Model Synchronization: Mappings, Tiles, and Categories

Zinovy Diskin

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

Abstract. The paper presents a novel algebraic framework for specification and design of model synchronization tools. The basic premise is that synchronization procedures, and hence algebraic operations modeling them, are *diagrammatic*: they take a configuration (diagram) of models and mappings as their input and produce a diagram as the output. Many important synchronization scenarios are based on diagram operations of square shape. Composition of such operations amounts to their *tiling*, and complex synchronizers can thus be assembled by tiling together simple synchronization blocks. This gives rise to a visually suggestive yet precise notation for specifying synchronization procedures and reasoning about them.

1 Introduction

Model driven software engineering puts models at the heart of software development, and makes it heavily dependent on intelligent model management (MMt) frameworks and tools. A common approach to implementing MMt tasks is to present models as collections of objects, and program model operations as operations with these objects; *object-at-a-time* programming is a suitable name for this style [1]. Since models may contain thousands of interrelated objects, object-at-a-time programming can be very laborious and error-prone. In a sense, it is similar to the infamous record-at-a-time programming in data processing, and has similar problems of being too close to implementation.

Replacing record- by relation-at-a-time frameworks has raised data processing technology to a qualitatively new level in semantic transparency and programmers' productivity. Similarly, we can expect that *model-at-a-time* programming, in which an engineer can think of MMt routines in terms of operations over models as integral entities, could significantly facilitate development of MMt applications [1]. This view places MMt into the realm of algebra: models are indivisible points and model manipulation procedures are operations with them.

Model synchronization tools based on special algebraic structures called *lenses* [2] can be seen as a realization of the algebraic vision. The lens framework was first used for implementing a bidirectional transformation language for synchronizing simple tree structures [3], and then employed for building synchronization

J.M. Fernandes et al. (Eds.): GTTSE 2009, LNCS 6491, pp. 92–165, 2011.

tools for more complex models closer to software engineering practice [4,5]. In [6], a lens-like algebraic structure was proposed to model semantics of QVT, an industrial standard for model transformation.

Lens-based synchronization is *discrete*: input data for a synchronizer consist of states of the models only, while mappings (deltas) relating models are ignored. More accurately, the synchronizer itself computes mappings based on keys and the structure of the models involved. However, in general a pair of models does not determine a unique mapping between them. To compute the latter, some context-dependent information beyond models may be needed, and hiding model mappings inside the tool rather than allowing the user to control them may compromise synchronization. For example, discrete composition of model transformations may be erroneous because in order to be composable, transformations must fit together on both models *and* mappings. In the paper we will consider several examples showing that model (and metamodel) mappings are crucial for model synchronization, and must be treated as first-class citizens not less important than models.

In algebraic terms, the arguments above mean that model mappings must be explicitly included in the arity shapes of MMt operations. A typical MMt universe should appear as a directed graph (nodes are models and arrows are mappings) that carries a structure of *diagrammatic* algebraic operations. The latter act upon configurations (diagrams) of models and mappings of predefined arity shapes: take a diagram as the input and produce a diagram as the output.

The world of diagram algebra essentially differs from the ordinary algebra. A single diagram operation may produce several nodes and arrows that must satisfy certain incidence relationships between themselves and input elements. Composition of such operations, and parsing of terms composed from them, are much more complex than for ordinary tuple-based single-valued operations. Fortunately, we will see that diagram operations appearing in many model synchronization scenarios have a square shape: the union of their input and output diagrams is a square composed of four arrows — we will call it a *tile*. Composition of such operations amounts to their *tiling*, and complex synchronization scenarios become *tiled*. Correspondingly, complex synchronizers can be assembled by tiling together simpler synchronizing blocks, and their architecture is visualized in a precise and intuitive way.

The main goal of the paper is to show the potential of the tile language for specifying synchronization procedures and for stating the laws they should satisfy. Tiles facilitate thinking and talking about synchronization; they allow us to draw synchronization scenarios on the back of an envelope, and to prove theorems about them as well. Specification and design with tiles are useful and enjoyable; if the reader will share this view upon reading the paper, the goal may be considered achieved.

How to read the paper. There are several ways of navigating through the text. The fastest one is given by the upper lane in Fig. 1: rectangles denote sections (of number n)) and arrows show logical dependencies between them.

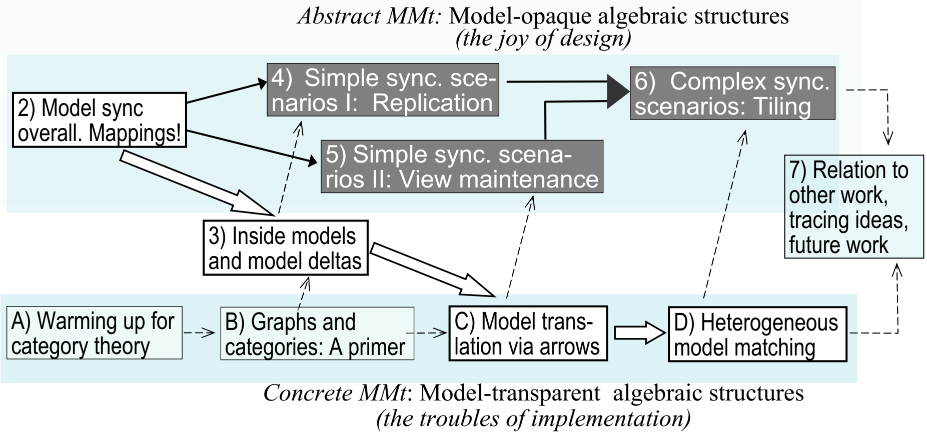

Abstract MMt: Model-opaque algebraic structures
(the joy of design)
2) Model sync overall. Mappings!
4) Simple sync. scenarios I: Replication
5) Simple sync. scenarios II: View maintenance
6) Complex sync. scenarios: Tiling
7) Relation to other work, tracing ideas, future work
3) Inside models and model deltas
A) Warming up for category theory
B) Graphs and categories: A primer
C) Model translation via arrows
D) Heterogeneous model matching
Concrete MMt: Model-transparent algebraic structures
(the troubles of implementation)

Fig. 1. Flow of Content

Section 2 is the beginning of the journey: it draws an overall picture of model synchronization, presents two simple examples (replica synchronization and view maintenance), and argues that mappings are of primary importance. It also warns the reader about the dangers of walking through *the arrow forest* and declares tile algebra and category theory as a means to meet the challenge.

The subsequent three upper sections present abstract algebraic models of the examples from Section 2, and develop them into an algebraic framework based on tiles. Models and model mappings are treated as opaque indivisible nodes and arrows, and synchronization procedures as abstract algebraic operations over them. Two families of such operations are considered for two basic scenarios: replication (Section 4) and view maintenance (Section 5). Section 6 shows how to build complex synchronizers by putting together basic blocks.

The upper three sections can be viewed as a mini-tutorial on building algebraic theories in the diagrammatic setting. We will see how to set signatures of diagram operations, state equational laws, and define diagram algebras intended to model synchronization tools. The goal is to present a toolbox of algebraic instruments and show how to use them; several exercises should allow the reader to give them a try. Except in subsection 6.2, the mathematics employed in the upper lane is elementary (although somewhat unusual).

The upper lane of the paper presents an *abstract* MMt framework: models and mappings are black-boxes without internal structure (hence its notation: black opaque nodes and arrows). This setting can be useful for a top-level architectural design of synchronization tools. A more refined (and closer to implementation) setting is presented in the *concrete* MMt branch of the paper formed by Sections 3,C,D connected by transparent arrows. In these sections we look inside models and mappings, consider concrete examples, and refine the abstract constructs of the upper lane by more "concrete" algebraic models. In more detail, Section 3 factorizes the fast route 2 → 4 (from examples in Section 2 to abstract constructs in Section 4) by providing a formal model for the internal structure of models and model deltas, and for delta composition as well (including deltas with inconsistencies!). Section C refines the fast route $2 \to 5$ into a "concrete" path $2 \to 3 \to C \to 5$ by providing an algebraic model for the view mechanism (also based on tiles); and Section D plays a similar role for Section 6 with a refined model of heterogenous matching.

Both frameworks — abstract and concrete — employ algebraic models and tiling. A principal distinction of the latter is that metamodels and metamodel mappings are explicitly included into algebraic constructs and play an essential role. Indeed, ignoring metamodels and their mappings hides semantic meaning of operations with heterogeneous models from the user and may provoke ad hoc solutions in building MMt-tools. Taking metamodels seriously brings onto the stage an entire new dimension and significantly complicates the technical side of mapping management. Use of category theory (CT) seems unavoidable, and two "concrete" sections C and D require certain categorical intuition and habits of arrow thinking not familiar to the MMt community.[1] Therefore, a special "starter" on CT was written (Sect. A), which motivates and explains the basics of arrow thinking. Section B is merely a technical primer on graphs and categories: it fixes notation and defines several basic constructs employed in the paper (but is not intended to cover all categorical needs). Even though the presentation in Sect. C and D is semi-formal, all together the four lower sections are much more technically demanding than the upper ones, and so are placed in the Appendix that may be skipped for the first reading.

Sections 7 presents diverse comments on several issues considered or touched on in the paper in a wider context. It also briefly summarizes contributions of the paper and their relation to other work. Section 8 concludes. Answers to exercises marked by * can be found on p. 143

A possible reading scenario the author has in mind is as follows. The reader is a practitioner with a solid knowledge of model synchronization, who knows everything presented in the paper but empirically and intuitively. He has rather vague (if any) ideas about diagram algebra and category theory, and is hardly interested in these subjects, yet he may be interested in a precise notation for communicating his empirical knowledge to his colleagues or/and students. He may also be interested in some mathematics that facilitates reasoning about complex synchronization procedures or even allows their mechanical checking. Such a reader would take a look through numerous diagrams in the paper with an approximate understanding of what they are talking about, and hopefully could find a certain parallelism between these diagrams and his practical intuition. Perhaps, he would remember some terms and concepts and, perhaps, would take a closer look at those concepts later on. Eventually, he may end up with a feeling that viewing model synchronization through the patterns of diagram algebra makes sense, and category theory is not so hopelessly abstract.

Now it is the reader's turn to see if this scenario is sensible.

[1] It could explain why many known algebraic approaches to MMt ignore the metamodeling dimension.

2 Model Sync: A Tangled Story

By the very nature of modeling, a system to be modeled is represented by a set of interrelated models, each one capturing a specific 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 essentially dependent on model synchronization.

This section presents *a tale* of model synchronization: we begin with a tangle, then follow it and get to an arrow forest, which we will try to escape by paving our way by tiles.

2.1 The Tangle of Relationships and Update Propagation

The task of model synchronization is schematically presented in Fig. 2. A snapshot of a design project appears as a heterogeneous collection $\mathcal{M}$ of models (shown by nodes $A, B, C...$) interrelated in different ways (edges $r_1, r_2, r_3...$). The diversity of node and edge shapes is a reflection of the diversity of models and the complexity of their mutual relationships that emerge in software design. The image of a tangle in the center of the figure is intentional.

Typically, models in a project's snapshot are only partially consistent, i.e., their relationships partially satisfy some predefined consistency conditions. That is, we suppose that inconsistencies are partially detected, specified and recorded for future resolution. Inconsistency specifications may be considered as part of the intermodel relationships and hence are incorporated into intermodel edges.

Now suppose that one of the models (say, A in the figure) is updated to a new state (we draw an arrow $u_A\colon A \to A'$), which may violate existing consistent relationships and worsen existing inconsistencies. To restore consistency or at least to reduce inconsistency, other related models must be updated as well (arrows $u_B\colon B \to B'$, $u_C\colon C \to C'$ *etc*). Moreover, relationships between models must also be updated to new states r'_i, $i = 1, 2, ...$, particularly by incorporating new inconsistencies. Thus, the initial update u_A is to be *propagated* from the updated model to other related models and relationships so that the entire related fragment ("section" $\mathcal{M}$ of the model space) is updated to state $\mathcal{M}'$. We call this scenario a *single-source update propagation.*

Another scenario is when several models (say, A, B, C) are updated concurrently, so their updates must be *mutually* propagated between themselves and other models and relationships. Such *multi-source* propagation is more complex because of possible conflicts between updates. However, even for single-source propagation, different propagation paths may lead to the same model and generate conflicts; cycles in the relationship graph confuse the situation even more. The relationship tangle generates a propagation tangle.

Propagation is much simpler in the binary case when only two interrelated models are considered. This is a favorite case of theoreticians. For binary situations, multi-source propagation degenerates into *bi-directional* (in contrast to *unidirectional* single-source propagation) — an essential simplification but still

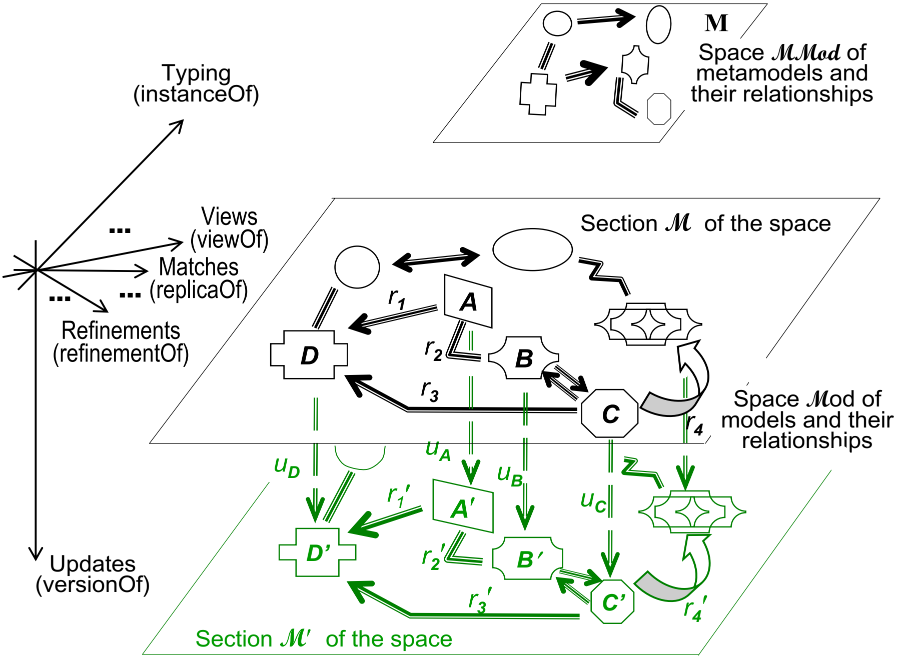

M
Space $\mathcal{MMod}$ of metamodels and their relationships
Typing (instanceOf)
Views (viewOf)
Matches (replicaOf)
Refinements (refinementOf)
Updates (versionOf)
...
Section $\mathcal{M}$ of the space
Space $\mathcal{M}$od of models and their relationships
A
B
C
D
r_1
r_2
r_3
r_4
u_A
u_B
u_C
u_D
A'
B'
C'
D'
r_1'
r_2'
r_3'
r_4'
Section $\mathcal{M}'$ of the space

Fig. 2. Models and their relationships: From a tangle to mD-space

a challenge [7]. Practical situations enjoy a mix of single- and multi-source, uni- and bi-directional propagations. We will generically refer to them as *synchronization procedures*.

The description above shows that understanding intermodel relationships is crucial for design of synchronization procedures, and it makes sense to establish a simple taxonomy. For the binary case, one model in relation to another model may be considered as its

- replica (e.g., a Google replica of a Microsoft Outlook calendar),
- updated version (two versions of the same replica),
- view (a business view of a calendar),
- refinement (an hourly refinement of a daily schedule),
- instance (an actual content of a diary book – the metamodel for the content).

The list could be be extended and gives rise to a family of binary relations $\mathcal{R}_i \subset \mathcal{M}od \times \mathcal{M}od$, $i = 1, 2..$ over the space of models $\mathcal{M}od$. Unfortunately, a more or less complete classification of such relations important for MMt seems to be missing from the literature.

An observation of fundamental importance for model synchronization is that intermodel relationships are not just pairs of models $(A, B) \in \mathcal{R}_i$, they are mappings $r\colon A \Rightarrow B$ linking models' elements. That is, edges in Fig. 2 have extension consisting of links. Roughly, we may think of an edge $r\colon A \Rightarrow B$ as a set of ordered pairs (a, b) with $a \in A$ and $b \in B$ being similar model elements (a class and a class, an attribute and an attribute *etc*). We may write such a pair $\ell{=}(a,b)$ as an arrow $a \xrightarrow{\ell:r} b$ and call it a *link* (note the difference in the bodies of arrows for mappings and links). In the UML jargon, links ℓ are called *instances* of r. In the arrow notation for links as above, the name of the very link ℓ may be omitted but the pointer to its type, $:r$, is important and should be there.

Table 1. Intermodel relationships & mappings

Relationship	Mapping
replicaOf	match
versionOf	update
viewOf	view trc.
instanceOf	typing

Table 1 presents a brief nomenclature of intermodel relations and mappings ('trc.' abbreviates 'traceability'). Normally mappings have some structure over the set of links they consist of, and we should distinguish between a mapping r and its extension $|r|$, i.e., the set links the mapping consists of. Yet we will follow a common practice and write $\ell \in r$ for $\ell \in |r|$. In general, a mapping's extra structure depends on the type of the relationship, and so mappings listed in the table are structured differently and operated differently.

2.2 Mappings, Mappings, Mappings...

In this section we consider how mappings work for synchronization. We will begin with two simple examples. The first considers synchronization of two replicas of a model. In the second, one model is a view of the other rather than an independent replica. Then we will discuss deficiencies of state-based synchronization. Finally, we discuss mathematics for mapping management.

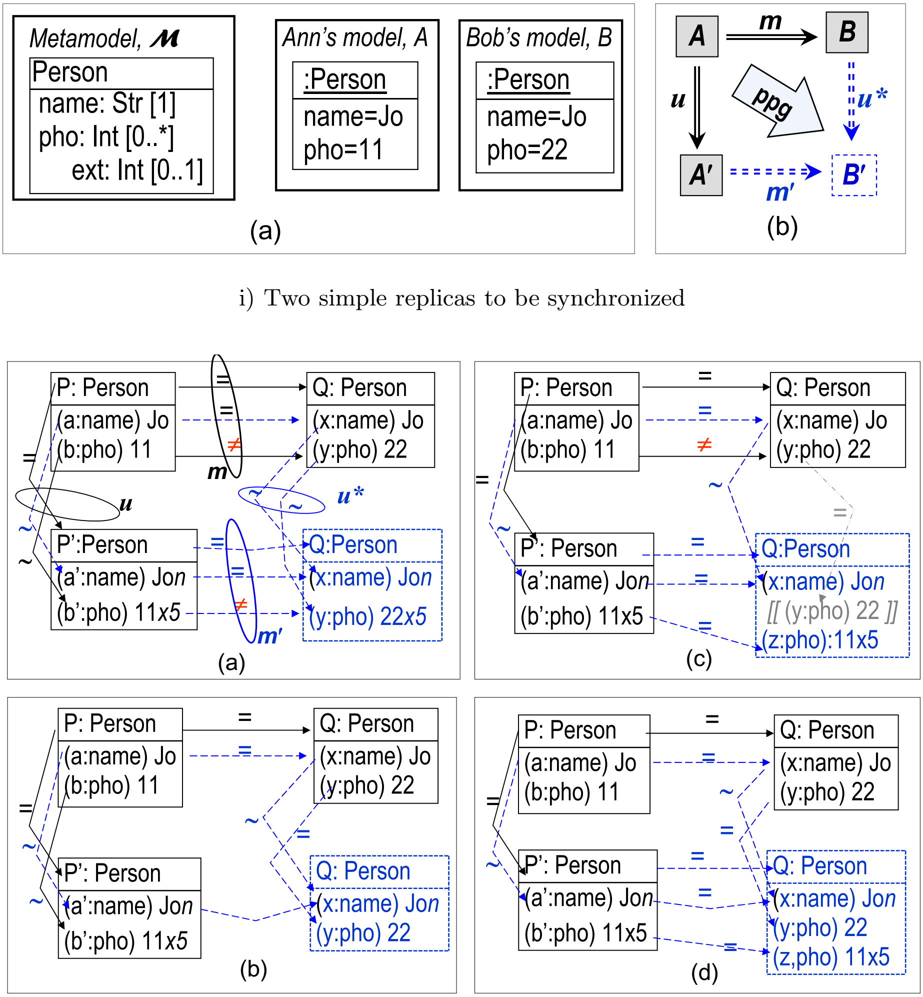
To make tracing examples easier, our sample models will be object diagrams, whose class diagrams thus play the role of metamodels (and the metamodel of class diagram is the meta-metamodel).

2.2.1 Replica Synchronization

Suppose that two developers, Ann and Bob, maintain their own replicas of a simple model Fig. 3i(a). The model consists of Person-objects with mandatory attribute 'name' and any number of 'phone's with an optional extension number 'ext' (see the metamodel in the leftmost square; attribute multiplicities are shown in square brackets).

Diagram in Fig. 3i(b) presents an abstract schema of a simple synchronization scenario. Arrow $m\colon A \Rightarrow B$ denotes some correspondence specification, or a *match*, between the models. Such specifications are often called *(symmetric) deltas*, and are computed by model differencing tools.[2] Similarly, arrow $u\colon A \Rightarrow A'$ denotes the delta between two versions of Ann's replica, and we call it an *update*. The task is to propagate this update to Bob's replica and update the match. That is, the propagation operation ppg must compute an updated model B' together with update u^* and new match m'. Note that derived arrows are dashed (and the derived node is blank, rather than shaded). When reading

[2] The term *directed delta* refers to an operational (rather than structural) specification: a sequence of operations (add, change, delete) transforming A to B (an edit log).



i) Two simple replicas to be synchronized

ii) Four cases with different input mappings

Fig. 3. Mappings do matter in update propagation

the paper in color, derived elements would be blue (because the color blue reminds us of machines and mechanical computation). We will continue with this pattern throughout the paper.

Fig. 3ii demonstrates that the results of update propagation depend on the input mappings u and m. All four cases presented in the figure have the same input models A, A', B, but different mappings m or/and u, which imply — as we will see — different outputs B', u^*, m'.

Consider Fig. 3ii(a). Models' elements (their OIDs) are denoted by letters $P, a, ..., Q, x, ...$. We match models by linking those elements that are different replicas of the same objects in the real world (note the label =). Some of such links are provided by the user (or a matching tool) while others can be derived using the metamodel. For example, as soon as elements $P@A$ and $Q@B$ are linked, their 'name' attributes must be linked too because the metamodel prescribes a mandatory unique name for any Person object. In contrast, linking the phone attributes $b@A$ and $y@B$ is an independent (basic rather then derived) datum because the metamodel allows a person to have several phones. The match shown in the figure says that b and y refer to the same phone. Then we have a conflict between models because they assign different numbers to the same phone.In such cases the link is labeled by (red) symbol $\neq$ signaling a conflict. The set of all matching links together with their labels is called a *matching mapping* or just a *match*, $m\colon A \Rightarrow B$.

An *update mapping* $u\colon A \Rightarrow A'$ specifies a delta between models in time. Mapping u in Fig. 3ii(a) consists of three links. Note that in general the OIDs of the linked (i.e., the "same") objects may be different if, for example, Ann first deleted object P but later recognized that it was a mistake and restored it from scratch as a fresh object P'. Then we must explicitly declare the "sameness" of P and P', which implies the sameness of their 'name' attributes. In contrast, the sameness of phone numbers is an independent datum that must be explicitly declared. Different values of linked attributes mean that the attribute was modified, and such links are labeled by $\sim$ (the update analog of $\neq$-label for matches).

Now we will consecutively consider the four cases of update propagation shown in Fig. 3ii. In all four cases, link $PP' \in u$ means that object P is not deleted, and hence its model B's counterpart, object Q, is also preserved (yet in Fig. 3ii =-links QQ in mapping u^* are skipped to avoid clutter.) However, Q's attribute values are kept unchange or modified according to mappings u and m.

Case (a). Name change in A is directly propagated to B, and addition of phone extension specified by u is directly propagated to u^*. The very phone number is not changed because match m declared a conflict, and our propagation policy takes this into account. A less intelligent yet possible policy would not propagate the extension and keep the entire y unchanged.

Case (b): conflicting link $b \rightarrow y$ is removed from the match, i.e., Ann and Bob consider different phones of Jo. Hence, the value of $y@B$ should not change.

Case (c): link $b \rightarrow b'$ is removed from mapping u, i.e., Jo's phone b was deleted from the model, and a new phone b' is added. Propagation of this update can be managed in different ways. For example, we may require that deletions are propagated over both =- or ≠-matching links, and then phone y must be deleted from B'. Or we may set a more cautious policy and do not propagate deletions over conflicting matching links. Then phone y should be kept in B' (this variant is shown in square brackets and is grey). Assuming that additions to A are always propagated to B, we must insert in B a new phone z "equal" to b'.

Case (d) is a superposition of cases (b,c): both links $b \to y$ and $b \to b'$ are removed from resp. m and u. A reasonable update policy should give us model B' as shown: phone y is kept because it was not matched to the deleted b, and phone z is the new b' propagated to B'. This result can be seen as a superposition of the results in (b) and (c), and our propagation policies thus reveal compatibility with mappings' superposition.

Discussion. In each of the four cases we have an instance of the operation specified in Fig. 3i(b): given an input diagram (u, m), an output diagram (u^*, m') is derived. What we call an update propagation policy is a precise specification of how to build the output for any input. Three points are worth mentioning.

