints) that require any semantic interpretation of node Actor (i.e., a set $[\![\, Actor \,]\!]$ of Actor-objects) to be exactly the disjoint union of sets $[\![\, Male \,]\!]$ and $[\![\, Female \,]\!]$.

We want to translate Person-models ($\boldsymbol{\mathcal{S}}$-instances) into Actor-models ($\boldsymbol{\mathcal{T}}$-instances). This intention makes sense if $\boldsymbol{\mathcal{T}}$-concepts are somehow "hidden" amongst $\boldsymbol{\mathcal{S}}$-concepts. For example, we may assume that Actor and Person refer to the same class in the real world.

The situation with Actor-concepts Male and Female is not so simple: they are not present in the Person-metamodel. However, although these concepts are not immediately specified in $\boldsymbol{\mathcal{S}}$, they can be *derived* from other $\boldsymbol{\mathcal{S}}$-concepts. We first derive new attributes /name and /qual by sequential arrow composition (see Fig. 1 with derived elements shown with dashed thin lines and with names prefixed by slash — a UML notation). Then, by the evident select-queries, we form two derived subclasses of class Person: mrPerson and msPerson.

Note that these two subclasses together with class Person satisfy the constraints disj, cov discussed above for metamodel $\boldsymbol{\mathcal{T}}$. It can be formally proved by first noting that enumeration {Mr,Mrs} is disjointly composed of singletons {Mr}, {Mrs}, and then using the property of Select queries (in fact, pullbacks) to preserve disjoint covering. That is, given (i) query specifications defining classes mrPesron, mrsPerson, and (ii) predicate declarations disj, cov for the triple ({Mr,Mrs},{Mr},{Mrs}), the same declarations for the triple (Person, mrPerson, mrsPerson) can be logically derived.

The process described above gives us an augmentation $Q[\boldsymbol{\mathcal{S}}] \supset \boldsymbol{\mathcal{S}}$ of the Person-metamodel $\boldsymbol{\mathcal{S}}$ with derived elements, where Q refers to the set of queries involved. Now we can relate Actor concepts Male and Female to derived Person-concepts mrPerson and mrsPerson. Formally, we set a total mapping $\boldsymbol{v}\colon \boldsymbol{\mathcal{T}} \to Q[\boldsymbol{\mathcal{S}}]$ that maps every $\boldsymbol{\mathcal{T}}$-element to a corresponding $Q[\boldsymbol{\mathcal{S}}]$-element. In Fig. 1, links constituting the mapping are shown by thin curly arrows. The mapping satisfies two important requirements: (a) the structure of the metamodels (incidence of

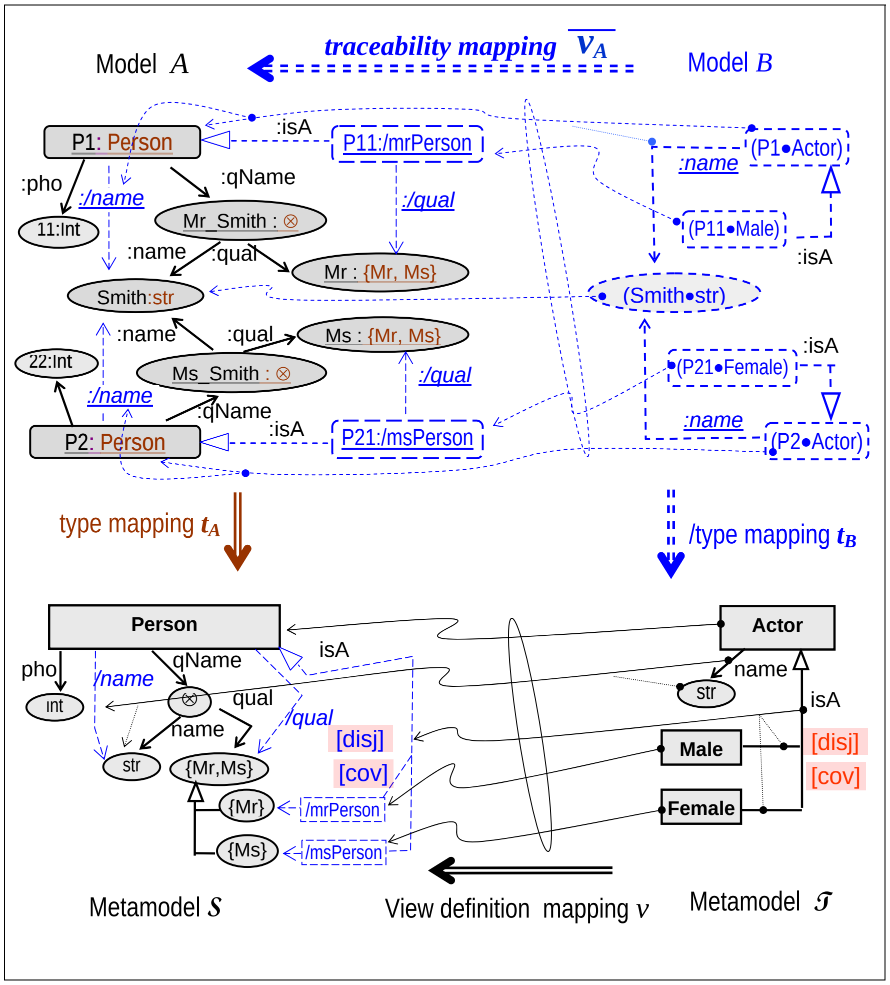


Fig. 1. Semantics of model translation via a metamodel mapping

nodes and arrows, and the isA-hierarchy) is preserved; and (b) the constraints in metamodel $\boldsymbol{\mathcal{T}}$ are respected (disj, cov-configuration in $\boldsymbol{\mathcal{T}}$ is mapped to disj,cov-configuration in $\boldsymbol{\mathcal{S}}$).

2.2 MT via tile algebra

This section shows that data specified above are sufficient to automatically translate any $\boldsymbol{\mathcal{S}}$-model into a $\boldsymbol{\mathcal{T}}$-model by applying to them two diagram operations. Since these operations are based on the square shape, they will be called *tile* operations; more about tile algebra is in [2].

1) Query execution. Query specifications used in augmenting $\boldsymbol{\mathcal{S}}$ with derived elements can be executed for $\boldsymbol{\mathcal{S}}$-models. For example, given a model A, each pair of A's arrows typed with :qName and :name produces a composed arrow typed with :/name (see the left upper quadrant in Fig. 1), and similarly any pair of some model's arrows :qName and :qual produces an arrow :/qual (these are not shown in the figure to avoid clutter). Then each object typed by :Person and having the value Mr along the arrow :/qual, is cloned and typed :/mrPerson.[1] The result is that the initial typing mapping $t_A\colon A \to \boldsymbol{\mathcal{S}}$ is extended to typing mapping $t_{Q[A]}\colon Q[A] \to \boldsymbol{Q}[\boldsymbol{\mathcal{S}}]$, in which $Q[A]$ and $\boldsymbol{Q}[\boldsymbol{\mathcal{S}}]$ denote augmentations of the model and the metamodel with derived elements.

Remark 1 (On notation). Note that a model is a pair $A = (D_A, t_a)$ with D_A its carrier graph and $t_A\colon D_A \to S$ its typing mapping. To easy notation (but abusing it), we will write A for D_A too. Note also that bold $Q[\boldsymbol{\mathcal{S}}]$ is a query *specification* while $Q[A]$ is provided by the query execution.

The extended typing mapping $t_{Q[A]}$ is again structure preserving. Moreover, it is a *conservative extension* of mapping t_A in the sense that (a) types of elements in A are not changed by $t_{Q[A]}$, and (b) each derived elements (from $Q[A]\setminus A$) has a new type (from $Q[\boldsymbol{\mathcal{S}}] \setminus \boldsymbol{\mathcal{S}}$). Formally, the inverse image of submodel $\boldsymbol{\mathcal{S}} \subset \boldsymbol{Q}[\boldsymbol{\mathcal{S}}]$ wrt. the mapping $t_{Q[A]}$ equals to A, and restriction of $t_{Q[A]}$ to A is again t_A.

The configuration we obtained is specified by the left square diagram in Fig. 2(a). Framed nodes and solid arrows denote the input for the operation of query execution, dashed arrows and non-framed nodes denote the result. Label qExe means that the entire square is produced by the operation; the names of arrows and nodes explicitly refer to query $\boldsymbol{Q}[]$ (whereas q is part of the label, not a separate name).

2) Retyping. The pair of mappings,

typing $t_{Q[A]}\colon Q[A] \to Q[\boldsymbol{\mathcal{S}}]$, and view $\boldsymbol{Q}[\boldsymbol{\mathcal{S}}] \xleftarrow{\boldsymbol{v}} \boldsymbol{\mathcal{T}}$,

[1] With a common semantics for inheritance, we should assign the new type label /mrPerson to the same object P1. To avoid multi-valued typing, inheritance is straightforwardly formalized by cloning the objects. In fact, such cloned objects are roles played by "real" objects.

$$\begin{tikzcd}
\boxed{A} \arrow[r, hook, dashed, "\overline{\mathsf{i}^Q_A}"] \arrow[d, "t_A"'] \arrow[dr, phantom, ":\mathsf{qExe}\Rrightarrow"] & Q[A] \arrow[d, dashed, "t_{Q[A]}"] \arrow[dr, phantom, ":\mathsf{pb}\Rrightarrow"] & B \arrow[l, dashed, "\overline{\boldsymbol{v}_{Q[A]}}"'] \arrow[d, dashed, "t_B"] \\
\boxed{\boldsymbol{\mathcal{S}}} \arrow[r, hook, "\mathsf{i}^Q"] & \boxed{Q[\boldsymbol{\mathcal{S}}]} & \boxed{\boldsymbol{\mathcal{T}}} \arrow[l, "\boldsymbol{v}"']
\end{tikzcd}$$

(a)

$$\Rightarrow$$

$$\begin{tikzcd}
\boxed{A} \arrow[d, "t_A"'] \arrow[dr, phantom, ":\mathsf{vExe}\Rrightarrow"] & A\!\upharpoonright_{\boldsymbol{v}} \arrow[l, Leftarrow, dashed, "\overline{\boldsymbol{v}_A}"'] \arrow[d, dashed, "\overline{t_A}"] \\
\boxed{\boldsymbol{\mathcal{S}}} & \boxed{\boldsymbol{\mathcal{T}}} \arrow[l, Leftarrow, "\boldsymbol{v}"']
\end{tikzcd}$$

(b)

$$\Rightarrow$$

$$\begin{tikzcd}
\boxed{\mathrm{A}{:}\boldsymbol{\mathcal{S}}} \arrow[d, "\bullet"'] \arrow[dr, phantom, ":\mathsf{vExe}\Rrightarrow"] & A\!\upharpoonright_{\boldsymbol{v}}{:}\boldsymbol{\mathcal{T}} \arrow[l, Lleftarrow, "\overline{\boldsymbol{v}_A}{:}\boldsymbol{v}"'] \arrow[d, dashed, "\bullet"] \\
\boxed{\boldsymbol{\mathcal{S}}} & \boxed{\boldsymbol{\mathcal{T}}} \arrow[l, Leftarrow, "\boldsymbol{v}"']
\end{tikzcd}$$

(c)

Fig. 2. Model translation via tile operations (the upper arrow in diagram (c) is derived and must be dashed but the Diagram software does not draw dashed triple arrows).

provide enough information for translating model $Q[A]$ into $\boldsymbol{\mathcal{T}}$-metamodel. All that we need to do is to assign to elements of $Q[A]$ new types according to the view mapping: if an element $e \in Q[A]$ has type $X = t_{Q[A]}(e) \in Q[\boldsymbol{\mathcal{S}}]$ and $X = \boldsymbol{v}(Y)$ for some type $Y \in \boldsymbol{\mathcal{T}}$, we set the new type of e to be Y. For example, since $Q[A]$-element $P11$ in Fig. 1 has type mrPerson, which (according to the view mapping $\boldsymbol{v}$) corresponds to type Male in $\boldsymbol{\mathcal{T}}$, this elements must be translated into an instance of type Male; we denote it by $(P11 \bullet Male)$. If no such $\boldsymbol{\mathcal{T}}$-type Y exists, the element e is not translated and lost by the translation procedure (e.g., phones of Person-objects). Indeed, non-existence of Y means that the X-concept of metamodel $\boldsymbol{\mathcal{S}}$ is beyond the view defined by mapping $\boldsymbol{v}$ and hence all X-instances are to be excluded from $\boldsymbol{v}$-views.

