GSDLAB TECHNICAL REPORT

From Lenses to Tiles: Model Synchronization via Double Categories

Zinovy Diskin

GSDLAB–TR 2009–12–15 December 2009

Generative Software Development Lab

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

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

From Lenses to Tiles: Model Synchronization via Double Categories [⋆]

Zinovy Diskin

Generative Software Development Lab,
University of Waterloo, Canada
zdiskin@gsd.uwaterloo.ca

Abstract. The paper describes a novel mathematical framework for model synchronization. It is based on diagram operations and can be considered an essential generalization of *lenses*—a popular algebraic approach to the view update problem. A distinctive feature of the framework is that both view and update mappings are first class citizens and explicitly occur in arity shapes of the operations of update translation. We show that lenses augmented with mappings give rise to double categories and discuss the benefits of this formalism.

1 Introduction

The task of synchronizing two heterogeneous artifacts is ubiquitous in software engineering. It appears in many contexts of data synchronization and conversion (e.g., HTML publishing, data warehousing, and object-relational mapping), in forward, reverse and roundtrip engineering, and in other scenarios of bidirectional model transformations [4].

Two synchronized heterogeneous artifacts often have some common abstraction—a *view*—in-between them. We may specify this case with the formula $B_1 \succ A \prec B_2$, where $\prec$ denotes the "viewOf" relationship. For example, B_1 could be an internal data format, B_2 an external layout, and A a common view free from details of both internal and layout representations. Or B_1 could be a complex behavioral model, A its structural projection, and B_2 a richer structural model. There are also many "half-way" situations $B \succ A$. For example, A is a reverse engineered model of code B, or B is a pretty-printing format of code A, or B is a database and A is its materialized view. We will use the terms *view maintenance* or *view synchronization* as generic names for the task of synchronization over a"viewOf" relationship.

Examples above show that studying view synchronization is important either itself or as a part of a general theory of synchronization. Algebraic approaches to the problem have a long history, from early work on the view update problem in databases [1], to Meerten's work on constraint maintenance for user interfaces

[⋆] A shorter conference version of this report was prepared, together with Michał Antkiewicz and Krzysztof Czarnecki, for FASE'2010. It was rejected by the PC

[16], to a recent series of works on the so called *lenses*. The latter is a crisp name for a pair of functions realizing view computation and backward update propagation in a coordinated way (the notion and the name were probably conceived in [17]). The lens framework enjoys well developed algebraic foundations [10, 19, 8], and has guided design of several successful DSLs for programming data synchronization for tree-based [10], string-based [2], and relational data [3].

Applications of lenses to software model synchronization expose some deficiencies of the framework. We will show in the paper that model synchronization requires accurate elementwise specifications of relationships between the base (B) and the view (A) models—a *view mapping*, and between the original and the updated models—an *update mapping*. Neither of these mappings are included into the lens formalism, however. At the same time, concrete lenses for trees and tables described in the literature implicitly employ such mappings: the mappings are set by default via name coincidence (or other available keys). For simple data structures like trees or tables, this implicit coordination mechanism may work well, but for semantically rich software models the mappings should be specified explicitly. Thus, software model synchronization requires the lens framework to be extended beyond the simple state-based setting and be augmented with view and update mappings. Further, since view mappings between models are indexed by the corresponding view definition mappings between metamodels, the universe of metamodels and metamodel mappings appears as an important part of the synchronization landscape. This universe is missing from the lens formalism and should be introduced to handle synchronization of software models.

Inserting mappings into lenses entirely changes their algebraic foundation. We leave the world of ordinary algebra (operations act on tuples) and come to the world of diagram algebra, in which operations act on and produce graphs (of models and model mappings in our context). It turns out that all necessary diagram operations act within a square-shaped graph, that is, have a part of the square as their input and the rest of the square as their output. We call these squares *tiles* (another crisp name for a formal construct we borrowed from [11]). We use tile operations to define a series of formal constructs modeling different aspects of view maintenance as shown in Fig. 1 (with arrows indicating generalization). We show that forward-, backward- and bi-maintainable view systems can be specified as *double categories*—an immediate two-dimensional generalization of ordinary categories, which enjoy two sorts of composition working together in a coherent way. Since compositionality is of primary importance for applications, the double category view on synchronization provides a useful algebraic framework. We also show that state-based view systems are equivalent to systems of sequentially composable very well-behaved lenses. It is an expected result but note that forward- and bi-maintainable systems live outside

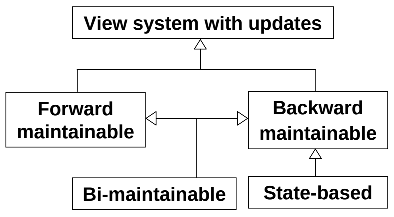


Fig. 1: Hierarchy of view systems

the state-based ones. Hence, the lens formalism covers only a part of a bigger synchronization landscape.

Relation to other work. Algebraic approaches to synchronization follow lenses in using the ordinary algebra framework [20, 19], or use category theory in the *operational* setting of TGG [18]. A series of works by Johnson and Rosebrugh [13] on the view update problem is also based on category theory but their goal is different from ours: they study conditions under which a reasonable view update policy can be inferred from the view definition. In contrast, we follow the lens idea that an update policy is specified by an operation put related to but not inferred from the view definition. As far as we know, applications of double categories are novel for the view synchronization problem. We have used double categories in our study of model versioning [9], where horizontal arrows were matches between homogeneous models rather than view mappings.

The rest of the paper is structured as follows. The next section presents two brief overviews: (i) lenses for the view update problem and (ii) basic definitions of category, double category and double functor, and notational conventions we use. Section 3 describes an example showing the deficiency of the lens framework and roughly outline the diagram algebra reformulation of lenses. The technical development is in Section 4. A brief final discussion of the results is in Section 4.5.

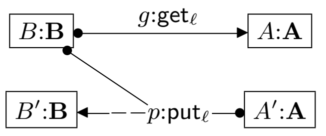
2 Background

Given a mapping (function) $f\colon X \to Y$, we write the value of f at argument x as either $f(x)$ or $(x)f$ or $x.f$ or $f.x$ choosing notation that makes formulas easier to read. A pair $(x, x.f)$ is called an *(application) instance* of f. For a binary mapping $f\colon X_1 \times X_2 \to Y$, an application instance is a triple $[x_1, x_2, (x_1, x_2)f]$.

2.1 Model synchronization and lenses

Suppose we have a space of base (source) models $\mathbf{B}$, a space of view (target) models $\mathbf{A}$, and a function $\mathsf{get}\colon \mathbf{B} \to \mathbf{A}$ ("getView"). Since views abstract away some data, there may be many different bases $B_1, B_2, ... \in \mathbf{B}$ having the same view $A = \mathsf{get}(B_i)$, $i = 1, 2, ...$ Hence, when a view A is updated to a new state A', propagating this update back to the base, i.e., finding B' such that $\mathsf{get}(B') = A'$, is not unique (or is not possible at all when the updated view goes beyond the set of legal views; not considered in this paper).

Thus, some *update policy* is required to choose one base state B' among the multitude of states whose views equal to A'. The lens framework assumes that such a policy amounts to a function $\mathsf{put}\colon \mathbf{A} \times \mathbf{B} \to \mathbf{B}$, which takes the updated view and the original base state and outputs an updated base



(a) Individual models

(GetPut) $(B.\mathsf{get}_\ell, B).\mathsf{put}_\ell = B$
(PutGet) $(A', B).\mathsf{put}_\ell.\mathsf{get}_\ell = A'$
(PutPut) $(A'', (A', B).\mathsf{put}_\ell).\mathsf{put}_\ell$
$= (A'', B).\mathsf{put}_\ell$

(b) Equational laws

Fig. 2: Lens operations and laws

state B' as shown in Fig. 2a: arrows g and p
denote application instances of get_ℓ and put_ℓ,
resp., (index ℓ indicates the lens ℓ). A pair of functions (get_ℓ,put_ℓ) coordinated in a suitable way is called a *lens*.