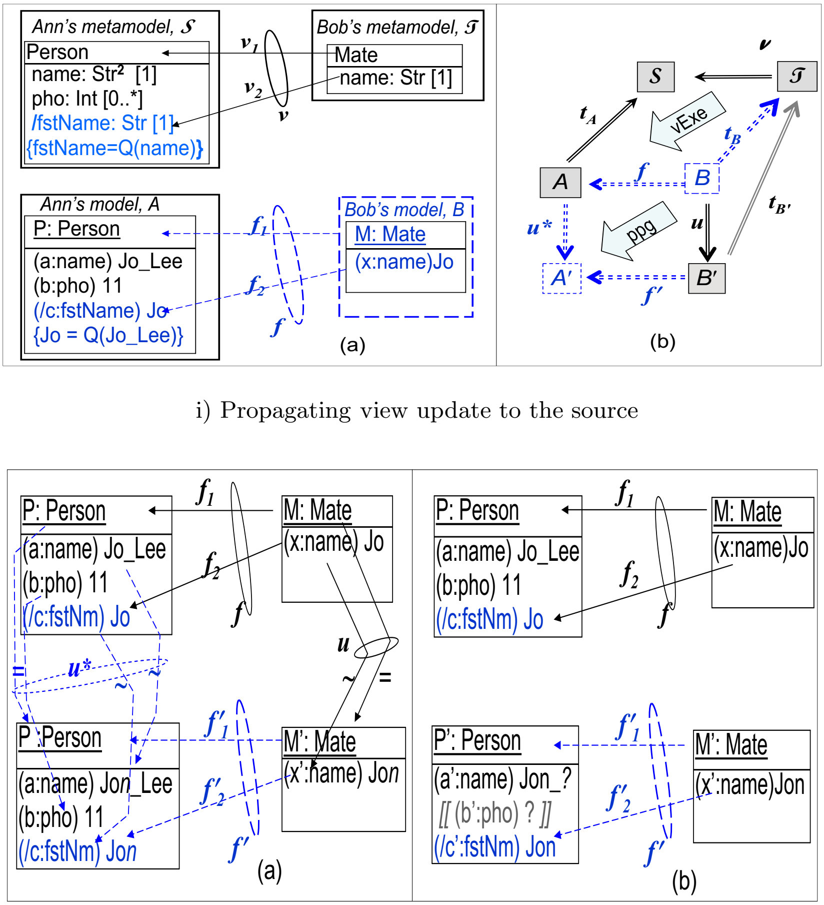
1. Policies are based on the metamodel: for example, a policy may prescribe different propagation strategies for different attributes (say, phone changes are propagated but name changes are not).
2. Recall that in cases (a,c) we discussed different possibilities of update propagation. They correspond to different policies rather than to different outputs of a single policy. That is, different policies give rise to different algebraic operations but a given policy corresponds to a deterministic operation producing a unique output for an input.
3. The mapping-free projection of the four cases would reveal a strange result: the same three input models A, B, A' generate different models B' for a given policy. That is, the mapping-free projection of a reasonable propagation procedure cannot be seen as an algebraic operation.

2.2.2 View Update Propagation

Now we consider a different situation when Bob's model is a view of Ann's one, see Fig. 4i(a). Ann is interested in objects called Persons, their full names, i.e., pairs (fstName, lstName), and phone numbers. Bob calls these objects Mates, and only considers their first names but call the attribute 'name'.

To specify this view formally, we first augment Ann's metamodel $\boldsymbol{\mathcal{S}}$ with a derived attribute 'fstName' coupled with the query specification Q defining this element. Query Q says "take the first component of a name"; formally, fstName$\overset{\text{def}}{=}$ Q(name)=proj_1(name). Then we map Bob's metamodel $\boldsymbol{\mathcal{T}}$ into Ann's one as shown in the figure, where the view definition mapping $\boldsymbol{v}: \boldsymbol{\mathcal{T}} \Rightarrow \boldsymbol{\mathcal{S}}$ consists of two links. Link v_1 says that Bob's class Mate is Ann's class Person. Link v_2 says that Mate attribute 'name' is Person's 'fstName' computed by query Q.

Now let A be a model over Ann's metamodel $\boldsymbol{\mathcal{S}}$ shown in the left lower corner of Fig. 4i(a). We may apply to it the query Q specified in the metamodel, and compute the derived attribute c = proj_1(Jo_Lee)=Jo. Then we select those elements of the model, whose types are within the range of mapping $\boldsymbol{v}$, and relabel them according to this mapping.



i) Propagating view update to the source

ii) Two cases with different update mappings

Fig. 4. Mappings do matter in update propagation cont'd

The result is shown in the right-lower corner as model B, and links $f_{1,2}$ trace the origin of its elements. These links constitute the traceability mapping $f = \{f_1, f_2\}$. In this way, having the view definition mapping $\boldsymbol{v}$, any Ann's model A (an instance of $\boldsymbol{\mathcal{S}}$) can be translated into a $\boldsymbol{\mathcal{T}}$'s instance B computed together with traceability mapping $f\colon B \Rightarrow A$. (A more complex example can be found in Sect. C.)

Thus, we have a diagram operation specified by square diagram $A\boldsymbol{\mathcal{S}}\boldsymbol{\mathcal{T}}B$ in Fig. 4i(b). It takes two mappings — view definition $\boldsymbol{v}$ and typing of the source model t_A, and produces model B (together with its typing t_B) and traceability mapping $f\colon B \Rightarrow A$. This is nothing but an arrow formulation of the view execution mechanism; hence the name vExe of the operation.

Now suppose that the view is updated with mapping $u\colon B \Rightarrow B'$, and we need to propagate the update back to the source as shown by the lower square in Fig. 4i(b). Update propagation is a different type of diagram operation, and it is convenient to consider the two diagrams as orthogonal: view execution is the top face of the semi-cube and propagation is the front. Note that an output element of operation vExe, mapping f, is an input element for operation ppg; diagram Fig. 4i(b) thus specifies substitution of one term into another (and we have an instance of tiling mentioned above).

Fig. 4ii presents two cases of update propagation. In case (a), the name of Mate-object M was modified, and this change is propagated to object P – the preimage of M in the source model. Elements of model A not occurring in the view are kept unchanged. In case (b), the update mapping is empty, which means that object M was deleted and a new object M' added to the model. Correspondingly, object P is also deleted and a new object P' is added to A. Since the view ignores last names and phones numbers, these attributes of P' are set to Unknown (denoted by ?). The attribute b' is shown in brackets (and grey) because a different propagation policy could simply skip P''s phone number as it is allowed by the metamodel (but the last name cannot be skipped and its value must be set to Unknown).

The results of Discussion at the end of the previous section applies to the view update propagation as well.

2.2.3 Why State-Based Synchronization Does Not Work Well

Examples above show that synchronization is based on mappings providing model alignment, particularly, update mappings. Nevertheless, *state-based* frameworks are very popular in data and model synchronization. Being state-based means that the input and the output of the synchronizer only include states of the models while update mappings are ignored. More accurately, model alignment is done inside the synchronizer, as a rule, on the basis of keys (names, identifying numbers or other relevant information, e.g., positions inside a predefined structure). However, this setting brings with it several serious problems.

First of all, update mappings cannot be, in general, derived from the states. Identification based on names fails in cases of synonymy or homonymy that are not infrequent in modeling. Identification numbers may also fail, e.g., if an employee quit and then was hired back, she may be assigned a new identification number. “Absolutely” reliable identification systems like SSNs are rarely available in practice, and even if they are, fixing a typo in a SSN creates synonymy. On the other hand, identification based on internal immutable OIDs also does not solve the problem if the models to be aligned reside in different computers. Even for models in the same computer, OID-based identification fails if an object was deleted but then restored from scratch with a new ID, not to mention the technological difficulties of OID-based alignment. Thus, update mappings cannot be computed entirely automatically, and there are many model differencing tools [8,9,10] employing various heuristics and requiring user assistance to fix the deficiencies of the automatic identification. In general, alignment is another story, and it is useful to separate concerns: discovering updates and propagating updates are two different tasks that must be treated differently and addressed separately.

Second, writing synchronization procedures is difficult and it makes sense to divide the task into simpler parts. For example, view update propagation over a complex view can be divided into composition of update propagations over the components as shown in Fig. 5: $\boldsymbol{\mathcal{X}}$ is some intermediate metamodel and view definition $\boldsymbol{v}$ is composed from parts, $\boldsymbol{v} = \boldsymbol{v}_1;\boldsymbol{v}_2$. It is reasonable to compose the procedure of update propagation over view $\boldsymbol{v}$ from propagation procedures over the component views as shown in the figure. It is a key idea for the lens approach to tree-based data synchronization [2], but lens synchronization is state-based and so two propagation procedures ppg_1 and ppg_2 can be composed if the output states of the first are the input for the second. Hence, the composed procedure will be erroneous if the components use different alignment strategies (e.g., based on different keys) and then we have different update mappings u_X^1, u_X^2 as shown in the figure.

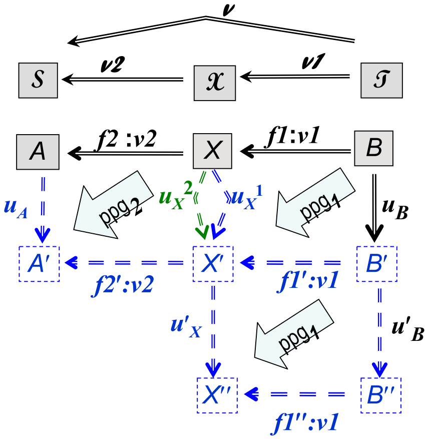


Fig. 5. Mappings do matter in update propagation (cont’d)

Finally, propagation procedures are often compatible with update composition: the result of propagating a composed update $u_B; u'_B$ is equal to composition of updates $u_X; u'_X$ obtained by consecutive application of the procedure. However, if alignment is included into propagation, this law rarely holds — see [11] for a detailed discussion.

2.3 The Arrow Forest and Categories

Mappings are two-folded constructs. On one hand, they consist of directed links and can be sequentially composed; the arrow notation is very suggestive in this respect. On the other hand, mappings are *sets* of links and hence enjoy set operations (union, intersection, difference) and the inclusion relation (defined for mappings having the same source and target). Mappings can also be composed in parallel: given $m_i\colon A_i \Rightarrow B_i$ $(i = 1, 2)$, we can build $m_1 \otimes m_2\colon A_1 \otimes A_2 \Rightarrow B_1 \otimes B_2$, where $\otimes$ may stand for Cartesian product or disjoint union (so that we have two types of parallel composition).

Mapping compositions complicate the *relationship tangle* in Fig. 2 even more: the set of basic relationships generates derived relationships. If the latter are not recognized, models remain unsynchronized and perhaps inconsistent. Living with inconsistencies [12] is possible if they are explicit and specified; implicit inconsistencies undermine modeling activities and their automation.

Thus, our tale of unraveling the tangle of relationships led us to an *arrow forest*. Updates, matches, traceability and typing mappings are all important for model synchronization. Together they give rise to complex structures whose intelligent mathematical processing is not evident and not straightforward.

In the paper we will only consider one side of the rich mapping structure: directionality and sequential composition. Even in this simplified setting, specifying systems of heterogeneous mappings needs special linguistic means: right concepts and a convenient notation based on them. Fortunately, such means were developed in category theory and are applicable to our needs (the reader may think of "paved trails in the arrow forest"); the concrete MMt sections of the paper will show how they work.

Arrows of different types interact in synchronization scenarios and are combined into tiles. The latter may be either similar and work in the same plane, or be "orthogonal" and work in orthogonal planes as, for example, shown in Fig. 4i(b). Complex synchronization scenarios are often multi-dimensional and involve combinations of low-dimensional tiles into higher-dimensional ones. For example, update propagation for the case of two heterogeneous models with evolving metamodels gives rise to a synchronization cube built from six 2D-tiles (Sect. 6.2). Higher-dimensional tiles are themselves composable and also form category-like structures. In this way the tangled collection of models and model mappings can be unraveled into a regular net in a multi-dimensional space, as suggested by the frame of reference on the left of Fig. 2. (Note that we do not assume any metric and the space thus has an algebraic rather than a geometric structure. Nevertheless, multi-dimensional visualization is helpful and provides a convenient notation.)

3 Inside Models and Model Deltas

Diagrammatic models employ a compact *concrete* syntax, which is a cornerstone of practical applications. This syntax hides a rich structure of relationships and dependencies between model's elements (*abstract* syntax), which does matter in model semantics, and in establishing relations between models as well. In this section we will take a look "under the hood" and consider structures underlying models (Sect. 3.1) and symmetric deltas (binary relations) between models (Sect. 3.3). To formalize inconsistencies, we introduce *object-slot-value* models and their mappings (Sect. 3.2). We will use the notions of graph, graph mapping (morphism) and span; their precise definitions can be found in Appendix B.

3.1 Inside Models: Basics of Meta(Meta)Modeling

A typical format for internal (repository) model representation is, roughly, a containment tree with cross-references, in fact, a directed graph. The elements of this graph have attributes and types; the latter are specified in the metamodel. An important observation is that assigning types to model elements constitutes a mapping $t\colon A \to M$ between two graphs underlying the model (A) and its metamodel (M) resp. What is usually called a *model graph* [9,10,13] is actually an encoding of a typing mapping t. Making this mapping explicit is semantically important, especially for managing heterogeneous model mappings.

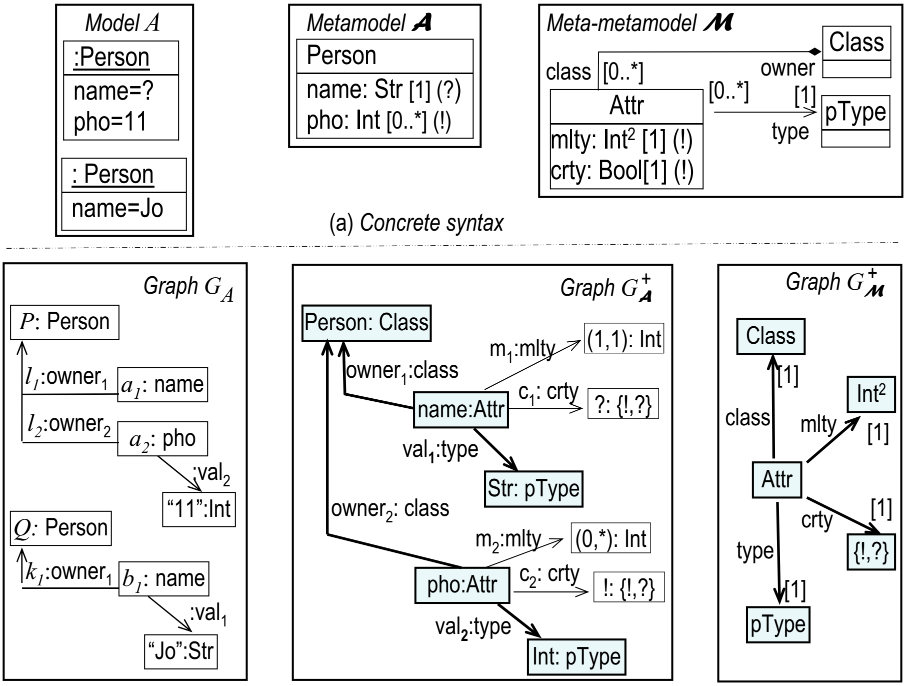
Example. The upper half of Fig. 6 presents a simple metamodel $\boldsymbol{\mathcal{A}}$ (in the middle) and its simple instance, model A (on the left), with a familiar syntax of class and object diagrams. The metamodel is a class diagram declaring class Person with two attributes. Expressions in square brackets are *multiplicities*: their semantic meaning is that objects of class Person have one and only one name (multiplicity [1..1] or [1] in short), and may have any number of phones, perhaps none (multiplicity [0..*]).

Symbols in round brackets are beyond UML and say whether or not the value of the attribute may be set to Unknown (*null*, in the database jargon). Marking an attribute by ? means that nulls are allowed: every person has a name but it may be unknown; we call attributes *uncertain*. An attribute is called *certain* (and marked by !) if nulls are not allowed and the attribute must always have an actual value. If a person has a phone, its number cannot be skipped.

Model A is an object instance of $\boldsymbol{\mathcal{A}}$. It declares two Person objects: one with an unknown name (which is allowed by the metamodel) and phone number 11, and the other with name Jo and without phones (which is also allowed). Symbol '?' is thus used as both a quasi-value (null) in the models and a Boolean value $? \in \{?, !\}$ in the metamodel.

In its turn, the metamodel is an instance of the meta-metamodel specified by a class diagram $\boldsymbol{\mathcal{M}}$ in the right upper corner. It says that metamodels can declare classes that own any number (perhaps, zero) of attributes, but each attribute belongs to one and only one class (this is a part of the standard semantics for "black diamond" asscoiations in UML). Each attribute is assigned one primitive type, a pair of integers specifying its multiplicity, and a Boolean value for certainty; neither of these can be skipped (marker !). We will use model element names (like Person, pho, etc) as OIds, and hence skip the (important) part of $\boldsymbol{\mathcal{M}}$ specifying element naming: certainty and uniqueness of names.

Remark 1. As is clear from the above, an attribute's multiplicity and certainty are orthogonal concepts. Below we will see that their distinction matters for model synchronization. It also matters for query processing and is well known in the database literature [14]. Surprisingly, the issue is not recognized in UML,



(a) *Concrete syntax*

(b) *Abstract syntax*

Fig. 6. From models to graphs

whose metamodel for class diagrams does not have the concept of certainty, and handbooks suggest modeling an attribute's uncertainty by multiplicity [0..1] [15].

Example cont'd: Abstract syntax. In the lower half of Fig. 6, the concrete syntax of model diagrams is unfolded into directed graphs: model elements are nodes and their relationships are arrows. We begin our analysis with the metamodel graph $G^+_{\mathcal{A}}$ (in the middle of the figure). Bold shaded nodes stand for the concepts (types) declared in the class diagram $\mathcal{A}$: class Person and its two attributes. Bold arrows relate attributes with their owning class and value domains. The bold elements together form an *instantiable* subgraph $G_{\mathcal{A}}$ of the entire graph $G^+_{\mathcal{A}}$. Non-instantiable elements of $G^+_{\mathcal{A}}$ specify constraints on the intended instantiations.

Graph G_A (the leftmost) corresponds to the object diagram A and specifies an instantiation of graph $G_{\mathcal{A}}$. Each G_A's element has a type (referred to after the colon) taken from graph $G_{\mathcal{A}}$. Nodes typed by Person are *objects* (of class Person) and nodes typed by attributes are *slots* (we use a UML term). Slots are linked to their owning objects and to values they hold. Slot a_1 is *empty*: there are no value links going from it. Thus, the abstract syntax structure underlying a class diagram is a graph $G^+_{\mathcal{A}}$ containing an instantiable subgraph $G_{\mathcal{A}}$ and noninstantiable constraints. A legal instance of graph $G_{\mathcal{A}}$ is a graph mapping $t_A : G_A \to G_{\mathcal{A}}$ satisfying all constraints from $G^+_{\mathcal{A}} \setminus G_{\mathcal{A}}$.

The same pattern applies to the pair $(G^+_{\mathcal{A}}, G_{\mathcal{M}})$, where $G_{\mathcal{M}}$ is the instantiable subgraph of graph $G^+_{\mathcal{M}}$ specifying the metametamodel (the rightmost in Fig. 6). Multiplicities in Fig. 6(b) are given in the sugared syntax with square brackets, and can be converted into nodes and arrows as it is done for graph $G^+_{\mathcal{A}}$; association ends without multiplicities are assumed to be [0..*] by default. Finally, there is a metameta... graph $G_{\mathcal{MM}}$ providing types and constraints for $G^+_{\mathcal{M}}$; it is not shown in the figure.

The entire configuration appears as a chain of graphs and graph mappings in Fig. 7. Horizontal and slanted arrows are typing mappings; vertical arrows are inclusions and symbols $\models$ remind us that typing mappings on the left-above of them must satisfy the constraints specified in the noninstantiable part. This compact specification is quite general and applicable far beyond our simple example. To make it formal, we need to formalize the notion of constraint and its satisfiability by a typing mappings. This can be done along the lines described in [16].

$$\begin{tikzcd}
G_A \arrow[r, "t_A"'] & G_{\mathcal{A}} \arrow[d, hook, "\models"'] \arrow[dr, dashed, "t_{\mathcal{A}}"] & \\
& G^+_{\mathcal{A}} \arrow[r, "t^+_{\mathcal{A}}"'] & G_{\mathcal{M}} \arrow[d, hook, "\models"'] \\
& & {G_{\mathcal{M}}}^+
\end{tikzcd}$$

Fig. 7. Models as graphs

Two models are called *similar* if they have the same metamodel, and hence all layers below the upper one are fixed. In our example, two object diagrams are similar if they are instances of the same class diagram.

3.2 Object-Slot-Value Models and Their Mappings

Our definition of models as chains of graph mappings does not distinguish between objects and values: they are just nodes in instance graphs. However, objects and values play different roles in model matching and updating, and for our further work we need to make their distinction explicit. Below we introduce *object-slot-value (osv)* models, whose mappings (morphisms) treat objects and values differently. This is a standard categorical practice: a distinction between objects is explicated via mappings (in Lawvere's words, "to *objectify* means *to mappify*").

In the previous section we defined a metamodel as a graph mapping $t^+_{\mathcal{A}} \colon G^+_{\mathcal{A}} \to G_{\mathcal{M}}$. Equivalently, we may work with the inverse mapping $(t^+_{\mathcal{A}})^{-1}$, which assigns to each element $E \in G_{\mathcal{M}}$ the set of those $G^+_{\mathcal{A}}$'s elements e for which $t^+_{\mathcal{A}}(e) = E$. It is easy to check that this mapping is compatible with incidence relationships between nodes and arrows and hence can be presented as a graph morphism $(t^+_{\mathcal{A}})^{-1} \colon G_{\mathcal{M}} \to \boldsymbol{Sets}$ into the universe of all sets and (total) functions between them. (Indeed, multiplicities in graph $G^+_{\mathcal{M}}$ require all its arrows to be functions). To simplify notation, below we will skip the metametamodel's syntax and write E instead of $(t^+_{\mathcal{A}})^{-1}(\mathsf{E})$ (where E stands for Class, Attr, type etc. elements in graph $G_{\mathcal{M}}$). Given a model, we will also consider sets Obj and Slot of all its objects and slots.

$$\begin{tikzcd}
{} & \mathsf{Obj} \arrow[r, "t^{\mathsf{obj}}"] & \mathsf{Class} & \mathsf{Int}^2 \\
{} & \mathsf{Slot} \arrow[u, "\mathsf{obj}"] \arrow[r, "t^{\mathsf{slt}}"] & \mathsf{Attr} \arrow[u, "\mathsf{class}"'] \arrow[ur, "\mathsf{mlty}"] \arrow[dr, "\mathsf{crty}"] \arrow[dd, "\mathsf{type}"'] & {} \\
\mathsf{Slot}^! \arrow[ur, hook, "i^!"] \arrow[dr, "\mathsf{val}"'] & & & \mathsf{Bool} \\
{} & \mathsf{Value} \arrow[r, "t^{\mathsf{val}}"] & \mathsf{pType} & {}
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
{} & \mathsf{Obj}A \arrow[rr, "f^{\mathsf{obj}}"] & & \mathsf{Obj}B \\
{} & \mathsf{Slot}A \arrow[u, "\mathsf{obj}_A"'] \arrow[rr, "f^{\mathsf{slt}}"] & & \mathsf{Slot}B \arrow[u, "\mathsf{obj}_B"'] \\
\mathsf{Slot}^!A \arrow[ur, hook, "i^!_A"] \arrow[urrr, dashed, "f^{\mathsf{slt}!}"] \arrow[rr, "f^{\mathsf{slt}!}"] \arrow[dr, "\mathsf{val}_A"'] & & \mathsf{Slot}^!B \arrow[ur, hook, "i^!_B"'] \arrow[dr, "\mathsf{val}_B"] & {} \\
{} & \mathsf{Value} \arrow[rr, equal] & & \mathsf{Value}
\end{tikzcd}$$

(b)

Fig. 8. Osv-models and their mappings

Definition 1. (Osv-models) An *object-slot-value model* is given by a collection of sets and functions (i.e., total single-valued mappings) specified by diagram Fig. 8(a); the hooked arrow $i^!$ denotes an inclusion. The functions are required to make the diagram commutative, and to satisfy two additional constraints (1,2) (related to mlty and crty) specified below after we discuss the intended interpretation of sets and functions in the diagram.

The bottom row gives a system of primitive types for the model, and the right "column" specifies a class diagram without associations (the metamodel). For example, model A in Fig. 6 is an instance of the osv-model definition with sets Class={Person}, Attr={name, pho}, pType={Str, Int} and Value consisting of all strings and all integers. Classes Int and Bool have their usual extension consisting, resp., of integers (including "infinity" *) and Boolean values (denoted by ?,!).[3] The functions are defined as follows: type(name) = Str, type(pho) = Int; class(name) = class(pho) = Person; mlty(name) = (1,1), mlty(pho) = (0,*); crty(name) = !, crty(pho)= ?

The left column specifies the "changeable/run-time" part of the model — an object diagram over the class diagram; hence, there are typing mappings t^{obj}, t^{slt} and the requirement for the upper square diagram to be commutative. For example, for model A in Fig. 6, we have sets $\mathsf{Obj} = \{P, Q\}$, $\mathsf{Slot} = \{a_1, a_2, b_1\}$ and functions: $t^{\mathsf{obj}}(P) = t^{\mathsf{obj}}(Q) =$ Person; $t^{\mathsf{slt}}(a_1) =$ name, *etc*; $\mathsf{obj}(a_1) = P$, *etc.*

Slots in set $\mathsf{Slot}^!$ are supposed to hold a real value extracted by function val. This value should be of the type specified for the attribute, and the lower polygon is also required to be commutative. Slots in set $\mathsf{Slot}^? \overset{\mathrm{def}}{=} \mathsf{Slot} \setminus \mathsf{Slot}^!$ are considered empty, and function val is not defined on them. For model A, we

[3] For a punctilious reader, values in classes Int and Bool live in the metalanguage and are different from elements of set Value.

have $\mathsf{Slot}^! = \{a_2, b_1\}$ $\mathsf{val}(a_2)$ = ’11’, $\mathsf{val}(b_1)$ = ’Jo’ whereas $a_1 \in \mathsf{Slot}^?$. We will continue to use our sugared notation $\mathsf{val}(s) =?$ for saying that slot $s \in \mathsf{Slot}^?$ and hence $\mathsf{val}(s)$ is not defined.

The following two conditions hold.
(1) For any attribute $a \in \mathsf{Attr}$ and object o with $t^{\mathsf{obj}}(o) = \mathsf{class}(a)$, if $\mathsf{mlty}(a) = (m, n)$, then $m \leq |\mathsf{obj}^{-1}(o)| \leq n$ (i.e., the number of a-slots that a $\mathsf{class}(a)$-object has must satisfy a’s multiplicity).
(2) If for a slot $s \in \mathsf{Slot}$ we have $s.t^{\mathsf{slt}}.\mathsf{crty} = 1$ (i.e., the attribute is certain), then $s \in \mathsf{Slot}^!$.

Definition 2. (Osv-model mappings) Let A, B be two osv-models over the same class diagram, i.e., they have the same right “column” in diagram Fig. 8(a) but different changeable parts distinguished by indexes A, B added to the names of sets and functions (see Fig. 8(b) where the class diagram part is not shown, and bottom double-line denotes identity). We call such models *similar.*

A *mapping* $f\colon A \to B$ of similar osv-models is a pair $f = (f^{\mathsf{obj}}, f^{\mathsf{slt}})$ of functions shown in Fig. 8(b) such that the upper square in the diagram commutes, and triangles formed by these functions and typing mappings (going into the “depth” of the figure) are also commutative: $f^{\mathsf{obj}}; t^{\mathsf{obj}}_B = t^{\mathsf{obj}}_A$ and $f^{\mathsf{slt}}; t^{\mathsf{slt}}_B = t^{\mathsf{slt}}_A$.