Thus, translation is just retyping of some of $Q[A]$-elements by $\boldsymbol{\mathcal{T}}$-types, and hence elements of the translated model B are, in fact, pairs $(e, Y) \in Q[A]\times\boldsymbol{\mathcal{T}}$ such that $t_{Q[A]}(e) = \boldsymbol{v}(Y)$. In Fig. 1, such pairs are denoted by a bullet between the components, e.g., P1•Actor is a pair (P1,Actor) *etc.* If we now replace bullets by colons, we come to the usual notation for typing mappings. The result is that elements of the original model are retyped by the target metamodel according to the view mapping, and if B denotes the result of translation, we may write

$$B \cong \big\{(e, Y) \in Q[A]\times\boldsymbol{\mathcal{T}}\colon\ t_{Q[A]}(e) = \boldsymbol{v}(Y)\big\} \tag{1}$$

We use isomorphism rather than equality because elements of B should be objects and links rather than pairs of elements. Indeed, the translator should create a new OId for each pair appearing in the right part of (1).

First components of pairs specified in (1) give us a traceability mapping $\overline{\boldsymbol{v}_A}\colon B \to A$ as shown in Fig. 1. Second components provide typing mapping $t_B\colon B \to \boldsymbol{\mathcal{T}}$. The entire retyping procedure thus appears as a diagram operation specified by the right square in Fig. 2(a): the input of the operation is a pair of mappings $(t_{Q[A]}, \boldsymbol{v})$, and the output is another pair $(\overline{\boldsymbol{v}_{Q[A]}}, t_B)$. The square is labeled pb because equation (1) specifies nothing but an instance of pullback operation.

Remark 2. If view $\boldsymbol{v}$ maps two different $\boldsymbol{\mathcal{T}}$-types $Y_1 \neq Y_2$ to the same $\boldsymbol{\mathcal{S}}$-type X, each element $e \in Q[A]$ of type X will gives us two pairs (e, Y_1) and (e, Y_2) satisfying the condition above and hence translation to $\boldsymbol{\mathcal{T}}$ would duplicate e. However, this duplication is reasonable rather than pathological: equality $\boldsymbol{v}(Y_1) =$

$\boldsymbol{v}(Y_2) = X$ means that in the language of $\boldsymbol{\mathcal{T}}$ the type X simultaneously plays two roles (those described by types Y_1 and Y_2) and hence each X-instance in $Q[A]$ must be duplicated in the translation. Further examples of how specification (1) works can be found in [6]. They show that the pullback operation is surprisingly "smart" and provides an adequate and predictive model of retyping.

Moreover, Since the construct of inverse image is also nothing but a special case of pullback, the postcondition for operation qExe stating that $t_{Q[A]}$ is a conservative extension can be formulated by saying that the square qExe is a pullback too. To be precise, if we apply pullback to the pair $(\mathsf{i}_A^Q, t_{Q[A]})$, we get the initial mapping t_A.

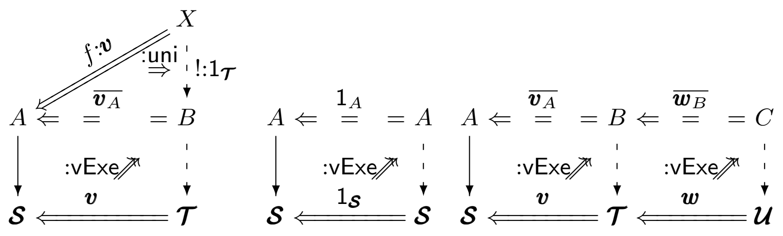
Constraints do matter. To ensure that view model B is a legal instance of the target metamodel $\boldsymbol{\mathcal{T}}$, view definition mapping $\boldsymbol{v}$ must be compatible with constraints declared in the metamodels. In our example in Fig. 1, the inheritance tree in the domain of $\boldsymbol{v}$ has two constraints disj,cov attached. Mapping $\boldsymbol{v}$ respects these constraints because it maps this tree into a tree (in metamodel $\boldsymbol{\mathcal{S}}$) that has the same constraints attached. Augmentation of model A with derived elements satisfies the constraints, $A \models \mathsf{disj} \wedge \mathsf{cov}$, because query execution (semantics) and constraint derivation machinery (pure logic, syntax) work in concert (the completeness theorem for the first order logic). Relabeling does nothing essential and model B satisfies the original constraint in $\boldsymbol{\mathcal{T}}$ as well (details can be found in [7]).

Arrow encapsulation. Query execution followed by retyping gives us the operation of view execution shown in Fig. 2(b). In the tile language, the outer tile vExe is the horizontal composition of tiles qExe and pb. Note that queries are "hidden" (encapsulated) within double arrows: their formal targets are ordinary models but in the detailed elementwise view their targets are models augmented with derived elements.

Diagram (c) present the operation in an even more encapsulated way. The top triple arrow denotes the entire diagram (b): the source and target nodes are models together with their typing mappings, and the arrow itself is the pair of mappings $(\boldsymbol{v}, \overline{\boldsymbol{v}_A})$. Although the source and the target of the triple arrow are typing mappings, we will follow a common practice and denote them by pairs (model:metamodel), e.g., $A{:}\boldsymbol{\mathcal{S}}$, leaving typing mappings implicit. Two vertical arrows are links, i.e., pairs $(A, \boldsymbol{\mathcal{S}})$, $(B, \boldsymbol{\mathcal{T}})$; a similar link from the top arrow to the bottom one is skipped. Note that diagram (c) actually presents a diagram operation: having a metamodel mapping $\boldsymbol{\mathcal{S}} \overset{\boldsymbol{v}}{\Longleftarrow} \boldsymbol{\mathcal{T}}$ and a model $A{:}\boldsymbol{\mathcal{S}}$, view execution produces a model $A{\restriction_{\boldsymbol{v}}}{:}\boldsymbol{\mathcal{T}}$ along with a traceability mapping (triple arrow) $\overline{\boldsymbol{v}_A}{:}\boldsymbol{v}$ encoding the entire diagram Fig. 2(b). An abstract formulation of this construction is called fibration; it will be central later in Sect. 5.

2.3 Properties of the view execution operation

The view mechanism has three remarkable algebraic properties.

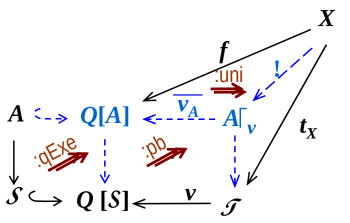


$$\begin{tikzcd}
 & & X \arrow[dll, Rightarrow, "f:\boldsymbol{v}"'] \arrow[d, dashed, "!:1_{\mathcal{T}}"] \arrow[dl, phantom, ":\mathsf{uni} \Rightarrow"] \\
A \arrow[d] & & B \arrow[ll, Rightarrow, "\overline{\boldsymbol{v}_A}"'] \arrow[d, dashed] \arrow[dll, phantom, ":\mathsf{vExe}"] \\
\boldsymbol{\mathcal{S}} & & \boldsymbol{\mathcal{T}} \arrow[ll, Rightarrow, "\boldsymbol{v}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[d] & & A \arrow[ll, Rightarrow, "1_A"'] \arrow[d, dashed] \arrow[dll, phantom, ":\mathsf{vExe}"] \\
\boldsymbol{\mathcal{S}} & & \boldsymbol{\mathcal{S}} \arrow[ll, Rightarrow, "1_{\boldsymbol{\mathcal{S}}}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[d] & & B \arrow[ll, Rightarrow, "\overline{\boldsymbol{v}_A}"'] \arrow[d, dashed] \arrow[dll, phantom, ":\mathsf{vExe}"] & & C \arrow[ll, Rightarrow, "\overline{\boldsymbol{w}_B}"'] \arrow[d, dashed] \arrow[dll, phantom, ":\mathsf{vExe}"] \\
\boldsymbol{\mathcal{S}} & & \boldsymbol{\mathcal{T}} \arrow[ll, Rightarrow, "\boldsymbol{v}"'] & & \boldsymbol{\mathcal{U}} \arrow[ll, Rightarrow, "\boldsymbol{w}"']
\end{tikzcd}$$

(b1) ExeUni (b2) IdExe (b3) ExeExe

Fig. 3. Laws of the view execution mechanism

1) Universality. Suppose we have a model X and a mapping $Q[A] \xleftarrow{f} X$ that maps some of X's elements to derived rather than basic elements of A as shown in Fig. 4.



$$\begin{tikzcd}
 & & & X \arrow[dll, "f"'] \arrow[dl, dashed, "!"] \arrow[ddl, "t_X"] \\
A \arrow[r, dashed, hook] \arrow[d] \arrow[dr, phantom, ":\mathsf{qExe}"] & Q[A] \arrow[d, dashed] \arrow[dr, phantom, ":\mathsf{pb}"] & A{\restriction_v} \arrow[l, dashed, "\overline{v_A}"'] \arrow[d, dashed] \arrow[ur, phantom, ":\mathsf{uni}"] & \\
\mathcal{S} \arrow[r, hook] & Q\,[\mathcal{S}] & \mathcal{T} \arrow[l, "v"'] &
\end{tikzcd}$$

Fig. 4. Universal property of view computation

The mapping must be compatible with typing so that the outer right square with vertex X is required to be commutative: $f;Q[t_A] = t_X;\boldsymbol{v}$. Then owing to the universal properties of pullbacks, there is a uniquely defined mapping $A{\restriction_{\boldsymbol{v}}} \xleftarrow{!} X$ such that the triangles commute (note that mapping ! is a homogeneous model mapping over identity $1_{\boldsymbol{\mathcal{T}}}\colon \boldsymbol{\mathcal{T}} \to \boldsymbol{\mathcal{T}}$).