Definition 1 (adapted from [10]) *A* well-behaved lens *ℓ is set by the following data. (i) Two sets, $^\bullet\ell$ and $\ell^\bullet$, called the domain and codomain of the lens. (ii) Functions $\mathsf{get}_\ell\colon {}^\bullet\ell \to \ell^\bullet$ and $\mathsf{put}_\ell\colon \ell^\bullet \times {}^\bullet\ell \to {}^\bullet\ell$, such that the laws (GetPut) and (PutGet) in Fig. 2b hold for any $B \in {}^\bullet\ell$ and $A' \in \ell^\bullet$. (iii) A well-behaved lens is called* very well-behaved*, if the (PutPut) law holds as well.*[1]

Compositionality is at the heart of lenses' applications to practical problems. Writing correct bidirectional transformation for complex views is laborious and error-prone. To manage the problem, a complex view is decomposed into a sequence of simple blocks $B \succ A_1 \succ A_2$... (views of views of views $\dots$), for each of which a correct lens can be found in a repository. A lens-based DSL provides the programmer with a number of operators of lens composition. One major such operator is sequential composition (for chains of views as above), and a fundamental result states that sequential composition of correct lenses is also a correct lens.

Construction 2 (Lens' composition [10]) *Given lenses ℓ_1 and ℓ_2 with $\ell_1^\bullet = {}^\bullet\ell_2$, the lens $\ell = \ell_1;\ell_2$ is defined as follows. $^\bullet\ell = {}^\bullet\ell_1$ and $\ell^\bullet = \ell_2^\bullet$. For any $B \in {}^\bullet\ell$, $B.\mathsf{get}_\ell = B.\mathsf{get}_{\ell_1}.\mathsf{get}_{\ell_2}$, and for any pair $(A', B) \in \ell^\bullet \times {}^\bullet\ell$, $(A', B).\mathsf{put}_\ell = (A', B.\mathsf{get}_{\ell_1}).\mathsf{put}_{\ell_2}.\mathsf{put}_{\ell_1}$. It is easy to check that ℓ is a (very) well-behaved lens as soon as both lenses ℓ_i are such.*

2.2 Categories, double categories, functors, and double functors

A *graph* G consists of a set of *nodes* G_0 and a set of *arrows* G_1 together with two mappings $\partial_i\colon G_1 \to G_0$, $i = 0, 1$. As usual, we write $a\colon N \to N'$ if $\partial_0(a) = N$ and $\partial_1(a) = N'$. Sometimes we write $a\colon\ N \to N' :: G$ to remind that a is an arrow in graph G. Two arrows a, b are called *parallel* if $\partial_i(a) = \partial_i(b)$, $i = 0, 1$.

A *category* $\boldsymbol{C}$ is a graph with (i) an associative operation of arrow composition denoted by ; (semi-colon), and (ii) a unary operation $\mathbf{1}\_$ that assigns to every node N an *identity/idle* loop $\mathbf{1}_N\colon N \to N$. These loops are units of the composition: $\mathbf{1}_{\partial_0(a)}; a = a = a; \mathbf{1}_{\partial_1(a)}$. Nodes in a category are called *objects*, and arrows are *morphisms.* The class of objects $\boldsymbol{C}_0$ can be also considered as a category, whose only arrows are identities (called *discrete category*).

A *functor* $f\colon \boldsymbol{C} \to \boldsymbol{C}'$ is a pair of functions $f_0\colon \boldsymbol{C}_0 \to \boldsymbol{C}'_0$, $f_1\colon \boldsymbol{C}_1 \to \boldsymbol{C}'_1$ that preserves incidence relations between nodes and arrows ($a.f_1.\partial_0{=}a.\partial_0.f_0$, etc.), arrow composition $(a;b).f_1{=}(a.f_1);(b.f_1)$ and idle loops $f_1(\mathbf{1}_N){=}\mathbf{1}_{N.f_0}$.

[1] We do not consider function $\mathsf{create}\colon \ell^\bullet \to {}^\bullet\ell$

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "a"'] \arrow[dr, phantom, "\searrow^{T}"] & B \arrow[d, "b"] \\
A' \arrow[r, "f'"'] & B'
\end{tikzcd}$$

A *double category* $\mathbb{D}$ is a four-sorted algebraic structure comprising a class $\mathbb{D}_0$ of *nodes (* also called *0-cells)*, classes $\mathbb{D}_1^{\mathsf{h}}$ of *horizontal* and $\mathbb{D}_1^{\mathsf{v}}$ of *vertical arrows (1-cells)*, and a class $\mathbb{D}_2$ of *squares* or *tiles (2-cells)* subject to incidence (source and target) conditions shown by the diagram on the right (that is, the vertical source of tile T is arrow f, the horizontal target is b etc.). Below we abbreviate *horizontal* and *vertical something* to *h-something* and *v-something.*

These data satisfy the following requirements. Nodes and h-arrows form a category denoted, again, by $\mathbb{D}_1^{\mathsf{h}}$, and similarly, we have a category $\mathbb{D}_1^{\mathsf{v}}$ so that $(\mathbb{D}_1^{\mathsf{h}})_0 = (\mathbb{D}_1^{\mathsf{v}})_0 = \mathbb{D}_0$. H-arrows considered as nodes and tiles as arrows between them form a category $\mathbb{D}_2^{\mathsf{v}}$ (in which tiles are composed vertically); v-arrows as nodes and tiles as arrows between them form category $\mathbb{D}_2^{\mathsf{h}}$ (in which tiles are composed horizontally). Thus, both categories have the same set of arrows, $(\mathbb{D}_2^{\mathsf{h}})_1 = (\mathbb{D}_2^{\mathsf{v}})_1 = \mathbb{D}_2$, but compositions are different and denoted by $;^{\mathsf{h}}$ and $;_{\mathsf{v}}$, respectively. We will often omit the subindex if no confusion arises.

The fact that $\mathbb{D}_2^{\mathsf{x}}$, x=h,v, are categories means that there are h-idle and v-idle tiles, which are the units of the respective compositions. Horizontal edges of h-idle tiles must be idle h-arrows; same holds for v-idle tiles. Finally, horizontal and vertical tile compositions are coordinated between themselves by an interchange law (see [14])[2].

A *double functor* $\boldsymbol{f} : \mathbb{D} \to \mathbb{E}$ is a quadruple of functions $\boldsymbol{f}_0, \boldsymbol{f}_1^{\mathsf{v}}, \boldsymbol{f}_1^{\mathsf{h}}, \boldsymbol{f}_2$, sending i-cells in $\mathbb{D}$ to respective i-cells in $\mathbb{E}$ ($i = 0, 1^{\mathsf{v}}, 1^{\mathsf{h}}, 2$), with preservation of all incidence relationships and such that pairs $(\boldsymbol{f}_0, \boldsymbol{f}_1^{\mathsf{x}})\colon \mathbb{D}_1^{\mathsf{x}} \to \mathbb{E}_1^{\mathsf{x}}$ and $(\boldsymbol{f}_1^{\mathsf{x}}, \boldsymbol{f}_2)\colon \mathbb{D}_2^{\mathsf{x}} \to \mathbb{E}_2^{\mathsf{x}}$ (x=v,h) are ordinary functors.

3 Mappings in model synchronization

Arrows in diagram Fig. 2a are tuples of models. However, components of these tuples (models A', B, B') are themselves complex structures consisting of their own elements, and there may be mappings between these structures relating their elements. In this section we consider a simple example showing that such mappings are important for synchronization.

3.1 Example: Do mappings really matter?

Fig. 3 presents a simple example of view synchronization. Models are simple structures consisting of objects with attributes. Attributes are name-and-value pairs. Model element identifiers (eids) use letters P,Q for the objects and a,b,..x,y,z for the attributes. The view is defined as follows: take all Person objects from the base model and remove their age attribute. The base and the view models have disjoint sets of eids because they are stored on different computers. For a while, ignore all the arrows in the diagram.

[2] Appendix reproduces this material for the reviewers' convenience but is excluded from the paper due to space limitations.

