Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond

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

[1]Network for Engineering of Complex Software-Intensive Systems for Automotive Systems (NECSIS),
McMaster University, Canada

[2]Generative Software Development Lab,
University of Waterloo, Canada

diskinz@mcmaster.ca   tom@maibaum.org

There is a hidden intrigue in the title. CT is one of the most abstract mathematical disciplines, sometimes nicknamed ”abstract nonsense”. MDE is a recent trend in software development, industrially supported by standards, tools, and the status of a new ”silver bullet”. Surprisingly, categorical patterns turn out to be directly applicable to mathematical modeling of structures appearing in everyday MDE practice. Model merging, transformation, synchronization, and other important model management scenarios can be seen as executions of categorical specifications.

Moreover, the paper aims to elucidate a claim that relationships between CT and MDE are more complex and richer than is normally assumed for ”applied mathematics”. CT provides a toolbox of design patterns and structural principles of real practical value for MDE. We will present examples of how an elementary categorical arrangement of a model management scenario reveals deficiencies in the architecture of modern tools automating the scenario.

Keywords: Model-driven engineering, mathematical modeling, category theory

1 Introduction

There are several well established applications of category theory (CT) in theoretical computer science; typical examples are programming language semantics and concurrency. Modern software engineering (SE) seems to be an essentially different domain, not obviously suitable for theoretical foundations based on abstract algebra. Too much in this domain appears to be ad hoc and empirical, and the rapid progress of open source and collaborative software development, service-oriented programming, and cloud computing far outpaces their theoretical support. Model driven (software) engineering (MDE) conforms to this description as well: the diversity of modeling languages and techniques successfully resists all attempts to classify them in a precise mathematical way, and model transformations and operations — MDE's heart and soul — are an area of a diverse experimental activity based on surprisingly weak (if any) semantic foundations.

In this paper we claim that theoretical underpinning of modern SE could (and actually quite naturally) be based on CT. The chasm between SE and CT can be bridged, and MDE appears as a “golden cut”, in which an abstract view of SE realities and concrete interpretations of categorical abstractions merge together: SE $\rightarrow$ MDE $\leftarrow$ CT. The left leg of the cospan is extensively discussed in the MDE literature (see [47] and references therein); prerequisites and challenges for building the right leg are discussed in the present paper. Moreover, we aim to elucidate a claim that relationships between CT and MDE are more complex and richer than is normally assumed for ”applied mathematics”. CT provides a toolbox of design patterns and principles, whose added value goes beyond such typical applications of mathematics to SE as formal semantics for a language, or formal analysis and model checking.

U. Golas, T. Soboll (Eds.): Proceedings of ACCAT 2012
EPTCS 93, 2012, pp. 1–21, doi:10.4204/EPTCS.93.1

Two aspects of the CT-MDE "marriage" are discussed in the paper. The first one is a standard argument about the applicability of a particular mathematical theory to a particular engineering discipline. To wit, there is a mathematical framework called CT, there is an engineering domain called MDE, and we will try to justify the claim that they make a good match, in the sense that concepts developed in the former are applicable for mathematical modeling of constructs developed in the latter. What makes this standard argument exciting is that the mathematical framework in question is known to be notoriously abstract, while the engineering domain is very agile and seemingly not suitable for abstract treatment. Nevertheless, the argument lies within the boundaries of yet another instance of the evergreen story of applying mathematics to engineering problems. Below we will refer to this perspective on the issue as Aspect A.

The second perspective (Aspect B) is less standard and even more interesting. It is essentially based on specific properties of categorical mathematics and on the observation that software engineering is a special kind of engineering. To wit, CT is much more than a collection of mathematical notions and techniques: CT has changed the very way we build mathematical models and reason about them; it can be seen as a toolbox of structural design patterns and the guiding principles of their application. This view on CT is sometimes called *arrow thinking*. On the other hand, SE, in general, and MDE, in particular, essentially depend on proper structuring of the universe of discourse into subuniverses, which in their turn are further structured and so on, which finally results in tool architectures and code modularization. Our experience and attempts to understand complex structures used in MDE have convinced us that general ideas of arrow thinking, and general patterns and intuitions of what a healthy structure should be, turn out to be useful and beneficial for such practical concerns as tool architecture and software design.

The paper is structured as follows. In Section 2 we present two very general A-type arguments that CT provides a "right" mathematical framework for SE. The second argument also gives strong prerequisites for the B-side of our story. Section 3.1 gives a brief outline of MDE, and Section 3.2 reveals a truly categorical nature of the cornerstone notions of multimodeling and intermodeling (another A-argument). In Section 4 we present two examples of categorical arrangement of model management scenarios: model merge and bidirectional update propagation. This choice is motivated by our research interests and the possibility to demonstrate the B-side of our story. In Section 5 we discuss and exemplify three ways of applying CT for MDE: understanding, design patterns for specific problems, and general design guidance on the level of tool architecture.