By encapsulating queries, i.e., hiding them inside double-arrows (see transition from diagram (a) to (b) in Fig. 2), we can formulate the property as shown in Fig. 3(b1), where arrows $f{:}\boldsymbol{v}$ and $!{:}1_{\boldsymbol{\mathcal{T}}}$ actually denote square diagrams whose vertical arrows are typing mappings and bottom arrows are pointed after semicolon. In the categorical jargon, it means that mapping $\overline{\boldsymbol{v}_A}$ is *weakly Cartesian*.

2) Unitality. If a view definition is given by the identity mapping, view execution is identity as well, as shown by diagram Fig. 3(b2)

3) Compositionality. Suppose we have a pair of composable metamodel mappings $\boldsymbol{v}\colon \boldsymbol{\mathcal{T}} \Rightarrow \boldsymbol{\mathcal{S}}$ and $\boldsymbol{w}\colon \boldsymbol{\mathcal{U}} \to \boldsymbol{\mathcal{T}}$, which defines $\boldsymbol{\mathcal{U}}$ as a view of a view of $\boldsymbol{\mathcal{S}}$. Execution of a composed view is normally composed from the execution of components so that for any $\boldsymbol{\mathcal{S}}$-model A we should have

$$\overline{\boldsymbol{v};\boldsymbol{w}_A} = \overline{\boldsymbol{w}_B}\,;\,\overline{\boldsymbol{v}_A} \text{ with } B \text{ standing for } A{\restriction_{\boldsymbol{v}}}$$

as shown in Fig. 3(b3).

3 Abstract multimodeling framework

Building the definition. An *abstract multimodeling framework* $\mathcal{F}_{\rm abstr}$ is a tuple of constructs defined below.

1) A category ***MMod*** whose objects are called *metamodels* and arrows are *metamodel mappings*.

2) Each metamodel M is assigned with two categories, one being a subcategory of the other, $[\![M]\!] \subset [\![M]\!]^?$. Intuitively, objects of $[\![M]\!]^?$ are structures properly typed over M but perhaps violating M's constraints (hence the question mark); we will call them *structural instances*. Objects of $[\![M]\!]$ are *(legal) models*: structural instances of M satisfying, in addition, all constraints in M.

We require all categories $[\![M]\!]^?$ to be closed under colimits (merging). This is the case for many classes of structures carrying metamodels and models like graphs or attributed graphs. But we do not require this property for $[\![M]\!]$. Examples in [1] show that in practically interesting situations $[\![M]\!]$ is *not* closed under colimits.

3) Any metamodel mapping $m\colon M \to N{::}\boldsymbol{MMod}$ is assigned with a *getView* functor $\mathsf{get}^?_m\colon [\![N]\!]^? \to [\![M]\!]^?$ that maps in the opposite direction (think of m as a view definition and $\mathsf{get}^?_m$ as its execution). Moreover, functor $\mathsf{get}^?_m$ is required to map legal N-models to legal M-models because we implicitly assume that mapping m is compatible with constraints declared in the metamodels. Below in Section 4 we will see how it works. Thus, the restriction of $\mathsf{get}^?_m$ to the category of models $[\![N]\!] \subset [\![N]\!]^?$ is a functor $\mathsf{get}_m\colon [\![N]\!] \to [\![M]\!]$

Moreover, if $m = \mathbf{1}_M$ is the identity mapping of metamodel M, then both $\mathsf{get}^?_m$ and get_m are equal to identity functors on $[\![M]\!]^?$ and $[\![M]\!]$ resp, and for two consecutive mappings $M \xrightarrow{m_1} N \xrightarrow{m_2} O$,

$$\mathsf{get}_{m_1;m_2} = \mathsf{get}_{m_2};\mathsf{get}_{m_1}\colon [\![O]\!] \to [\![M]\!]$$

(a sequentially composed view definition is executed consecutively). The same condition holds for $\mathsf{get}^?$. To be precise, we must require natural isomorphism rather than equality in the equation above: our formal framework assumes that the result of view computation is an ordinary model like other models (the view is "materialized"), and hence its elements have arbitrarily chosen object identifiers. Then the result of view computation is defined up to isomorphism, and composition is also preserved up to isomorphism.

4) Any injective mapping $\mathsf{i}\colon M \to N$ is assigned with a *retyping* functor $\mathsf{rtp}^?_\mathsf{i}\colon [\![M]\!]^? \to [\![N]\!]^?$ (think of retyping described in [1, Section 3.2]). Note that in contrast to operation get, $\mathsf{rtp}^?$ maps structural instances (particularly, models) to structural instances (not necessarily models): if even an instance A is an M-model, we cannot guarantee that $\mathsf{rtp}^?_\mathsf{i}(A)$ would satisfy all constraints in N. However,we require that for any i, functor $\mathsf{rtp}^?_i$ be the left adjoint to $\mathsf{get}^?_i$.

Similarly to get, we require $\mathsf{rtp}^?_{\mathbf{1}_M}$ to be the identity functor on $[\![M]\!]^?$, and for two consecutive mappings m_1, m_2 as above,

$$\mathsf{rtp}^?_{m_1;m_2} = \mathsf{rtp}^?_{m_1};\mathsf{rtp}^?_{m_2}\colon [\![M]\!]^? \to [\![O]\!]^?.$$

We will write an abstract multimodeling framework in a short form as a tuple $\mathcal{F}_{\mathrm{abstr}} = (\boldsymbol{MMod}, \mathsf{get}^?, \mathsf{get}, \mathsf{rtp}^?)$ assuming that the $[\![\_]\!]$-part of the construction is "included" into get, and the $[\![\_]\!]^?$ part is "included" into ($\mathsf{get}^?$,$\mathsf{rtp}^?$). That is, for an object (metamodel) $M \in \boldsymbol{MMod}_0$, we set $[\![M]\!] = \mathsf{get}_0(M)$ and $[\![M]\!]^? = \mathsf{get}^?_0(M) = \mathsf{rtp}^?_0(M)$, which will make mappings $\mathsf{get}^?$,get,$\mathsf{rtp}^?$ functors from $\boldsymbol{MMod}$ into "the" category of all categories $\mathbf{Cats}$.

Partial metamodel mappings. A *partial mapping* $m\colon M \rightharpoonup N$ between metamodels (note the harpoon-like arrow head) is, formally, a diagram $M \xleftarrow{\mathsf{i}_m} D_m \xrightarrow{f_m} N$ with $D_m \subset M$ a metamodel called the *domain* of m (while M is the *source* of m), i_m is the corresponding inclusion, and f_m is an ordinary (total) metamodel mapping (the *function* of m).

Evidently, sequential composition $\mathsf{rtp}^?_{\mathsf{i}_m} \circ \mathsf{get}^?_{f_m}$ provides a functor $[\![M]\!]^? \leftarrow [\![N]\!]^?$ translating N's structural instances and their mappings into M's ones. Thus, operations $\mathsf{get}^?$ and $\mathsf{rtp}^?$ together provide model translation over partial mappings. By some abuse of notation, we will denote this composition by $\mathsf{get}^?_m$ (so that the actual meaning of $\mathsf{get}^?_m$ depends on whether m is a total or a partial mapping).

4 Multimodels and their consistency

Let $\mathcal{F}_{\mathrm{abstr}} = (\boldsymbol{MMod}, \mathsf{get}, \mathsf{rtp}^?)$ be an abstract multimodeling framework.

Homogeneous multimodels and their consistency. A *homogeneous multimodel* over $\mathcal{F}_{\mathrm{abstr}}$ is a pair $(M, \mathcal{A})$ with $M \in \boldsymbol{MMod}$ a metamodel and $\mathcal{A}$ a diagram in $[\![M]\!]$. Below, we will often write such a pair as $\mathcal{A}{:}M$.

For simplicity, we will assume that this diagram has a multispan shape (see Appendix). Then a homogeneous multimodel $\mathcal{A}$ can be seen as a set $\mathsf{Feet}\mathcal{A}$ of base models with a systems of correspondence spans over them.

A multimodel is called *discrete* if $\mathcal{A}$ is simply a set of models without correspondence spans. It means that we *a priori* assume that all local models are entirely independent.

A multimodel is *consistent* if colimit $\Sigma\mathcal{A}$ of the multispan $\mathcal{A}$ (which always exists in $[\![M]\!]^?$) satisfies M's constraints, i.e., $\Sigma\mathcal{A} \in [\![M]\!]$.

Heterogeneous multimodels and their consistency. A *heterogeneous multimodel* is a pair $(\mathcal{A\!A}, \mathcal{S})$, in which

$$\mathcal{A\!A} = (\mathcal{A}_1{:}M_1 \dots \mathcal{A}_n{:}M_n)$$

is a family of homogeneous multimodels $\mathcal{A}_i$ (multispans) with their metamodels $M_i \in \boldsymbol{MMod}$ ($i = 1..n$), and $\mathcal{S}$ is a diagram in $\boldsymbol{MMod}$ called the *metamodel schema* of the multimodel.

Consistency of a heterogeneous multimodel is much more involved than in the homogeneous situation, and we will begin with two simplifying assumptions.

1) Each homogeneous multimodel $\mathcal{A}_i$ is discrete, i.e., is a set of models without mappings between them.

2) The metamodel schema $\mathcal{S}$ is a multispan, that is, a set of total and partial spans over the set of metamodels $\{M_1 ... M_n\}$ considered as feet (see Appendix).

With these assumptions, the algorithm for checking global consistency is as follows.

The global consistency of $\mathcal{AA}$ is checked at the heads of all spans in $\mathcal{S}$. That is, for each span S in $\mathcal{S}$ we perform the following procedure.

Let $H = \mathsf{head}S$ be the head of S. First, we project to the space $[\![H]\!]^?$ of structural H-instances all models $\mathcal{A}_i$, whose metamodels M_i are reachable from H by the legs of the span. If the span is total, projecting is provided by the view mechanism. If the span is partial, projecting needs both view execution and model retyping as explained above. In this way we obtain a set of instances

$$\mathcal{A}_H = \{\mathsf{get}_{m_i}(A) \colon\ (m_i \colon H \to M_i) \in \mathsf{legs}S,\ A \in \mathcal{A}_i\} \subset [\![H]\!]^?$$

Second, instances in $\mathcal{A}_H$ are matched by a multispan (i.e., a family of correspondence spans) $\mathcal{C}_H$. Note that $\mathcal{C}_H$-data are provided by the user and are, in fact, part of the multimodel's state. That is, a multimodel is actually a triple $(\mathcal{AA}, \mathcal{S}, \mathcal{CC})$ with $\mathcal{CC} = \{\mathcal{C}_H \colon\ H = \mathsf{head}S, S \in \mathcal{S}\}$ a family of multispans indexed by $\mathcal{S}$.