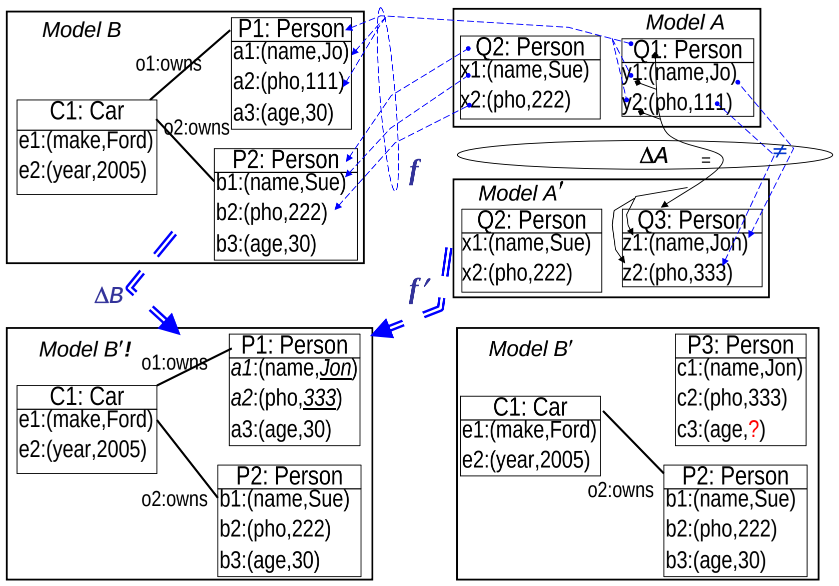


Fig. 3: State-based (B') and mapping-based (B'!) synchronization results

Suppose that model A was updated to a new state A', and we need to propagate the update back to B. In the state-based synchronization setting based on eids, the update from A to A' consists of deletion of object Q1 with its attributes and addition of new object $Q3$ with new attributes. Since the view is merely a projection, the backward propagation is easy: object P1 (corresponding to Q1) is deleted from B, and a new object P3 (corresponding to Q3) is added. The result is model B' placed under model A'. The age-attribute of object P3 is unknown, hence the null value denoted by ? (question mark).

Now suppose that the actual update from A to A' was more complicated. The user first deleted object Q1 but later recognized that was a mistake. The user then created a new object Q3 with the same values of attributes (but, of course, new eids), and later updated the attribute values. For an accurate specification of such an update, we must augment the correspondence based on eids with additional correspondence links Q1→Q3, etc., as shown in the figure by thin curly arrows labeled by =. In addition, two dashed links labeled by $\neq$ show modification of attribute values. Together, all these links constitute a specification called an *update mapping*: it is denoted by arrow $\Delta A\colon A \Rightarrow A'$, whose double-body indicates that the arrow has an extension. To be accurate, we must also explicate the correspondence between the base and the view model as shown by dashed links from model A to B. These links constitute a *view mapping* $f\colon A \Rightarrow B$ (double-body again indicates the existence of extension).

In this and other diagrams below, dashed lines (blue with color display) indicate mappings that are computed (derived). The view mapping is derived because it is computed from a given view definition and base model B—below we will consider this construction in detail. In contrast, the update specification cannot be computed and must be given explicitly (though modification links are, in fact, computed by direct comparison of the values). Having these data, the backward update is as follows. We read the view mapping $f\colon\ A \Rightarrow B$ in the backward direction from B to A and compose it with the update mapping: this will show what remains unchanged and what must be modified in the base model. The result is the state $B'!$ of the base model, which is different from B'. The base update $\varDelta B$ and the updated view mapping f' can be also computed (bold bent arrows in the diagram), and present an important part of the backward update propagation. The latter thus appears as a *diagram operation* $\mathsf{put}_{\boldsymbol{v}}$, whose application instance is shown in Fig. 4c.

Such an instance has a pair of arrows $(\varDelta A, f)$ as its input, and another pair $(\varDelta B, f')$ (shown by dashed lines), completing the square, as its output. Similarly, we define get-operation shown in Fig. 4b: the input is a base update $\varDelta B$ and the output is a triple of arrows completing the square.

Finally, the last ingredient of instance specifications is typing of view mappings f and f' by a metamodel mapping $\boldsymbol{v}$ shown in Fig. 4a. This mapping does not occur in the notion of a lens, and we discuss it in the next section.

$$\begin{tikzcd}
\boldsymbol{\mathcal{N}} & \boldsymbol{\mathcal{M}} \arrow[l, Leftarrow, "\boldsymbol{v}"']
\end{tikzcd}$$

(a) Space of metamodels

$$\begin{tikzcd}
B{:}\boldsymbol{\mathcal{N}} \arrow[d, Rightarrow, "\varDelta B"'] & A{:}\boldsymbol{\mathcal{M}} \arrow[l, Leftarrow, dashed, "f{:}\boldsymbol{v}"'] \arrow[d, Rightarrow, dashed, "\varDelta A"] \arrow[dl, phantom, "g{:}\mathsf{get}_{\boldsymbol{v}}\ \Rightarrow"] \\
B'{:}\boldsymbol{\mathcal{N}} & A'{:}\boldsymbol{\mathcal{M}} \arrow[l, Leftarrow, dashed, "f'{:}\boldsymbol{v}"]
\end{tikzcd}$$

(b) An instance of $\mathsf{get}_{\boldsymbol{v}}$

$$\begin{tikzcd}
B{:}\boldsymbol{\mathcal{N}} \arrow[d, Rightarrow, dashed, "\varDelta B"'] & A{:}\boldsymbol{\mathcal{M}} \arrow[l, Leftarrow, "f{:}\boldsymbol{v}"'] \arrow[d, Rightarrow, "\varDelta A"] \arrow[dl, phantom, "p{:}\mathsf{put}_{\boldsymbol{v}}\ \Leftarrow"] \\
B'{:}\boldsymbol{\mathcal{N}} & A'{:}\boldsymbol{\mathcal{M}} \arrow[l, Leftarrow, dashed, "f'{:}\boldsymbol{v}"]
\end{tikzcd}$$

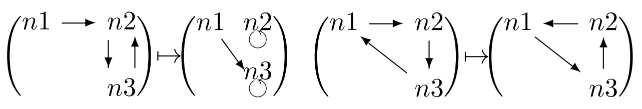
(c) An instance of $\mathsf{put}_{\boldsymbol{v}}$

Fig. 4: Tile operations

3.2 Do metamodels matter?

For lenses, view computation (function get) is merely a data transformation. Models are also data, but these data have precise semantic meaning encoded by the metamodel, and hence, not every models-as-data transformation is semantically meaningful. If $[\![\boldsymbol{\mathcal{M}}]\!]$ and $[\![\boldsymbol{\mathcal{N}}]\!]$ are spaces of models determined by metamodels $\boldsymbol{\mathcal{M}}$ and $\boldsymbol{\mathcal{N}}$, resp., then a transformation $\mathsf{get}\colon [\![\boldsymbol{\mathcal{N}}]\!] \to [\![\boldsymbol{\mathcal{M}}]\!]$ is semantically valid if it is compatible with some semantically meaningful relationship between $\boldsymbol{\mathcal{M}}$ and $\boldsymbol{\mathcal{N}}$. For view computation, such a relationship should be a certain *view specification* $\boldsymbol{v}$, which somehow defines metamodel $\boldsymbol{\mathcal{M}}$ as a view to metamodel $\boldsymbol{\mathcal{N}}$, and we may write $\boldsymbol{\mathcal{M}} \prec_{\boldsymbol{v}} \boldsymbol{\mathcal{N}}$. Specification $\boldsymbol{v}$ and transformation get are different artifacts so that it makes sense to ask whether get is compatible with $\boldsymbol{v}$. In other words, we argue that view *definitions* must be separated from view *computations*: the former live in the world of metamodels while the latter leave in the world of models.