In addition, the following two conditions hold.
(3) Let $f^{\mathsf{slt}}!\colon \mathsf{Slot}^! A \to \mathsf{Slot} B$ be the composition $i^!_A; f^{\mathsf{slt}}$, i.e., the restriction of function f^{slt} to subset $\mathsf{Slot}^! A$. We require function $f^{\mathsf{slt}}!$ to map a non-empty slot to a non-empty slot. Then we actually have a total function $f^{\mathsf{slt}!}\colon \mathsf{Slot}^! A \to \mathsf{Slot}^! B$, and the upper diamond in diagram (b) is commutative.
(4) The lower diamond is required to be commutative as well: a non-empty slot with value x is mapped to a non-empty slot holding the same value x.

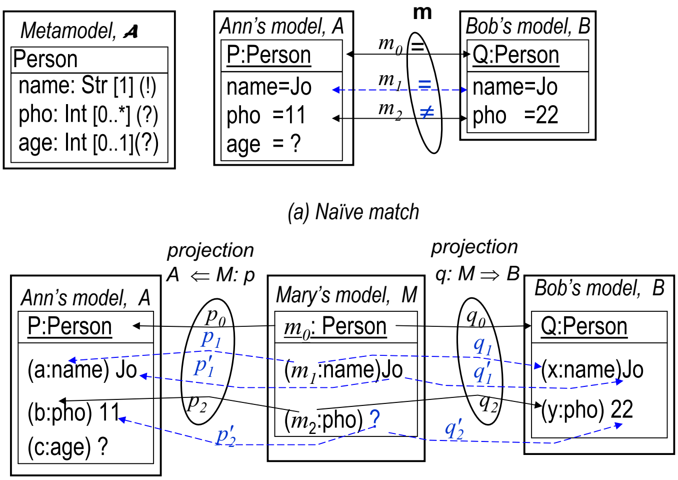
To simplify notation, all three components of mapping f will often be denoted by the same symbol f without superscripts.

Remark 2. Condition (3) says nothing about B-slot $f^{\mathsf{slt}}(s)$ for an empty A-slot $s \in \mathsf{Slot}^? A$: it may be be also empty, or hold a real value. That is, a slot with ’?’ can be mapped to a slot with either ’?’ or a real value (but a slot with a real value v is mapped to a slot holding the same v by condition (4)).

Commutativity of diagram Fig. 8(b) is the key point of Definition 2 and essentially ease working with model mappings. (Categorically, commutativity means that model mappings are *natural transformations*). This advantage comes for a price: condition (4) prohibits change of attribute values in models related by a mapping, and hence we need to model attribute changes somehow differently. We will solve this problem in the next section.

3.3 Model Matching via Spans

Comparing two models to discover their differences and similarities is an important MMt task called *model differencing* or *matching.* Since absolutely reliable keys for models’ elements are rarely possible in practice, model matching tools



(a) Naïve match

(b) Reified match

Fig. 9. Reification of matches

usually employ complex heuristics and AI-based techniques (like, e.g., similarity flooding [17]), which are tailored to specific kinds of models or/and to specific contexts of model comparison [8,10,18]. Whatever the technic is used for model matching, the result is basically a set of matching links between the models' elements. Such sets have a certain structure, and our goal in this section is to specify it formally.

A simple example of model matching is shown in Fig. 9(a). Two similar models are matched by a family **m** of links $m_{0,1,2}$ between model elements (objects and slots). Linking slots implies linking their values; hence we have two additional links m'_1 : Jo → Jo and m'_2 : 11 → 22. The latter link shows a *conflict* between the models.

All matching links respect typing: we cannot match an attribute and an object, or two attributes belonging to unmatched classes. The set of matching links is itself structured similarly to models being matched, and hence can be seen as a new model, say, M as shown in Fig. 9(b). (Name M stands for Mary — an MMt administrator who did the comparison of Ann's and Bob's models.) In the UML jargon, this step can be called *reification* of links: each one becomes an object holding two references (p and q) to the matched elements.

Note that some matching links can be derived from the others. For example, the metamodel says that all Person objects must have one 'name' slot. Then as soon as we have objects P and Q matched, their name slots must be automatically matched (the link is thus derived and shown dashed). In contrast, since several phone slots are possible for a person, matching link m_2 between slots $b@A$ and $y@B$ is an independent datum (solid line).

Whatever the way two slots are matched, their matching means that they should hold the same value. If it is not the case, for example, note different numbers in slots $b@A$ and $y@B$, we have a conflict between models. This conflict is represented by setting the value in slot $m_2@M$ to '?' (which is allowed by the metamodel $\boldsymbol{\mathcal{A}}$ in Fig. 9(a)). Note that the metamodel also allows us to skip attribute 'phone', but then we would not have any record of the conflict. By introducing a slot for the conflicting attributes but keeping it empty, we make the conflict explicit and record it in model M. Moreover, two conflicting slots in models A, B can be traced by links $m_2.p^{\mathsf{slt}}$, $m_2.q^{\mathsf{slt}}$. Then we may continue to work with models A, B leaving the conflict resolution for a future processing (as stated by the famous *Living with inconsistencies* principle [12]).

Note that if models were conflicting at their name-attributes, we should resolve this conflict at once because the metamodel in Fig. 9(a) does not allow having null values for names. In this way metamodels can regulate which conflicts can be recorded and kept for future resolution, and which must be resolved right away. Note also that whether two models are in conflict or consistent is determined by the result of their matching, and hence is not a property of the pair itself.

Definition 3. (Osv-model match) Let A, B be two similar osv-models. An *(extensional) model match* is an osv-model M together with two injective model mappings $A \xleftarrow{p} M \xrightarrow{q} B$ (see Fig. 10).

A match is called *complete*, if for any slot $m \in \mathsf{Slot}M$ the following holds:
(*) if $m.p \in \mathsf{Slot}^!A$, $m.q \in \mathsf{Slot}^!B$ and $\mathsf{val}^A(m.p) = \mathsf{val}^B(m.q)$, then $m \in \mathsf{Slot}^!M$.

That is, if a matching slot m links two slots with the same real value, m is not empty (and holds the same value as well by Definition 2).

The term *extensional* refers to the fact that in practice model matches may have some extra (*non-extensional*) information beyond data specified above; we will discuss the issue later in Sect. 4.1. In this section we will say just 'match'.

$$\begin{tikzcd}
\mathsf{Obj}A & \mathsf{Obj}M \arrow[l, "p^{\mathsf{obj}}"'] \arrow[r, "q^{\mathsf{obj}}"] & \mathsf{Obj}B \\
\mathsf{Slot}A \arrow[u] & \mathsf{Slot}M \arrow[l, "p^{\mathsf{slt}}"'] \arrow[r, "q^{\mathsf{slt}}"] \arrow[u] & \mathsf{Slot}B \arrow[u] \\
\mathsf{Slot}^!A \arrow[u, hook] & \mathsf{Slot}^!M \arrow[l, "p^{\mathsf{slt!}}"'] \arrow[r, "q^{\mathsf{slt!}}"] \arrow[u, hook] & \mathsf{Slot}^!B \arrow[u, hook]
\end{tikzcd}$$

Fig. 10. Matching two osv-models

Completion and consistency of matches. Any incomplete match M can be completed up to a uniquely defined complete match M^* containing M: $\mathsf{Obj}M^* = \mathsf{Obj}M$, $\mathsf{Slot}M^* = \mathsf{Slot}M$, and $\mathsf{Slot}^!M^* \supset \mathsf{Slot}^!M$. We first set $\mathsf{Slot}^!M^* = \mathsf{Slot}^!M$. Then for any slot $m \in \mathsf{Slot}^?M$ we compute two values, $x(m) = \mathsf{val}^A(m.p)$ and $y(m) = \mathsf{val}^B(m.q)$. If x, y are both real values and $x = y$, we move m into set $\mathsf{Slot}^!M^*$ and set $\mathsf{val}_M(m) = x$, otherwise m is kept in $\mathsf{Slot}^?M^*$. Below we will assume that any match is completed.

For a match M and a slot $m \in \mathsf{Slot}^\sharp M$, there are three cases of relationships between values $x(m)$ and $y(m)$ defined above. (Case A): both values are real but not equal; it means a real conflict between the models. (B): if exactly one of the values is null, say, x, we have an *easy* conflict that can be resolved by propagating real value y from B to A. Let $\mathsf{Slot}^{\sharp}M \subset \mathsf{Slot}^{?}M$ denotes the set of slots for which either (A) or (B) holds.

(C) If both values are nulls, the models do not actually conflict although slot m is empty.

Definition 4. (Consistency) Models A and B are called *consistent wrt. their (complete) match M* if set $\mathsf{Slot}^{\sharp}M$ is empty. (That is, all matched slots either hold a real value or link two empty slots, but the situation of linking two slots with different values is excluded). As a rule, we will say in short that a match M is consistent.

Remark 3. Links in a match can be labeled according to some four-valued logic: no conflict between two real values, no conflict because two nulls, a real conflict (between two real values), and an easy conflict between a value and a null. We leave investigation of this connection for future work.

3.4 Symmetric Deltas and Their Composition

What was described above in terms of matching models understood as replicas, may be also understood in terms of model updates. The following terminology borrowed from category theory will be convenient.

A configuration like $A \xleftarrow{p} M \xrightarrow{q} B$ is called a *model span*: model M is the *head*, models A, B are *feet* and mappings p, q are the *legs* or *projections*. A model span consists of three *set spans*, i.e., spans whose nodes are sets and legs are functions, see Fig. 10. Thus, a (complete) model match is just a (complete) model span whose legs are injections.

Let $A \xleftarrow{p} M \xrightarrow{q} B$ be a complete model span. We may interpret it as an update specification with A and B being the *states* of some fixed model before and after the update. Then elements in sets $\mathsf{Obj}M$ and $\mathsf{Slot}M$ link elements that were kept, A's elements beyond the range of p are elements that were deleted, B's elements beyond the range of q were inserted, and elements from set $\mathsf{Slot}^{\sharp}M$ (of "conflicting" links) show the attributes that were changed. Now we will call a complete span with injective legs a *(symmetric) delta*, and interpret it as either an (extensional) match or an update.

A delta as specified by Fig. 10 is a symmetric construct, but to distinguish the two models embedded into it, we need to name them differently. Say, we may call model A the *left* or better the *source*) model, and model B the *right* or better the *target* model. It is suggestive to denote a delta by an arrow $\mathbf{\Delta} : A \Rightarrow B$, whose double-body is meant to remind us that a whole triple-span diagram (Fig. 10) is encoded. The same diagram can be read in the opposite direction from the right to the left, which means that delta $\mathbf{\Delta}$ can be inverted into delta $\mathbf{\Delta}^{-1} : B \Rightarrow A$ (see Appendix B, p. 151 for a precise definition).

Suppose we have two consecutive deltas

$$A \xRightarrow{\mathbf{\Delta}_1} B \xRightarrow{\mathbf{\Delta}_2} C \text{ with } \mathbf{\Delta}_1 = (A \xleftarrow{p_1} M_1 \xrightarrow{q_1} B) \text{ and } \mathbf{\Delta}_2 = (B \xleftarrow{p_2} M_2 \xrightarrow{q_2} C)$$

between Ann's, Bob's and, say, Carol's models. To compose them, we need to derive a new delta $A \overset{\mathbf{\Delta}}{\Longrightarrow} C$ from deltas $\mathbf{\Delta}_1$ and $\mathbf{\Delta}_2$.

Since deltas are complete spans, each of them is determined by two set spans, Δ_i^{obj} and Δ_i^{slt}, $i = 1, 2$, which can be sequentially composed. The reader may think of deltas as representations of binary relations, and their composition as the ordinary relational composition $\bowtie$; a precise formal definition of delta composition via the so called *pullback* operation is in Appendix B, p. 152.

In this way we derive a new osv-model N determined by sets $\mathsf{Obj}N \overset{\text{def}}{=} \mathsf{Obj}M_1 \bowtie \mathsf{Obj}M_2$ and $\mathsf{Slot}N \overset{\text{def}}{=} \mathsf{Slot}M_1 \bowtie \mathsf{Slot}M_2$, and by function $\mathsf{obj}_N \colon \mathsf{Slot}N \to \mathsf{Obj}N$ defined in the natural way (via the universal property of pullbacks; this is where the categorical formulation instantly provides the required result). Projections are evident and thus we have two set spans $\mathbf{\Delta} = (A \xleftarrow{p^{\mathsf{x}}} \mathsf{x}N \xrightarrow{q^{\mathsf{x}}} C)$ with $\mathsf{x} = \mathsf{obj}, \mathsf{slt}$. These data give us a span N with empty set $\mathsf{Slot}^{!}N$. However, we can complete N as described above (we let N denote the completion too), and so obtain a new delta $\mathbf{\Delta} = (A \xleftarrow{p} N \xrightarrow{q} C)$ between models. Associativity of so defined composition follows from associativity of span composition (Appendix B). In addition, a complete span $A \leftarrow A \rightarrow A$ whose legs consist of identity functions between sets is a unit of composition. We have thus proved

Theorem 1. *The universe of osv-models and symmetric deltas between them is a category.*

Exercise 1. Explain why $\mathsf{Slot}^{!}N \supseteq \mathsf{Slot}^{!}M_1 \bowtie \mathsf{Slot}^{!}M_2$ but equality does not necessarily hold.

4 Simple Update Propagation, I: Synchronizing Replicas

By a replica we understand a maintained copy of a model, and assume that replication is *optimistic*: replicas are processed independently and may conflict with each other, which is optimistically assumed to appear infrequently [19]. Then it makes sense to record conflicts to resolve them later, and continue to work with only partially synchronized replicas. The examples considered in Section 2 (Fig. 3) are simple instances of replica synchronization. We have considered them in a *concrete* way by looking inside models and their mappings. The present section aims to build an abstract algebraic framework in which models and mappings are treated as indivisible points and arrows.

Subsection 4.1 introduces the terminology and basic notions of replica synchronization; there is an overlap with the previous section that renders the present section independent. Subsection 4.2 develops a basic intuition for the algebraic approach to modeling synchronization. Subsections 4.3 and 4.4 proceed with algebraic modeling as such: constructing algebraic theories and algebras (instances of theories).

4.1 Setting the Stage: Delta × Delta = Tile

Terminology. There are two main types of representations for model differences: operational and structural, which are usually called *directed* and *symmetric* deltas respectively [20]. The former is basically a sequence (log) of edit operations: add, change, delete (see, e.g., [21]). The latter is a specification of similarities and differences of the two models ([22]).

A symmetric delta can be seen *extensionally* as a family of matching links, in fact, as a binary relation; in the previous section we formalized symmetric deltas as *(complete) spans* (reified binary relations). Besides extension, a symmetric delta may contain *non-extensional* information: matching links can be annotated with authorship, time stamps, update propagation constraints and the like. We also call deltas *mappings* and denote them by arrows (even symmetric deltas, see Sect. 3.4).

Now suppose we have two replicas of the same model maintained by our old friends Ann and Bob, Fig. 11. Nodes A, B are snapshots of Ann's and Bob's replica at some time moment, when we want to compare them. The horizontal arrow m denotes a relationship — match— between the replicas. We interpret matches as symmetric deltas (spans) with, perhaps, some additional (non-extensional) data. Ann and Bob work independently and later we have two updated versions A' and B' with arrows a and b denoting the corresponding updates. We may interpret updates structurally as symmetric deltas. Or we may interpret them operationally as directed deltas (edit logs).

The four deltas m, a, m', b are mutually related by *incidence relationships*: $\partial_{\mathsf{s}}m = \partial_{\mathsf{s}}a, \partial_{\mathsf{t}}m = \partial_{\mathsf{s}}b$, etc. (where $\partial_{\mathsf{s}}x, \partial_{\mathsf{t}}x$ denote the source and the target of arrow x), and together form a structure that we call a *tile*. The term is borrowed from a series of work on behavior modeling [23], and continues the terminological tradition set up by the Harmony group's *lenses* — naming synchronization constructs by geometric images.

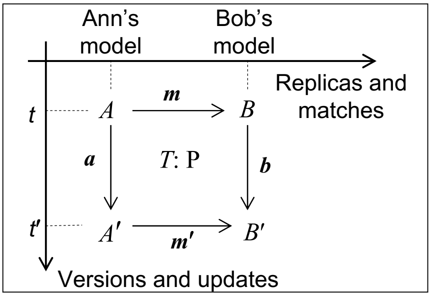


Fig. 11. The space of model versioning

Visually, a tile is just a square formed by arrows with correspondingly sorted arrows. To avoid explicit sorting of arrows in our diagrams, we will always draw them with updates going vertically and matches horizontally. A tile can be optionally labeled by the name of some tile's property (predicate) P. Expression $T{:}P$ means that T has property P, i.e., $T \models P$ or $T \in [\![P]\!]$ with $[\![P]\!]$ denoting the extension of P. The name of the tile may be omitted but the predicate label should be there if $T \models P$ holds.

The tile language: matches vs. updates. To keep the framework sufficiently general, we do not impose any specific restrictions on what matches and updates really are, nor do we assume that they are similar specifications. For example, matches may be annotated with some non-extensional information that does not make sense for updates, e.g., priorities of update propagation (say, 'name' modifications are propagated from Ann's model to Bob's, while 'phones' are not) or "matching ranks" (how much we are sure that elements $e@A$ and $e'@B$ are the same, see [24] for a discussion). Furthermore, we may have matches defined structurally (with annotations or not) whereas updates operationally.

Therefore, we do not suppose that matches can be sequentially composed with updates (and vice versa). But of course updates can be composed with updates, and matches with matches, although match composition can be non-trivial, if at all well-defined, because of non-extensional information. For example, let $m^+ \colon X \to Y$ denotes a match consisting of symmetric delta (relation, span) m augmented with some non-extensional information. For two consecutive matches $m_1^+ \colon A \to B$, $m_2^+ \colon B \to C$, their extensional parts can be composed as relations producing delta $m = m_1; m_2 \colon A \to C$, but to make m into a match m^+ we need to compose somehow non-extensional parts of the matches. We leave the issue for the future work and in this paper will not compose matches.

The situation with updates is simpler. Either they are interpreted structurally as symmetric deltas (spans), or operationally as edit logs, they are sequentially composable in the associative way. For symmetric deltas it is shown in Sect. 3.4; and it is evident for edit logs (whose composition is concatenation).

In addition, we assume that for every model A there are an *idle update* $1_A^b \colon A \to A$ that does nothing, and an *identity match* $1_A^{\mathsf{h}} \colon A \to A$ that identically matches model A to itself. For the structural interpretation of arrows, both idle updates and identity matches are nothing but spans whose legs are identity mappings (and no extra non-extensional information is assumed for matches). For the operational interpretation, idle/identity arrows are empty edit logs.

Thus, in the abstract setting we have a structure consisting of two reflexive graphs, ***Modmch*** of models and *matches*, and ***Modupd*** of *models* and *updates*, which share the same class of objects ***Mod*** but have different arrows. Moreover, arrows in graph ***Modupd*** are composable (associatively) and ***Modupd*** is a category. We will call such a structure a *1.5-sorted category* and denote it by $\mathbb{M}$**od** (if ***Modmch*** also were a category, $\mathbb{M}$**od** would be a *two-sorted category*)(see Sect. B).

Simple synchronization stories via tiles. Despite extreme simplicity of the language introduced above, it allows us to describe some typical replication situations as shown in Fig. 12. The diagrams in the figure can be seen as specifications of use cases ("stories") that have happened, or may happen, in some predefined context. The meaning of these stories is easily readable and explained in the captions of the diagrams (a-d). In diagram (b), symbol $\cong$ denotes the predicate of being an *isomorphic* match (i.e., we assume that a subclass $[\![\cong]\!]$ of arrows in graph ***Modmch*** is defined).

The stories could be made more interesting if we enrich our language with diagram predicates, say, P_{h} and P_{v}, allowing us to compare matches and updates. Then, for example, by declaring that tile T belongs to the class $[\![P_{\mathsf{h}}]\!]$ (as shown by diagram (c)*), we say that match m' is "better" than m. Such predicates can be

$\begin{tikzcd} O \arrow[r, "1_O"] \arrow[d, "a"'] & O \arrow[d, "b"] \\ A \arrow[r, "m"'] & B \end{tikzcd}$	$\begin{tikzcd} A \arrow[r, "m"] \arrow[d, "a"'] & B \arrow[d, "b"] \\ A' \arrow[r, "m'\colon \cong"'] & B' \end{tikzcd}$	$\begin{tikzcd} A \arrow[r, "m"] \arrow[d, "1_A"'] & B \arrow[d, "1_B"] \\ A \arrow[r, "m'"'] & B \end{tikzcd}$	$\begin{tikzcd} A \arrow[r, "1_A"] \arrow[d, "a"'] & A \arrow[d, "b"] \\ A' \arrow[r, "1_{A'}"'] & A' \end{tikzcd}$	$\begin{tikzcd} A \arrow[r, "m"] \arrow[d, "1_A"'] \arrow[dr, phantom, "T{:}P_{\mathsf{h}}"] & B \arrow[d, "1_B"] \\ A \arrow[r, "m'"'] & B \end{tikzcd}$
(a) diverge	(b) converge	(c) revision of match	(d) revision of update	(c)* improvement of match

Fig. 12. Several replication stories via tiles

seen as arrows between arrows, or *2-arrows*, and give rise to a rich framework of *2-categories* and *bicategories* (see, e.g., [25]). We leave this direction of modeling replication for future work.

The "historian's" view on synchronization scenarios, even with comparison predicates, is not too interesting. The practice of model synchronization is full of automatic and semi-automatic operations triggered automatically or by the user's initiation. Thus, we need to enrich our language with synchronization operations.

4.2 Update Propagation via Algebra: Getting started

As discussed in Sect. 2.2, algebraic operations modeling synchronization procedures should be *diagrammatic*: they take a configuration (diagram) of matches and updates that conform to a predefined *input* pattern, and add to it new matches and updates conforming to a predefined *output* pattern. These new elements are to be thought of as computed or derived by the operation. In this section we consider how diagram operations work with a typical example, and develop a basic intuition about the algebraic approach to modeling synchronization.

Update propagation: A sample diagram operation. Propagating updates from one replica to another is an important synchronization scenario. We model it by diagram operation fPpg shown in Fig. 13(a). The operation takes a match m between replicas and an update a of the source replica, and produces an update b of the target replica and a new match m'. The input/output arrows are shown by solid/dashed lines resp.; the direction of the operation is shown by the doubled arrow in the middle. (To be consistent, we should also somehow decorate node B' but we will not so so.)

We write $(b, m') = \mathsf{fPpg}(a, m)$ and call the quadruple of arrows (tile) $T = (a, m, b, m')$ an *application instance* of the operation. Other pairs of input arrows will give other application instances of the same operation; hence, notation $T{:}\mathsf{fPpg}$. (The name T is omitted in the diagram). This notation conforms to labeling tiles by predicates introduced earlier. Operation fPpg defines a predicate fPpg^* of square shape: for a quadruple of arrows (a, m, b, m') forming a square, we set $\mathsf{fPpg}^*(a, m, b, m')$ is true iff $(b, m') = \mathsf{fPpg}(a, m)$; in this case we say that the quadruple (a, m, b, m') is a fPpg-tile. Later we will omit the star superindex.

$$\begin{tikzcd}
A \arrow[r, "m"] \arrow[d, "a"'] \arrow[dr, phantom, "\Searrow\mathsf{fPpg}"] & B \arrow[d, dashed, "b"] \\
A' \arrow[r, dashed, "m'"'] & B'
\end{tikzcd}$$

(a) general instance

$$\begin{tikzcd}
A \arrow[r, "m"] \arrow[d, "a"'] \arrow[dr, phantom, "\Searrow\mathsf{fPpg}"] & B \arrow[d, dashed, "1_B"] \\
A' \arrow[r, dashed, "m'"'] & B
\end{tikzcd}$$

(b) block

$$\begin{tikzcd}
A \arrow[r, "m"] \arrow[d, "a"'] \arrow[dr, phantom, "\Searrow\mathsf{fPpg}"] & B \arrow[d, dashed, "b"] \\
A' \arrow[r, dashed, "m'\colon\cong"'] & B'
\end{tikzcd}$$

(c) copy

Fig. 13. Forward update propagation (a) and its two special cases (b,c)

Below we will also use the dot-notation for function applications, $(b, m') = (a, m).\mathsf{fPpg}$ to ease reading complex formulas. Since the operation produces two elements, we need special *projection* operations, upd and mch, that select the respective components of the entire output tuple: $b = (a, m).\mathsf{fPpg}.\mathsf{upd}$ and $m' = (a, m).\mathsf{fPpg}.\mathsf{mch}$.

Update policies and algebra. There are two extreme cases of update propagation with fPpg.

One is when nothing is propagated and hence the output update is idle as shown in diagram Fig. 13(b). Then propagation amounts to *rematching*: updating the match from m to m'. If this special situation, i.e., equality $(a, m).\mathsf{fPpg}.\mathsf{upd} = 1_B$, holds for any update a originating at m's source, we have a very strong propagation policy that actually *blocks* replica B wrt. updates from A.

The opposite extremal case is when the entire updated model is propagated and overwrites the other replica as shown in diagram (c). A milder variant would be to propagate the entire A' but not delete the unmatched part of B, then match m' would be an embedding rather than isomorphism.

In-between the two extremes there are different propagation policies as discussed in Sect. 2.2.1. The possibility of choice is in the nature of synchronization problems: as a rule, some fragments of information are missing and there are several possible choices for model B'. To make computation of model B' deterministic, we need to set one or another propagation policy. Yet as soon as a policy is fixed, we have an algebraic operation of arity shown in Fig. 13(a). Thinking algebraically, a policy *is* an operation (cf. Discussion in Sect. 2.2.1).

Remark 4. So-called *universal* properties and the corresponding operations (see Appendix A.1) are at the heart of category theory. It explains attempts to model update policies as universally defined operations [26]. However, our examples show that, in general, a propagation policy could not be universally defined simply because many policies are possible (while universally defined operations are unique up to isomorphism).

Algebra: action vs. "history". The mere assertion that some components of a story specified by a tile are derived from the other components may be a strong statement. Let us try to retell our simple synchronization stories in Fig. 12 in an algebraic way.

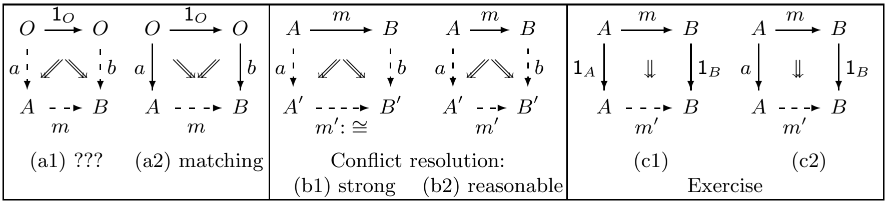


Fig. 14. Replication stories and algebra: Fig. 12 processed algebraically

Diagram Fig. 14(a1) says that three arrows (a, m, b) are produced by applying some operation to the identity match, that is, in fact, to model O. This is evidently meaningless because triple (a, m, b) cannot be derived from O alone. In contrast, diagram (a2) is a reasonable operation: given two updates of the same source, a match between them can be computed based on the information provided by the input data.