Third, for each multispan $\mathcal{C}_H$ its colimit is computed, that is, all instances in $\mathcal{A}_H$ are merged modulo the correspondence $\mathcal{C}_H$ into a structural instance

$$\varSigma\mathcal{C}_H = \left(\biguplus \mathcal{A}_H\right) / \mathcal{C}_H\ \in [\![H]\!]^?.$$

Finally, for each span we check whether $\varSigma\mathcal{C}_H \in [\![H]\!]$, i.e., whether the colimit $\biguplus \mathcal{A}_H / \mathcal{C}_H$ satisfies all constraints declared in metamodel H.

Definition 1 (global consistency) The multimodel $(\mathcal{AA}, \mathcal{S}, \mathcal{CC})$ is considered to be *(globally) consistent* if $\varSigma\mathcal{C}_H \in [\![H]\!]$, $H = \mathsf{head}S$, for all spans S in $\mathcal{S}$.

Now we return to our two simplifying assumptions.

1) The general case with $\mathcal{A}_i$ being diagrams rather than sets can be treated similarly to the above. The key point is that translation operations get and $\mathsf{rtp}^?$ are functors, that is, they translate not only instances but also instance mappings, and hence correspondence diagrams as well. Then the projection $\mathcal{A}_H \subset [\![H]\!]^?$ will be a diagram rather than a set of instances, and diagram $\mathcal{C}_H$ will provide a second level correspondence structure. As colimit operation consumes any sort of input diagrams, the algorithm works well for the general case too.

2) Building a *generic* consistency checking algorithm for the case of metamodel schema being more complex than a set of spans is harder. When we have spans over spans like in the example in [1], we should have a possibility to declare commuttaivity of some diagrams in the metamodel schema. Then the

metamodel schema becomes a construct called a *sketch* [8] rather than a graph. Generalization in this direction is a work in progress.

5 Concrete multimodeling framework

In a nutshell, a *concrete multimodeling framework* (CMF) consists of three components: (i) a *base category* $\mathbb{G}$ of graph-like structures to be thought of as the carriers of metamodels and models, (ii) a *constraint language* cntr together with binary relations $\models$ of constraints' satisfiability by a model, and (iii) a *query language* $\mathbf{Q}$ together with operations of query execution over a model. We will consecutively consider the corresponding constructs

5.1 The carrier structure.

We fix a category $\mathbb{G}$, whose objects are to be thought of as graphs, or many-sorted (colored) graphs, or attributed graphs [9]. 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 [8], and hence possessing limits, colimits, and other important properties. We will call $\mathbb{G}$-objects *'graphs'*.

For a 'graph' G thought of as a metamodel, an *instance* of G is a pair $A = (D_A, t_A)$ with D_A another 'graph' and $t_A\colon D_A \to G$ a mapping (arrow in $\mathbb{G}$) to be thought of as *typing*. An *instance mapping* $f\colon A \to B$ is a 'graph' mapping $f\colon D_A \to D_B$ commuting with typing: $f; t_B = t_A$. This defines the category $\mathbb{G}/\mathrm{G}$ of G-instances (in category theory such categories are called *sliced*).

Any mapping $m\colon G_1 \to G_2 :: \mathbb{G}$ determines a functor

$$\mathsf{pb}(m)\colon \mathbb{G}/G_2 \to \mathbb{G}/G_1$$

built with pullback operation in the standard way (see e.g.[10, p.48]); note that the arrow is reversed. To ease notation, given an instance $B \in \mathbb{G}/G_2$, we will often write $\mathsf{pb}_m(B)$ for $\mathsf{pb}(m)(B)$.

We can join all local categories $\mathbb{G}/G$ into one category $\mathbb{G}^{\cdot\to\cdot}$ whose objects are $\mathbb{G}$-arrows $t\colon D \to G$ and morphisms $t_1 \to t_2$ are pairs $f_D\colon D_1 \to D_2$, $f_G\colon G_1 \to G_2$ such that $f_D; t_2 = t_1; f_G$. In other words, morphisms in $\mathbb{G}^{\cdot\to\cdot}$ are commutative squares. It is well known that the codomain assignment cod: $\mathbb{G}^{\cdot\to\cdot} \to \mathbb{G}$ is a fibration, whose Cartesian lifting is given by pullbacks (see Background section below).

The retyping operation $\mathsf{rtp}^?$ is defined by composition: given an instance $A \in \mathbb{G}/G_1$ and a 'graph' mapping $m\colon G_1 \to G_2 :: \mathbb{G}$, we define $\mathsf{rtp}^?_m(A) = t_A; m$. It is easy to see that it makes $\mathsf{rtp}^?_m$ a functor:

$$\mathsf{rtp}^?_m\colon \mathbb{G}/G_1 \to \mathbb{G}/G_2.$$

It is well known that functor $\mathsf{rtp}^?_m$ is a left adjoint to pb_m for any m [11].

Background: Fibrations. Let $\boldsymbol{p}\colon \boldsymbol{E} \to \boldsymbol{B}$ be a functor. We say that an arrow $e\colon E_1 \to E_2 :: \boldsymbol{E}$ is *above* arrow $b :: \boldsymbol{B}$ if $\boldsymbol{p}(e) = b$. Arrow e is called *vertical* (wrt. $\boldsymbol{p}$) if it is above an identity arrow in $\boldsymbol{B}$.

An arrow $e\colon E_1 \to E_2$ is called *(weakly) Cartesian* wrt. $\boldsymbol{p}$, or $\boldsymbol{p}$-Cartesian, if for any arrow $e'\colon E' \to E_2$ to the same target s.t. $\boldsymbol{p}(e) = \boldsymbol{p}(e')$, there is a unique vertical arrow $!\colon E' \to E_1$ with $!; e = e'$. The prefix '$\boldsymbol{p}$-' will be often skipped if the functor $\boldsymbol{p}$ is clear.

It is easy to prove that all Cartesian arrows with the same target are isomorphic (i.e., their domains are isomorphic and the respective triangle commutes). All identity arrows in $\boldsymbol{E}$ are Cartesian.

Functor $\boldsymbol{p}$ is called a *fibration* if

(a) For any object $E' \in \boldsymbol{E}_0$ and arrow $v\colon V \to \boldsymbol{p}E' :: \boldsymbol{B}$, there is a Cartesian arrow $e\colon E \to E'$ s.t. $\boldsymbol{p}(e) = v$. We say that arrow e is the *Cartesian lifting* of v at object E'.

(b) Sequential composition of two Cartesian arrows is again Cartesian.

Thus, properties of the view mechanism in Fig. 3 mean that the functor assigning to each model its metamodel is a fibration.

5.2 Constraints

We follow the lines of the institution theory and define constraints abstractly via a functor $\mathsf{cntr}\colon \mathbb{G} \to \mathbf{Sets}$. To ease understanding, we first consider how this functor acts on objects, and then proceed with mappings.

Constraints and metamodels. We assume that for any 'graph' G (to be thought of as the carrier of some metamodel), there are defined (i) a set $\mathsf{cntr}(G)$ of all *constraints* that can be specified over G, and (ii) a binary *satisfiability* relation

$$\models_G \subset \mathbb{G}/G \times \mathsf{cntr}(G)$$

between G's instances and constraints. For an instance $A \in \mathbb{G}/G$ and a constraint $c \in \mathsf{cntr}(G)$, we write $A \models_G c$ for $(A, c) \in \models_G$ and say that instance A satisfies the constraint c.

A *metamodel* is a pair $M = (G_M, C_M)$ with $G_M \in \mathbb{G}$ a carrier graph and $C_M \subset \mathsf{cntr}(G_M)$ a set of constraints. Instances of G_M, i.e., 'graphs' typed by G_M, are called *structural instances* of M, and we write $[\![\, M \,]\!]^?$ for the class $\mathbb{G}/G_M$. Structural instances that satisfy all constraints in C_M are *(legal)models* of M, and we write $[\![\, M \,]\!]$ for the class of all models, $[\![\, M \,]\!] \overset{\text{def}}{=} \left\{ A \in [\![\, M \,]\!]^? \colon\ A \models C_M \right\}$.

Constraint translation and metamodel mappings. Now we assume that cntr also acts on mappings: if $m\colon G \to G' :: \mathbb{G}$ is a 'graph' mapping, then $\mathsf{cntr}(m)$ is a function that translates any constraint $c \in \mathsf{cntr}(G)$ that may be declared over G to a constraint $c' = \mathsf{cntr}(m)(c)$ over G'.

Informally, mapping $\mathsf{cntr}(m)$ works as follows. A constraint c declares some property P_c of the corresponding fragment $G_c \subset G$ of 'graph' G. Mapping m

maps the fragment G_c into a fragment $m(G_c) \in G'$ of the target 'graph'. Since m is structure preserving, $m(G_c)$ has a structure similar to G_c, and hence property P can be declared for $m(G_c)$ as well. Hence, we have a constraint $c' = [m(G_c), P]$ over 'graph' G', and this constraint is the value of function $\mathsf{cntr}(m)$ at argument c. These considerations can be made precise within the framework of *generalized sketches* (or *diagram predicate graphs, dp-graphs*) described in [7].

Thus, translation $\mathsf{cntr}(m)$ does not change the property and simply substitutes a piece of G' for the corresponding piece of G. Then, if a structural instance $B \in [\![G']\!]^?$ of the target graph satisfies constraint $c' = \mathsf{cntr}(m)(c)$, then its "inverse image" $A = \mathsf{pb}_m(B) \in [\![G]\!]^?$ must satisfy constraint c. That is,

$$B \models c' \text{ implies } A \models c. \tag{2}$$

This statement is nothing but the fundamental *translation axiom* of the institution theory, which does not define what constraints are, and how they are translated, but postulates the translation axiom (see Section 6 for a brief primer on the notion of institution).

A *metamodel mapping* $m\colon (G, C) \to (G', C')$ is a 'graph' mapping $m\colon G \to G'$ compatible with the constraints: $\mathsf{cntr}(m)(c) \in C'$ for all $c \in C$. Hence, if $B \in [\![(G', C')]\!]$ is a model of the metamodel $M' = (G', C')$, then structural instance $\mathsf{pb}_m(B) \in [\![(G, C)]\!]^?$ is actually a legal model of $M = (G, C)$, that is, $\mathsf{pb}_m(B) \in [\![M]\!]$.

Our interpretation of constraint translation also requires to postulate

$$\mathsf{cntr}(m; n) = \mathsf{cntr}(m); \mathsf{cntr}(n)$$

for any two consecutive mappings $G_1 \xrightarrow{m} G_2 \xrightarrow{n} G_3$, and

$$\mathsf{cntr}(1_G) = 1_{\mathsf{cntr}(G)}.$$

It implies that the class of all metamodels and their mappings is a category, which we will denote by ***MMod***.

Heterogeneous models and their mappings. In the heterogeneous setting, we redefine the notion of *model* as a pair $A = (M_A, t_A)$ with $M = (G_A, C_A)$ a metamodel and $t_A\colon D_A \to G_A$ its legal model (instance), $t_A \models C_A$.