Consider the following example. Let both metamodels define graphs but $\boldsymbol{\mathcal{M}}$-graphs are the following views to $\boldsymbol{\mathcal{N}}$-graphs: given an $\boldsymbol{\mathcal{N}}$-graph B, its $\boldsymbol{\mathcal{M}}$-view A has the same nodes but A-arrows are pairs of sequentially composable B-arrows. Two simple examples of computing such $\boldsymbol{\mathcal{N}}$-views are presented in the diagrams below (where symbol $\circlearrowleft$ denotes loop arrows):



(1) g_1:get (2) g_2:get

In each of the cells (i), i=1,2, we have a pair of graphs (B_i, A_i) considered as instances of function get. Our goal is to define this function syntactically, and Computation of such views can be defined by a metamodel mapping as shown in Fig. 5. To save space, we employ concrete syntax normally used for graphs also for metamodels: an arrow replaces an element together with two (source and target) references to nodes. Composable pairs of arrows (paths of length 2) are formally defined by the following query Q:

$$2Path \overset{\text{def}}{=} \{(x,y)\in Arr\times Arr \mid \partial_1(x) = \partial_0(y)\}\,,\ \partial_0(x,y) \overset{\text{def}}{=} \partial_0(x),\ \ \partial_1(x,y) \overset{\text{def}}{=} \partial_1(y).$$

The query determines a derived arrow 2Path that can be added to metamodel $\boldsymbol{\mathcal{N}}$ as shown in the left upper quadrant of the figure; we denote the augmented metamodel by $Q(\boldsymbol{\mathcal{N}}) \supset \boldsymbol{\mathcal{N}}$. However, the query definition does not complete the view definition. We need to specify how elements of the metamodel $\boldsymbol{\mathcal{M}}$ are interpreted in $\boldsymbol{\mathcal{N}}$. This is done by specifying a mapping $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}}$ that maps metamodel $\boldsymbol{\mathcal{M}}$ to metamodel $Q(\boldsymbol{\mathcal{N}})$, that is, maps elements of $\boldsymbol{\mathcal{M}}$ to either basic or derived elements of $\boldsymbol{\mathcal{N}}$ as shown in the figure.

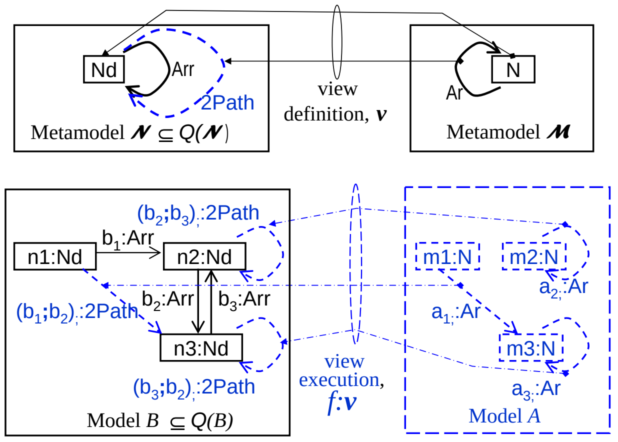


Fig. 5: Example of view definition and execution

The view definition $\boldsymbol{v}$ can be executed for any model B over $\boldsymbol{\mathcal{N}}$. We first execute the query Q for model B and add the computed elements to B: the result is an augmented model $Q(B) \supset B$ typed by the augmented metamodel $Q(\boldsymbol{\mathcal{N}})$ (see the left lower quadrant of the figure). Then we relabel, by reading the mapping backward, the elements of $Q(B)$ whose types (labels) occur in the image of mapping $\boldsymbol{v}$. For example, since arrow (b2;b3) in $Q(B)$ is typed by arrow 2Path in $Q(\boldsymbol{\mathcal{N}})$, and metaarrow Ar from $\boldsymbol{\mathcal{M}}$ is mapped to 2Path, the arrow (b2;b3) is relabeled by Ar. The relabeled elements do not replace the originals, they are copied to a new location where the materialized view is to be stored. The result of all these acts of copying with relabeling will be some $\boldsymbol{\mathcal{M}}$-model A, as shown in the lower right quadrant of the figure. Since each of the A-elements can be traced back to a $Q(B)$-element, we have a mapping $f\colon A \Rightarrow Q(B)$. We can consider this mapping as an "instance" of view definition $\boldsymbol{v}$. Thus, what was a pair of models (B, A) in the lens framework, has become a mapping $f\colon A \Rightarrow Q(B)$, which is a set of pairs of models' elements, in our framework. More details about the construction of view execution and its formalization can be found in [7]. If typing models by metamodels is considered a morphism of the corresponding structures, then relabeling procedure can be formally specified by application of the categorical pullback operation. Details and examples can be found in [6, 7].

Suppose we have a composable pair of view definitions $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}}, \boldsymbol{w}\colon \boldsymbol{\mathcal{N}} \Rightarrow \boldsymbol{\mathcal{O}}$, i.e., $\boldsymbol{v}$ defines a view of the view defined by $\boldsymbol{w}$. These two view mappings can be composed by substituting $\boldsymbol{v}$-query into $\boldsymbol{w}$-query, and we thus obtain a view definition $\boldsymbol{v};\boldsymbol{w}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{O}}$. A normal execution of this definition for a model $C{:}\boldsymbol{\mathcal{O}}$ will execute the view $\boldsymbol{w}$ followed by execution of $\boldsymbol{v}$:

(ExeExe) $\quad \overline{(\boldsymbol{v};\boldsymbol{w})_C} = (\overline{\boldsymbol{w}_C});(\overline{\boldsymbol{v}_B})$ with $B = C\!\upharpoonright_{\boldsymbol{w}}$

4 Maintainable view systems

In this section we will define all notions from Fig. 1 and describe our main results.

4.1 Setting the stage: View systems with updates

Let $\boldsymbol{Modview}_0$ denote a class of objects called *models*. We denote update mappings by ordinary arrows $a\colon A \to A'$ (and thus stop using our previous convention to use double-body for arrows with extension) and we always draw them vertically with the updated model below the original. We denote view mappings by double arrows $f\colon A \Rightarrow B$ and we always draw them horizontally.

We do not make any specific assumptions about the nature of the updates. Whatever they are, they have a source and a target, and consecutive updates can be composed, hence the arrow notation. It is also reasonable to assume that for every model A there is a special *idle update* $\mathbf{1}_A$ that does nothing. We summarize these requirements by saying that we have a category $\boldsymbol{Modupd}$ of *models* and their *updates*. Similarly, we introduce a category $\boldsymbol{Modview}$ of *models* and *view mappings*, as discussed in Section 3.2 (in which an identity arrow is a model considered as the identical view of itself).

Each model is typed over its metamodel, and we write $A{:}\boldsymbol{\mathcal{M}}$ or $A_{\boldsymbol{\mathcal{M}}}$ to explicitly point to the metamodel $\boldsymbol{\mathcal{M}}$. Each view mapping $f\colon A_{\boldsymbol{\mathcal{M}}} \Rightarrow B_{\boldsymbol{\mathcal{N}}}$ is determined by the corresponding *view definition* mapping $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}}$ in the space of metamodels, and we write $f{:}\boldsymbol{v}$. We thus have a typing functor $\tau_{\text{view}}\colon \boldsymbol{Modview} \to \boldsymbol{MMod}$ into the category of metamodels and view definition mappings. As for updates, we assume that for any update $a\colon A \to A'$, its source and target models have the same metamodel, say, $\boldsymbol{\mathcal{M}}$, and then we write $a{:}\mathbf{1}_{\boldsymbol{\mathcal{M}}}$ to indicate that updates do not change the metamodel. Correspondingly, we have another typing functor $\tau_{\text{upd}}\colon \boldsymbol{Modupd} \to \boldsymbol{MMod}_0$ into the discrete category of metamodels.

Our first main observation is that constructs described above can be compactly specified in the language of double categories. We make category $\boldsymbol{MMod}$ a double category $\mathbb{MM}\boldsymbol{od}$ whose horizontal category is $\boldsymbol{MMod}$, the vertical category is discrete $\boldsymbol{MMod}_0$, and squares are diagrams of the shape shown in Fig. 6b.

$$\begin{tikzcd}
B \arrow[d, "b{:}\mathbf{1}_\bullet"'] \arrow[dr, phantom, "\swarrow^{D}"] & A \arrow[l, Leftarrow, "f{:}\boldsymbol{v}"'] \arrow[d, "a{:}\mathbf{1}_\circ"] \\
B' & A' \arrow[l, Leftarrow, "f'{:}\boldsymbol{v}"']
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\bullet \arrow[d, "\mathbf{1}_\bullet"'] \arrow[dr, phantom, "\swarrow^{\mathbf{T}}"] & \circ \arrow[l, Leftarrow, "\boldsymbol{v}"'] \arrow[d, "\mathbf{1}_\circ"] \\
\bullet & \circ \arrow[l, Leftarrow, "\boldsymbol{v}"']
\end{tikzcd}$$

(b)

Fig. 6: View systems via double categories

We also make the collection of models and mappings (both updates and views) a double category $\mathbb{M}\boldsymbol{od}$, whose horizontal category is ***Modview***, the vertical category is ***Modupd*** and squares are diagrams of the shape shown in Fig. 6a. Such squares can be trivially composed horizontally and vertically (because τ_{view} and τ_{upd} are functors), and the required idle squares are diagrams with idle arrows for the either horizontal or vertical sides. All laws of double categories trivially hold. Then typing amounts to a double functor from $\mathbb{M}\boldsymbol{od}$ to $\mathbb{MM}\boldsymbol{od}$, whose horizontal and vertical components are τ_{view} and τ_{upd}.