Diagram (b1) says that any two matched replicas can be made isomorphic. It is a very strong statement: we assume that all conflicts can be resolved, and differences between replicas can be mutually propagated in a coherent way. A more reasonable algebraic model of conflict resolution is specified by diagram (b2): the result of the operation is just another match m' presumably better (with less conflicts) than m. Augmenting the language with constructs formally capturing the meaning of "better" (e.g., 2-arrows) would definitely be useful, and we leave it for future work.

Exercise 2 (*). Diagrams (c1,c2) present two algebraic refinements of the synchronization story specified in Fig. 12(c). Explain why diagram (c1) does not make much sense whereas (c2) specifies a reasonable operation. *Hint*: Note an important distinction of diagram (c1) from diagram (b2).

4.3 An Algebraic Toolbox for a Replica Synchronization Tool Designer

Suppose we are going to build a replica synchronization tool. Before approaching implementation, we would like to specify what synchronizing operations the tool should perform, and what behavior of these operations the tool should guarantee; indeed, predictability of synchronization results is important for the user of replication/versioning tools (cf. [3]). Hence, we need to fix a signature of operations and state the laws they must obey; in other words, we need to fix a suitable *algebraic theory*. The tool itself will be an *instance* of the theory, that is, an *algebra*: sorts of the theory will be interpreted by classes of replicas the tool operates on, and operations will be interpreted by actual synchronization procedures provided by the tool.

Two main ingredients constituting an algebraic theory are a *signature* of operations with assigned arity shapes, and a set of *equational laws* prescribing the intended behavior of the operations. In ordinary algebra, operation arities are

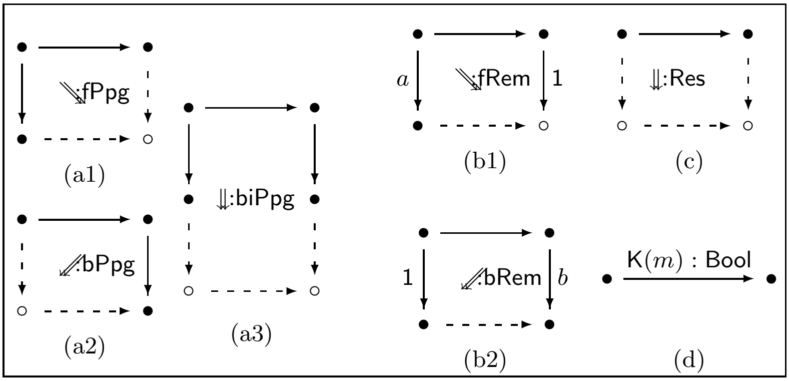


Fig. 15. Replica synchronization operations: update propagation (a), rematching (b), conflict resolution (c), and Boolean test for consistency of matches (d)

sorted sets; in diagram algebra, arities are sorted graphs but the principal ideas and building blocks remain the same. In this section we specify a pool of diagram operations for modeling synchronization procedures, and a pool of laws that they should, or may want, to satisfy. Together they are meant as an algebraic toolbox with which a tool designer can work.

The carrier structure. All our operations will be defined over 1.5-sorted categories, i.e., two-sorted reflexive graphs with arrows classified into horizontal (matches) and vertical (updates); the latter are composable and form a category.

Operations. A precise definition of a diagram operation over a two-sorted graph is given in Sect. B. For the present section it is sufficient to have a semi-formal notion described above.Recall that in order to avoid explicit sorting of arrows in our diagrams, we draw them with geometrically vertical/horizontal arrows being formally vertical/horizontal.

Figure 15 presents a signature of operations intended to model synchronization procedures. The input/output arrows are distinguished with solid/dashed lines, and input/output nodes are black/white.

Diagrams Fig. 15(a1,a2) show operations of *forward* and *backward* update propagation. The former was just considered; the latter propagates updates against the direction of match and is a different operation. For example, if the replica at the source is in some sense superior to the replica at the target, forward propagation may be allowed to propagate deletions whereas the backward one is not. Diagram (a3) specifies bi-directional update propagation. It takes a match and two parallel updates and mutually propagates them over the match; the latter is then updated accordingly.

Diagrams Fig. 15(b1,b2) show operations of forward and backward *rematching.* If for a given match $m\colon A \to B$, one of the replicas, say, A, is updated, we may want to recompute the match but do not change the other replica B. This scenario is modeled by operation fRem in Fig. 15(b1), where the update of the other replica is set to be idle. Thus, operation fRem actually has two arguments (the left update and the upper match) and produces the only arrow — an updated match (at the bottom). The backward rematch works similarly in the opposite direction. The operation of bidirectional rematching does not make sense (Exercise 1 above). If we were modeling both matches and updates by relations (spans), then rematch would nothing but sequential span composition Sect. 3.3. However, as we do not compose updates and matches, we model their composition by a special tile operations.

Finally, Fig. 15(c) specifies operation Res of *conflict resolution*. It takes a match between two replicas that, intuitively, may be inconsistent, and computes updates a, b necessary to eliminate those conflicts that can be resolved automatically without user's input.

Other synchronization operations are possible, and the signature described above is not intended to be complete. Neither is it meant to be fully used in all situations. Rather, it is a pool of operations from which a tool designer may select what is needed.

Predicates. To talk about consistency of matches, we need to enrich our language with a *consistency* predicate (think of strongly consistent matches from Sect. 3.3).

Diagram (d) presents it as a Boolean-valued operation: for any match m a Boolean value is assigned, and we call m *consistent* if $\mathsf{K}(m) = 1$. (The letter K is taken from "*K*onsistency": denoting the predicate by C would better fit the grammar but be confusing wrt. terms Classes and Constraints.) In our diagrams we will write $m{:}\mathsf{K}$ for $\mathsf{K}(m) = 1$. Semantically, we have a class of consistent matches $\mathbf{K} = \{m\colon\ \mathsf{K}(m) = 1\}$.

Remark 5. Consistency is often considered as a binary predicate K' on models: replicas (A, B) are *consistent* if $\mathsf{K}'(A, B)$ holds [6]. Our definition is essentially different and moves the notion of consistency from pairs of replicas to matches. Indeed, as discussed in sections 2.2, 3.3, multiple matches between replicas are possible, and it is a match $m\colon A \to B$ that makes the pair (A, B) consistent or inconsistent.

Remark 6. The presence of predicates makes our theory non-algebraic. A standard way to bring it back to algebra is to define predicates via equations between operations, if it is possible. Another approach is to work in the framework of order-sorted algebra [27].

Equational laws. Equations the operations must satisfy are crucial for algebraic modeling. Without them, algebraic theories would define too broad classes of algebras encompassing both adequate and entirely inadequate algebraic models.

Equational laws for diagram operations can be concisely presented by diagrams as well. Consider, for example, diagram Fig. 16(a1), whose arrows are labeled by names (identifiers) of matches and updates. The names express the following equation: for any match m, $\mathsf{fPpg}(1_{\partial_s m}, m) = (1_{\partial_t m}, m)$. This is a general mechanism: if all arrows in the tile have different names, the tile specifies a generic instance of the operation without any restrictions, but the presence of common names amounts to equational constraints like above.

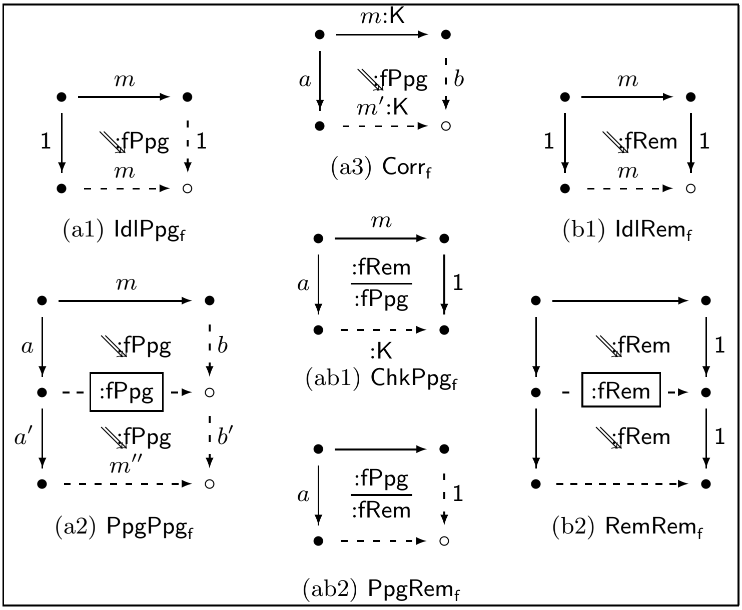


Fig. 16. Replica synchronization: the laws

The equation expressed by Fig. 16(a1) has a clear interpretation: given a match m, the idle update on the source is propagated into the idle update on the target while the match itself is not changed. We call the law $\mathsf{IdlPpg_f}$ following a general pattern of naming such laws by concatenating the operation names (take the idle update and propagate it; index f refers to forward propagation). The pattern was invented by the Harmony group for lenses and turned out very convenient.

Diagram Fig. 16(a2) displays two fPpg-tiles vertically stacked (ignore the boxed label for a while). It means that the output match of the upper application of fPpg is the input match for the lower application. Since updates are composable, the outer rectangle in the diagram is also a tile whose updates are $a;a'$ and $b;b'$. Now the boxed label says that the outer tile is also an application instance of fPpg. (In more detail, given a match m and two consecutive updates a, a' on its source, we have $\mathsf{fPpg}(a;a', m) = (b;b', m'')$ where $(b, m') = \mathsf{fPpg}(a, m)$ (name m' is hidden in the diagram) and $(b', m'') = \mathsf{fPpg}(a', m')$.) We will phrase this as follows: if the two inner tiles are fPpg, then the outer tile is also fPpg (note also the name of the law). Thus, composed updates are propagated componentwise.

Diagram Fig. 16(a3) says that if $(b, m') = \mathsf{fPpg}(a, m)$ and $m \in \mathsf{K}$, then $m' \in \mathsf{K}$ as well: consistency of matches is not destroyed by update propagation. We call an update propagation *correct* if it satisfies this requirement, hence the name of the law. Note the conditional nature of the law: it says that the resulting match is consistent if the original match is consistent but does not impose any obligations if the original match is inconsistent. This formulation fixes the problem of the unconditional correctness law stated in [6].

Exercise 3. Explain the meaning of diagrams (b1) and (b2) in Fig. 16

Now we consider laws regulating interaction between the two operations. The law specified by diagram (ab1) is conditional. The argument of the premise and the conclusion is the entire tile, and the diagram says: if a tile is an instance of fRem with output match satisfying K, then the tile is also an instance of fPpg. Formally, $\mathsf{fRem}^*(T)$ implies $\mathsf{fPpg}^*(T)$ for a tile of the shape shown in the diagram (recall that starred names denote predicates defined by operations). That is, if $m' = \mathsf{fRem}(a, m) \in \mathsf{K}$ then $\mathsf{fPpg}(a, m) = (m', 1_{\partial_\mathsf{t} m})$. The meaning of the law is that if we update the source, and the updated match m' is consistent, then nothing should be propagated to the target. This is a formal explication of the familiar requirement on update propagation: "first check, then enforce" (cf. Hippocraticness in [6]). Hence the name of the law, $\mathsf{ChkPpg}_{.}$

Exercise 4. Explain the meaning of diagram (ab2) Fig. 16

Exercise 5 (*). Formulate some laws for the operation of conflict resolution, and specify them diagrammatically.

There is no claim that the set of laws we have considered is complete: other reasonable laws can be formulated. The goal was to show how to specify equational laws, and how to interpret them, rather then list them "all".

4.4 Replica Synchronization Tools as Algebras

In this section, we build a simple algebra intended to model a replica synchronization tool as it was explained at the beginning of Sect. 4.3.

We first fix a theory (= signature + laws). For the signature, we take four operations (to be precise, operation symbols) ($\mathsf{fPpg}, \mathsf{bPpg}, \mathsf{fRem}, \mathsf{bRem}$) with arities specified in Fig. 15. These operations can be interpreted over any 1.5-sorted category encompassing any number of replicas. However, we assume that our synchronization tool will only work with two replicas propagating updates from one to the other and back. Hence, we need to specify a specific 1.5-sorted category adequate to our modest needs.

Definition 5. A *(binary) replication lane* $\mathbf{r}$ is given by the following data.

(a) Two categories, $\mathbf{A}$ and $\mathbf{B}$, whose objects are called *replicas* (or *models*), and arrows are *updates*. (For a category $\mathbf{X}$, its classes of objects and arrows are denoted by, resp., $\mathbf{X}_0$ and $\mathbf{X}_1$.) Specifically, objects of $\mathbf{A}$ are called *source* replicas and those of $\mathbf{B}$ the *target* ones.

(b) A set $\mathbf{M}$ whose elements are called *matches* from $\mathbf{A}$- to $\mathbf{B}$-replicas, and two functions (*legs*), $\mathbf{A}_0 \overset{\partial_\mathsf{s}}{\leftarrow} \mathbf{M} \overset{\partial_\mathsf{t}}{\rightarrow} \mathbf{B}_0$, from matches to replicas. If for a match $m \in \mathbf{M}$, $\partial_\mathsf{s}(m) = A$, $\partial_\mathsf{t}(m) = B$, we write $m \colon A \to B$.

(c) A set $\mathbf{K} \subset \mathbf{M}$ of *consistent* matches.

Figure 17 visualizes the definition: updates are vertical, and matches are horizontal or slanted (solid or dotted-dashed for being consistent or inconsistent).

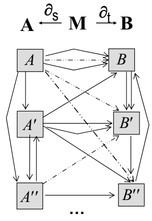


Fig. 17. Replica lane

We denote a replication lane by a bulleted arrow $\mathbf{r}$: $\mathbf{A} \mathrel{-\!\!\bullet\!\!\rightarrow} \mathbf{B}$. If replicas are considered within the same versioning space, categories $\mathbf{A}$ and $\mathbf{B}$ coincide, and we call the lane *unary*, $\mathbf{r}$: $\mathbf{A} \mathrel{-\!\!\bullet\!\!\rightarrow} \mathbf{A}$.

Now we define an algebra over a replica lane.

Definition 6. A *diagonal replica synchronizer* is a pair $\delta = (\mathbf{r}^\delta, \Sigma^\delta_{\mathsf{brSync}})$ with $\mathbf{r}^\delta$ a replica lane and $\Sigma^\delta_{\mathsf{brSync}} = (\mathsf{fPpg}^\delta, \mathsf{bPpg}^\delta, \mathsf{fRem}^\delta, \mathsf{bRem}^\delta)$ a quadruple of diagram operations over $\mathbf{r}^\delta$ of the arities specified in Fig. 15. The name *diagonal* refers to the fact that propagation operations act along diagonals of operation tiles, and bidirectional propagation (for parallel updates) is not considered.

It is convenient to denote a replica synchronizer by an arrow δ : $\mathbf{A} \mathrel{-\!\!\bullet\!\!\rightarrow} \mathbf{B}$ whose source and target refer to the source and target of the replica lane $\mathbf{r}^\delta$.

A diagonal synchronizer is called *well-behaved (wb)* if the pair $(\mathsf{fPpg}, \mathsf{fRem})$ satisfies the laws $\mathsf{IdlPpg_f}$,$\mathsf{Corr_f}$,$\mathsf{IdlRem_f}$,$\mathsf{ChkPpg_f}$ specified in Fig. 16, and the pair $(\mathsf{bPpg}, \mathsf{bRem})$ satisfies the backward counterparts of those laws. A wb diagonal synchronizer is called *very well-behaved (vwb)* if the laws $\mathsf{PpgPpg_f}$,$\mathsf{RemRem_f}$ and their backward counterparts hold too.

Modularization of the set of laws provided by the notions of wb and very wb synchronizer is somewhat peculiar from the categorical standpoint because it joins unitality (preservation of units of composition, ie, idle updates) with other laws but separates it from compositionality, and the very terms are not very convenient. However, this modularization and terminology follow the terminology for lenses [2] and make comparison of our framework with lenses easier (see [28] for an analysis of these laws in the discrete setting).

The definition above is intuitively clear but its precise formalization needs a careful distinction between syntax and semantics of a diagram operation, see Sect. B.

5 Simple Update Propagation II: Forward and Backward View Maintenance

In this section we consider synchronization of a source model and its view. The content is parallel to replica synchronization and the algebraic model is developed along the same lines. Yet view synchronization is essentially different from replica synchronization.

5.1 View vs. Replica Synchronization

Examples in Sect. 2.2 and Appendix C show that a view definition can be modeled by a metamodel mapping $\boldsymbol{\mathcal{S}} \xleftarrow{\boldsymbol{v}} \boldsymbol{\mathcal{T}}$ that sends elements of the view (target) metamodel $\boldsymbol{\mathcal{T}}$ to basic or derived elements of the source metamodel $\boldsymbol{\mathcal{S}}$.[4] In addition, the mapping must be compatible with the structure of the metamodels

[4] Derived elements of $\boldsymbol{\mathcal{S}}$ are, in fact, queries against $\boldsymbol{\mathcal{S}}$ seen as a data schema.

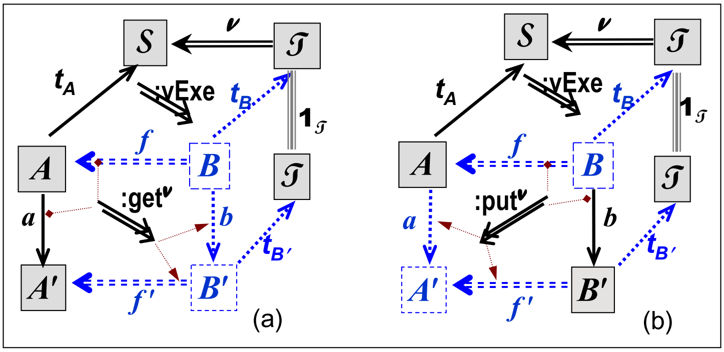


Fig. 18. View maintenance: Forward (a) and backward (b) update propagation

(and send a class to a class, an attribute to an attribute *etc.*) Such a view definition can be executed for any instance A of $\boldsymbol{\mathcal{S}}$, and produce a $\boldsymbol{v}$-view of A, i.e., a $\boldsymbol{\mathcal{T}}$-instance denoted by $A\restriction_{\boldsymbol{v}}$, along with a traceability mapping $A \xleftarrow{\overline{\boldsymbol{v}_A}} A\restriction_{\boldsymbol{v}}$ (see Sect. C for details).[5] In fact, we have a diagram operation specified by the top face of cube (a) in Fig. 18, where $B = A\restriction_{\boldsymbol{v}}$ and $f = \overline{\boldsymbol{v}_A}$.

If the source A is updated, the update is propagated to the view by operation $\mathsf{get}^{\boldsymbol{v}}$ ("getView") shown in the front face of the cube Fig. 18(a). The operation takes a source update a and view mapping f, and produces a view update b together with a new view traceability mapping f'. A reasonable requirement is to have $f' = \overline{\boldsymbol{v}_{A'}}$ and $B' = A'\restriction_{\boldsymbol{v}}$. In the database literature, such operations have been considered as *view maintenance* [29].

If the view is updated via $b\colon B \to B'$ (the front face of cube Fig. 18(b)), we need to update the source correspondingly and find an update $a\colon A \to A'$ such that $B' = A'\restriction_{\boldsymbol{v}}$; simultaneously, a new traceability mapping $f' = \overline{\boldsymbol{v}_{A'}}$ is computed. Since normally a view abstracts away some information, many updates a may satisfy the condition. To achieve uniqueness, we need to consider additional aspects of the situation (metamodels, view definition, the context) — this is the infamous view update problem that has been studied in the database literature for decades [30]. Yet we assume that somehow an update propagation policy ensuring uniqueness is established, and hence we have an operation $\mathsf{put}^{\boldsymbol{v}}$ ("put update back") specified by the front face of the cube. Names 'get' and 'put' are borrowed from the lens framework [2], but in the latter neither update nor view mappings are considered. Also, lenses' operation get corresponds to our vExe_0.

Despite similar arity shapes of bidirectional pairs (get,put) in view synchronization and (fPpg,bPpg) in replica synchronization, the two tasks are different.

First we note that in the view update situation, consistency relation $\mathbf{K}$ can be derived rather than independently postulated: we set

$$\mathbf{K} \overset{\text{def}}{=} \left\{ A \xleftarrow{f} B\colon\ f = \overline{\boldsymbol{v}_A} \right\}. \tag{Cons}$$

[5] View $A\restriction_{\boldsymbol{v}}$ can be seen as $\boldsymbol{v}$-*projection* of model A to space of $\boldsymbol{\mathcal{T}}$-models, hence symbol $\restriction$ denoting restriction.

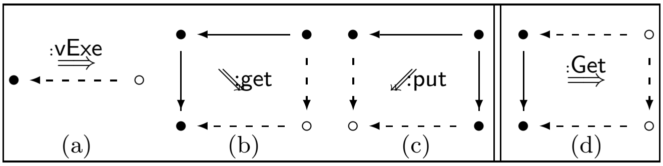


Fig. 19. View synchronization: the signature

Next we assume that the view is entirely dependent on the source: once the source is updated, the view is automatically recomputed so that the source update does not create inconsistency. On the other hand, if the view is updated, it at once becomes inconsistent with the source since only one view corresponds to the source. Hence, there is no need for the "first-check-then-enforce" principle, and any view update must be propagated back to the source to restore consistency.

The result is that in contrast to replica synchronization, it is reasonable to assume that view update propagation always acts on consistent matches as shown by the front faces of cubes in Fig. 18(a,b), and produces consistent matches. We may thus ignore inconsistent matches completely. It implies that the correctness and "first-check-then-enforce" laws of replica synchronization become redundant, and we do not need rematching operations. This setting greatly simplifies the theory of update propagation over views. The rest of the section described the basics of such a theory.

5.2 The Signature and the Laws

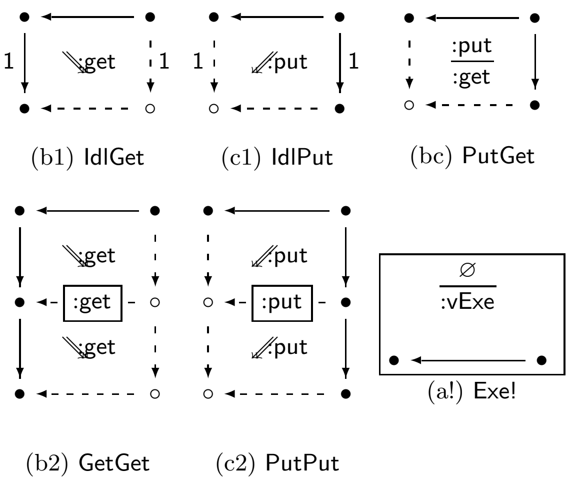
Figure 19 (a,b,c) presents arity shapes of the three operations we will consider. As before, the input nodes and arrows are black and solid, the output ones are white and dashed. The meaning of the operations is clear from the discussion above. Operation (d) will be discussed later.

Figure 20 specifies some laws the three operations must satisfy. The laws IdlGet, IdlPut, GetGet, and PutPut in cells (b1,b2,c1,c2) are quite similar to the respective laws for forward and backward propagation discussed in Sect. 4. They say that idle updates on one side result in idle updates on the other side, and composition of updates is propagated componentwise.

The PutGet law in cell (bc) states that any put-tile is automatically a get-tile. In the string-based notation, if $(a, f') = \mathsf{put}(b, f)$ then $(b, f') = \mathsf{get}(a, f)$.

The Exe! law in cell (a!) states that any match (the empty premise) is a correct view traceability mapping produced by vExe applied to the target of the match. This implies that put and get only apply to correct matches as discussed above. We could a priori postulate this, and rearrange operation get into operation Get specified in Fig. 19(d), which both computes the views and propagates updates. It is a possible way to go (cf. the functorial approach to the view update problem [26]), but this paper explores a different setting, in which vExe computes the view model only and get propagate updates using view traceability mappings.

Exercise 6. Formulate the horizontal counterparts of GetGet and PutPut, and explain their meaning. *Hint*: consider a composed view definition in Fig. 5.



(b1) IdlGet (c1) IdlPut (bc) PutGet

(b2) GetGet (c2) PutPut (a!) Exe!

Fig. 20. View synchronization: the laws

5.3 View Synchronization

Definition 6. A *view lane* $\mathbf{v}$ is given by the following data.

(a) Two categories $\mathbf{A}$ and $\mathbf{B}$, whose objects are called *models* and arrows are *updates*. Objects of $\mathbf{A}$ are called *source* models and those of $\mathbf{B}$ are *views*.

(b1) A span of sets, $\mathbf{A}_0 \overset{\partial_\mathsf{t}}{\leftarrow} \mathbf{V} \overset{\partial_\mathsf{s}}{\rightarrow} \mathbf{B}_0$ with ∂_t and ∂_s being total functions giving the *target* and the *source* for each *view traceability* mapping $f \in \mathbf{V}$. We write $A \xleftarrow{f} B$ if $\partial_\mathsf{t}(f) = A$ and $\partial_\mathsf{s}(f) = B$.

(b2) An operation $\mathsf{vExe}\colon \mathbf{A}_0 \to \mathbf{V}$ of *view execution* such that for any model $A \in \mathbf{A}_0$ and any mapping $v \in \mathbf{V}$, the following two laws hold:

(ExeDir) $\partial_\mathsf{t}\mathsf{vExe}(A) = A$

(Exe!) if $\partial_\mathsf{t} v = A$, i.e., $A \xleftarrow{v} B$, then $v = \mathsf{vExe}(A)$

Thus, for any $A \in \mathbf{A}_0$ we have a unique traceability mapping $A \xleftarrow{\mathsf{vExe}(A)} B$ targeting A, and any traceability mapping is of this form.

We denote the composition $\mathsf{vExe}; \partial_\mathsf{s}$, which gives the source of the arrow $\mathsf{vExe}(A)$, by vExe_0. Then, given a source A, its view $B = \mathsf{vExe}_0(A)$.

Evidently, $A \neq A'$ implies $\mathsf{vExe}(A) \neq \mathsf{vExe}(A')$, but it may happen that $B = \mathsf{vExe}_0(A) = \mathsf{vExe}_0(A')$ for different A, A' because view abstracts away some information.

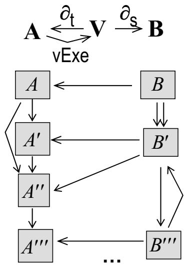


Fig. 21. View lane

Fig. 21 visualizes the definition: updates are vertical arrows, and view traceability mappings are horizontal. (Compare this figure with Fig. 17 and note the difference between the carrier structures for replication and view updates.)