A *model mapping* $f\colon B \to A$ is a pair of mappings,

$$f_{\text{meta}}\colon M_B \to M_A \text{ and } f_{\text{inst}}\colon D_B \to D_A,$$

the first being a metamodel mapping and the second an instance mapping, such that the square commutes: $t_B; f_{\text{meta}} = f_{\text{inst}}; t_A$. In this way we obtain the category of all (heterogeneous) models and their mappings, which we denote by ***Mod***.

Moreover, it is easy to see that owing to translation axiom (2) we still have a codomain fibration $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$ (with Cartesian lifting via pullbacks), which projects a model A to its metamodel (G_A, C_A), and a model mapping f to its meta-component f_{meta}.

5.3 Queries I: Preliminary discussion

We begin in the institution theory fashion and declare a functor $\mathsf{quer}\colon \boldsymbol{MMod} \to \mathbf{Sets}$. However, this is a very poor model because it says nothing about the fundamental properties of queries — the possibility to substitute one query into another. Capturing substitutions in an abstract way needs some technical work to be done. It is presented below for the general case of any category $\boldsymbol{C}$ for $\boldsymbol{MMod}$; and queries can be understood as sets of terms built over $\boldsymbol{C}$-objects as the carriers.

Queries are ordered. For any object $M \in \boldsymbol{C}_0$, we assume a partial order on the set $\mathsf{quer}M$: $Q_1 \leq Q_2$ means, informally, that query Q_2 subsumes query Q_1 (like, e.g., set of terms term $\{(a{+}b){*}c, a{*}c{*}d\}$ built over carrier $M = \{a, b, c, d, e\}$ subsumes $\{a + b, a * c\}$). Thus, $\mathsf{quer}M$ is required to be a poset. Moreover, this poset is a join semi-lattice because the union of two sets of terms over a carrier is again a set of terms over the same carrier. We will denote join of queries Q_1 and Q_2 by $Q_1 * Q_2$ (because informally this join is like conjunction — both queries are used), and let $\mathbf{Poset}^*$ denotes the category of posets with finite joins. Query translation is required to respect this structure and we thus postulate a functor $\mathsf{quer}\colon \boldsymbol{C} \to \mathbf{Poset}^*$.

For a map $f\colon M \to N :: \boldsymbol{C}$ and query $Q \in \mathsf{quer}M$, we will denote query $\mathsf{quer}f(Q) \in \mathsf{quer}N$ by Q^f.

Queries add derived elements, and are composable. Let M be a $\boldsymbol{C}$-object (think of a metamodel or a database schema). Then applying a query $Q \in \mathsf{quer}M$ can be described by adding to M derived elements specified by Q. In other words, any query Q is assigned with inclusion $\eta_Q\colon M \hookrightarrow M.Q$. As always, it is technically easier to work with monic arrows (monics) rather than inclusions; we may then think that elements of M are considered as queries extracting those elements, and each element of M can be then considered as trivially derived.

If $Q \in \mathsf{quer}M$ and $Q^+ \in \mathsf{quer}M.Q$, then we require existence of a unique query $Q' \in \mathsf{quer}M$ s.t. $\eta_Q;\eta_{Q^+} = \eta_{Q'}$. We denote this query Q' by $Q.Q^+$ and require $Q \leq Q.Q^+$. Conversely, if $Q_1 \leq Q_2$, we require existence of a unique query $Q^+ \in \mathsf{quer}M.Q_1$ such that $\eta_{Q_1};\eta_{Q^+} = \eta_{Q_2}$. We denote this query Q^+ by Q_2/Q_1 and require $Q_1.(Q_2/Q_1) = Q_2$. Note that this equality cannot be derived: although the commutative triangle of η-arrows consists of monic arrows, we do not exclude the situation when different queries have the same η-arrow.

Background: Algebraic theories in extension form. Let $\boldsymbol{C}$ be a category. An *algebraic theory in extension form* over $\boldsymbol{C}$ is a triple $\mathbf{T} = (\mathsf{T}, \eta, \_^\#)$, in which

$\mathsf{T}\colon \boldsymbol{C}_0 \to \boldsymbol{C}_0$ maps $\boldsymbol{C}$-objects to $\boldsymbol{C}$-objects;

$\eta\colon \boldsymbol{C}_0 \to \boldsymbol{C}_1$ assigns to each $\boldsymbol{C}$-object A a $\boldsymbol{C}$-arrow $\eta_A\colon A \to \mathsf{T}A$

$\_^{\#}\colon \boldsymbol{C}(\_, \mathsf{T}\_) \to \boldsymbol{C}(\mathsf{T}\_, \mathsf{T}\_)$ assigns to a $\boldsymbol{C}$-arrow $f\colon B \to \mathsf{T}A$ an "extension" $f^{\#}\colon \mathsf{T}B \to \mathsf{T}A$, such that the following three axioms hold (see the diagram below):

$$\begin{tikzcd}
 & & & \mathsf{T}C \arrow[dl, "g^{\#}"'] \\
\mathsf{T}A \arrow[dr, "(\eta_A)^{\#}"] & & \mathsf{T}B \arrow[dl, "f^{\#}"'] & C \arrow[l, "g"'] \arrow[u, "\eta_C"'] \\
A \arrow[u, "\eta_A"] \arrow[r, "\eta_A"] & \mathsf{T}A & B \arrow[l, "f"'] \arrow[u, "\eta_B"'] &
\end{tikzcd}$$

$$(\eta_A)^{\#} = 1_{\mathsf{T}A} \tag{3}$$

$$\eta_B; f^{\#} = f \tag{4}$$

$$(g; f^{\#})^{\#} = g^{\#}; f^{\#} \tag{5}$$

Lemma 1 (Manes [12]). *The notions of monad and algebraic theory in extension form are equivalent.*

Because of this result, we will call the construct above a *monad in extension form* or just a *monad.*

Given a monad $\mathbf{T}$ over $\boldsymbol{C}$, a *Kleisli mapping* from $\boldsymbol{C}$-object B to C-object A is a $\boldsymbol{C}$-arrow of the following type: $f\colon B \to \mathsf{T}A$. Kleisli mappings can be composed as shown in the diagram above: $g; f \overset{\text{def}}{=} g; f^{\#}$, and it is easy to prove that their composition is associative. We thus have the *Kleisli category* $\boldsymbol{C}_{\mathbf{T}}$ of monad $\mathbf{T}$: it has the same objects as $\boldsymbol{C}$ but Kleisli mappings as morphisms.

Query language: Syntactical part. Example in Section 2 suggests the following definition.

A *query language* is a monad $\mathbf{QD} = (\mathsf{QD}, \eta^{\text{spec}}, \_^{\#})$ over the category of metamodels $\boldsymbol{MMod}$ (notation $\mathbf{QD}$ is a single letter that refers to "Query Definition"). This monad augments each metamodel $M = (G_M, C_M)$ with all possible derived elements, together with all new constraints they satisfy, and we have inclusion:

$$\eta_M^{\text{def}}\colon M \hookrightarrow M.\mathsf{QD} = (G_M.\mathsf{QD}, C_M.\mathsf{QD}),$$

where we write the mapping symbol on the right of the argument.

Query language: Semantic part (query execution). Example in Section 2 demonstrates that query execution is an operation that acts in concert with query definition. Hence, we may join query definition (a monad $\mathbf{QD}$ on $\boldsymbol{MMod}$) and query execution into a single new monad $\mathbf{Q} = (\mathsf{Q}, \eta, \_^{\#})$ over the category of models $\boldsymbol{Mod}$. Each component of this monad consists of two components, syntactical and semantical, which work together as follows. Mapping $\mathsf{Q}\colon \boldsymbol{Mod}_0 \to \boldsymbol{Mod}_0$ is a pair $(\mathsf{QD}, \mathsf{QE})$, which augments each model A with all

$$\begin{tikzcd}
 & & & D_B.\mathsf{QE} \arrow[dll, "{f_{\mathrm{inst}}}^{\#}"'] \arrow[dd, "t_B.\mathsf{QE}"] \\
D_A \arrow[r, hook, "\eta_A^{\mathrm{exe}}"] \arrow[dd, "t_A"'] \arrow[ddr, phantom, "{[\mathsf{PB}]}"] & D_A.\mathsf{QE} \arrow[dd, "t_A.\mathsf{QE}"'] & D_B \arrow[l, "f_{\mathrm{inst}}"'] \arrow[ur, hook] \arrow[dd, "t_B"] & \\
 & & & M_B.\mathsf{QD} \arrow[dll, "{f_{\mathrm{meta}}}^{\#}"'] \\
M_A \arrow[r, hook, "\eta_A^{\mathrm{def}}"'] & M_A.\mathsf{QD} & M_B \arrow[l, "v = f_{\mathrm{meta}}"] \arrow[ur, hook] &
\end{tikzcd}$$

Fig. 5. Query monad, formally.

$$\begin{tikzcd}
 & D_A.(Q_1{*}Q_2)^{\mathrm{exe}} \arrow[dd, dashed] & & D_A.Q_1^{\mathrm{exe}}.Q_2^{\mathrm{exe}} \arrow[ll, Rightarrow, "(\eta^{\mathrm{exe}}_{Q_1*Q_2})^{\#}"'] \arrow[dd, dashed] & & D_B.Q_2^{\mathrm{exe}} \arrow[ll, "{f_{\mathrm{inst}}}^{\#}"'] \arrow[dd, "t_B.Q_2^{\mathrm{exe}}"] \\
D_A \arrow[ur, hook, "\eta^{\mathrm{exe}}_{Q_1*Q_2}"] \arrow[rr, hook, "\eta^{\mathrm{exe}}_{Q_1}"] \arrow[dd, "t_A"'] & & D_A.Q_1^{\mathrm{exe}} \arrow[ur, hook, "\eta^{\mathrm{exe}}_{Q_2}"] \arrow[dd] & & D_B \arrow[ll, "f_{\mathrm{inst}}"'] \arrow[ur, hook, "\eta^{\mathrm{exe}}_{Q_2}"] \arrow[dd] & \\
 & M_A.(Q_1{*}Q_2)^{\mathrm{def}} & & M_A.Q_1^{\mathrm{def}}.Q_2^{\mathrm{def}} \arrow[ll, Rightarrow, dashed] & & D_B.Q_2^{\mathrm{exe}} \arrow[ll, dashed, "{f_{\mathrm{meta}}}^{\#}"'] \\
M_A \arrow[ur, hook, dashed, "\eta^{\mathrm{def}}_{Q_1*Q_2}"] \arrow[rr, hook, "\eta^{\mathrm{def}}_{Q_1}"'] & & M_A.Q_1^{\mathrm{def}} \arrow[ur, hook, dashed, "\eta^{\mathrm{def}}_{Q_2}"] & & M_B \arrow[ll, "f_{\mathrm{meta}}"] \arrow[ur, hook, "\eta^{\mathrm{def}}_{Q_2}"] &
\end{tikzcd}$$

Fig. 6. Query monad "physics".

its derived elements respectively typed as shown by the left square diagram in Fig. 5 (notation QE is a single letter that refers to "Query Execution"). Symbol PB means that the square is a pullback, that is, D_A is exactly the inverse image of 'subgraph' $G_A \subset G_A.\mathsf{QD}$ under mapping η_A^{def} and hence queries do not affect the original data. Note that arrows η_A^{def} and η_A^{exe} are components of a single ***Mod***-arrow $\eta_A\colon A \to \mathsf{Q}A$.