Definition 3 *A* view system with updates *is a double functor* $\boldsymbol{\tau}\colon \mathbb{M}\boldsymbol{od} \to \mathbb{MM}\boldsymbol{od}$ *with the vertical category* $\mathbb{MM}\boldsymbol{od}_1^{\boldsymbol{v}}$ *being discrete. We denote categories* $\mathbb{M}\boldsymbol{od}_1^{\boldsymbol{h}}$, $\mathbb{M}\boldsymbol{od}_1^{\boldsymbol{v}}$ *and* $\mathbb{MM}\boldsymbol{od}_1^{\boldsymbol{h}}$ *by, resp.,* ***Modview, Modupd*** *and* ***MMod****, and functors* $\boldsymbol{\tau}_1{}^{\boldsymbol{h}}$, $\boldsymbol{\tau}_1{}^{\boldsymbol{v}}$ *by* τ_{view}, τ_{upd} *as above. Given a metamodel* $\boldsymbol{\mathcal{M}}$*, the class of models* $A \in \mathbb{M}\boldsymbol{od}_0$ *such that* $\boldsymbol{\tau}(A) = \boldsymbol{\mathcal{M}}$ *is denoted by* $[\![\boldsymbol{\mathcal{M}}]\!]$*. If we add to this class all vertical arrows (updates) a s.t.* $\boldsymbol{\tau}_1^{\boldsymbol{v}}(a) = \boldsymbol{1}_{\boldsymbol{\mathcal{M}}}$*, we make* $[\![\boldsymbol{\mathcal{M}}]\!]$ *a category (of* $\boldsymbol{\mathcal{M}}$*-models and their updates).*

There is not too much content in this reformulation because compositionality of squares in $\mathbb{M}\boldsymbol{od}$ is trivial and says nothing more than about separate compositionality of h- and v-arrows. Yet it compactly specifies numerous incidence relations, and provides convenient foundation for our further work in the paper.

The definition above says that horizontal arrows in $\mathbb{M}\boldsymbol{od}$ are typed by view definitions (arrows in ***MMod***) but it says nothing about how to, given a view definition $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}}$ and a model $B{:}\boldsymbol{\mathcal{N}}$, execute the view definition for B such that it results in a view model $B\!\upharpoonright_{\boldsymbol{v}}$ together with a view mapping $\overline{\boldsymbol{v}_B}\colon B\!\upharpoonright_{\boldsymbol{v}} \Rightarrow B$. In category theory, such type of operations is well studied in the theory of *fibrations*[12] and is usually called *(Cartesian) lifting*, hence the over-bar notation. Model $B\!\upharpoonright_{\boldsymbol{v}}$ can be understood as the reduction (restriction) of model B to metamodel $\boldsymbol{\mathcal{N}}$ along the view mapping $\boldsymbol{v}$, hence, symbol $\upharpoonright$.

Importantly, lifting is compatible with arrow composition. Suppose we have a composable pair of view definitions $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}}$, $\boldsymbol{w}\colon \boldsymbol{\mathcal{N}} \Rightarrow \boldsymbol{\mathcal{O}}$, i.e., $\boldsymbol{v}$ defines a view of the view defined by $\boldsymbol{w}$. These two view mappings can be composed by substituting $\boldsymbol{v}$-query into $\boldsymbol{w}$-query, and we thus obtain a view definition $\boldsymbol{v};\boldsymbol{w}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{O}}$. A normal execution of this definition for a model $C{:}\boldsymbol{\mathcal{O}}$ executes view $\boldsymbol{w}$ followed by execution of $\boldsymbol{v}$: $\overline{(\boldsymbol{v};\boldsymbol{w})_C} = (\overline{\boldsymbol{w}_C});(\overline{\boldsymbol{v}_B})$ with $B{=}C\!\upharpoonright_{\boldsymbol{w}}$. Of course, $\overline{(\mathbf{1}_{\boldsymbol{\mathcal{M}}})_B} = \mathbf{1}_B$.

An accurate categorical formulation of lifting (can be found in [12]) goes beyond the elementary categorical apparatus we use in the paper. However, we will use lifting a view definition to an actual view execution for only one (but important) result of the paper—Theorem 9. For the rest of the paper, the reader can use simple Definition 3. Theorem 9 requires a more precise definition of view systems with updates:

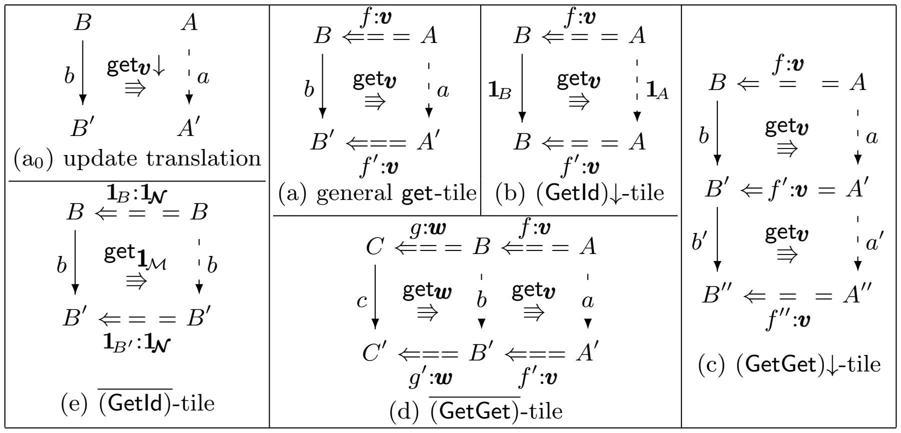

(a$_0$) update translation
(a) general get-tile
(b) (GetId)↓-tile
(c) (GetGet)↓-tile
(d) $\overline{\text{(GetGet)}}$-tile
(e) $\overline{\text{(GetId)}}$-tile

Fig. 7: Diagram operation of incremental view maintenance (a) and its laws (b,c,d,e)

Definition 4 (Definition 3 completed) *A* view system with updates *is a double functor* $\boldsymbol{\tau}\colon \mathbb{M}\boldsymbol{od} \to \mathbb{MM}\boldsymbol{od}$ *with category* $\mathbb{MM}\boldsymbol{od}_1^{v}$ *being discrete and functor* $\boldsymbol{\tau}_1^{h}\colon \mathbb{M}\boldsymbol{od}_1^{h} \to \mathbb{MM}\boldsymbol{od}_1^{h}$ *being a split fibration.*

4.2 Forward maintainable view systems

Let $f\colon A \Rrightarrow B$:: ***Modview*** be a mapping defined by view $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rrightarrow \boldsymbol{\mathcal{N}}$:: ***MMod***, and the base model B is updated. The updated model B' provides an updated view $f'\colon A' \Rrightarrow B'$ computed by executing the same view definition $\boldsymbol{v}$ for model B'. However, it may be expensive to recompute the view A' from scratch, and a better way is to compute an update a to A out of the update b to B. Various algorithms of update translation have been intensively studied by the database community in the area of incremental view maintenance [15]. These algorithms essentially depend on the queries involved in the view definition, and interaction of updates and views may be quite complicated. Our goal in this section is to formulate the most basic laws regulating update translation and their interaction with views in a precise algebraic way. We will begin with ordinary algebra, in which terms are strings, and then reformulate the framework in terms of diagrammatic operations.