Definition 8. A *view synchronizer* over a lane $\mathbf{v}$ is a pair of diagram operations $\lambda = (\mathsf{get}, \mathsf{put})$ whose arities are specified in Fig. 19. Notation λ reminds us *l*enses.

We will denote view synchronizers by arrows $\lambda\colon \mathbf{A} \to \mathbf{B}$ and write $\partial_\mathsf{s}\lambda$ for $\mathbf{A}$ and $\partial_\mathsf{t}\lambda$ for $\mathbf{B}$. With this notation, operations get of forward view maintenance and vExe of view computation go in the direction of arrow λ whereas the backward operation put goes in the opposite direction. Thus, although vExe computes from $\mathbf{A}$ to $\mathbf{B}$, all view traceability mappings computed by vExe are directed from $\mathbf{B}$ to $\mathbf{A}$.

A view synchronizer is called *well-behaved (wb)* if the pair $(\mathsf{get}, \mathsf{put})$ satisfies the laws IdlGet IdlPut, and PutGet specified in Fig. 20. A wb synchronizer is called *very well-behaved* if the laws GetGet and PutPut hold as well.

Exercise 7. Prove that in the discrete setting (mappings are just pairs of models), a (very) wb view synchronizer becomes a (very) wb lens [2].

Exercise 8. Let $\lambda_1 : \mathbf{A} \to \mathbf{B}$, $\lambda_2 : \mathbf{B} \to \mathbf{C}$ be view synchronizers defined in Sect. 5.2. Define a view synchronizer $\lambda_1; \lambda_2 : \mathbf{A} \to \mathbf{C}$ and prove that it is (very) well-behaved as soon as the components λ_i are such.
Hint: Define $\mathbf{V}^\lambda \overset{\text{def}}{=} \big\{(v_1, v_2)_A \colon\ v_1 = \mathsf{vExe}^{\lambda_1}(A), v_2 = \mathsf{vExe}^{\lambda_2}(v_1.\partial_\mathsf{s}), A \in \mathbf{A}_0\big\}$ and $\mathsf{vExe}^\lambda(A) \overset{\text{def}}{=} (v_1, v_2)_A$.

6 Complex Update Propagation: Managing Heterogeneity

In this section we consider scenarios in which the operation of update propagation is assembled from simpler propagation blocks.

6.1 Synchronization of Heterogeneous Models

Suppose that models to be synchronized are instances of different metamodels, for example, we need to keep in sync a class diagram and a sequence diagram. If one of the models is updated, say, a method in the class diagram is renamed, we need to update the sequence diagram and rename messages calling for the renamed method. Thus, we need to propagate updates across a match between heterogeneous (non-similar) models.

We will approach this problem by adapting constructions developed in Sect. 4 for homogeneous replication. Surprisingly, a precise realization of this idea is not too complicated. We will first find "the right" constructs using the metamodels, and then proceed with algebras over spaces models like in the previous section.

Matching. Discussion in Appendix D shows that heterogeneous model matching is based on metamodel matching via a span $\boldsymbol{o} = \boldsymbol{\mathcal{A}} \xleftarrow{\boldsymbol{v}} \boldsymbol{\mathcal{O}} \xrightarrow{\boldsymbol{w}} \boldsymbol{\mathcal{B}}$ in the space of metamodels, where $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are metamodels of models to be synchronized, $\boldsymbol{\mathcal{O}}$ is a metamodel specifying their overlap, and mappings $\boldsymbol{v}$,$\boldsymbol{w}$ are view definitions that make $\boldsymbol{\mathcal{O}}$ a common view to $\boldsymbol{\mathcal{A}}$, $\boldsymbol{\mathcal{B}}$. Recall that each metamodel $\boldsymbol{\mathcal{M}}$ determines a 1.5-sorted category $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{M}})$ whose objects are $\boldsymbol{\mathcal{M}}$-instances (models), vertical arrows are their updates and horizontal arrows are matches (Sect. 4.1). To simplify notation, we will use the following abbreviations. For a given span $\boldsymbol{o} = \boldsymbol{\mathcal{A}} \xleftarrow{\boldsymbol{v}} \boldsymbol{\mathcal{O}} \xrightarrow{\boldsymbol{w}} \boldsymbol{\mathcal{B}}$, bold letters $\mathbf{A}$, $\mathbf{B}$ denote the *vertical* categories (of updates) in $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{A}})$ and $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{B}})$ resp; bold letter $\mathbf{O}$ denotes the *horizontal* graph (of matches) in $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{O}})$.

We assume that the metamodel span is consistent, that is, there are no conflicts between the metamodels.

Definition 9. A *heterogeneous match* of type $\boldsymbol{o}$ is a triple $h = (A, m, B)$ with $A \in \mathbf{A}_0$, $B \in \mathbf{B}_0$, and $m\colon A\restriction_{\boldsymbol{v}} \to B\restriction_{\boldsymbol{w}}$ a match between the corresponding projections in graph $\mathbf{O}$. Match h is called *consistent* if match m is such.

Given a metamodel span $\boldsymbol{o}$, we will denote heterogeneous matches of type $\boldsymbol{o}$ by arrows $A \xrightarrow{h:\boldsymbol{o}} B$ or $h_{\boldsymbol{o}}\colon A \to B$. The typing discipline then implies that models A and B are instances of metamodels $\boldsymbol{\mathcal{A}} = \partial_{\mathsf{s}}\boldsymbol{o}$ and $\boldsymbol{\mathcal{B}} = \partial_{\mathsf{t}}\boldsymbol{o}$ resp.

Propagation. Suppose we are given a matched heterogeneous pair of models $h_{\boldsymbol{o}}\colon A \to B$. If one of the models, say A, is updated and consistency between models gets worse, we may want to propagate update $a\colon A \to A'$ to model B and restore consistency as much as possible. Thus, we need to compute an update $b\colon B \to B'$ along with an updated match $h'_{\boldsymbol{o}}\colon A' \to B'$ of the same type $\boldsymbol{o}$.

If both legs of the span $\boldsymbol{o}$ are maintainable views, and the replication space $\mathbb{M}\mathbf{od}(\boldsymbol{\mathcal{O}})$ is equipped with synchronization, a reasonable idea would be to compose update propagation from A to B from the blocks provided by synchronization mechanisms of $\boldsymbol{v}$, $\boldsymbol{\mathcal{O}}$, and $\boldsymbol{w}$. That is, having lenses $\lambda_{\boldsymbol{v}}$ and $\lambda_{\boldsymbol{w}}$, and a homogeneous replica synchronizer δ over $\mathbb{M}\mathbf{od}(\mathcal{O})$, we may try to build a heterogeneous replica synchronizer spanning model spaces $\mathbf{A}$ and $\mathbf{B}$. The rest of the section is devoted to a precise realization of this idea.

After metamodels have helped us to figure out the right concepts, we may forget about them and work within model spaces only.

Definition 10. A *triple lane* $\mathbf{t}$ is a pair of view lanes $(\mathbf{v}^l, \mathbf{v}^r)$ referred to as the *left* and the *right* lanes, with a replica lane in-between them:

$$\begin{tikzcd} \mathbf{A} \arrow[r, "\mathbf{v}^l"] & \mathbf{O} \arrow[r, "\mathbf{r}", "\bullet" marking] & \mathbf{O} & \mathbf{B}. \arrow[l, "\mathbf{v}^r"'] \end{tikzcd}$$

Categories $\mathbf{A}$, $\mathbf{B}$ are called the *ends* of the triple lane and category $\mathbf{O}$ is the *overlap.*

A *triple synchronizer* τ over a triple lane $\mathbf{t}$ is a pair of view synchronizers for the pair of view lanes and a diagonal replica synchronizer for the replica lane: $\tau = (\lambda^l, \delta, \lambda^r)$ with $\begin{tikzcd} \mathbf{A} \arrow[r, "\lambda^l"] & \mathbf{O} \arrow[r, "\delta", "\bullet" marking] & \mathbf{O} & \mathbf{B}. \arrow[l, "\lambda^r"'] \end{tikzcd}$

A triple synchronizer is called *(very) well-behaved* if all its three components are such.

Theorem 2. *Any triple synchronizer $\tau = (\lambda^l, \delta, \lambda^r)$ gives rise to a diagonal replica synchronizer Δ_τ. Moreover, the latter is (very) well-behaved as soon as all three components are such.*

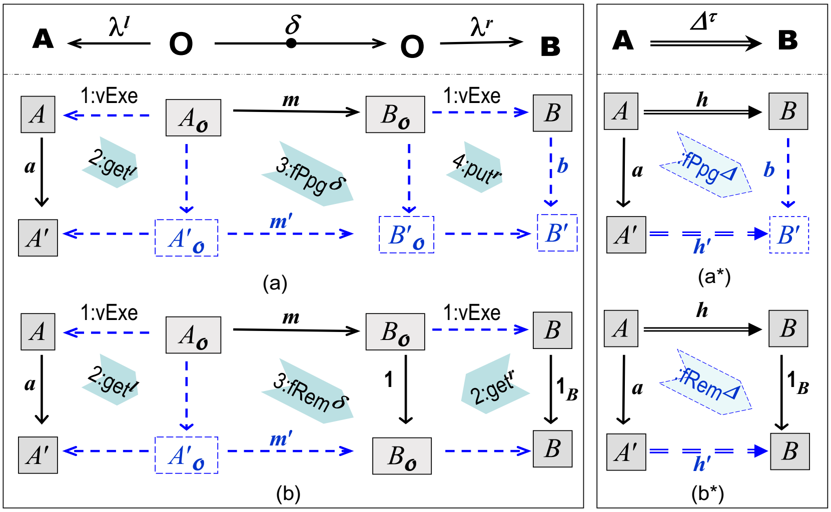


Fig. 22. Heterogeneous update propagation

The principle idea of the proof is easy and well explained by Fig. 22.

The binary replica lane for $\varDelta_\tau$ is formed by the ends of $\mathbf{A}$ and $\mathbf{B}$ of the triple lane, and with the class of matches formed by heterogeneous matches (A, m, B) as described in Definition 6.1. The subclass of consistent matches is also described in Definition 6.1.

The four operations of diagonal update propagation specified in Fig. 15 are defined by tiling the corresponding operations of the three component synchronizers: Fig. 22 shows this for forward propagation (a) and rematching (b). Applications of the operations are numbered, and concurrent applications have the same number. Algebraically, diagrams (a) and (b) specify terms that can be abbreviated by diagrams (a*) and (b*). It is exactly similar to definitions by equality in the ordinary algebra: when we write, say, $\varDelta(x, y) \overset{\text{def}}{=} ax * (b_1x + b_2y) * cy$ with x, y variables and a, b, c fixed coefficients, expression $\varDelta(x, y)$ can be considered as an abbreviation for the term on the right-hand side of equality symbol. Backward propagation and rematching are defined in exactly the same way but in the opposite direction.

Finally, we need to check that composed operations in diagrams (a*,b*) and their backward analogs satisfy the laws specified in Fig. 16. With tiling notation, this check is straightforward. Ancient Indian mathematicians used to prove their results by drawing a picture and saying "Look!". The reader is encouraged to follow this way and appreciate the benefits of diagram algebra. □

Similarly to unidirectional heterogeneous update propagation, heterogeneous bi-directional operation can be built from lenses and bi-directional synchronization over the overlap as suggested by Fig. 23 (where $\delta^\Leftrightarrow$ denotes the operation of homogeneous bi-directional update propagation). A special case of this construction for synchronizing data presented by trees was described in [3].

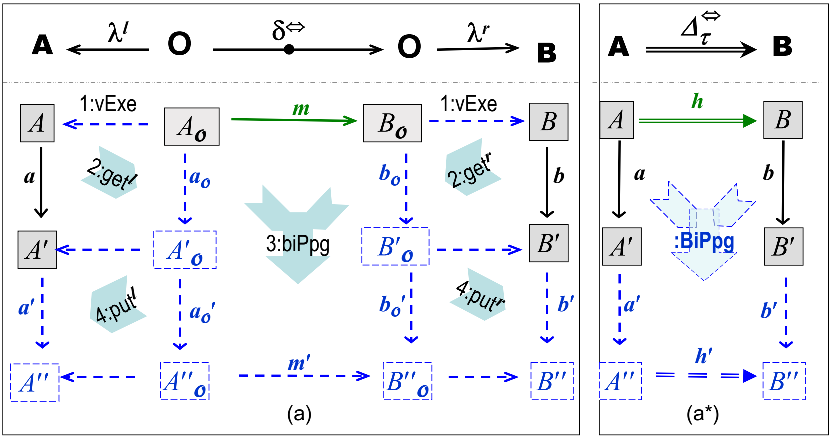


Fig. 23. Heterogeneous bi-directional update propagation

Exercise 9 (*). Define an algebra for modeling synchronization of *materialized* views, for which view data are managed independently, and inconsistency with the source is possible (though undesirable). *Hint*: The possibility of inconsistency makes this case somewhat similar to replication (Sect. 4) and distinct from ordinary views (Sect. 5.2).

6.2 Synchronization with Evolving Metamodels: A Sketch

First we note that typing can be considered as a specific kind of match. Then model adaptation to metamodel evolution can be described as backward diagonal propagation as shown by Fig. 24 (in which superscript ϵ stands for "evolution"). Arrow u encodes an ordinary (update) span in the space of metamodels. Arrow a is a span whose head is an instance of u's head, and the legs are heterogeneous model mappings over u's legs as described in Sect. D.1.

$$\begin{tikzcd}
A \arrow[r, "t_A"] \arrow[d, dashed, "a"'] \arrow[dr, phantom, ":\mathrm{bPpg}^{\epsilon}"] & \boldsymbol{\mathcal{A}} \arrow[d, "u"] \\
A' \arrow[r, dashed, "t_{A'}"'] & \boldsymbol{\mathcal{A}}'
\end{tikzcd}$$

Fig. 24. Typing as matching

Now consider a heterogeneous pair of replicas $A{:}\boldsymbol{\mathcal{A}}$ and $B{:}\boldsymbol{\mathcal{B}}$, and suppose that metamodels may change. A typical scenario is shown in Fig. 25(a). The upper face of the cube specifies a heterogeneous match defined in Sect. D.2. Suppose that metamodel $\boldsymbol{\mathcal{A}}$ is updated with $u{:}\,\boldsymbol{\mathcal{A}} \to \boldsymbol{\mathcal{A}}'$. This update can be propagated in two directions.

In the first one, update u is propagated over the left face of the cube and results in update $a{:}\ A \to A'$ adapting model A to the change. In the second direction, update u is first propagated to metamodel $\boldsymbol{\mathcal{B}}$ along the match $\boldsymbol{o}$ by the ordinary replica synchronization mechanisms (Sect. 5) but now working with the metamodels rather than models. This gives us the back face of the cube and update $v{:}\,\boldsymbol{\mathcal{B}} \to \boldsymbol{\mathcal{B}}'$ of the right metamodel. The latter is then propagated to model B by the model adaptation mechanism now applied to the right face of the cube.

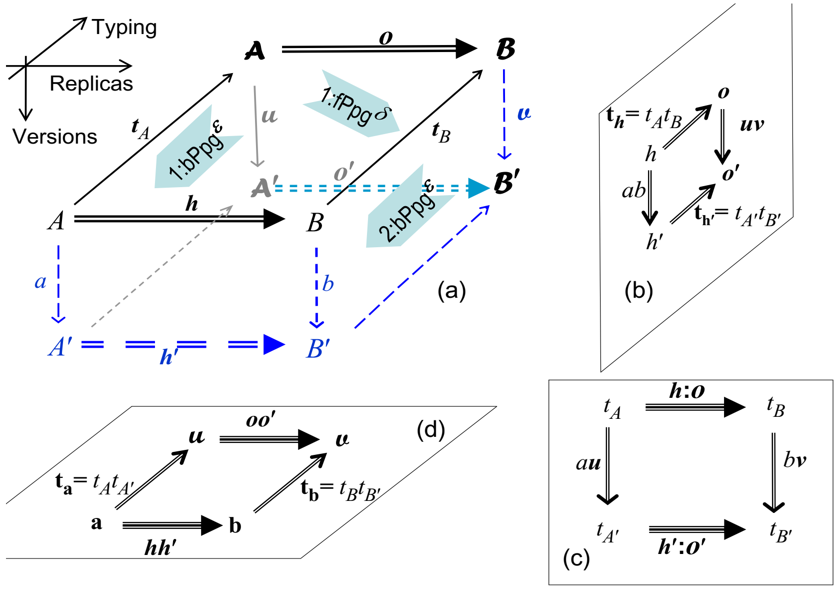


Fig. 25. 3D-synchronization with evolving metamodels

In this way we get two parallel updates a and b at the ends of match h. Having the metamodel span $\boldsymbol{o}' = (\mathcal{A}' \overset{\boldsymbol{v}'}{\leftarrow} \mathcal{O}' \overset{\boldsymbol{w}'}{\rightarrow} \mathcal{B}')$ at the back face, we may project models A' and B' to their common overlap space $\mathbb{M}\mathbf{od}(\mathcal{O}')$ thus arriving at models $A'_{\mathcal{O}} = \boldsymbol{v}'\restriction_{A'}$ and $B'_{\mathcal{O}} = \boldsymbol{w}'\restriction_{B'}$. Having match $m\colon A_{\mathcal{O}} \to B_{\mathcal{O}}$ (occurring into h) and all other information provided by the cube, we may derive a match $m'\colon A'_{\mathcal{O}} \to B'_{\mathcal{O}}$ by applying the corresponding operation to nodes and arrows of the cube (in its de-abbreviated form with all models and model mappings explicated). This would be a typically categorical exercise in diagram chasing. A theoretical obstacle to be watched is that categories involved must be closed under the required operations. Practically, it means that the required operations have to be implemented.

Thus, synchronization scenarios with evolving metamodels are deployed within a *three-sorted* graph with three sorts of arrows: vertical (updates), horizontally frontal (matches) and horizontally “deep” (typing). Since updates and types are composable, we actually have a *2.5-sorted* category. If heterogeneous match composition is also defined, we have a thin *triple* category. This pattern could be probably extended for other types of relationships between models. Hence, general synchronization scenarios are mutli-dimensional and are deployed within n-sorted graphs and categories. *Multi-dimensional category theory (mdCT)* appears to be an adequate mathematical framework for multi-dimensional synchronization.[6]

[6] Md-category seems to be a new term. The term *higher-dimensional* categories is already in use and refers to md-categories with weaker compositional laws: unitality and associativity of composition hold up to canonic isomorphisms [31]. In fact, hdCT is a different discipline, and mdCT is a proper, and very simple, sub-theory of hdCT.

2D-projections. To manage the complexity of 3D-synchronization, it is useful to apply a classic idea of descriptive geometry and study 2D-projections of the 3D-whole. We can realize the idea by arrow encapsulation, that is, by treating arrows of some sort as objects (nodes) and faces between those arrows as morphisms (complex arrows). There are three ways of applying this procedure corresponding to the three ways of viewing the cube (see the frame of reference in the left-upper corner).

Viewing the cube along the axis of Replicas means that we consider match arrows as nodes, the top and bottom faces as "deep" arrows, and the front and back faces as vertical arrows. In this view, the cube becomes a tile shown in diagram (b). If we treat typing mappings as specific matches, these tiles become similar to replica synchronization tiles from Sect. 4

In the view along the Typing axis, typing mappings are nodes, the top and bottom faces are horizontal arrows, and the left and right faces are vertical arrows (but of type different from vertical arrows of the Replicas-view). The result is shown in diagram (c). These tiles are similar to heterogeneous replication considered above but with evolving metamodels.

Finally, in the Versions view, updates are nodes, the front and back faces are new horizontal arrows, and the left and right faces are new deep arrows as shown in diagram (d). Such tiles can be seen as structures for specifying "dynamic typing", in which typing arrows are actually couples of original and updated typing mappings.

Tiles of each of the three sorts can be repeated in the respective directions and we come to three two-sorted graphs $\mathbb{G}_{\mathsf{x}}$ with $\mathsf{x} = R, T, V$ for the Replicas, Typing, Version axes. Each of the graphs is a universe for its own synchronization scenarios with different contexts. Yet there may be many similarities in the algebras of operations, and there may be core algebraic structures common to all three views. We leave this for future work.

7 Relation to Other Work, Brief Discussions, Future Work

The paper is a part of an ongoing research project on model synchronization with the Generative Software Development Lab at the University of Waterloo The project started with the GTTSE'07 paper [32] by Antkiewicz and Czarnecki, which outlined a broad landscape of heterogeneous synchronization, provided a range of examples, and introduced a notation that can be seen as a precursor of synchronization tiles. The project has been further developed in [33,11,34], and in several papers currently in progress. The present paper aims to specify a basic mathematical framework for the project, and to offer a handy yet precise notation.

Of course, this is only the short prehistory of the paper. Synchronization spans a wide range of specification problems, and the present paper (in its attempt to set a sufficiently general framework) inevitably intersects with many ideas and approaches, and builds on them. These "pre-histories" and intersections are briefly reviewed below without any aspiration to be complete. Directions for future work are also discussed.

7.1 Abstract MMt or *Model-at-a-Time* Programming

Synchronization via algebra. Analysis of synchronization problems in abstract algebraic setting is a long-standing tradition in the literature on databases and software engineering. It can be traced back to early work on the view update problem, particularly to seminal papers by Bancilhon and Spyratos [35], Dayal and Bernstein [30] and Gottlob *et al* [36]. This algebraic style was continued by Meertens in [37] in the context of constraint maintenance, and more recently was further elaborated in the work of the Harmony group on bi-directional programming languages and data synchronization [2,3,38,39,40]. An adaptation of the approach for bi-directional model transformations was developed by Stevens [6,41] and Xiong *et al* [4,42]; an analysis of the corresponding algebraic theories can be found in [28]. Paper [32] mentioned above, and an elegant relational model of bi-directional data transformations [43] by Oliveira are also within this algebraic trend.

Two features characterize the framework: (A) model mappings are not considered or implicit; and (B) metamodels (and their mappings) are either ignored or only considered extensionally —- a metamodel defines its class of instances and may be forgotten afterwards (e.g., see [3]).

Feature (A) makes the framework discrete and subject to the critique in Section 2. Feature (B) significantly simplifies technicalities but hides semantics of model translation and makes it difficult to manage heterogeneity in a controlled way.[7] The abstract MMt part of the present paper also does not include metamodels. However, the latter are central for the concrete MMt part that motivates and explains several important constructs in the abstract part.

Generic MMt. A broad vision of the model-at-a-time approach to the database metadata management was formulated by Bernstein *et al* in [44,1]. They coined the term *generic model management*, stressed the primary importance of mappings, and described several major operations with models and mappings necessary to establish a core MMt framework. This work originated a research direction surveyed in [45]. Interestingly, synchronization operations are not included in the core framework as scoped by Bernstein *et al.*

Although mappings are first-class citizens in generic MMt, a typical algebraic setting is discrete: the universe in which operations act is a set (of models and mappings) rather than a graph; the same setting is used in Manifesto [46] by Brunet *et al.* Not surprisingly, neither diagram algebra nor category

[7] Dayal and Bernstein's work [30] is a notable exception. It does use update, traceability and typing links (and, in fact, is remarkably categorical in its approach to the problem). However, these links are not organized into mappings (not to mention more advanced arrow encapsulation techniques), and technicalities become hardly manageable. The categorical potential of the paper remained undiscovered.

theory are employed in generic MMt so far; few exceptions like [47] by Alagic and Bernstein, and [48] by the present author remain episodes without follow-up. The purely extensional treatment of data schemas (feature (B) of synchronization frameworks above) is also typical for generic MMt [49]; papers [47,48] are again exceptions.

Tiles and tiling. *Tile systems* were developed by Ugo Montanari *et al* (see [50] and references therein) as a general algebraic framework for modular description of concurrent systems. The tiles represent modules and can be thought of as computations (or conditional rewriting rules) with side effects. The two horizontal arrows of a tile are the initial and the final states of the module, and the two vertical arrows are the trigger and the effect. This interpretation works for our tiles: modules are connected pairs of models, matches are their states, the input update is a trigger and the output one is the effect. However, there are important distinctions between the two tile frameworks. For the brief discussion below, we will refer to them as to *c-tiles* and *s-tiles*, with *c* standing for *concurrency* and *s* for synchronization.

(a) C-tiles have an interior in the sense that different c-tiles may have the same four-arrow boundary whereas our s-tiles are merely quadruples of arrows (in the categorical jargon, they are *thin*).

(b) Montanari *et al* only deal with operations on tiles as integral entities (*tiling-in-the-large*), and consider their vertical, horizontal and parallel composition. In contrast, we have been looking inside tiles and considered algebraic operations that produce tiles from arrows (*tiling-in-the-small*). We have also considered vertical composition-in-the-large in our XyzXyz laws, and horizontal composition in Exercise 6 on p.126.

(c) Three composition operations over c-tiles assume they are homogeneous units, and so we have *homogeneous* tiling. In contrast, our complex scenarios in Sect. 6 present *heterogeneous* tiling: a big tile is composed from smaller tiles of different types.

A perfectly adequate mathematical framework for homogeneous tiling is *double categories* [25], or *two-sorted categories* (Appendix B) for thin tiles; their s-interpretation is described in [33]. Heterogeneous tiling requires more refined algebraic means and a real diagram algebra. A general formal definition of a diagram operation appears in [51] and is specified in detail in [52]; in the present paper it is formulated in a slightly different but equivalent way. Parsing terms composed of diagram operations is discussed in [52, Appendix A].

A few historical remarks. Elements of the tile language in the context of model synchronization can be found in Antkiewicz and Czarnecki [32], and even earlier in Lämmel [53]; my paper [28] also deals with s-tiles but in the discrete setting. Operations of update propagation and conflict resolution are considered in [32] but without any equational laws. The language of s-tiles is explicitly introduced in Diskin *et al* [33] with a focus on 2D-composition and double categories. A general framework for specifying synchronization procedures via tile algebra in this paper is novel.

The 2-arrow structure. Introducing a partial order on mappings, and then ordering matches and updates, is important for model synchronization (see p. 116) and should be a part of the tile language. The issue is omitted in the paper and left for future work; some preliminary remarks are presented below.

By the principles of arrow thinking, ordering should be modeled by arrows, and we thus come to arrows between arrows or *2-arrows.* If mappings are spans, 2-arrows are ordinary arrows between their heads, but the entire structure becomes a 2-category and hence a much richer structure than an ordinary category. Another approach suggested by an anonymous referee is to work with so called *allegories* [54] rather than categories, in which morphisms are to be thought of as binary relations rather than functions. However, an important feature of any set of matching links — its structure being similar to the structure of models — is lost if mappings are simply morphisms in an allegory. Another (arguable) advantage of the span model of mappings is that it is technically easier to work with 2-categories of spans than with allegories.

Parallel updates. This synchronization scenario is very important yet omitted in the paper and left for the future work. It is a challenging problem, whose algebraic treatment needs a more elaborated framework than simple algebraic models we used. An initial attempt and some results can be found in [42].