Extension operation $\_^{\#}$ also has two components, but to ease notation we will denote them by the same symbol. The "physics" of this operation is more complicated, and to analyze it we will use Fig. 6, in which instead of the biggest-possible "query" Q (including all possible queries), we consider ordinary queries Q_i.

Let $A = (t_A.M_A)$ be a model and Q_1 a query against it. That is, we have a query definition Q_1^{def} against metamodel M_A, and an operation of query execution $Q_1^{\rm exe}$ for instance t_A. Together this data produce the front-left square diagram in Fig. 6; recall that the diagram is a pullback.

Let $A.Q_1 \xleftarrow{f} B$ be an injective model mapping, that is, a pair of injective mappings $(f_{\rm meta}, f_{\rm inst})$ making the front right square in Fig. 6 commutative. Suppose that we have another query Q_2 against model B (the right-most face of the prism in Fig. 6), and we want to build mapping $A.Q_1.Q_2 \xleftarrow{f^\#} B.Q_2$. That is, we need to build two mappings, ${f_{\rm meta}}^\#$ and ${f_{\rm inst}}^\#$ as shown in the figure.

Building mapping ${f_{\rm meta}}^\#$ is easy: on the level of metamodels we only deal with query definitions, that is, terms, and ${f_{\rm meta}}^\#$ is given by term substitution. To wit: we homomorphically extend mapping $f_{\rm meta}$ to the bigger domain by adding elements of Q_2 to the codomain (and so the bottom-right square is a pushout). Note also that metamodel $M_A.Q_1^{\rm def}.Q_2^{\rm def}$ equals (up to a natural isomorphism), to metamodel $M_A.(Q_1{*}Q_2)^{\rm def}$, where $*$ denotes the operation of term substitution.

We cannot apply substitution on the level of instances because here we deal with real operations rather than terms. Suppose, however, that our queries are *monotonic*, that is, preserve inclusion of datasets over which they operate [13, p.42]. The right upper-face square Fig. 6 shows that monotonicity of query Q_2 provides existence of mapping ${f_{\rm inst}}^\#$. Note also that it is reasonable to assume that execution of a composed query $Q_1{*}Q_2$ equals to composition of the respective executions, which gives us equality of models $A.Q_1.Q_2$ and $A.(Q_1{*}Q_2)$ (as shown by the back-face left square of the prism).

Finally, by combining together all possible monotonic queries Q_i into one biggest-possible monotonic query Q, we come to data shown in Fig. 5. In other words, it makes sense to define a monotonic query language as a monad $\mathbf{Q} = (\mathsf{Q}, \eta, \_^\#)$ over category of models ***Mod***. With a concrete definition of what is a monotonic query language, we should be able to prove that each such a language gives rise to a monad as above.

Query translation. Suppose that the front-right square in Fig. 5 is a pullback, that is, model (D_B, t_B) is the inverse image of model $(D_A.\mathsf{QE}, t_A.\mathsf{QE})$ (recall that $f_{\rm meta}$ is injective). Then the back "diagonal" square should be also a pullback because of the following.

Let object (metamodel) $Q^{\rm def} = f_{\rm meta}(M_B.\mathsf{QD})$ be the image of object $M_B.\mathsf{QD}$ under mapping $f_{\rm meta}$; it can be seen as a query against the model A (a part of the biggest-possible query Q). The part of model $D_A.\mathsf{QE}$, whose types (provided by $t_A.\mathsf{QE}$) are outside of $Q^{\rm def}$, does not influence execution of $Q^{\rm def}$, it is out of its scope. Then executing $Q^{\rm def}$ for model A, i.e., selecting $D_A.Q^{\rm exe}$ inside of $D_A.\mathsf{QE}$, and then projecting it back by pulling back along ${f_{\rm meta}}^\#$, gives the same (up to iso) result as if we first pull back along $f_{\rm meta}$, and then execute $Q^{\rm def}$ against B, that is, build the right-most face of the prism in Fig. 5. Thus, if the front-right square is a pullback, then the back-diagonal square is a pullback as well.

Definition 2 (Query language) Let $\boldsymbol{p}$: ***Mod*** $\to$ ***MMod*** be a fibration (of models over metamodels). A *(monotonic) query language* is a monad $\mathbf{Q} = (\mathsf{Q}, \eta, \_^\#)$

$$\begin{tikzcd}
\text{Models :} & \boldsymbol{Mod}_{\mathbf{Q}} \arrow[r, shift left, "U_{\mathbf{Q}}"] \arrow[d, "\boldsymbol{p}_{\mathbf{Q}}"'] & \boldsymbol{Mod} \arrow[l, shift left, "F_{\mathbf{Q}}"] \arrow[r, shift left, "U_{\mathrm{cntr}}"] \arrow[d, "\boldsymbol{p}"'] & \boldsymbol{Mod}^{?} \equiv \mathbb{G}^{\cdot\to\cdot} \arrow[l, shift left, "F_{\mathsf{cntr}}"] \arrow[d, "\boldsymbol{p}^{?} \equiv \mathrm{cod}"] \\
\text{conformsTo :} & & & \\
\text{Metamodels :} & \boldsymbol{MMod}_{\mathbf{QD}} \arrow[r, shift left, "U_{\mathbf{QD}}"] & \boldsymbol{MMod} \arrow[l, shift left, "F_{\mathbf{QD}}"] \arrow[r, shift left, "UM_{\mathsf{cntr}}"] & \mathbb{G} \arrow[l, shift left, "FM_{\mathsf{cntr}}"]
\end{tikzcd}$$

Fig. 7. The universe of models and metamodels within a concrete metamodeling framework

over $\boldsymbol{Mod}$, satisfying two *query translation axioms*:

$$\text{for any model } A,\ \eta_A \text{ is } \boldsymbol{p}\text{-Cartesian} \tag{6}$$

$$\text{if } f\colon B \to A.\mathsf{Q} \text{ is } \boldsymbol{p}\text{-Cartesian, then } f^{\#}\colon B.\mathsf{Q} \to A.\mathsf{Q} \text{ is } \boldsymbol{p}\text{-Cartesian as well} \tag{7}$$

Lemma 2. *Given a metamodeling framework $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$, and a query language $\mathbf{Q}$ over it, we have a fibration $\boldsymbol{p}_{\mathbf{Q}}\colon \boldsymbol{Mod}_{\mathbf{Q}} \to \boldsymbol{MMod}_{\mathbf{Q}}$ of the respective Kleisli categories.*

*Proof.*Cartesian lifting is given by the front-face diagram in Fig. 5, that is, given a view definition v, first the query involved is executed, and then the result is relabeled. Axiom (6) ensures that identity is lifted to identity. Condition (b) of the definition of fibration is provided by axiom (7) (and the definition of Kleisli arrow composition). □

In other words, translation axioms ensure good algebraic properties of the view mechanism described by diagrams in Fig. 3.

5.4 Constraints and queries together: Summary.

A concrete modeling framework (CMF) is a triple

$$\mathcal{F} = (\mathbb{G}, (\mathsf{cntr}, \models), \mathbf{Q}),$$

consisting of three parts: the structure $\mathbb{G}$, the constraint language $(\mathsf{cntr}, \models)$, and the query language $\mathbf{Q}$.

1) The structural part $\mathbb{G}$ gives us the standard codomain fibration cod: $\mathbb{G}^{\cdot\to\cdot} \to \mathbb{G}$. We prefer to write it as $\boldsymbol{p}^{?}\colon \boldsymbol{Mod}^{?} \to \mathbb{G}$ because in our context objects of category $\mathbb{G}^{\cdot\to\cdot}$ are instances that are properly typed but may violate constraints; more accurately, constraints are simply absent so far.

2) The pair $(\mathsf{cntr}, \models)$ adds the possibility to declare constraints in metamodels (functor cntr) and check their validity for instances (relations $\models$). Together, data 1) and 2) define the right commutative square diagram in Fig. 7, whose nodes are categories and arrows are functors. Vertical arrows are fibrations: they just

project the meta-component of $\boldsymbol{Mod}$-objects and arrows; the translation axiom plays a crucial role in making projection $\boldsymbol{p}$ a fibration. Arrows going from left to right are functors that forget about constraints, that is, provide the Underlying structure for constraint declaration. Arrows going from right to left are functors that add the empty set of constraint for Free. Each of the upper functors is a right adjoint to the lower one.

Finally, pairs of U-fucntors and F-functors are fibration morphisms: them map Cartesian arrows into Cartesian arrows.

3) Monad $\mathbf{Q} = (\mathsf{Q}, \eta, \_^{\#})$ over category $\boldsymbol{Mod}$ is a formal model of the query (view) mechanism, i.e., the possibility to define queries over metamodels and execute them for models. The definition works well if all queries are monotonic. Since arrows in $\boldsymbol{Mod}$ are, in fact, pairs of arrows (making the respective squares commutative), each component of $\mathbf{Q}$ contains two components working in concert. More accurately, each component of monad $\mathbf{Q}$ has a purely syntactical (definitional) part operating over metamodels. We may project these syntactical components out and obtain monad $\mathbf{QD}$ (of query definitions) over category of metamodels $\boldsymbol{MMod}$ (of model definitions). Moreover, due to query translation axioms, fibration $\boldsymbol{p}$ gives rise to fibration $\boldsymbol{p}_{\mathbf{Q}}\colon \boldsymbol{Mod}_{\mathbf{Q}} \to \boldsymbol{MMod}_{\mathbf{QD}}$ between the Kleisli categories of the two monads.

These data are summarized by the left commutative square in Fig. 7. Again, horizontal arrows going from left to right are functors that provide the Underlying structure for posing and executing queries. Arrows going from right to left are functors that provide additional "empty" structure for Free, now identical views rather than empty sets of constraints. Each of the upper functors is a right adjoint to the lower one — this is a standard material of categorical algebra [12]. The pairs of U- and F-functors are again fibration morphisms (axiom (6) is crucial for this result).

Thus, a CMF gives rise to data specified by Fig. 7.

From concrete to abstract frameworks. Let $\mathcal{F} = (\mathbb{G}, \mathsf{cntr}, \models, \mathbf{Q})$ be a concrete multimodeling framework as defined above. It gives rise to an abstract multimodeling framework $\mathcal{F}^{@} = (\boldsymbol{MMod}^{@}, \mathsf{get}^{?}, \mathsf{get}, \mathsf{rtp}^{?})$ in the following way.