Update translation via string-based algebra. Given a view definition $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rrightarrow \boldsymbol{\mathcal{N}}$, we can specify declaratively an algorithm translating updates of base models to updates of their view models by a mapping "getUpdate", $\mathsf{get}_{\boldsymbol{v}}{\downarrow}$, sending base updates $b\colon B \to B'$ to view updates $a = \mathsf{get}_{\boldsymbol{v}}{\downarrow}(b)\colon A \to A'$, as shown in diagram Fig. 7(a$_0$).

These two laws regulate interaction between update translation and update composition:

(GetId)$\downarrow$ $\quad$ $\mathsf{get}_{\boldsymbol{v}}{\downarrow}(\mathbf{1}_B) = \mathbf{1}_A$

(GetGet)$\downarrow$ $\quad$ $\mathsf{get}_{\boldsymbol{v}}{\downarrow}(b; b') = b.\mathsf{get}_{\boldsymbol{v}}{\downarrow}(b); \mathsf{get}_{\boldsymbol{v}}{\downarrow}(b')$

Diagrams Fig. 7(b,c) help to explain the meaning of the laws (consider, for a while, that labels $\mathsf{get}_{\boldsymbol{v}}$ and $\mathsf{get}_{\boldsymbol{v}}{\downarrow}$ denote the same operation and ignore horizontal arrows). The first law is evident: "do nothing with the base" implies "do nothing with the view". The second law says that translation of a sequentially composed update can be composed from translations of the components. This is a non-trivial requirement that may not hold for some views; e.g., in general it does not hold for views defined with the so called non-distributive relational queries [15] (but may still hold for some classes of updates). We leave these cases for future work and below consider "good" views and updates for which both laws hold.

These two laws regulate interaction between update translation and view composition (see diagrams Fig. 7d,e):

$\overline{\text{(GetId)}}$ $\quad$ $b.\mathsf{get}_{(\mathbf{1}_{\mathcal{M}})}{\downarrow} = b$

$\overline{\text{(GetGet)}}$ $\quad$ $b.\mathsf{get}_{(\boldsymbol{v};\boldsymbol{w})}{\downarrow} = b.\mathsf{get}_{\boldsymbol{v}}{\downarrow}.\mathsf{get}_{\boldsymbol{w}}{\downarrow}$

The first one is trivial. The second is very important: when it holds, translation algorithms for complex queries/views can be composed from simpler pieces.

Update translation via diagram algebra. As we discussed in Section 3.2, views come together with backward traceability mappings $f\colon A \Rightarrow B$. It is convenient to include these mappings in the output of operation $\mathsf{get}_{\boldsymbol{v}}{\downarrow}$. It makes the latter a full-fledged diagram operation $\mathsf{get}_{\boldsymbol{v}}$: given a view definition mapping $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}}$, operation $\mathsf{get}_{\boldsymbol{v}}$ takes any arrow $b{:}\mathbf{1}_{\boldsymbol{\mathcal{N}}}$ and produces three arrows forming a square shown in Fig. 7(a) (derived arrows are dashed). We call such a square a get-*tile*. Formally, we introduce a diagram predicate $\mathsf{get}_{\boldsymbol{v}}{}^*$ of square arity, and say that a corresponding quadruple of arrows $T = (b, a, f, f')$ satisfies $\mathsf{get}_{\boldsymbol{v}}{}^*$ if arrows a, f, f' are produced by operation $\mathsf{get}_{\boldsymbol{v}}$ applied to b.[3] To ease notation, we denote both the diagram operation and the predicate by the same symbol.

All our equational laws can be concisely formulated in the tile language. Diagrams (b) and (e) are nothing but the laws (GetId)$\downarrow$ and $\overline{\text{(GetId)}}$. Diagrams (c) and (d) express exactly the laws (GetGet)$\downarrow$ and $\overline{\text{(GetGet)}}$—if we read them as follows: the outer rectangle is a $\mathsf{get}_{\boldsymbol{v}}$-tile as soon as the inner squares are such.

Definition 5 *Let $\boldsymbol{\tau}\colon \mathbb{M}\boldsymbol{od} \to \mathbb{MM}\boldsymbol{od}$ be a view system with updates. A view $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}} :: \boldsymbol{MMod}$ is called* (incrementally) forward-maintainable *if it is assigned with a diagram operation $\mathsf{get}_{\boldsymbol{v}}$ of arity shown in Fig. 7(a) such that the laws shown in diagrams (b) and (c) hold. The view system is called* forward-maintainable *if each view is such and, in addition, the laws specified in diagrams (d,e) hold.*

We will to reformulate this definition in terms of double categories. Hence, we need a correctly defined horizontal composition of get-tiles. We would prefer to consider the vertical source of a $\mathsf{get}_{\boldsymbol{v}}$-tile to be an original base update, and the

[3] Analogously, a binary operation $+$ generates a ternary predicate $+^*$ with $+^*(x, y, z)$ iff $z = x + y$.

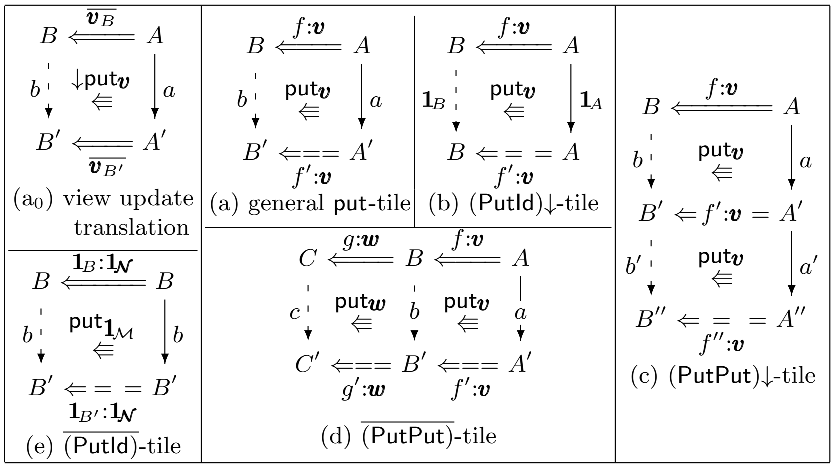


Fig. 8: Diagram operation of incremental view update propagation (a) and its laws (b,c,d,e)

computed view update to be the vertical target. Then according to definition of double category, the horizontal sides of the tile should go in the direction opposite to that shown in diagram Fig. 7(a). Such reversed view mappings are nothing but arrows in the category $\boldsymbol{Modview}^{\mathrm{op}}$, that is, the horizontal category of get-tiles is $\boldsymbol{Modview}^{\mathrm{op}}$ rather than $\boldsymbol{Modview}$. All laws of double categories [14] (see Appendix) immediately follow from get-operation laws in Fig. 7; thus:

Proposition 6 *A forward maintainable view system gives rise to a double category* $\mathbb{G}\boldsymbol{et}$ *whose horizontal category is* $\boldsymbol{Modview}^{\mathrm{op}}$*, vertical category is* $\boldsymbol{Modupd}$*, and squares are get-tiles. Moreover, typing amounts to a double functor* $\boldsymbol{\tau}_{\mathsf{get}}\colon \mathbb{G}\boldsymbol{et} \to \mathbb{MM}\boldsymbol{od}^{\mathrm{op}}$*, where* $\mathbb{MM}\boldsymbol{od}^{\mathrm{op}}$ *is the double category whose horizontal category is* $\boldsymbol{MMod}^{\mathrm{op}}$*.*

4.3 Backward maintainable view systems

Having the machinery of diagram operations in place, we immediately proceed with formal definitions.

Definition 7 *Let* $\boldsymbol{\tau}\colon \mathbb{M}\boldsymbol{od} \to \mathbb{MM}\boldsymbol{od}$ *be a view system with updates. A view* $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}} :: \boldsymbol{MMod}$ *is called* updatable *or* backward maintainable *if it is assigned with a diagram operation* $\mathsf{put}_{\boldsymbol{v}}$ *of arity shown in Fig. 8(a) such that the laws shown in diagrams (b) and (c) hold (dashed arrows are derived). A view system with updates is called* updatable *or* backward maintainable *if each view is updatable and, in addition, the laws specified in diagrams (d,e) hold.*

Note that any updatable view does behave well w.r.t. the PutGet-law of lenses by the very arity shape of operation put (diagram (a)). The GetPut-law is enforced by diagram (b). The diagram (c) (to be read as "if the two inner squares are put-tiles, then the outer one is a put-tile as well) is an analog of the

PutPut-law. To set a more precise relationship between updatable views and lenses, we need a more refined notion of view system with updates.