Lenses, view synchronization and categories. The Harmony group's paper [55] was seminal. It presented a basic algebraic framework in a very transparent way; and coined several vigorous names: *get* and *put* for the two main operations, GetPut, PutGet, PutPut for equational laws imposed on these operations, and *lens* for the resulting "bi-directional" algebra. In fact, the paper set up a pattern for algebraic models of update propagation.

The basic lens framework is enriched with update mappings in [11]. Algebras introduced in [11] operate on both models and update mappings, and are called *u-lenses* with 'u' standing for updates. Earlier, a similar framework was developed by Johnson and Rosebrugh [26]. For them, updates are also arrows, a model space is a category, and a view is a functor. However, they work in the concrete rather than abstract MMt setting, and focus on conditions ensuring uniqueness of update policies. As discussed in Sect. 4.2, this setting may be very restrictive in practice.

View synchronizers of the present paper can be seen as *ut-lenses* since they operate on two types of mappings: updates and traceability. Moreover, given a view definition language with well-behaved operations of update propagation defined for any view mapping, both tile systems, of all get-tiles and of all put-tiles, give rise to two-sorted categories, say, $\mathbb{G}\boldsymbol{et}$ and $\mathbb{P}\boldsymbol{ut}$ (see Fig. 20 and Exercise 6 on p.127). In addition, the PutGet law entails inclusion $\mathbb{P}\boldsymbol{ut} \subset \mathbb{G}\boldsymbol{et}$. Proving these results is not difficult and will appear elsewhere.

Multi-dimensional synchronization. The ideas of constructing 3D-tiles (synchronization cube on p. 132), and more generally of the multi-dimensional nature of synchronization problems, seem to be new. The paper only presents a vision (Sect. 6.2), and even the initial steps are still waiting for a precise specification.

With dynamic interpretation of horizontal arrows as transformations (rather than structural mappings), 2D-projections of the synchronization cube can be seen as *coupled transformations* considered in [53], and have probably been studied by different communities in different contexts, e.g., [56]. If vertical arrows (updates) are interpreted dynamically, then the front and back faces of the cube become close to triple graph transformations [57] (with the third graph hidden in the match). Stating precise relationships is a future work.

7.2 Concrete Model Management

Inside models: constraints as diagrams predicates. For a rich software model, specifying its abstract syntax ”tree” as a mathematical object is not as easy as it may seem. One of the challenges is how to specify and manage constraints, which populate model graphs with non-instantiable elements. In the paper we have only considered very simple constraints declared for a single arrow (multiplicities). However, there are other practically important constraints involving several arrows, e.g., invertibility of two mappings going in the opposite directions, uniqueness of identification provided by a family of mappings with a common source (a key), and many other conditions that constraint languages (like OCL) allow us to specify.

A general approach to the problem is to specify such constraints as *diagram predicates* [51] and treat models as graphs with diagram predicates, *dp-graphs*. A principal distinction of this approach from the attributed typed graphs (ATGs) [58] is that a constraint is an independent object referring to respective nodes and edges rather than an attribute of a node or an edge. Theoretical advantages of the approach are its universality and proximity to an established framework of the so called *sketches* developed in categorical logic (see [16] for details). The approach was shown to be useful in schema integration [59], conceptual modeling [60], and fixing known problems with UML associations [61]. An accurate algebraic model of metamodeling with diagrammatic constraints is an important direction for future work.

Homogeneous model mappings and deltas. Specifying (symmetric) deltas is a known issue, e.g., [9,10,62,63]. A major challenge is how to formally specify model *changes*: *modifications*, if we interpret deltas as updates, or *conflicts*, if deltas are matches. A well-known idea is to treat a modification of an element as its deletion followed by insertion; but it is a simplistic treatment. The approach developed in the paper (for our OSV-models) is more adequate and still simple but is not straightforward. First, value-preserving model mappings are defined; then changes are specified by spans built from two value-preserving mappings but having empty slots. This treatment of changes seems correlating with ATG transformations but a precise comparison needs some technical work to be done. Generalization of the idea for more practically interesting (and hence more complex) models than simple OSV-models is important future work.

Heterogeneous model mappings and deltas. Precise specification of operations on heterogeneous models and model mappings is a rarity in the literature because of semantic and technical difficulties. It is managed in the paper by specifying heterogeneous models as chains of graphs and graph mappings; model mappings then appear as multi-layer commutative diagrams. The idea seems to be more or less evident but I am not aware of its realization in the literature.

Despite their frightening appearance, universes of multi-layer complex objects and mappings are well-known in CT under the name of *arrow categories*. They are well-studied and behave very well. Unfortunately, constraints may be an obstacle: while any model is a chain of graph mappings, not any such chain is a model because it may violate the constraints. It implies that the universe of models may be *not* closed wrt. some operations, e.g., merging (colimit) [34]. How graph-based constraints declared in a metamodel affect the properties of the corresponding universes of models is a big issue studied in categorical logic. Its adaptation to metamodeling with diagram predicates (as constraints) [16] is important future work.

Model translation (MT) and fibrations. The algebraic model of MT proposed in the paper is generic and formulated for any metamodel language, including an associated query language. In this model, MT is treated as a view computation, and is entirely determined by the corresponding metamodel mapping considered as a view definition. The idea was first described in [48]; the description in the present paper is more accurate and detailed. It culminates in the statement that the view mechanism (for monotonic queries) makes the functor projecting heterogeneous models and mappings to their metamodel components a split *fibration* — a construct well known and studied in CT.

Fibrational formulation can be seen as dual to the familiar *functorial semantics*: a model is a functor from the metamodel (theory) to some semantic category, e.g., of sets and relations. Functorial semantics is quite popular in the Algebraic Specification community [64], and is basic for the categorical approach to the view update problem developed in [26], but it may seem foreign for a model transformation engineer accustomed to work with metamodeling patterns. The latter assume that a model is given by a (typing) mappings *to* the metamodel rather than *from* it. Fibrations fit perfectly in this framework, but offer much more. Practical modeling situations often comprise instances at several levels, say, objects, classes, and the metamodel for the latter (e.g., a simple sequence diagram is a three-level structure of this kind [65]). Specification of multilevel modeling is quite manageable with fibrations: composition of fibrations is again a fibration (this is a well-known result). In contrast, functorial semantics becomes hard to manage when we consider more than one pair (theory, model).

8 Conclusion

Building theoretical foundations for model synchronization is a challenging problem. Among the factors contributing to its complexity are heterogeneity of models to be synchronized, the multitude and heterogeneity of their relationships, and interactions between different dimensions of synchronization. The paper aims to show that algebraic models based on diagram operations can be an effective means to manage the complexity of the problem.

Two lines of approaches and results are presented. The first one is *abstract*: models and model mappings are treated as indivisible (black-box) nodes and arrows, on which synchronization procedures operate. The machinery used is algebra of tile operations and tiling as term substitution. The abstract line culminates in Sect. 6, which shows how complex synchronizers can be assembled by tiling together simple components. The second line is *concrete*: it provides algebraic models for (white-box) complex structures underlying models and model mappings. The machinery is essentially categorical: arrow categories (for heterogeneous models and their mappings) and fibrations (for the view mechanism). Tile algebra is applicable here as well.

The tile framework offers a handy notation with formal semantics, and a toolbox of constructs amenable to algebraic manipulations and hence to automated computer processing. This benefit package may be very appealing for a software engineer.

Synchronization scenarios considered in the paper are deployed on 2D-planes of a 3D-space populated by models and model mappings (and a 3D-scenario with evolving metamodels is sketched in Sect. 6.2). The three dimensions correspond to the three kinds of intermodel relationships (and mappings) that were considered: replication (matches), versioning (updates), metamodeling (typing). Other kinds of relationships can give rise to new dimensions of the space and synchronization procedures spanning it. Handy yet precise tile notation and the corresponding algebraic framework can be an invaluable tool for multi-dimensional synchronization.

Acknowledgements. I'm grateful to the organizers of GTTSE'09 for the invitation to lecture at the school and for the chance to enjoy its unique atmosphere — stimulating, challenging, and warm. I'd like to thank the same team for organizing qualified and friendly reviewing, and their enormous editor's patience. I'm indebted to the anonymous referees for valuable feedback and criticism, which greatly improved the structure and presentation of the paper. Special thanks to the referee who also provided detailed technical comments and suggestions.

Different parts of the paper were presented at our seminars at the University of Waterloo, and I'm grateful to Michał Antkiewicz, Krzysztof Czarnecki and Yingfei Xiong for valuable feedback and constructive criticism. Several discussions of the paper's general ideas and technical fragments with Tom Maibaum were thought-provoking and invigorating. In addition, Tom read the entire manuscript and provided many helpful remarks. Finally, thanks to GTTSE'09 students and participants for their attention, stimulating questions, and discussions that supported my enthusiasm about the practical use of abstract diagrams.

Financial support was provided by the Ontario Research Fund.

References

1. Bernstein, P.: Applying model management to classical metadata problems. In: Proc. CIDR 2003, pp. 209–220 (2003)
2. Foster, J.N., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
3. Foster, J.N., Greenwald, M., Kirkegaard, C., Pierce, B., Schmitt, A.: Exploiting schemas in data synchronization. J. Comput. Syst. Sci. 73(4), 669–689 (2007)
4. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE, pp. 164–173 (2007)
5. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ICFP, pp. 47–58 (2007)
6. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
7. Czarnecki, K., Foster, J., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional transformations: A cross-discipline perspective. [66], 260–283
8. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Redmiles, D., Ellman, T., Zisman, A. (eds.) ASE, pp. 54–65. ACM, New York (2005)
9. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-Specific Models. European J. of Information Systems 16, 349–361 (2007)
10. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: ESEC/SIFSOFT FSE, pp. 295–304 (2007)
11. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 61–76. Springer, Heidelberg (2010)
12. Balzer, R.: Tolerating inconsistency. In: ICSE, pp. 158–165 (1991)
13. Melnik, S., Rahm, E., Bernstein, P.: Developing metadata-intensive applications with Rondo. J. Web Semantics 1, 47–74 (2003)
14. Dyreson, C.E.: A bibliography on uncertainty management in information systems. In: Uncertainty Management in Information Systems, pp. 415–458 (1996)
15. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2004)
16. Diskin, Z., Wolter, U.: A diagrammatic logic for object-oriented visual modeling. Electron. Notes Theor. Comput. Sci. 203(6), 19–41 (2008)
17. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In: ICDE, pp. 117–128. IEEE Computer Society, Los Alamitos (2002)
18. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. [67], 326–340
19. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (2005)
20. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Software Eng. 28(5), 449–462 (2002)
21. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
22. Ohst, D., Welle, M., Kelter, U.: Differences between versions of uml diagrams. In: ESEC / SIGSOFT FSE, pp. 227–236. ACM, New York (2003)

23. Gadducci, F., Montanari, U.: The tile model. In: Proof, Language, and Interaction, pp. 133–166 (2000)
24. Sabetzadeh, M., Easterbrook, S.: An algebraic framework for merging incomplete and inconsistent views. In: 13th Int. Conference on Requirement Engineering (2005)
25. Kelly, G., Street, R.: Review of the elements of 2-categories. In: Category Seminar, Sydney 1972/73. Lecture Notes in Math., vol. 420, pp. 75–103 (1974)
26. Johnson, M., Rosebrugh, R.: Fibrations and universal view updatability. Theor. Comput. Sci. 388(1-3), 109–129 (2007)
27. Goguen, J.A., Meseguer, J.: Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. 105(2), 217–273 (1992)
28. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: [67], pp. 21–36. Springer, Heidelberg (2008)
29. Liefke, H., Davidson, S.: View maintenance for hierarchical semistructured data. In: Kambayashi, Y., Mohania, M., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, pp. 114–125. Springer, Heidelberg (2000)
30. Dayal, U., Bernstein, P.: On the correct translation of update operations on relational views. TODS 7(3), 381–416 (1982)
31. Leinster, T.: Higher Operads, Higher Categories. Cambridge University Press, Cambridge (2004)
32. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: [68], pp. 3–46
33. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: Algebraic foundations and the tile notation. In: ICSE 2009 Workshop on Comparison and Versioning of Software Models, pp. 7–12. ACM, New York (2009), doi: 10.1109/CVSM.2009.5071715, ISBN: 978-1-4244-3714-6
34. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: First International Workshop on Model-Driven Interoperability, MDI 2010, pp. 42–51. ACM, New York (2010), doi: 10.1145/1866272.1866279, ISBN: 978-1-4503-0292-0
35. Bancilhon, F., Spyratos, N.: Update semantics of relational views. TODS 6(4), 557–575 (1981)
36. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM TODS 13(4), 486–524 (1988)
37. Meertens, L.: Designing constraint maintainers for user interaction (1998), http://www.kestrel.edu/home/people/meertens/
38. Bohannon, A., Foster, J.N., Pierce, B., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: POPL, pp. 407–419 (2008)
39. Bohannon, A., Pierce, B., Vaughan, J.: Relational lenses: a language for updatable views. In: PODS (2006)
40. Hofmann, M., Pierce, B., Wagner, D.: Symmetric lenses. In: POPL (2011)
41. Stevens, P.: A landscape of bidirectional model transformations. [68], 408–424
42. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. [66], 213–228
43. Oliveira, J.N.: Transforming data by calculation. [68], 134–195
44. Bernstein, P., Halevy, A., Pottinger, R.: A vision for management of complex models. SIGMOD Record 29(4), 55–63 (2000)
45. Bernstein, P., Melnik, S.: Model management 2.0: manipulating richer mappings. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD 2007, pp. 1–12. ACM Press, New York (2007)

46. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: GaMMa (2006)
47. Alagic, S., Bernstein, P.: A model theory for generic schema management. In: Ghelli, G., Grahne, G. (eds.) DBPL 2001. LNCS, vol. 2397, p. 228. Springer, Heidelberg (2002)
48. Diskin, Z.: Mathematics of generic specifications for model management. In: Rivero, L., Doorn, J., Ferraggine, V. (eds.) Encyclopedia of Database Technologies and Applications, pp. 351–366. Idea Group, USA (2005)
49. Melnik, S., Bernstein, P., Halevy, A., Rahm, E.: Supporting executable mappings in model management. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 167–178. ACM Press, New York (2005)
50. Bruni, R., Meseguer, J., Montanari, U.: Symmetric monoidal and cartesian double categories as a semantic framework for tile logic. Mathematical Structures in Computer Science 12(1), 53–90 (2002)
51. Diskin, Z.: Towards algebraic graph-based model theory for computer science. Bulletin of Symbolic Logic 3, 144–145 (1997)
52. 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), http://www.cs.toronto.edu/~zdiskin/Pubs/TR-9602.pdf
53. Lammel, R.: Coupled software transformation. In: Workshop on Software Evolution and TRanformation (2004)
54. Freyd, P., Scedrov, A.: Categories, Allegories. Elsevier Science Publishers, Amsterdam (1990)
55. Foster, J., Greenwald, M., Moore, J., Pierce, B., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view update problem. In: POPL, pp. 233–246 (2005)
56. Berdaguer, P., Cunha, A., Pacheco, H., Visser, J.: Coupled schema transformation and data conversion for xml and sql. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 290–304. Springer, Heidelberg (2006)
57. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
58. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph tranformations. Springer, Heidelberg (2006)
59. Cadish, B., Diskin, Z.: Heterogenious view integration via sketches and equations. In: Michalewicz, M., Raś, Z.W. (eds.) ISMIS 1996. LNCS (LNAI), vol. 1079, pp. 603–612. Springer, Heidelberg (1996)
60. Diskin, Z., Kadish, B.: Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. Data & Knowledge Engineering 47, 1–59 (2003)
61. Diskin, Z., Easterbrook, S., Dingel, J.: Engineering associations: from models to code and back through semantics. In: Paige, R., Meyer, B. (eds.) TOOLS Europe 2008, LNBIP, vol. 11. Springer, Heidelberg (2008)
62. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A metamodel independent approach to difference representation. JOT 6 (2007)
63. Abi-Antoun, M., Aldrich, J., Nahas, N.H., Schmerl, B.R., Garlan, D.: Differencing and merging of architectural views. Autom. Softw. Eng. 15(1), 35–74 (2008)
64. Ehrig, H., Große-Rhode, M., Wolter, U.: Application of category theory to the area of algebaric specifications in computer science. Applied Categorical Structures 6, 1–35 (1998)
65. Liang, H., Diskin, Z., Dingel, J., Posse, E.: A general approach for scenario integration. [67], 204–218

66. Paige, R.F. (ed.): ICMT 2009. LNCS, vol. 5563. Springer, Heidelberg (2009)
67. Czarnecki, K., Ober, I., Bruel, J., Uhl, A., Völter, M. (eds.): MODELS 2008. LNCS, vol. 5301. Springer, Heidelberg (2008)
68. Lämmel, R., Visser, J., Saraiva, J. (eds.): GTTSE 2007. LNCS, vol. 5235. Springer, Heidelberg (2008)
69. Barr, M., Wells, C.: Category theory for computing science. Prentice Hall, Englewood Cliffs (1995)
70. Adamek, J., Herrlich, H., Strecker, G.: Abstarct and concrete categories. The joy of cats. TAC Reprints, No.17 (2007)
71. Bruni, R., Gadducci, F.: Some algebraic laws for spans. Electr. Notes Theor. Comput. Sci. 44(3) (2001)
72. Diskin, Z.: Model transformation as view computation: an algebraic approach. Technical Report CSRG-573, the University of Toronto (2008)
73. Manes, E.: Algebraic Theories. Graduate Text in Mathematics. Springer, Heidelberg (1976)
74. Jacobs, B.: Categorical logic and type theory. Elsevier Science Publishers, Amsterdam (1999)

Answers to *-Exercises

Exercise 2 (p. 119) Diagram Fig. 14(c1) says that from a match between replicas a new match can be computed *without* changing the replicas. This situation is typical and is a built-in procedure in many differencing tools. However, it cannot be modeled by an algebraic operation of the arity shown in the figure: to recompute a match a new information is required. That is, we may have a reasonable "binary" operation $(m, X) \bullet\!\!\longrightarrow m'$ with the second argument standing for contextual information about replicas, but the "unary" operation $m \bullet\!\!\longrightarrow m'$ is not too sensible. In contrast, the operation specified by diagram (c2) is quite reasonable and may be called *rematching*: having one of the replicas updated, we recompute the match based on data in the original match and the update.

Exercise 4(p. 123)
Three reasonable laws the operation should satisfy are specified in Fig. 26. Diagram (a1) states that nothing is done with consistent replicas. Diagram (a2) says that conflict resolution is an idempotent operation. Match m' produced by the operation is not supposed to be necessarily consistent: some of the conflicts embodied in match m may need additional information and user's input, and hence cannot be resolved automatically. Yet everything that could be done automatically is done with the first run of the operation. Diagram (a3) says that resolution is complete in the sense that nothing can be propagated in the tile produced by Res.

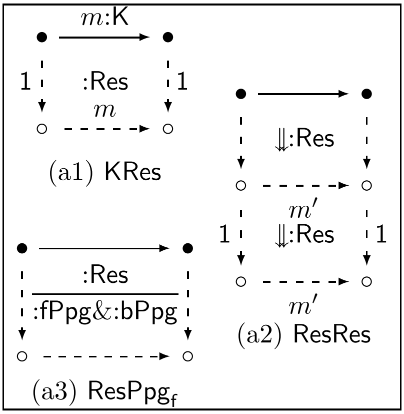


Fig. 26. Laws of conflict resolution

Exercise 9 (p. 131) Synchronization of materialized views can be considered as a particular cases of triple synchronization, in which one view (say, the right one) is identity. Formal definitions are as follows.

A *semi-triple lane* is a pair $\mathbf{st} = (\mathbf{v}, \mathbf{r})$ with $\mathbf{v}$ a view and $\mathbf{r}$ a replica lane coordinated as follows: $\mathbf{A} \xrightarrow{\mathbf{v}} \mathbf{B} \overset{\mathbf{r}}{\longrightarrow\!\!\bullet\!\!\rightarrow} \mathbf{B}$. A *semi-triple synchronizer* σ over a semi-triple lane $\mathbf{st}$ is a pair $\sigma = (\lambda, \delta)$ of a view synchronizer λ over the view lane and a diagonal replica synchronizer δ over the replica lane. A semi-triple synchronizer is called *(very) well-behaved* if its two components are such.

The following result is analogous to Theorem 2.

Theorem 3. *Any semi-triple synchronizer $\sigma = (\lambda, \delta)$ gives rise to a diagonal replica synchronizer Δ_σ. Moreover, the latter is (very) well-behaved as soon as its two components are such.*

Appendices
Concrete MMt and Category Theory

Several words about category theory (CT) are in order. CT provides a number of patterns for structure specification and operation. Since models and model mappings are rich structures, and MMt needs to operate them, CT should be of direct relevance for MMt. Of course, this theoretical prerequisite requires practical justification and examples.

Two fundamental categorical ideas are used in the paper.

Encapsulation 1: "To objectify means to mappify". The internal structure of models and model mappings is encapsulated. Models are considered as indivisible objects (points), and mappings as indivisible morphisms (arrows) between them. Mappings of the same type can be sequentially composed and form a category (a graph with associatively composable arrows). Although objects are encapsulated, the categorical language provides sufficient means to recover the internal structure of objects via mappings adjoint to them. For example, a special family of mappings with a common source object makes this object similar to a relation (and its "elements" can be thought of as tuples). Dually, a special family of mappings with a common target object makes it similar to a disjoint union (and its "elements" can be thought of as "either..or" variants). The next section shows how it works.

Encapsulation 2: Arrow categories. Repeatable constructions consisting of several models and mappings are considered as new complex objects or arrows, which can again be encapsulated and so on. In this way we come to categories whose objects (nodes) themselves consist of arrows, while morphisms (arrows) are complex diagrams. For example, a model is, in fact, a typing mapping, and a traceability mapping is a commutative square diagram like the top face of cube Fig. 3(b). Deltas-as-spans denoted by arrows are another simple example. We will build progressively more complex arrow categories in the subsequent sections C and D. Formalization of the sketch presented in Sect. 6.2 requires even more complex arrow encapsulating constructions.

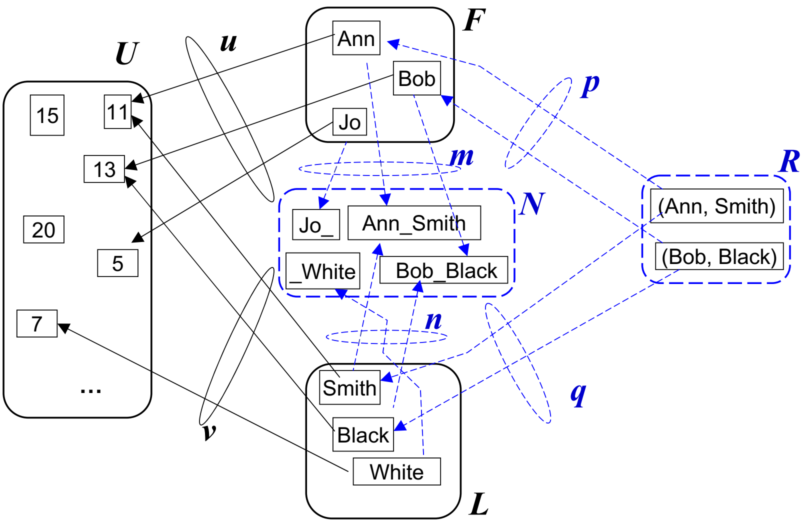


Fig. 27. Matching and merging sets via elements

A Match and Merge as Diagram Operations: Warming Up for Category Theory

This section aims to give a notion of how basic categorical patterns can work in MMt. We will begin with a very simple model of models by considering them as sets of unstructured elements (points), and discuss matching and merging sets. Then we will reformulate the example in abstract terms and come to categories.

A.1 Matching and Merging via Elements

Suppose that our models are sets of strings denoting names, and we have two sets, F of First and L of Last names, of some group of people as shown in Fig. 27. We also assume that for each of the sets, different elements refer to different persons. It does not exclude the situation when an F-name and an L-name refer to the same person, but without additional information, sets F and L are entirely unrelated and disjoint. To match the sets, we map them into some common universe U, say, by assigning to each string the social security number (SSN) of the corresponding person as shown in the left part of the figure. Following UML, we call such assignments *(directed) links* and denote them by arrows (Ann→11, Bob→13 and so on); speaking formally, links are just ordered pairs. Similar (i.e., having the same source and target) links are collected into *mappings*, $u\colon F \Rightarrow U$ and $v\colon L \Rightarrow U$, which are denoted by double-body arrows to distinguish them from link-arrows. We call triple (U, u, v) a *matching cospan* between sets F and L, set U is its *head* and mappings u, v are the *legs*.

Now we may form set $R = \{(x,y)\colon\ u(x) = v(y)\}$ consisting of those pairs of names, which are mapped to the same SSN. This set is equipped with two projection mappings $p\colon R \Rightarrow F$, $q\colon R \Rightarrow L$ giving the components of the pairs. The way we built R implies that sequential compositions of mappings $p;u$ and $q;v$ are equal: $(x,y).p.u = (x,y).q.v$ for any element $(x,y) \in R$. The triple (R,p,q) is called a *correspondence span* or *matching span* between sets F and L; set R is its *head* and mappings p,q are the *legs*. To show that the components of the span are *derived* from mappings u,v, they are denoted by dashed lines (blue with a color display).

Each pair (x,y) in the head R of the span says that actually elements $x.p \in F$ and $y.q \in L$ refer to the same object of the real world (at least, to the same SSN). Hence, we may be interested in merging sets F and L without duplication of information, that is, by gluing together the first and last names of the same person. Set N of names in the middle of Fig. 27 presents the result. It is formed by first taking disjoint union of sets F and L, and then gluing together those elements, which are declared to be the same by the span. For example, we join Ann and Smith since there is a pair (Ann, Smith) in set R. Since there are two elements in R, set N has four (rather than six) elements. Note also mappings $m\colon F \Rightarrow N$ and $n\colon L \Rightarrow N$ embedding the original sets into the merge.

How joined names are formed is a matter of taste: Ann_Smith, or Ann*Smith or AnnSmith will all work to show that Ann and Smith are two different representations of the same object. In fact, all work is done by inclusion mappings m and n that map Ann and Smith to the same element in N. Similarly, the concrete nature of elements in set R does not matter: it is mappings p,q that do the job and specify that elements of R are pairs. Hence, strictly speaking, sets R and N may be defined up to isomorphism: the internal structure of their elements is not important.

Since the internal structure of elements in sets R and N is not important, it is tempting to try to rewrite the entire construction in terms of sets and mappings only, without elements at all. Such a *pointfree* rewriting, apart of satisfying purely intellectual curiosity, would be practically useful too. If we were able to specify object matching and merging only based on mappings between objects without use of their internal structure, we would have generic patterns of match and merge working similarly for such diverse objects as sets, graphs, typed attributed graphs and so on. The benefits are essential and justify some technical work to be done.