From $\mathcal{F}$, we derive data specified in Fig. 7 as it was explained above. Then we define $\boldsymbol{MMod}^{@} = \boldsymbol{MMod}_{\mathbf{Q}}$, and generate from fibrations $\boldsymbol{p}^{?}$ and $\boldsymbol{p}_{\mathbf{Q}}$ indexed categories $\mathsf{get}^{?}\colon \boldsymbol{MMod}^{@op} \to \mathbf{Cats}$ and $\mathsf{get}\colon \boldsymbol{MMod}^{@op} \to \mathbf{Cats}$ in the standard way [10]. Mapping $\mathsf{rtp}^{?}$ is given by postcomposition (Section 5.1).

It follows then that the notion of global consistency defined in Section 4 is applicable to any concrete framework.

6 Relation to other work and historical remarks.

A general discussion of work related to heterogeneous multimodeling can be found in [1]. The goal of the present section is to trace the history of the formal framework.

Both main notions — of an abstract and concrete multimodeling frameworks (AMF and CMF) — are built within the range of ideas introduced into computer science in the early 80s by Joseph Goguen and Rod Burstall [14] under the name of the *institution theory*. We will briefly review the main definitions.

Institutions and specification frames. An *institution* is a quadruple $I = (\boldsymbol{Sign}, \mathsf{sen}, \mathsf{mod}, \models)$ with $\boldsymbol{Sign}$ a category of *signatures* (think of signatures of operations, or FOL signatures, or the like), $\mathsf{sen}\colon \boldsymbol{Sign} \to \mathbf{Sets}$ and $\mathsf{mod}\colon \boldsymbol{Sign}^{\mathrm{op}} \to \mathbf{Cats}$ are functors that assign to each signature the set of all logical sentences that can be declared over this signature, and the category of its *models*, respectively. $\models$ is a mapping that assigns to each signature Σ a binary *satisfiability* relation $\models_\Sigma \subset \mathsf{mod}_0(\Sigma) \times \mathsf{sen}(\Sigma)$.[2] Given a signature morphism $\sigma\colon \Sigma_1 \to \Sigma_2$, functor $\mathsf{mod}(\sigma)\colon \mathsf{mod}(\Sigma_2) \to \mathsf{mod}(\Sigma_1)$ is often called *reduction* of Σ_2-models to Σ_1-models along σ.

These ingredients must satisfy the following *translation axiom*: for any signature morphism (translation) $\sigma\colon \Sigma_1 \to \Sigma_2$, any sentence $\phi \in \mathsf{sen}\Sigma_1$ and any model $A \in \mathsf{mod}\Sigma_2$,

$$\text{(TA)} \qquad A.\mathsf{mod}(\sigma) \models_{\Sigma_1} \phi \text{ iff } A \models_{\Sigma_2} \phi.\mathsf{sen}(\sigma)$$

A *theory* or *specification* is a pair $T = (\Sigma, \Phi)$ with $\Sigma \in \boldsymbol{Sign}_0$ a signature and $\Phi \subset \mathsf{sen}(\Sigma)$ a set of sentences over it. A *model* of T is a model A of Σ satisfying all sentences in Φ: $A \models_\Sigma \phi$ for all $\phi \in \Phi$. We write $\mathsf{mod}_0(T)$ for the class of all T-models. A *theory morphism* $T_1 \to T_2$ is a signature morphism $m\colon \Sigma_1 \to \Sigma_2$ such that $\mathsf{sen}(\sigma)(\Phi_1) \subset \Phi_2$. Translation axiom (TA) ensures that functor $\mathsf{mod}(m)$ maps from category $\mathsf{mod}(\Sigma_2)$ to category $\mathsf{mod}(\Sigma_2)$. That is, we have a functor (indexed category) $\mathsf{mod}\colon \boldsymbol{Thr} \to \mathbf{Cats}$ defined on the category $\boldsymbol{Thr}$ of all theories. This indexed category is often called the *specification frame* generated by institution I.

AMF	**Institution**
	signature
constraint	sentence
metamodel	theory
model	model
view mechanism	reduction
retyping	

Table 1. Comparison of AMFs and institutions

Abstract multimodeling framework. AMF is a construct in-between the notions of institution and specification frame derived from an institution (see Table 1). On one hand, if we consider only functors get between categories of legal models $[\![\,..\,]\!]$ and forget about structural instances $[\![\,..\,]\!]^?$, we come exactly to the notion of specification frame. Availability of two categories of models is, in fact, a way of modeling constraints (logical sentences) without mentioning them,

[2] Expressions $\mathbf{C}_0$ and $\mathbf{C}_1$ denote the class of all objects and all arrows of a category $\mathbf{C}$ resp.

that is, purely extensionally.[3] In addition, the notion of AMF assumes availability of the left adjoint (retyping) functor, $\mathsf{rtp}^? \dashv \mathsf{get}^?$. For slice categories, this is a well-known idea first presented (probably) in [11].

Note that we do not require the retyping functor to map legal models to legal models. The latter condition only holds for special (Horn) logics [**?**], and hence the existence of right adjoint to get is a much stronger condition. It is postulated in the so called *liberal* institutions [15].

Concrete multimodeling framework. In a nutshell, the idea behind CMF is to enrich the notion of institution with an algebraic model of query languages. This enrichment may be seen necessary even from the purely logical perspective because normally the set of all sentences is a freely generated algebra in some predefined variety, and a model is a homomorphism into an algebra "extracted" from a semantic structure (a set or a Kripke frame). Indeed, this is a basic premise of the algebraic logic in the sense of Polish school.

An early attempt to capture the inductive nature of syntax is Goguen and Burstall's *parchments* [16]; however, they only dealt with syntax. Another early attempt to capture *both* syntax and semantics by introducing a monad T over the category of signatures so that sentences are elements of T-free algebras and models are homomorphisms into a T-algebra can be found in [17] (where several such monads = algebraic theories popular in algebraic logic are considered.) A much more intelligent and general elaboration of these ideas can be found in [18]; and a database-oriented digest is in [19]. The same idea of introducing a monad and going along the lines of algebraic logic can also be found in [20][4].

However, considering models as homomorphisms *from* signatures (dataschemas, metamodels) into semantic algebras does not fit in the standards of metamodeling, where a model/instance is typically a (typing) mapping from a semantic structure *to* its schema. A precise elaboration of this switch in semantics for constraints only (no queries) can be found in [7]. The present report presents the *to*-semantics (typing) for the case when both constraints and queries are considered. Thus, it has taken more than fifteen years to distill finally a manageable notion of an institution with queries. The crucial idea that was not easy to recognize is that although query execution definitely lives in semantics (Eilenberg-Moore algebras) rather than in syntax (Kleisli), considering the query language monad over the arrow category allows us to capture both syntax and semantics via the Kleisli construction.

7 Instead of conclusion: Future work.

1) Accurate formal proofs of the results presented above are still to be completed. I do not anticipate any principle problems but several seemingly non-essential details skipped in the above presentation need to be written down and checked.

[3] One of the reviewers of our paper [1], marked the elegance of the idea.

[4] without any reference to [17], which I do know from personal communication was essentially used, at least, as the starting point.

2) The key feature of the approach is that heterogeneous consistency checking is reduced to homogeneous with a minimal amount of metamodel merging; the latter is unavoidable if we want to treat inter-metamodel constraints (see [1]) yet the approach is *as local as possible*. An alternative approach (let's call it *total*) would be to merge all metamodels into one global metamodel, consider all models as (partial) instances of that global metamodel, and correspondingly check their global consistency wrt. the global metamodel.

The latter idea seems to be the most immediate and direct specification of out intuition of *what* global consistency of heterogeneous multimodel should be. Hence, the total approach to global consistency could serve as a basic *definition* of what the global consistency is. Then we need to *prove* that our local definition of global consistency is indeed equivalent to the total one. Although there are strong formal arguments that this is indeed the case, an accurate formal proof is not easy and is a work in progress.

3) The notion and algorithm for global consistency checking described in the report assume that the metamodel schema is a set of spans. In practice, metamodels may overlap in more complex ways, and hence the metamodel schema should be considered, in general, to be a graph. Generalization of the framework for this situation is an important issue to be addressed.

4) The notion of concrete multimodeling frameworks defined in Section 5 actually defines a rather abstract construct. It is less abstract than an abstract framework of Section 3 in that constraints and queries are taken into account, but they are modeled in a fairly abstract way. There are neither predicate nor operation symbols in the framework (and so neither formulas nor terms are explicit), only summarizing/cumulative effects of having constraints and queries are specified while their syntax is ignored. It implies that much work is to be done in order to prove that a particular multimodeling framework/tool is indeed an instance of the notion of concrete framework.

To facilitate this work, it is useful to have a less abstract notion of the multimodeling framework, in which the syntax of constraint and query declarations would be captured. However, we need a fairly abstract model of syntax, which would be applicable to a wide diversity of multimodeling languages and tools in practical use. An abstract model of syntax for constraints is provided by the notion of generalized sketch (or dp-graph), whose presentation tailored towards the metamodeling patterns can be found in [7]. A similar model for queries is described in [21] but in the database context, in which semantics of a metamodel is given by a functor from the metamodel into category **Sets** (*from*-semantics). Hence, it is important to redescribe the notion of query for the metamodeling context within the *to*-semantics (typing-mapping-based).

References

1. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In Dingel, J., Solberg, A., eds.: Models in Software Engineering, Workshops and Symposia at MODELS 2010. Reports and Revised Selected Papers. (2011) to appear.

2. Diskin, Z.: Model synchronization: mappings, tile algebra, and categories. In R. Lämmel et al., ed.: Postproceedings GTTSE 2009. Volume 6491 of LNCS., Springer
3. Melnik, S., Bernstein, P., Halevy, A., Rahm, E.: Supporting executable mappings in model management. In: SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (2005) 167–178
4. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Exploiting schemas in data synchronization. J. Comput. Syst. Sci. **73**(4) (2007) 669–689
5. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: GTTSE. (2007) 3–46
6. Diskin, Z.: Model transformation as view computation: an algebraic approach. Technical Report CSRG-573, University of Toronto (2008) http://ftp.cs.toronto.edu/pub/reports/csrg/582/TR-582-MTisVM.pdf.
7. Diskin, Z., Wolter, U.: A diagrammatic logic for object-oriented visual modeling. Electron. Notes Theor. Comput. Sci. **203**(6) (2008) 19–41
8. Barr, M., Wells, C.: Category theory for computing science. Prentice Hall (1995)
9. Ehrig, H., Ehrig, K., Prange, U., Taenzer, G.: Fundamentals of Algebraic Graph Transformation. (2006)
10. Jacobs, B.: Categorical logic and type theory. Elsevier Science Publishers (1999)
11. Freyd, P.: Aspects of topoi. Bull.Austral.Math.Soc. **7** (1972) 1–72
12. Manes, E.: Algebraic Theories. No.26 in Graduate Text in Mathmetics. Springer Verlag (1976)
13. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
14. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of ACM **39**(1) (1992) 95–146
15. Tarlecki, A.: Institutions: An abstract framework for formal specifications. In Astesiano, E., Kreowski, H., Krieg-Brukner, B., eds.: Alegbraic foundations of system specification. (1998)
16. J.Goguen, Burstall, R.: A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In: Category Theory and Computer Programing. Volume 240 of Springer LNCS. (1986) 313–333
17. Diskin, Z.: Algebraizing institutions: Incorporating algebraic logic methodology into the institution framework for building of specifications. Technical Report 9404, Frame Inform Systems/LDBD, Riga, Latvia (1994) http://www.cs.toronto.edu/ zdiskin/Pubs/TR-1994(algebraizingInstitutions).pdf.
18. Diskin, Z.: A unified functorial framework for propositional-like and equational-like logics. In: First Workshop on Abstract Algebraic Logic, WAAL'97, Centre de Recerca Matematica, Barcelona, Spain, 1997. Volume 10 of Quaderns. (1998) 26–50
19. Diskin, Z., Kadish, B.: A graphical yet formalized framework for specifying view systems. In: Advances in Databases and Information Systems, ADBIS'97. (1997) 123–132 `www.uitcinc.com/articles/article9.html`.
20. Voutsadakis, G.: Categorical abstract algebraic logic: algebraizable institutions. Applied Categorical Structures **10** (2002) 531–568
21. Diskin, Z.: Databases as diagram algebras: Specifying queries and views via the graph-based logic of skethes. Technical Report 9602, Frame Inform Systems, Riga, Latvia (1996) doubloadable from http://www.cs.toronto.edu/ zdiskin/Pubs/TR-9602.pdf.
22. Sabetzadeh, M., Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. Requir. Eng. **11**(3) (2006) 174–193

A Appendix. Networks of model interaction

This section consists of a series of simple definitions and their direct consequences. We will not split them into numerous explicit Definitions and simply point to defined notions by italicizing them.