Definition 8 *An updatable view $\boldsymbol{v}$ is called* state-based *if $\mathsf{put}_{\boldsymbol{v}}$-operation translates any two parallel view updates $a_1, a_2\colon A \Rightarrow A'$ into two parallel base updates $b_1, b_2\colon B \Rightarrow B'$.*

Theorem 9 **(i)** *Any state-based updatable view $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}}$ generates a well-behaved lens $\ell = \ell(\boldsymbol{v})$ by setting ${}^{\bullet}\ell = [\![\boldsymbol{\mathcal{M}}]\!]$, $\ell^{\bullet} = [\![\boldsymbol{\mathcal{N}}]\!]$, $\mathsf{get}_{\ell}(B) = B{\restriction}_{\boldsymbol{v}}$ and $\mathsf{put}_{\ell}(A', B) = \partial_1[{\downarrow}\mathsf{put}_{\boldsymbol{v}}(a, \overline{\boldsymbol{v}_B})]$ for an arbitrary $a\colon B{\restriction}_{\boldsymbol{v}} \to A'$ (operations ${\restriction}_{\boldsymbol{v}}$ and $\overline{\boldsymbol{v}}$ were defined in Section 4.1).* **(ii)** *Moreover, horizontal composition of state-based updatable views provides sequential composition of lenses: $\ell(\boldsymbol{w}) = \ell(\boldsymbol{v}_1); \ell(\boldsymbol{v}_2)$ for any composed view $\boldsymbol{w} = \boldsymbol{v}_1; \boldsymbol{v}_2$.*

Proof. (i) Since the view is state-based, definition of put_{ℓ} does not depend on the choice of a. The lens laws Definition 1 easily follow from get- and put-tiles laws Fig. 7 and Fig. 8. To wit: lens's (GetPut) follows from (GetId)$\downarrow$ and (PutId)$\downarrow$ laws for tiles (diagrams (b) in Fig.7,8) . Lens's (PutGet) is guaranteed by the very arity shape of put-tiles. Lens's (PutPut) holds owing to tile's (PutPut)$\downarrow$ Fig. 8(c).

(ii) Use $\overline{\text{(PutPut)}}$ in Fig. 8(d) $\square$

We now reformulate the notion of updatable view system in terms of double categories.

Proposition 10 *An updatable view system gives rise to a double-category $\mathbb{P}\boldsymbol{ut}$ whose horizontal category is **Modview**, the vertical category is **Modupd**, and squares are put-tiles. Moreover, typing amounts to a double-category morphism $\boldsymbol{\tau}_{put}\colon \mathbb{P}\boldsymbol{ut} \to \mathbb{MM}\boldsymbol{od}$.*

The proof is simple: the double category laws follow from put-operation laws in Fig. 7.

4.4 Bi-maintainable view systems

Definition 11 *Let $\boldsymbol{\tau}\colon \mathbb{M}\boldsymbol{od} \to \mathbb{MM}\boldsymbol{od}$ be a view system with updates. A view $\boldsymbol{v}\colon \boldsymbol{\mathcal{M}} \Rightarrow \boldsymbol{\mathcal{N}} :: \boldsymbol{MMod}$ is called* bi-maintainable *if it is both forward- and backward maintainable (Definitions 5, 7) and, in addition, the following (PutGet)-law holds: applying operation $\mathsf{get}_{\boldsymbol{v}}$ to any put-tile T (i.e., a tile produced by operation $\mathsf{put}_{\boldsymbol{v}}$) does not produce new elements. Formally,*

(PutGet) $\quad$ $\mathsf{put}_{\boldsymbol{v}}(T)$ *implies* $\mathsf{get}_{\boldsymbol{v}}(T)$

where $\mathsf{put}(T)$ means that the tile T is a put-tile and the same for $\mathsf{get}(T)$. A view system with updates is called very well bi-maintainable *if every view is such.*

We remind that the lens PutGet-law is captured in the notion of updatable view (Put-tiles). The tile PutGet-law specified above is strictly stronger: it requires that backward translation of views updates be compatible not only with views but with forward update translation as well. Propositions 6 and 10 together with the (PutGet) law entail

Theorem 12 *A very well bi-maintainable view system gives rise to the commutative diagram of double categories and double functors shown on the right (the upper arrow denotes inclusion of double categories with reversing of h-arrows, and the bottom arrow is the identity functor reversing h-arrows):*

$$\begin{tikzcd} \mathbb{P}\boldsymbol{ut} \arrow[r, hook, "\mathrm{op}"] \arrow[d, "\tau_{put}"'] & \mathbb{G}\boldsymbol{et}^{\mathrm{op}} \arrow[d, "\tau_{get}"] \\ \mathbb{MM}\boldsymbol{od} \arrow[r, "\mathrm{op}"] & \mathbb{MM}\boldsymbol{od}^{\mathrm{op}} \end{tikzcd}$$

4.5 Discussion

Note the conceptual transparency of the last diagram. Powerful two-dimensional composition machinery is packed into the fact that nodes are double categories and arrows are double functors. The upper inclusion is nothing but the (PutGet) law w.r.t. update translation, and reversal of arrows (note the index 'op') indicates that operations get and put work in the opposite directions. Finally, typing functors and commutativity show that all operations are compatible with typing. Briefly, diagram operations of get and put exhibit good composition properties, and are mutually semi-inverse: we have the (PutGet) law but (GetPut) does not hold. This compact presentation shows that double categories can be convenient for building a sound algebraic theory of model synchronization.

The framework can be also useful in practice. Indeed, two-dimensional compositionality is of primary importance for applications. It allows us to decompose complex updates over complex views into elementary pieces and then build the entire view synchronization task from simple blocks. Correctness of "playing synchronization lego" in the double category framework is guaranteed by an important result called *Pasting Lemma*. Roughly, it says that the result of tiling together squares in a double category does not depend on the order in which composition is evaluated [5].

5 Conclusion

We presented a framework for view synchronization based on diagram operations and tiles. The framework extends lenses with metamodels and view and update mappings, supporting both forward and backward update incrementality. The framework is based on double categories, which a) concisely and naturally encode conditions to be satisfied by mappings and diagram operations, b) guarantee certain desirable properties of synchronization systems, and c) allow for formal reasoning about complex synchronization scenarios. In future work, we will integrate the presented view-update tiles with homogeneous synchronizer tiles [9] into a comprehensive framework for heterogeneous synchronization.

References

1. F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6(4):557–575, 1981.
2. A. Bohannon, J. N. Foster, B. Pierce, Al. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, pages 407–419, 2008.

3. A. Bohannon, B. Pierce, and J. Vaughan. Relational lenses: a language for updatable views. In *PODS*, 2006.
4. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, 2009.
5. R. Dawson and R. Pare. General associativity and general compostion for double categories. *Cahiers de topologie et géométrie différentielle catégoriques*, 34:57–79, 1993.
6. Z. Diskin. Mathematics of generic specifications for model management. In Rivero, Doorn, and Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 351–366. Idea Group, 2005.
7. Z. Diskin and J. Diengel. A metamodel independent framework for model transformation: Towards generic model management patterns in reverse engineering. In *ATEM*, 2006.
8. Zinovy Diskin. Algebraic models for bidirectional model synchronization. In *MoDELS*, pages 21–36, 2008.
9. Zinovy Diskin, Krzysztof Czarnecki, and Michal Antkiewicz. Model-versioning-in-the-large: Algebraic foundations and the tile notation. In *CVSM*, 2009.
10. J. N. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.
11. Fabio Gadducci and Ugo Montanari. The tile model. In *Proof, Language, and Interaction*, pages 133–166, 2000.
12. B. Jacobs. *Categorical logic and type theory.* Elsevier Science Publishers, 1999.
13. Michael Johnson and Robert D. Rosebrugh. Fibrations and universal view updatability. *Theor. Comput. Sci.*, 388(1-3):109–129, 2007.
14. G.M. Kelly and R. Street. Review of the elements of 2-categories. In *Category Seminar, Sydney 1972/73*, Lecture Notes in Math., 420, pages 75–103, 1974.
15. H. Liefke and S. Davidson. View maintenance for hierarchical semistructured data. In *DaWaK*, pages 114–125, 2000.
16. L. Meertens. Designing constraint maintainers for user interaction. Available from `http://www.kestrel.edu/home/people/meertens/`, 1998.
17. B. Pierce and A. Schmitt. Lenses and view update translation. Technical report, University of Pennsylvania, 2003.
18. Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *ICGT*, pages 411–425, 2008.
19. P. Stevens. Towards an algebraic theory of bidirectional transformations. In *ICGT*, pages 1–17, 2008.
20. Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *ASE*, pages 164–173, 2007.