A.2 Matching and Merging via Arrows

Matching. Figure 28(a) presents a more abstract view of our matching construction. Nodes denote sets, and arrows are mappings (functions) between them. Double-frames of nodes and double-bodies of arrows remind us that they have *extension*, i.e., consist of elements (points and links respectively).

Labels in square brackets denote *diagram predicates*, that is, properties of arrow diagrams on which these labels are "hung". Label [=] is assigned to the entire square diagram and declares its commutativity, that is, the property $p;u = q;v$ (i.e., in terms of elements, $r.p.u = r.q.v$ for any $r \in R$). Label [key] is assigned to the arrow span (p,q) and declares the following property: for any $r_1, r_2 \in R$,

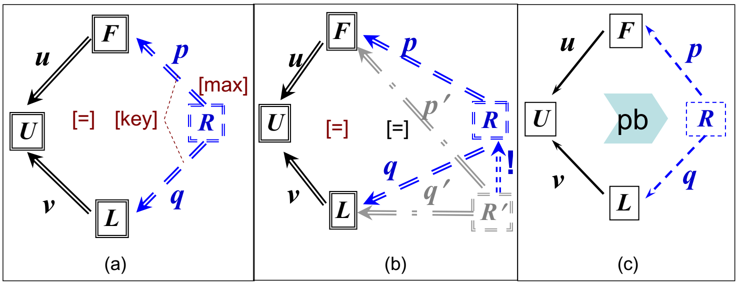


Fig. 28. Matching sets via arrows

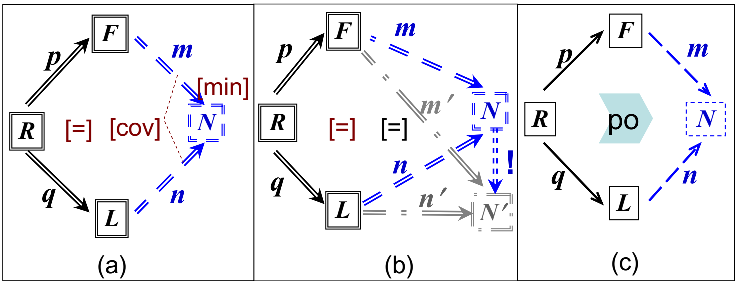


Fig. 29. Merging sets via arrows

$r_1 = r_2$ iff $r_1.p = r_2.p$ and $r_1.q = r_2.q$. That is, the pair of mappings (p, q) can identify the elements in set R, hence the name for the predicate. (In category theory, such families of mappings are called *jointly monic*). Note that any subset of set R defined in Fig. 27 will satisfy predicates [=] and [key]. Hence, to ensure that set R in Fig. 28 is indeed R defined in Fig. 27, we add one more predicate [max] stating R's maximality. Formally, it may be formulated as follows: for any other key span (p', q') as shown in Fig. 28(b), which makes the entire square commutative, there is a mapping $!\colon R' \to R$ such that $!; p = p'$ and $!; q = q'$.

Thus, we have reformulated the task of matching sets in terms of mappings, their composition and predicate [key]. However, the latter also can be expressed via mappings and composition!

Suppose that span (p, q) is not required to be a key, but has the following property: for any other span (p', q') (also not assumed to be a key), which makes the entire square commutative, there is a *unique* mapping $!\colon R' \to R$ such that $!; p = p'$ and $!; q = q'$. This maximality property is distinct from that previously formulated by the uniqueness requirement, and this is what does the job. That is, we can prove that uniqueness of ! implies the [key] property of span (p, q). Given an element $r' \in R'$, let $f' = r'.p'$ and $l' = r'.q'$ be its "names". To ensure commutativity conditions: $!; p = p'$ and $!; q = q'$, function ! must map r' into any element r of R with the same names: $r.p = f'$ and $r.q = l'$. If span (p, q) is not a key, there may be several such elements r and hence several functions ! providing commutativity. Hence, ! is unique iff span (p, q) is a key.

Thus, we may replace predicates [key] and [max] of span (p, q) in Fig. 28(a) by the uniqueness property: for any other span (p', q') that makes the entire square commutative (Fig. 28b), there is a unique mapping $!\colon R' \to R$ such that $!; p = p'$ and $!; q = q'$. In category theory such properties are called *universal*. The entire matching construction can now be formulated in abstract terms as follows. Given a cospan (u, v), a span with the same feet is derived and together with the original cospan makes a commutative square with the universal property described above. An operation producing a universal span from a matching cospan is called *pullback* (because it pulls two arrows back). The result is shown in Fig. 28(c) which depicts abstract nodes and arrows (single lines) whose internal structure is invisible.

Is pullback indeed an operation, i.e., does it indeed result in a uniquely determined span? The answer is almost positive: the result of a pullback is defined up to isomorphism. The proof can be found in any CT-textbook, e.g., [69][Theorem 5.2.2], and essentially uses associativity of arrow composition. Other constructions based on universal properties are also defined up to isomorphism.

Merging. Our construction of merging sets can be processed in a similar way. Figure 29 presents the ideas in parallel to Fig. 28. Diagram predicate [cov] declared for cospan (m, n) says that the two mappings jointly cover the target, that is, any element $e \in N$ is either in the image of mapping m or n or both. We replace this predicate by the following universal property: for any other cospan (m', n') making the entire square commutative, there exists a unique mapping $!\colon N \to N'$ such that $m; ! = m'$ and $n; ! = n'$. Indeed, if set N would contain an element e beyond the union of images of m, n, mapping ! could map this e to any element of N' without destroying the commutativity conditions.

Thus, we can define set merge in terms of mappings, their composition and the universal property of minimality. The operation that takes a span and produces a cospan making a commutative square with the minimal universal property is called *pushout* (as it pushes arrows out). The construction is dual to the construction of pullback in the sense that all arrows in the diagrams are reversed, and universal maximality is replaced by universal minimality.[8] Particularly, the result of pushout is also defined up to isomorphism.

Summary. We have defined matching and merging sets via mappings (functions) between sets and their sequential composition. Of course, to define the latter, we still need the notion of element: composition of mappings $f\colon A \to B$ and $g\colon B \to C$ is defined by setting $x.(f; g) \overset{\text{def}}{=} (x.f).g$ for all elements $x \in A$. However, if we consider some universe of abstract objects (nodes) and abstract associatively composable mappings (arrows) between them, then we can define pullback and pushout operation as described above. Such graphs are called *categories* and, thus, the notions of match and merge can be defined for any category irrespective of the internal structure of its objects. The next sections provides precise definitions.

[8] It can be made perfectly dual if we formulate the predicate [cov] in a different way exactly dual to predicate [key].

B Graphs, Categories and Diagrams: A Primer

In this section we fix notation and terminology about graphs and categories. We also accurately define diagrams and diagram operations.

Graphs and graph mappings. A *(directed multi-)graph* $\boldsymbol{G}$ consists of a set of *nodes* $\boldsymbol{G}_0$ and a set of *arrows* $\boldsymbol{G}_1$ together with two functions $\partial_{\mathsf{x}}\colon \boldsymbol{G}_1 \to \boldsymbol{G}_0$, $\mathsf{x} = s, t$. For an arrow a we write $a\colon N \to N'$ if $\partial_{\mathsf{s}} a = N$ and $\partial_{\mathsf{t}} a = N'$. The set of all arrows $a\colon N \to N'$ is denoted by $\boldsymbol{G}(N, N')$ or, sometimes, by $(N \to N')$ if graph G is given by the context.

A *graph mapping (morphism)* $f\colon \boldsymbol{G} \to \boldsymbol{G}'$ is a pair of functions $f_i\colon \boldsymbol{G}_i \to \boldsymbol{G}'_i$, $i = 0, 1$, compatible with incidence of nodes and arrows: $\partial_{\mathsf{s}} f_1(a){=}f_0(\partial_{\mathsf{s}} a)$ and $\partial_{\mathsf{t}} f_1(a){=}f_0(\partial_{\mathsf{t}} a)$ for any arrow $a \in \boldsymbol{G}_1$.

A graph is *reflexive* if every node N has a special *identity* loop $1_N\colon N \to N$. In other words, there is an operation $1_{\_}\colon \boldsymbol{G}_0 \to \boldsymbol{G}_1$ (with argument placed at the under-bar subscript) s.t. $\partial_{\mathsf{s}} 1_N = N = \partial_{\mathsf{t}} 1_N$ for any node N. If arrows are understood behaviorally (rather than structurally) as actions or transitions, identity loops may be also called *idle* (actions that do nothing and do not change the state). A reflexive graph *mapping (morphism)* is a graph mapping $f\colon \boldsymbol{G} \to \boldsymbol{G}'$ respecting identities: $f_1(1_N) = 1_{f_0(N)}$ for any node $N \in \boldsymbol{G}_0$.

Categories and functors. A *category* $\boldsymbol{C}$ is a reflexive graph $|\boldsymbol{C}|$ with an operation of *arrow composition* denoted by ; (semi-colon): for any pair of sequentially composable arrows $a\colon M \to N$ and $b\colon N \to O$, a unique arrow $a; b\colon M \to O$ is defined. Composition is required to be *associative*: $(a; b); c = a; (b; c)$ for any triple a, b, c of composable arrows; and *unital*: $1_{\partial_0(a)}; a = a = a; 1_{\partial_1(a)}$ for any arrow a.

Nodes in a category are usually called *objects*, and arrows are often called *morphisms*. Both a category $\boldsymbol{C}$ and its underlying graph $|\boldsymbol{C}|$ are normally denoted by the same letter $\boldsymbol{C}$. Thus, $\boldsymbol{C}_0$ and $\boldsymbol{C}_1$ denote the classes of all objects and all morphisms resp. The class of objects $\boldsymbol{C}_0$ can also be considered as a *discrete* category, whose only arrows are identities.

A category is called *thin* if for any pair of nodes (N, N') there is at most one arrow $a\colon N \to N'$. It is easy to see that a thin category is nothing but a preordered set with $N \leq N'$ iff there is an arrow $A\colon N \to N'$. Transitivity and reflexivity are provided by arrow composition and idle loops resp.

A *functor* $f\colon \boldsymbol{C} \to \boldsymbol{C}'$ between categories is a morphism of the underlying reflexive graphs that preserves arrow composition $f_1(a; b) = (f_1 a); (f_1 b)$.

Two-sorted graphs and 1.5-sorted categories. A *two-sorted graph* is a graph $\mathbb{G}$ whose arrows are classified into *horizontal* and *vertical*. That is, we have two disjoint graphs $\mathbb{G}_1^{\mathsf{h}}$ and $\mathbb{G}_1^{\mathsf{v}}$ sharing the same class of nodes $\mathbb{G}_0$. A two-sorted graph is *reflexive* if each node has both the *vertical* and the *horizontal* identity. A *two-sorted graph morphism (mapping)* is a graph mapping respecting arrow sorts.

A *two-sorted category* is a two-sorted reflexive graph $\mathbb{G}$ whose horizontal and vertical graphs are categories. Since horizontal composition (of matches) may be problematic, in the paper we deal with *1.5-sorted categories*: two-sorted reflexive graphs in which only vertical arrows are composable and form a category.

Flat vs. deep graphs and categories. There are two ways of interpreting elements of graphs and categories that we will call *flat* and *deep*. According to a flat interpretation, elements of a graph do not have an internal structure, they are symbols/tokens that can be drawn on paper. A visual representation/picture of such a graph drawn on paper is practically equivalent to the graph itself (up to inessential visual nuances like sizes of nodes and thickness of arrows).

According to a deep interpretation, nodes of a graph are thought of as sets endowed with some structure, for example, plain sets with empty structure, or sets with a partial order (posets), or vector spaces, or flat graphs, or models over a given metamodel M. Correspondingly, arrows are thought of as structure-preserving mappings, e.g., functions between sets, monotone functions between posets, linear mappings between vector spaces, graph morphisms, symmetric deltas. As a rule, deep arrows are associatively composable and deep graphs are indeed categories, e.g., ***Sets*** (of sets and functions), ***Rels*** (of sets and relations), ***Posets*** (of posets and monotone functions), ***Graphs*** (of graphs and graph mappings), $\boldsymbol{Moddel}_{\mathsf{sym}}(M)$ (of M-models and symmetric deltas between them).

The description above is rough and overly simplistic. Making it more precise and intelligent needs a careful setting for logical and set-theoretical foundations, and goes far beyond our goals in the paper. Note, however, that we were talking about possible *interpretations* of elements constituting a category but the very definition of a category says nothing about "depth" of its objects and arrows.[9] Hence, any result proven for a general category (possessing some property P) is applicable to any flat or deep category (possessing P). For example, when we deal with category ***Modupd*** of models and updates, our results are applicable to any formalization of model and update as soon as we have a category.

As a rule, deep categories are infinite and cannot be drawn on paper (think of all sets or all M-models). However, we can draw a graph representing a small fragment of an infinite category, and further use and manipulate this representation in our reasoning about its deep referent. For example, nodes and arrows of a graph drawn on paper could refer to models and deltas, and operations over them correspond to synchronization procedures. Precise specification of these syntax-semantics relationships may be non-trivial. In the paper we deal with the following particular case of the issue: arity shapes of diagram operations are flat graphs whereas their carriers are deep. The next section provides an accurate formalization of this situation.

Diagrams. When different nodes or different arrows of a graph drawn on paper bear the same name, e.g., in Fig. 30(a1,a2), these names are *labels* "hung" on elements of the underlying graph rather than their unique names (the latter are unique identifiers of graph elements and cannot be repeated). Hence, in the

[9] It can be formalized in terms of so called *constructs* and *concrete* categories explained in book [70] (with care and elegance).

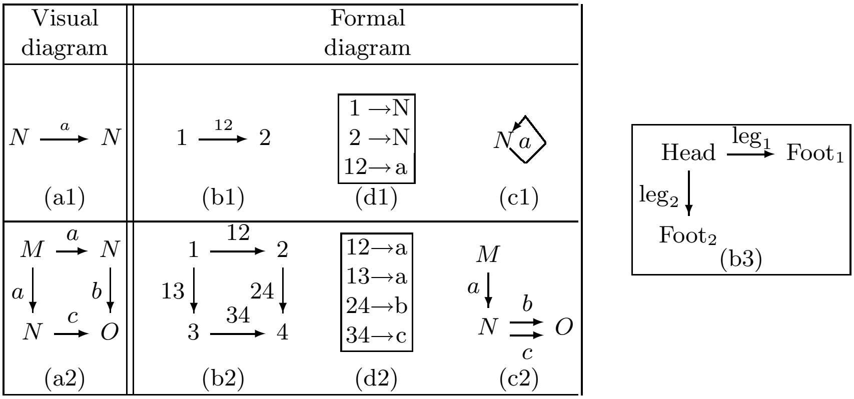


Fig. 30. Diagrams visually (a) and formally (b,d,c)

graphical image Fig. 30(a1), we have two unnamed different nodes (understood "flatly" as tokens) with the same label N. This label may be just another "flat" token, or the name (identifier) of a semantic object, e.g., a model; the formalization below does not take this into account (but in situations we deal with in the paper, labels are interpreted "deeply" as semantic objects).

It is convenient to collect all labels into a graph, and treat labeling as a graph mapping $D_1\colon (b1) \to (c1)$ with (b1) and (c1) being graphs specified in Fig. 30(b1,c1) and mapping D_1 defined by table (d1), i.e., $D_1(1) = D_1(2) = N$, $D_1(12) = a$. Thus, image (a1) that we call a diagram consists of three components: graph (b1) called the *shape* of the diagram, graph (c1) called the *carrier*, and a graph mapping (d1) — the *labeling.* Since the shape and the carrier are actually referred to by the mapping, the latter alone can be called a *diagram* (it is a standard categorical terminology). Indeed, the graphical image — visual diagram shown in (a1) — is nothing but a compact presentation of mapping D_1 defined up to isomorphism of the shape.

For another example, visual diagram in Fig. 30(a2) encodes the formal diagram of shape (b2) in the carrier graph (c2) with labeling $D_2\colon (b2) \to (c2)$ given by table (d2) (it is a graph morphism indeed).

What was earlier called a span in graph G, is actually a diagram $D\colon (b3) \to G$ with graph (b3) in Fig. 30 being the arity shape (the head of span D is node $D(\text{Head}) \in G$ etc.) Any span can be inverted: the *inverse* of D is another span $D^\dagger\colon (b3) \to G$ defined as follows: $D^\dagger(\text{leg}_1) = D(\text{leg}_2)$ and $D^\dagger(\text{leg}_2) = D(\text{leg}_1)$. Below we will call spans arity shapes (i.e. graphs isomorphic to (b3)) also *spans.*

Diagram operation over sorted graphs. Syntactically, a diagram operation is defined by its symbol (name), say, op, and a span of two-sorted graphs: $A_{\mathsf{op}} = (\mathbb{I}\text{n}_{\mathsf{op}} \xleftarrow{p} \mathbb{IO}_{\mathsf{op}} \xrightarrow{q} \mathbb{O}\text{ut}_{\mathsf{op}})$ whose legs are injections. The left foot specifies the *input* arity of the operation, the right one is the *output*, and the head is their intersection. For example, the operation of forward propagation considered above is specified by Fig. 31(a). The input arity is a span, the output arity is a cospan, and the head consists of two nodes.

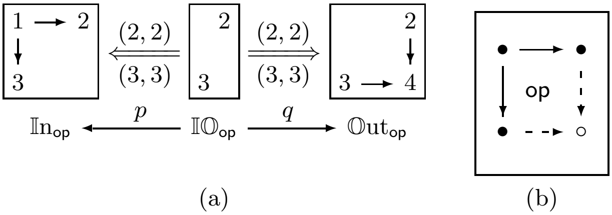

1
2
3
(2,2)
(3,3)
2
3
(2,2)
(3,3)
2
3
4
$\mathbb{I}\mathrm{n}_{\mathsf{op}}$
p
$\mathbb{IO}_{\mathsf{op}}$
q
$\mathbb{O}\mathrm{ut}_{\mathsf{op}}$
op
(a)
(b)

Fig. 31. Mechanism of diagram operations

We may merge both arities together (via pushout) and represent the arity span as a two-sorted graph $\mathbb{I}\mathrm{n}\mathbb{O}\mathrm{ut}$ with a designated subgraph $\mathbb{I}\mathrm{n}$ of *basic* elements. For the forward propagation example, this construction is specified in Fig. 31(b): the basic subgraph is shown with black nodes and solid arrows, elements beyond the basic subgraph are white and dashed. We can restore graph $\mathbb{O}\mathrm{ut}$ and the original arity span by subtracting graph $\mathbb{I}\mathrm{n}$ from $\mathbb{I}\mathrm{n}\mathbb{O}\mathrm{ut}$ so that both formulations are equivalent. Previously we used the latter formulation because it is intuitive and compact.

Semantic interpretation of an operation is given by a pair $\sigma = (\mathbb{G}^\sigma, \mathsf{op}^\sigma)$ with $\mathbb{G}^\sigma$ a two-sorted graph being the *carrier* of the operation, and

$$\mathsf{op}^\sigma \colon (\mathbb{I}\mathrm{n}_{\mathsf{op}} \to \mathbb{G}^\sigma) \to (\mathbb{O}\mathrm{ut}_{\mathsf{op}} \to \mathbb{G}^\sigma),$$

the operation as such, being a total function between the functional spaces in round brackets. That is, any instantiation $i \colon \mathbb{I}\mathrm{n}_{\mathsf{op}} \to \mathbb{G}^\delta$ of op's input in the carrier generates a unique instantiation $o \colon \mathbb{O}\mathrm{ut}_{\mathsf{op}} \to \mathbb{G}^\delta$ of op's output, and we set $\mathsf{op}^\sigma(i) = o$. Moreover, both instantiations are required to be equal on their common part $\mathbb{IO}_{\mathsf{op}}$, that is, $p;i = q;o$. In this way, the notion of diagram operation (its syntax and semantics) can be defined for any category (of "graphs").

The same idea is applicable to two-sorted graphs: both the shape and the carrier are two-sorted graphs and labeling must respect sorting. If we treat diagram Fig. 31(a2) as a two-sorted diagram, it would be incorrect because horizontal arrow 12 from the shape is mapped to vertical arrow a in the carrier.

Span composition. Categories are graphs, and hence the notion of a diagram, particularly, a span, applies to them as well. However, spans in categories are much more interesting than in graphs because we can sequentially compose them. Fig. 32 presents two consecutive spans between sets A, B, C. We may think of elements in the heads as bidirectional links and write $a \leftarrow r \rightarrow b$ for $r \in R_1$ if $p_1(r) = a$ and $q_1(r) = b$; and similarly for elements in R_2. If two such links $a \leftarrow r_1 \rightarrow b \in R_1$ and $(b \leftarrow r_2 \rightarrow c) \in R_2$ have a common end $b \in B$, we may compose them and form a new link $a \leftarrow r \rightarrow c$ denoted by $r_1;r_2$. By collecting together all such composed links, we form a new set R, which is equipped with two projections $(A \xleftarrow{p} R \xrightarrow{q} C)$. In addition, by the condition of compositionality, set R is equipped with another pair of projections $(R_1 \xleftarrow{p_2'} R \xrightarrow{q_1'} R_2)$ as shown in the figure, and it is easy to see that the upper square diagram is a pullback.

Note also that projections p and q are compositions $p = p_2'; p_1$ and $q = q_1'; q_2$. Now we may define the notion of span composition for any category having PBs, and achieve a remarkable generality.

There are however some hidden obstacles in this seemingly simple definition. Since pullbacks are defined up to iso(morphism), composition of spans is also defined up to iso. We may choose some canonical representatives in each of the iso classes, but then associativity of composition cannot be guaranteed. In fact, associativity would hold up to a canonic isomorphism too. It makes the universe of objects with arrows being spans a so called *bicategory* rather than a category, and essentially complicates the technical side.

$$\begin{tikzcd}
& & R \arrow[dl, dashed, "p_2'"'] \arrow[dr, dashed, "q_1'"] \arrow[dd, phantom, "{[\mathrm{PB}]}"] & & \\
& R_1 \arrow[dl, "p_1"'] \arrow[dr, "q_1"] & & R_2 \arrow[dl, "p_2"'] \arrow[dr, "q_2"] & \\
A & & B & & C
\end{tikzcd}$$

Fig. 32. Span composition

To avoid this, it is reasonable to consider spans up to isomorphism of their heads: it does not matter what are the OIDs of the head's elements. It is straightforward to check that composition of spans defined up to isomorphism of their heads is associative (details can be found in [71]).

Spans we deal with in the paper are special: their legs are injective mappings. It is known that if an input arrow in a PB-square is injective, the parallel output arrow is injective too ("monics are stable under PBs"). Hence, legs p_2', q_1' are injections, which implies that legs p, q are also injective as compositions of injections.

C Model Translation via Tiles

This section shows that model translation (MT) can be treated as a view computation, whose view definition is given by a corresponding metamodel mapping. Moreover, this construction can be modeled by tile operations, and gives rise to a well-known categorical construct called a *fibration.*

C.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 [49,3,32]. 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. 33. Oval nodes refer to value types. The domain of the attribute 'qName' is a Cartesian product (note the label ⊗) 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. 33 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. 33, links constituting the mapping are shown by thin curly arrows. The mapping satisfies

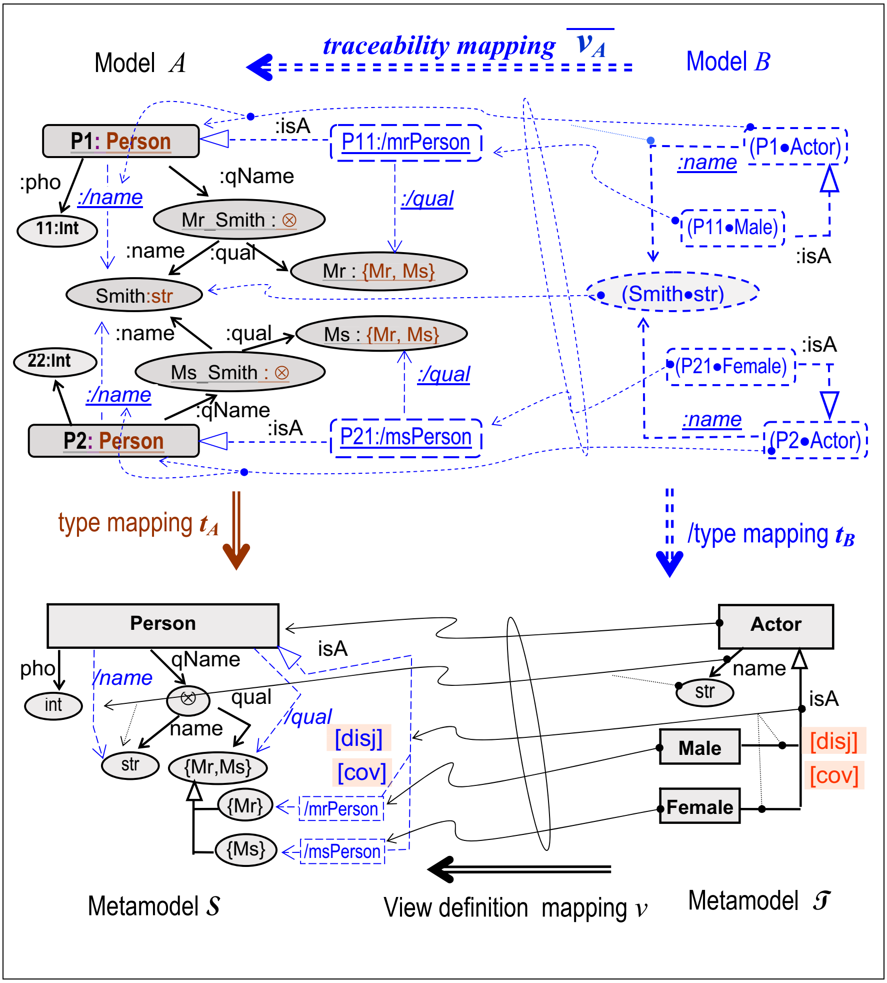


Fig. 33. Semantics of model translation via a metamodel mapping

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

(a)

$\Rightarrow$

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

(b)

$\Rightarrow$

$$\begin{tikzcd}
\boxed{\mathrm{A}{:}\boldsymbol{\mathcal{S}}} \arrow[d, "{:}\mu"'] \arrow[dr, phantom, "{:}\mathsf{vExe}\Nearrow"] & A{\restriction}_{\boldsymbol{v}}{:}\boldsymbol{\mathcal{T}} \arrow[l, Rightarrow, "\overline{\boldsymbol{v}_A}{:}\boldsymbol{v}"'] \arrow[d, dashed, "{:}\mu"] \\
\boxed{\boldsymbol{\mathcal{S}}} & \boxed{\boldsymbol{\mathcal{T}}} \arrow[l, Rightarrow, "\boldsymbol{v}"']
\end{tikzcd}$$

(c)

Fig. 34. Model translation via tile operations (the upper arrow in diagram (c) is derived and must be dashed but the Diagram software does not draw triple arrows)

two important requirements: (a) the structure of the metamodels (incidence of 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 $\boldsymbol{\mathcal{S}}$).