Spans and cospans. A *category* $\boldsymbol{C}$ is a directed graph with composable arrows; $\boldsymbol{C}_0$ and $\boldsymbol{C}_1$ denote the classes of its nodes and arrows resp. A *diagram* in a category $\boldsymbol{C}$ is a graph mapping $D\colon G_D \to \boldsymbol{C}$ with graph G_D called the *shape* of the diagram. We will follow presentation in [22] and identify diagram D with its image $\{D(e) : e \in G_D\}$. It is imprecise but makes presentation simpler. The difference becomes essential when D maps different elements in G_D into the same element in $\boldsymbol{C}$; we will make special reservations about it where appropriate.

In this section we will consider several diagrams important for heterogeneous multimodeling. To ease understanding, $\boldsymbol{C}$-elements may be interpreted as graphs and graph mappings.

An *n-ary span* ($n \geq 1$) in $\boldsymbol{C}$ is a subgraph S consisting of n-arrows with a common source. The latter is called the *head* of the span and denoted by H_S, arrows p_{Si} are *projections* or *legs* , and their targets F_{Si}, ($i = 1..n$) are *feet*; the index S may be skipped. For example, the triple of arrows going out of node H in Fig. 8 is a ternary span.

We write $\mathsf{feet}S$ and $\mathsf{legs}S$ for sets $\{F_1..F_n\}$ and $\{p_1..p_n\}$ resp. We will say that a span is *over* its feet, and will often refer to a span by referring to its head. Our main interpretation of span's feet is by a set of similar software artifacts to be matched, and the head is an artifact encompassing a set of n-ary matching links between feet.

An *n-ary cospan* is defined similarly but arrows go from the feet to the *cohead* and are called *coprojections* or *colegs*. For example, the triple of arrows in Fig. 8 heading into C is a ternary cospan.

$$\begin{tikzcd}
H \arrow[d, "p_1"'] \arrow[dr, "p_2"] \arrow[drr, "p_3"] & & \\
F_1 \arrow[d, dashed, "q_1"'] & F_2 \arrow[dl, dashed, "q_2"] & F_3 \arrow[dll, dashed, "q_3"] \\
C & &
\end{tikzcd}$$

Fig. 8. Sample span

Colimits of spans. A *cospan completion* of n-ary span $(H, p_1..p_n)$ is an n-ary cospan $(C, q_1..q_n)$ over the same feet (see Fig. 8). We call the entire diagram formed by n legs and n colegs an *n-ary diamond*, and write $D = (S, C)$ for such diagrams. A cospan completion is *commutative* if the corresponding diamond is commutative: $p_1; q_1 = \ldots = p_n; q_n$. As a rule, we will only deal with commutative cospan completions and call them simply completions to ease wording.

A cospan completion C of S is called a *colimit* of S if it is *universally minimal* amongst the class of all (commutative) cospan completions. Universal minimality means that for any other completion C', there is one and only one arrow $!\colon C \to C'$ such that the entire diagram commutes: $q_i; ! = q'_i$. In this case, we also call C a *colimit completion* of S, and diamond $D = (S, C)$ a *colimit diamond.*

It can be shown that if at least one colimit completion exists, then all such completions are canonically isomorphic. Hence, colimit completion can be considered an operation up to isomorphism: it takes a span S as input and produces its cospan completion denoted by ΣS — we choose one amongst the class of isomorphic completions.

Informally, the idea is that we first disjointly merge the feet of the span, and then glue together those elements that are related via the head and legs of the span. For the example in Fig. 8, if nodes are sets and arrows are functions, then we glue together elements $f_i \in F_i$, $i = 1, 2, 3$ if $p_i(h) = f_i$ for some $h \in H$; this gives us an element in the colimit ΣS. Since projections are not necessarily injections, colimit computation maybe be more complex but it can be shown that if $\boldsymbol{C}$ is the category of graphs then any span has colimit. Moreover, if all legs are injective graph mappings, all colegs of ΣS are also injections. In addition, colegs of the colimit jointly cover the cohead, i.e., any element in the cohead is in the range of at least one coleg.

Because colimit of a span is, basically, a disjoint union of its feet factorized by some equivalence relation determined by the head (together with projections), we will often denote the head of the colimit cospan ΣS by $\uplus\mathcal{F}/S$ with $\mathcal{F}$ denoting the family of S's feet, and say that this head is the result of merging $\mathcal{F}$ modulo the correspondence S.

Partial spans. Informally, a *partial span* is a span some of whose legs may be partially defined mappings. Though intuitively clear, a precise formal explication of the notion of partial mapping needs some work.

A *partial mapping* $m\colon A \rightharpoonup B$ between graphs A, B is a triple (i_m, D_m, f_m) specified by diagram $A \xleftarrow{i_m} D_m \xrightarrow{f_m} B$ with $D_m \subset A$ a graph called the *domain* of m (while A is the *source* of m), i_m is the inclusion mapping, and f_m a total graph mapping (the *function* of m). (Actually this definition works for any category in which the notion of sub-object and inclusion is defined). A total mapping is a particular case for which $D_m = A$ and $i_m = 1_A$. A partial mapping m is *injective* if its function f_m is injective.

$$\begin{tikzcd}
 & H \arrow[dl, "m_1"'] \arrow[d] \arrow[dr, "m_3"] & \\
F_1 & F_3 & F_2
\end{tikzcd}$$

(a) a partial span S

$$\begin{tikzcd}
 & \underline{H^{\Pi}_{S'}} \arrow[dl, dashed, "\dot{i}^{\Pi}_1"'] \arrow[dr, dashed, "\dot{i}^{\Pi}_2"] & \\
D_1 \arrow[dd, "f_1"'] \arrow[ddr, "\dot{i}_1"] \arrow[dr, dashed, "\dot{i}^{\Sigma}_1"] & & D_2 \arrow[dd, "f_2"] \arrow[ddl, "\dot{i}_2"'] \arrow[dl, dashed, "\dot{i}^{\Sigma}_2"'] \\
 & \underline{H^{\Sigma}_{S'}} \arrow[d, "!_{S'}"] & \\
F_1 & H & F_2
\end{tikzcd}$$

(b) computing overlaps

Colimits of partial spans are intricate. We first need to compute limits (intersections) of the domains, and then proceed with the colimit procedure (see the inset figure). After all, we hide all intricacies inside of implementation and for the user partial legs and partial spans appear as just subtypes of their total counterparts.

Multispans. The notion of multispan aims to generalize examples considered in [1]. The goal is to accurately specify the idea that there can be multiple correspondences over the same set of models.

Briefly, a *multi-span* is a set of (total and partial) spans sharing their feet. In more detail, let $\mathcal{F}$ be a set of objects (in a category $\boldsymbol{C}$ carrying the diagrams). A *multi-span* over $\mathcal{F}$ is a finite set $\mathcal{S}$ of spans with $\mathsf{feet}S \subset \mathcal{F}$ for all $S \in \mathcal{S}$. Hence,

$\mathsf{feet}\mathcal{S} \overset{\text{def}}{=} \bigcup\{\mathsf{feet}S\colon\ S \in \mathcal{S}\} \subset \mathcal{F}$ but equality is not required, that is, there may be elements in $\mathcal{F}$ not occurring in any span. We write $\mathsf{Feet}\mathcal{S}$ for $\mathcal{F}$ and $\mathsf{Feet}_0\mathcal{S}$ for $\mathsf{Feet}\mathcal{S} \setminus \mathsf{feet}\mathcal{S}$. (For unification, we may consider set $\mathsf{Feet}_0\mathcal{S}$ as a discrete span S_0 without head and include it into $\mathcal{S}$; then $\mathsf{feet}\mathcal{S} = \mathsf{Feet}\mathcal{S} = \mathcal{F}$).

In multimodeling, different spans usually have different heads and legs, $H^S \neq H^{S'}$ and $\mathsf{legs}S \cap \mathsf{legs}S' = \emptyset$ for $S \neq S'$ (as shown in Fig. 9). We write $\mathsf{Heads}\mathcal{S}$ and $\mathsf{Legs}\mathcal{S}$ for sets $\{H^S\colon\ S \in \mathcal{S}\}$ and $\{l\colon\ l \in \mathsf{legs}S, S \in \mathcal{S}\}$ resp. Multi-span is called *discrete* if set $\mathsf{Heads}\mathcal{S}$ (and hence $\mathsf{Legs}\mathcal{S}$) is empty.

$$\begin{tikzcd}
 & & H_2 \arrow[dl] \arrow[d] & H_1 \arrow[dl] \arrow[d] \arrow[dr] & \\
F_5 \arrow[ddrr, dashed] & F_4 \arrow[dr, dashed] & F_3 \arrow[d, dashed] \arrow[dr, dashed] & F_2 \arrow[d, dashed] & F_1 \arrow[dl, dashed] \\
 & & C_2 \arrow[d, dashed] & C_1 \arrow[dl, dashed] & \\
 & & \Sigma^! & &
\end{tikzcd}$$

Fig. 9. Sample multispan

Lemma 3. *For any pair $H \in \mathsf{Heads}\mathcal{S}$ and $F \in \mathsf{Feet}\mathcal{S}$ there is at most one leg $l_{HF}\colon H \to F$ in $\mathsf{Legs}\mathcal{S}$.*

In more complex situations, we may have spans over spans, that is, a foot of a span may be the head of another span. Then we may have several arrow paths between the same head and foot. In such cases we require commutativity, i.e., equality of arrow compositions along the paths. Then the conclusion of the lemma restores.