A Appendix. Graphs, categories and double categories

A *graph* G consists of a set of *nodes* G_0 and a set of *arrows* G_1 together with two mappings $\partial_i \colon G_1 \to G_0$, $i = 0, 1$. As usual, we write $a \colon N \to N'$ if $\partial_0(a) = N$ and $\partial_1(a) = N'$. Sometimes we write $a \colon\ N \to N' :: G$ to remind that a is an arrow in graph G. Two arrows a, b are called *parallel* if $\partial_i(a) = \partial_i(b)$, $i = 1, 2$.

A *category* $\boldsymbol{C}$ is a graph with (i) an associative operation of arrows composition denoted by ;, and (ii) a unary operation that assigns to every node N a special an *identity/idle* loop $\mathbf{1}_N \colon N \to N$. These loops are units of the composition: $\mathbf{1}_{\partial_0(a)}; a = a = a; \mathbf{1}_{\partial_1(a)}$. Nodes in a category are called *objects*, and arrows are *morphisms.* The class of objects $\boldsymbol{C}_0$ can be also considered as a category, whose only arrows are identities. This trivial category is denoted by $\boldsymbol{C}_0$ too.

A *double category* $\mathbb{D}^{\mathsf{s}}_i$ a four-sorted algebraic structure consisting of *nodes, horizontal arrows, vertical arrows* and *squares* or *tiles* subject to incidence (source and target) conditions shown by the diagram below (the vertical source of tile T is arrow f, the horizontal target is b etc.). We abbreviate *horizontal* and *vertical something* to *h-something* and *v-something.*

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "a"] \arrow[dr, phantom, "\searrow^{T}"] & B \arrow[d, "b"] \\
A' \arrow[r, "f'"] & B'
\end{tikzcd}$$

Nodes and h-arrows form a category $\boldsymbol{A}_{\mathsf{h}}$, and nodes and v-arrows form a category $\boldsymbol{A}_{\mathsf{v}}$ (so that $\boldsymbol{A}_{\mathsf{h}}0{=}\boldsymbol{A}_{\mathsf{v}}0$). H-arrows considered as nodes, and tiles as arrows between them, form a category $\boldsymbol{T}_{\mathsf{v}}$ (in which tiles are composed vertically); v-arrows as nodes and tiles as arrows between them form a category $\boldsymbol{T}_{\mathsf{h}}$ (in which tiles are composed horizontally). Thus, both categories have the same set of arrows, $\boldsymbol{T}_{\mathsf{h}}1{=}\boldsymbol{T}_{\mathsf{v}}1$, but compositions are different and denoted by, resp., $;^{\mathsf{h}}$ and $;_{\mathsf{v}}$. We will often omit the subindex if no confusion arises.

The fact that $\boldsymbol{T}_{\mathsf{v}}$ and $\boldsymbol{T}_{\mathsf{h}}$ are categories means that there are h-idle and v-idle tiles, which are the units of the respective compositions. Boundaries (sources and targets) of these tiles must be compatible with idle arrows (both horizontal and vertical). Finally, horizontal and vertical tile compositions are coordinated between themselves by an interchange law. These conditions can be found in, say, [14]and reproduced in Fig. 9.

In more detail, ***Idle tiles*** are formed as shown in Fig. 9(a1,b1), and their formation is compatible with identity arrows Fig. 9(a1+b1) and with tile composition Fig. 9(a2,b2).

Interchange law. For any four tiles incident to each other as shown in Fig. 9(ab)$_3$, equality specified at the bottom of the cell holds.

One of the basic results about double categories is

Pasting Lemma [5]. In any double category having so called *factorization*, composition of compatible tiles in any order gives the same result.

An illustrating example is shown in Fig. 9(d): Pasting lemma states that

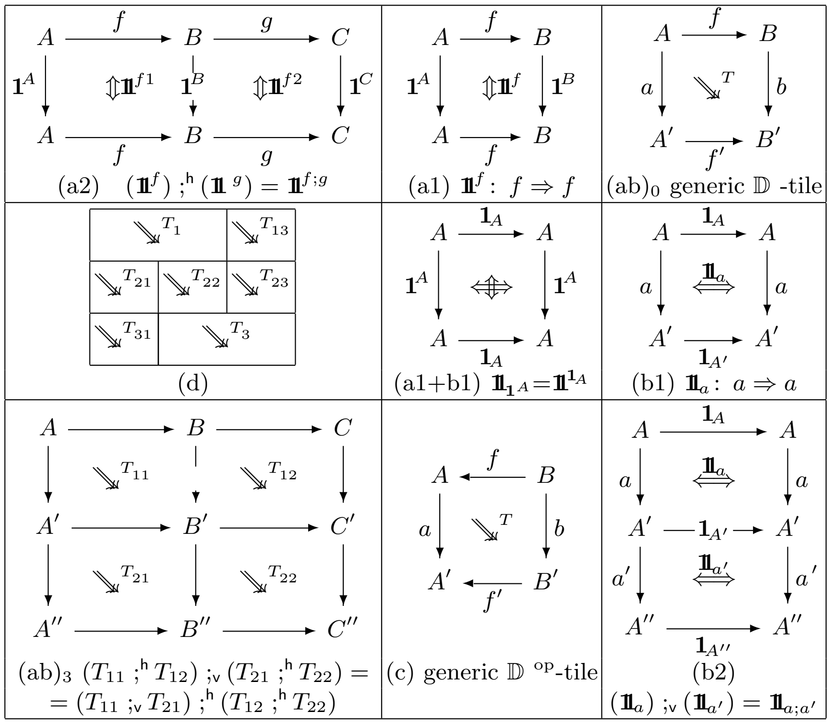

(a2) $(\mathbb{1}^f)\ ;^{\mathsf{h}}(\mathbb{1}^g) = \mathbb{1}^{f;g}$
(a1) $\mathbb{1}^f : f \Rightarrow f$
(ab)$_0$ generic $\mathbb{D}$ -tile
(d)
(a1+b1) $\mathbb{1}_{\mathbf{1}^A} = \mathbb{1}^{\mathbf{1}_A}$
(b1) $\mathbb{1}_a : a \Rightarrow a$
(ab)$_3$ $(T_{11}\ ;^{\mathsf{h}} T_{12})\ ;_{\mathsf{v}} (T_{21}\ ;^{\mathsf{h}} T_{22}) = = (T_{11}\ ;_{\mathsf{v}} T_{21})\ ;^{\mathsf{h}} (T_{12}\ ;^{\mathsf{h}} T_{22})$
(c) generic $\mathbb{D}^{\text{op}}$-tile
(b2) $(\mathbb{1}_a)\ ;_{\mathsf{v}} (\mathbb{1}_{a'}) = \mathbb{1}_{a;a'}$

Fig. 9: Double category definition and laws

$(T_1T_{13})\ ;_{\mathsf{v}} (T_{21}T_{22}T_{23})\ ;_{\mathsf{v}} (T_{31}T_3) = [(T_1\ ;_{\mathsf{v}} T_{21}T_{22})(T_{13}\ ;_{\mathsf{v}} T_{23})]\ ;_{\mathsf{v}} (T_{31}T_3)$, where for to ease reading we omit the symbol of horizontal composition (other ways of composition are possible as well but the result remains the same).