Now we will show that data specified above are sufficient to automatically translate any $\boldsymbol{\mathcal{S}}$-model into a $\boldsymbol{\mathcal{T}}$-model via two tile operations.

C.2 MT via Tile Algebra

1) Query execution. Query specifications used in augmenting $\boldsymbol{\mathcal{S}}$ with derived elements can be executed for $\boldsymbol{\mathcal{S}}$-models. For example, each pair of some model's arrows typed with :qName and :name produces a composed arrow typed with :/name, 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.[10] The result is that the initial typing mapping $t_A\colon A \to \boldsymbol{\mathcal{S}}$ is extended to typing mapping $t_{Q[A]}\colon Q[A] \to Q[\boldsymbol{\mathcal{S}}]$, in which $Q[A]$ and $Q[\boldsymbol{\mathcal{S}}]$ denote augmentations of the model and the metamodel with derived elements.

This extended typing mapping is again structure preserving. Moreover, it is a *conservative extension* of mapping t_A in the sense that types of elements in A are not changed by $t_{Q[A]}$. Formally, the inverse image of submodel $\boldsymbol{\mathcal{S}} \subset Q[\boldsymbol{\mathcal{S}}]$ 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. 34(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 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

[10] 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.

$Q[\boldsymbol{S}] \xleftarrow{\boldsymbol{v}} \boldsymbol{T}$, provide enough information for translating model $Q[A]$ into $\boldsymbol{T}$-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 $X = t_{Q[A]}(e) in Q[\boldsymbol{S}]$ and $X = \boldsymbol{v}(Y)$ for some type $Y \in \boldsymbol{T}$, we set the new type of e to be Y. For example, since $Q[A]$-element $P11$ in Fig. 33 has type mrPerson, which (according to the view mapping $\boldsymbol{v}$) corresponds to type Male in $\boldsymbol{T}$, this elements must be translated into an instance of type Male; we denote it by $(P11 \bullet Male)$. If no such $\boldsymbol{T}$-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 $\boldsymbol{S}$ 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 $\boldsymbol{T}$-types, and hence elements of the translated model B are, in fact, pairs $(e, Y) \in Q[A] \times \boldsymbol{T}$ such that $t_{Q[A]}(e) = \boldsymbol{v}(Y)$. In Fig. 33, 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

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

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. 33. Second components provide typing mapping $t_B \colon B \to \boldsymbol{T}$. The entire retyping procedure thus appears as a diagram operation specified by the right square in Fig. 34(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 discussed in Sect. A.1.

Remark 7. If view $\boldsymbol{v}$ maps two different $\boldsymbol{T}$-types $Y_1 \neq Y_2$ to the same $\boldsymbol{S}$-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 $\boldsymbol{T}$ 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 $\boldsymbol{T}$ 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 [72]. They show that the pullback operation is surprisingly "smart" and provides an adequate and predictive model of retyping.[11]

[11] 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 $(\mathsf{i}^Q_A, t_{Q[A]})$, 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. 33, 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 [16]).

Arrow encapsulation. Query execution followed by retyping gives us the operation of view execution shown in Fig. 34(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. Diagram Fig. 34(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. 34(b). We will return to this construction later in Sect. D.3.

C.3 Properties of the View Execution Operation

The view execution operation has three remarkable properties.

1) Unitality. If a view definition is given by the identity mapping, view execution is identity as well, as shown by diagram Fig. 35(b1).

2) Compositionality. Suppose we have a pair of composable metamodel mappings $\boldsymbol{v}1 {:} \boldsymbol{\mathcal{T}} \Rightarrow \boldsymbol{\mathcal{S}}$ and $\boldsymbol{v}2 {:} \boldsymbol{\mathcal{U}} \rightarrow \boldsymbol{\mathcal{T}}$, which defines $\boldsymbol{\mathcal{U}}$ as a view of a view of $\boldsymbol{\mathcal{S}}$. Clearly, execution of a composed view is composed from the execution of components so that for any $\boldsymbol{\mathcal{S}}$-model A we should have

$$\overline{\boldsymbol{v}1;\boldsymbol{v}2_A} = \overline{\boldsymbol{v}2_B}\,;\,\overline{\boldsymbol{v}1_A} \text{ with } B \text{ standing for } A{\restriction_{\boldsymbol{v}1}}$$

as shown in Fig. 35(b2). Formal proof of this fact needs an accurate definition of query specifications (see [52] for details), and then it will be a standard exercise in categorical algebra (with so called Kleisli triples). Details will appear elsewhere.

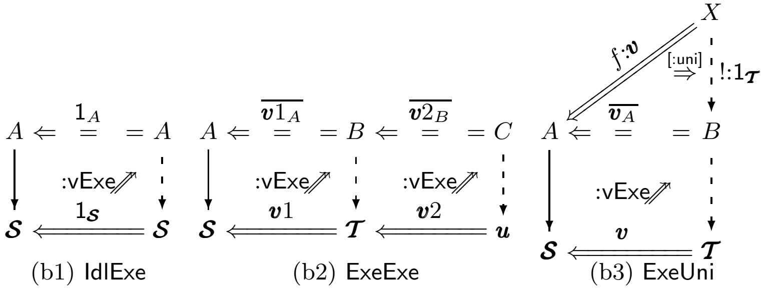


Fig. 35. Laws of the view execution mechanism

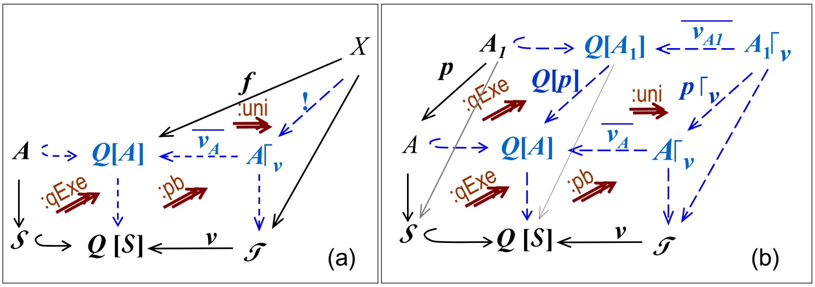


Fig. 36. Universal property of the view mechanism

3) 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. 36(a). The mapping must be compatible with typing so that the outer right square is required to be commutative. Then owing to the universal properties of pullbacks, there is a uniquely defined mapping $A\upharpoonright_{\boldsymbol{v}} \xleftarrow{!} X$ such that the triangles commute (note that mapping ! is a homogeneous model mapping over identity $1_{\mathcal{T}} \colon \mathcal{T} \to \mathcal{T}$).

By encapsulating queries, i.e., hiding them inside double-arrows (see transition from diagram (a) to (b) in Fig. 34), we can formulate the property as shown in Fig. 35(b3), where arrows $f{:}\boldsymbol{v}$ and $!{:}1_{\mathcal{T}}$ actually denote square diagrams whose vertical arrows are typing mappings and bottom arrows are pointed after semicolon.

View mechanism and updates. Universality of view execution has a remarkable consequence if queries are *monotonic*, i.e., preserve inclusion of datasets. Such queries have been studied in the database literature (e.g., [29]), and it is known that queries without negation are monotonic.

In our terms, a query Q is *monotonic* if any injective model mapping $A \xleftarrow{p} A_1$ between two $\boldsymbol{\mathcal{S}}$-models gives rise to an injective mapping $Q[A] \xleftarrow{Q[p]} Q[A_1]$ between models augmented with derived elements. This is illustrated by the left-upper square in Fig. 36(b). Applying retyping to models $A\upharpoonright_{\boldsymbol{v}}$ and $A_1\upharpoonright_{\boldsymbol{v}}$ provides the rest of the diagram apart from arrow $p{\upharpoonright}_{\boldsymbol{v}}$. To obtain the latter, we apply the universal property of $A{\upharpoonright}_{\boldsymbol{v}}$ (specified in diagram (a)) to mapping $\overline{\boldsymbol{v}_{A_1}};Q[p]$ (in the role of mapping f in diagram (a)), and get mapping $p{\upharpoonright}_{\boldsymbol{v}}$. If the view definition mapping is injective, then traceability mappings are injective too (PBs preserve monics), and hence $p{\upharpoonright}_{\boldsymbol{v}}$ is also injective. Thus, execution of views based on monotonic queries translates mappings as well. Moreover, if model updates are spans with injective legs, then view execution translates updates too: just add the other leg $q\colon A_1 \to A'$ and apply the same construction.

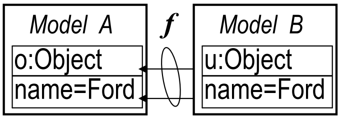
D Heterogeneous Model Mapping and Matching

Suppose that models A and B to be synchronized are instances of different metamodels, $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ respectively; we write $A{:}\boldsymbol{\mathcal{A}}$ and $B{:}\boldsymbol{\mathcal{B}}$. The metamodels may be essentially different, e.g., a class diagram and an activity diagram, which makes matching their instances structurally difficult. It even makes sense to reformulate the question ontologically: what is a match of non-similar models?

In Sect. 3.3 we modeled homogeneous matches by spans of homogeneous model mappings. We will apply the same idea to the heterogeneous case; hence, we first need to define heterogeneous model mappings.

D.1 Simple Heterogeneous Mappings

Model mappings are sets of links between model elements, and by *simple* mappings we mean those *not* involving derived elements. The first requirement for links to constitute a correct mapping is their compatibility with model structure: a class may be linked to a class, an attribute to an attribute etc. However, not all structurally correct mappings make sense.



Consider a mapping between two simple models in the inset figure. The mapping is structurally correct but it is not enough in the world of modeling, in which model elements have meaning encoded in metamodels. For example, Fig. 37(a) introduces possible metamodels for models A, B, and we at once see that sending an Employee to a Car is not meaningful. It could make sense if the concepts (classes) Employee and Car were “the same”, in which case it must be explicitly specified by a corresponding mapping between the metamodels. Without such a mapping, the metamodels are not related and it may be incorrect to map an Employee to a Car. Thus, diagram Fig. 37(a) presents an incorrect model mapping.

An example of a correct model mapping is shown in diagram (b). We first build a metamodel mapping and map concept Employee to Person. It is then legitimate to map instances of Employee to instances of Person. Thus, a correct model mapping is a pair of mappings $(f, \mathbf{f})$ commuting with typing : $f;t_A = t_B;\mathbf{f}$.

Arrow encapsulation. An abstract schema of the example is shown in Fig. 37(c). As shown in Sect. 3.1, models are chains of graph mappings realizing

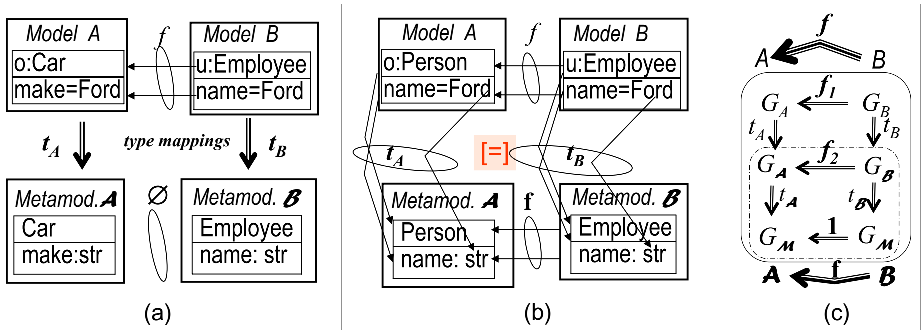


Fig. 37. Model mappings: incorrect (a), correct (b) and abstractly (c)

typing. Mapping $A \xleftarrow{f} B$ is a triple $(f_1, f_2, 1_{G_{\boldsymbol{\mathcal{M}}}})$ commuting with typing mappings, i.e., f can be considered as a two-layer commutative diagram (framed by the bigger rounded rectangle). The lower layer (the smaller frame) is the metamodel mappings $\boldsymbol{\mathcal{B}} \xleftarrow{\mathbf{f}} \boldsymbol{\mathcal{A}}$. If metamodels $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ were instances of different meta-metamodels, $\boldsymbol{\mathcal{M}}$, $\boldsymbol{\mathcal{N}}$, we could still build a reasonable mapping f by introducing the third component $\boldsymbol{\mathcal{M}} \xleftarrow{f_3} \boldsymbol{\mathcal{N}}$ commuting with typing mappings of $\boldsymbol{\mathcal{M}}$ and $\boldsymbol{\mathcal{N}}$ to their common meta-metametamodel.

Irrespectively of the number of layers, a mapping between models $A \xleftarrow{f} B$ contains a corresponding mapping $\boldsymbol{\mathcal{B}} \xleftarrow{\mathbf{f}} \boldsymbol{\mathcal{A}}$ between metamodels (which contains a mapping between metametamodels). Hence, models and model mappings can be projected to metamodels and their mappings by erasing the upper layer. This projection is evidently a graph morphism μ: ***Modmap*** → ***MModmap*** from the graph of models and their simple mappings to the graph of metamodels and their simple mappings.

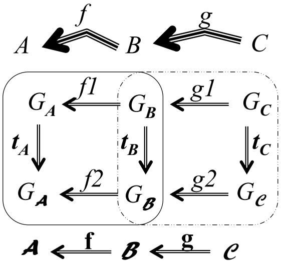


Fig. 38. Model mapping composition

Composition. Model mappings can be composed componentwise as shown in Fig. 38. The outer rectangle diagram is commutative as soon as the two inner squares are such. Hence, composition of two legal model mappings is again a legal model mapping. Associativity is evident, and the identity mapping consists of two identities $1_{G_A}: G_A \Rightarrow G_A$ and $1_{\boldsymbol{\mathcal{A}}}: G_{\boldsymbol{\mathcal{A}}} \Rightarrow G_{\boldsymbol{\mathcal{A}}}$. Hence, graphs of (meta)models and their mappings introduced above are categories. Moreover, projection μ: ***Modmap*** → ***MModmap*** is evidently compatible with composition and identities and so is a functor.

D.2 Matching Heterogeneous Models

Consider a simple example shown in Fig. 39, where matching links between two models are set in a naive way, and compare it with naive match in Fig. 9(a) on p.111. The first peculiarity of match in Fig. 39 is that objects of different types are matched (an Employee and a Personnel). Moreover, attributes are matched *approximately* meaning that their values somehow correspond but cannot be neither equal nor inequal because their relationship is more complex. Though intuitive, such matches do not conform to a type discipline, and their formal meaning is unclear.

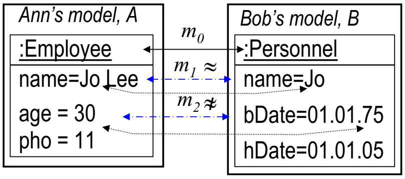
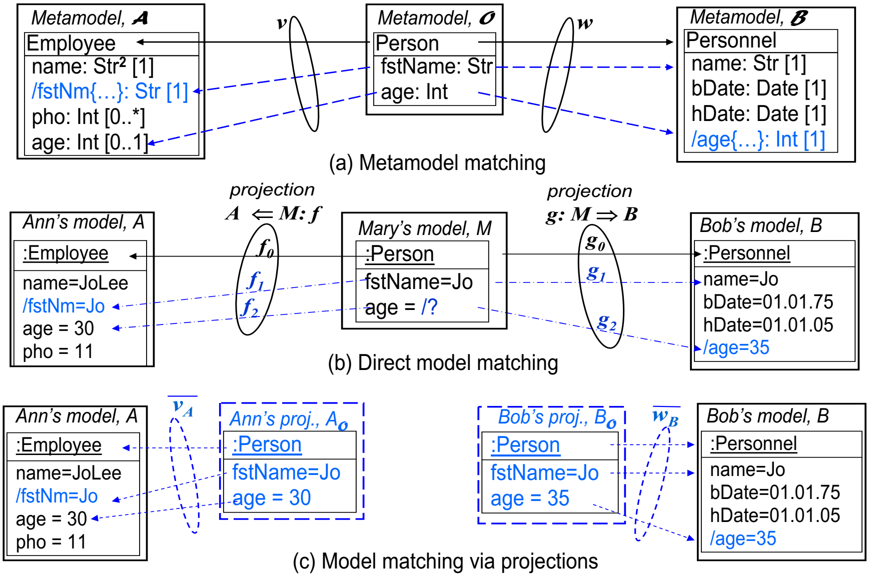


Fig. 39. Heterogeneous matching

Heterogeneous model mapping were defined above by including into them metamodel mappings. We may try to apply the same idea for heterogeneous matching: first match the metamodels, and then proceed with models. That is, we begin with making metamodels explicit and building a correspondence span between them as shown in Fig. 40(a).



(a) Metamodel matching

(b) Direct model matching

(c) Model matching via projections

Fig. 40. Matching a heterogeneous pair of models

The head of the span, metamodel $\mathcal{O}$, specifies the concepts common for both metamodels (we will say a *metamodel overlap*), and the legs are projection mappings. A basic concept in one metamodel, e.g., attribute 'age' in metamodel $\mathcal{A}$, may be a derived concept in the other metamodel: there is no attribute 'age' in metamodel $\mathcal{B}$ but it can be derived from attribute 'bDate' with a corresponding query. Similarly, we may specify a query to the metamodel $\mathcal{A}$, which defines a new attribute 'fstNm' (firstName). (Ellipsis in figurative brackets near derived attributes in Fig. 40(a) refer to the corresponding query specifications.) Thus, the legs of a correspondence span may map elements in the head to derived elements in the feet.

Now we can reify the match in Fig. 39 by the span in Fig. 40(b). The feet are models A, B augmented with derived elements; the latter are computed by executing queries specified in the metamodels (recall that a derived element in a metamodel is a query definition). The legs are heterogeneous model mappings whose metamodel components are specified in diagram (a) (typing mappings are not shown). These mappings are similar to simple heterogeneous mappings considered in Sect. D.1 but may map to derived elements; we call them *complex.*

Metamodel mappings are view definitions that can be executed for models (Sect. C). By executing view $\boldsymbol{v}$ for model A, and view $\boldsymbol{w}$ for model B, we project the models to the space of $\boldsymbol{\mathcal{O}}$-instances as shown in Fig. 40(c): the view models are denoted by $A_{\boldsymbol{\mathcal{O}}} \overset{\text{def}}{=} A{\upharpoonleft}_{\boldsymbol{v}}$ and $B_{\boldsymbol{\mathcal{O}}} \overset{\text{def}}{=} B{\upharpoonleft}_{\boldsymbol{w}}$ (and their frames are dashed to remind us that these models are derived rather than set independently). We also call the views *projections* to the overlap space. Note that along with view models, view execution computes also traceability mappings $\overline{\boldsymbol{v}_A}$ and $\overline{\boldsymbol{w}_B}$.

There are evident mappings from the head M to projections $A_{\boldsymbol{\mathcal{O}}}$, $B_{\boldsymbol{\mathcal{O}}}$ (not shown in the figure to avoid clutter). The existence of these mappings can be formally proved by the universal property of pullbacks as described in Sect. C.3.

$$\begin{tikzcd}
\boldsymbol{\mathcal{A}} \arrow[rr, Leftarrow, "\boldsymbol{v}"] & & \boldsymbol{\mathcal{O}} \arrow[rr, Rightarrow, "\boldsymbol{w}"] & & \boldsymbol{\mathcal{B}} \\
A \arrow[u, "t_A"] \arrow[rr, Leftarrow, "f"] \arrow[d, equal] & & M \arrow[u, "t_B"'] \arrow[rr, Rightarrow, "g"] \arrow[dl, dashed, "p{:}1\boldsymbol{\mathcal{O}}"'] \arrow[dr, dashed, "q{:}1\boldsymbol{\mathcal{O}}"] & & B \arrow[u, "t_M"'] \arrow[d, equal] \\
A \arrow[r, Leftarrow, dashed, "\overline{\boldsymbol{v}_A}"] & A{\upharpoonleft}_{\boldsymbol{v}} & & B{\upharpoonleft}_{\boldsymbol{w}} \arrow[r, Rightarrow, dashed, "\overline{\boldsymbol{w}_B}"] & B
\end{tikzcd}$$

(a) Extensional match

$$\begin{tikzcd}
A \arrow[r, Leftarrow, "\overline{\boldsymbol{v}_A}"] & A{\upharpoonleft}_{\boldsymbol{v}} \arrow[r, "\mathbf{m}"] & B{\upharpoonleft}_{\boldsymbol{w}} \arrow[r, Rightarrow, "\overline{\boldsymbol{w}_B}"] & B
\end{tikzcd}$$

(b) General match

Fig. 41. From hetero- to homogeneous matches

An abstract schema of the construction is shown in Fig. 41(a): the top row shows the metamodel overlap, the bottom row is the result of its execution, and the middle row is the correspondence span. Double-bodies of arrows remind us that mappings are complex, i.e., may map to derived elements in their targets.

Two slanted arrows are derived by the universal property of view traceability mappings (produced by pullbacks). Note that triple (M, p, q) is a homogeneous correspondence span in the space of $\boldsymbol{\mathcal{O}}$-models. It gives us an extensional match between models $A{\upharpoonleft}_{\boldsymbol{v}}$ and $B{\upharpoonleft}_{\boldsymbol{w}}$. We may add to this span non-extensional information (as discussed in Sect. 4.1) and come to diagram Fig. 41(b), in which arrow $\mathbf{m}$ denotes a general match between homogeneous models. Note that mappings $\overline{\boldsymbol{v}_A}$ and $\overline{\boldsymbol{w}_B}$ are derived whereas match $\mathbf{m}$ is an independent input datum.

D.3 Complex Heterogeneous Model Mappings

Simple heterogeneous model mappings defined above give rise to a functor $\mu\colon \boldsymbol{Modmap} \to \boldsymbol{MModmap}$. The goal of this section is to outline, semi-formally, how this description can be extended for complex mappings involving derived elements.

Let **QL** be a query language, that is, a signature of diagram operations over graphs. It defines a graph $\boldsymbol{MModmap}^{\mathbf{QL}}$ of metamodels and their complex mappings described in Sect. C. Similarly, we have graph $\boldsymbol{Modmap}^{\mathbf{QL}}$ of models and their complex mappings like, e.g., pairs mappings $(f, \boldsymbol{v})$ and $(g, \boldsymbol{w})$ shown in Fig. 40(b). (Recall that we actually deal with commutative square diagrams: $f;t_A = t_m;\boldsymbol{v}$ and $g;t_B = t_M;\boldsymbol{w}$.)

By encapsulating typing mappings inside nodes, and metamodel mappings inside arrows, we may rewrite the upper half of diagram Fig. 41(a) as shown in Fig. 42.

$$\begin{tikzcd}
\boldsymbol{\mathcal{A}} & \boldsymbol{\mathcal{O}} \arrow[l, Rightarrow, "\boldsymbol{v}"'] \arrow[r, Rightarrow, "\boldsymbol{w}"] & \boldsymbol{\mathcal{B}} \\
A{:}\boldsymbol{\mathcal{A}} \arrow[u, "{:}\mu^{\mathbf{QL}}"] & M{:}\boldsymbol{\mathcal{O}} \arrow[l, Rightarrow, "f{:}\boldsymbol{v}"'] \arrow[r, Rightarrow, "g{:}\boldsymbol{w}"] \arrow[u, "{:}\mu^{\mathbf{QL}}"] & B{:}\boldsymbol{\mathcal{B}} \arrow[u, "{:}\mu^{\mathbf{QL}}"]
\end{tikzcd}$$

Fig. 42. Encapsulation of complex heterogeneous mappings

A warning about arrow notation is in order. Graph mappings in Fig. 37(c) are denoted by double arrows to distinguish them from links (single-line arrows), and diagrams of graph mappings are triple arrows. Complex mappings add one more dimension of encapsulation — derived elements, and hence mappings $\boldsymbol{v}$, $\boldsymbol{w}$ should be denoted by triple arrows while mappings-diagrams $f{:}\boldsymbol{v}$, $g{:}\boldsymbol{w}$ by quadruple arrows. To avoid this monstrous notation, we sacrifice consistency. It is partially restored by using bullet-end arrows for links: the latter may be thought of as arrows with "zero-line" bodies.

Thus, similarly to simple heterogeneous model mappings, complex ones contain complex metamodel mappings and hence there is a graph morphism

$$\mu^{\mathbf{QL}} : \boldsymbol{Modmap}^{\mathbf{QL}} \to \boldsymbol{MModmap}^{\mathbf{QL}}$$

(vertical links in Fig. 42 are its instances). We want to turn the two graphs above into categories (and $\mu^{\mathbf{QL}}$ into a functor), i.e., we need to define composition of complex mappings.

Composition of complex metamodel mappings is easy and amounts to term substitution. As mentioned above in Sect. C.2, with an accurate definition of a query langauge's syntax, compositionality of metamodel mappings is a routine exercise in categorical algebra (with the so called *Kleisli triples* [73]). It turns graph $\boldsymbol{MModmap}^{\mathbf{QL}}$ into a category (the *Kleisli category* of the *monad* defined by the query language).

Defining composition of complex model mappings is much harder because we need to compose query executions, i.e., application instances of operations rather than terms (definitions of operations). It can be done relatively easily for monotonic queries defined above on p.159 (details will appear elsewhere). Thus, if all queries are monotonic, graph $\boldsymbol{Modmap}^{\mathbf{QL}}$ can also be turned into a category, whose arrows are square diagrams similar to those shown in Fig. 38. We thus have a functor $\mu^{\mathbf{QL}} : \boldsymbol{Modmap}^{\mathbf{QL}} \to \boldsymbol{MModmap}^{\mathbf{QL}}$ that maps models and model mappings to their embedded metamodel parts.

The view mechanism is a "play-back" operation specified in Fig. 34(c) such that three laws in Fig. 35 are satisfied. Together these requirements mean that functor $\mu^{\mathbf{QL}}$ is a *(split) fibration* — a construct well-known in CT [74, Exercise 1.1.6]. The fibrational formulation of metamodeling (including the the view mechanism) allows us to use many results of the rich theory of fibrations [74]. In a sense, it is a culmination of the concrete MMt branch of the paper: a multitude of complex data is encapsulated and cast in a very compact algebraic formulation.

Note that we did not formally prove the fibrational statement above. It is an observation suggested by our examples and semi-formal constructions in Sect. C and D rather than a theorem. To turn it into a theorem, we need a formal definition of queries and query execution, and then a formal specification of our considerations above; it is a work in progress. Part of this work is presented in [52] for the case of *functorial semantics* — a model is a functor *from* the metamodel to some semantic category of sets and mappings between them, which is dual to the usual metamodeling via a typing mapping (see beginning of Sect. 3.2).