2 Two very general perspectives on SE and Mathematics

2.1 The plane of Software × Mathematics

The upper half of Fig. 1 presents the evolution of software engineering in a schematic way, following Mary Shaw [48] and José Fiadeiro [25]. Programming-in-the-head refers to the period when a software product could be completely designed, at least in principle, "inside the head" of one (super intelligent) programmer, who worked like a researcher rather than as an engineer. The increasing complexities of problems addressed by software solutions (larger programs, more complex algorithms and data structures) engendered more industrially oriented/engineering views and methods (e.g., structured programming). Nevertheless, for Programming-in-the-small, the software module remained the primary goal and challenge of software development, with module interactions being simple and straightforward (e.g., procedure calls). In contrast, Programming-in-the-large marks a shift to the stage when module composition becomes the main issue, with the numbers of modules and the complexity of module interaction enormously increased. This tendency continued to grow and widened in scope as time went on, and

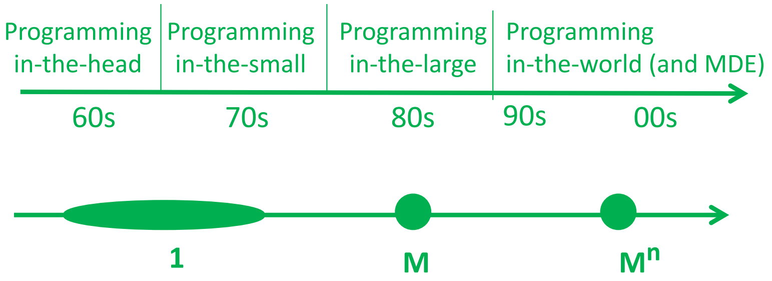

Figure 1: Evolution of software (M refers to Many/Multitude)

today manifests itself as Programming-in-the-world. The latter is characterized by a large, and growing, heterogeneity of modules to be composed and methods for their composition, and such essentially zdmodiffyin-the-largelarge modern technologies as service orientation, open source and collaborative software development, and cloud computing.

The lower part of Fig. 1 presents this picture in a very schematic way as a path from 1 to M to M^n with M referring to multiplicity in different forms, and degree n indicating the modern tendencies of growth in heterogeneity and complexity.

MDE could be seen as a reaction to this development, a way of taming the growth of n in a systematic way. Indeed, until recently, software engineers may feel that they could live without mathematical models: just build the software by whatever means available, check and debug it, and keep doing this throughout the software's life. (Note that the situation in classical (mechanical and electrical) engineering is essentially different: debugging, say, a bridge, would be a costly procedure, and classical engineers abandoned this approach long time ago.) But this gift of easily built systems afforded to SEs is rapidly degrading as the costs of this process and the liability from getting it wrong are both growing at an enormous rate. By slightly rephrasing Dijkstra, we may say that precise modeling and specification become a matter of death and life rather than luxury.

These considerations give us the vertical axis in Fig. 2, skipping the intermediate point. The horizontal axis represents the evolution of mathematics in a similar simplified way. Point 1 corresponds to the modern mathematics of mathematical structures in the sense of Bourbaki: what matters is operations and relations over mathematical objects rather than their internal structure. Skipped point M corresponds to basic category theory: the internal structure of the entire mathematical structure is encapsulated, and mathematical studies focus on operations and relations over structures considered as holistic entities. The multitude of higher degree, M^{∞}, refers to categorical facilities for reflection: enrichment, internalization, higher dimensions, which can be applied *ad infinitum*, hence, ∞-degree.

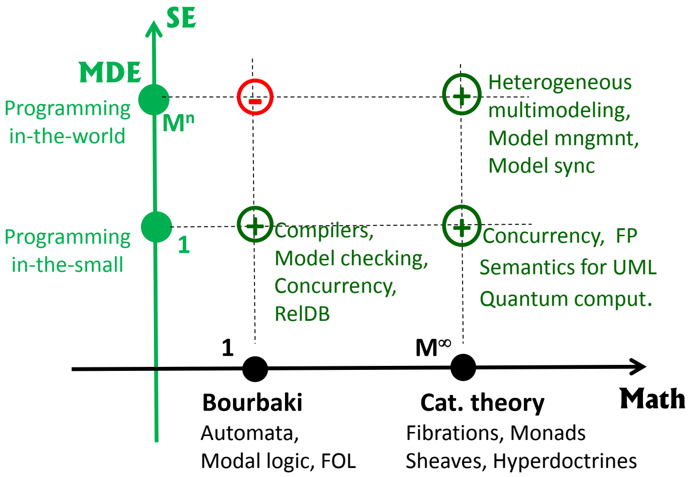

Figure 2: Software engineering and mathematics

This (over-simplified) schema gives us four points of Math×SE interaction. Interaction (1,1) turned out to be quite successful, as evidenced by such theory-based practical achievements as compilers, model checking, and relational DB theory. As for the point $(1, M^n)$, examining the literature shows that attempts at building theoretical foundations for MDE based on classical 1-mathematics were not successful. A major reason seems to be clear: 1-mathematics does not provide an adequate machinery for specifying and reasoning about inter-structural relationships and operations, which are at the very heart of modern software development. This point may also explain the general skepticism that a modern software engineer, and an academic teaching software engineering, feel about the practicality of using mathematics for modern software design: unfortunately, the only mathematics they know is the classical mathematics of Bourbaki and Tarski.

On the other hand, we view several recent applications of categorical methods to MDE problems [2, 5, 17, 37, 21, 45, 44, 22, 19, 43, 46] as promising theoretical attempts, with great potential for practical application. It provides a firm plus for the (M^∞, M^n)-point in the plane.

Moreover, as emphasized by Lawvere, the strength of CT based modeling goes beyond modeling multi-structural aspects of the mathematical universe, and a categorical view of a single mathematical structure can be quite beneficial too. This makes point $(M^\infty, 1)$ in the plane potentially interesting, and indeed, several successful applications at this point are listed in the figure.

2.2 Mathematical modeling of engineering artifacts: Round-tripping abstraction vs. waterfall based abstraction

Figure Fig. 3(a) shows a typical way of building mathematical models for mechanical and electrical engineering domains. Meta-mathematics (the discipline of modeling mathematical models) is not practically needed for engineering as such. The situation dramatically changes for software engineering. Indeed, category theory (CT) could be defined as a discipline for studying mathematical structures: how to specify, relate and manipulate them, and how to reason about them. In this definition, one can safely remove the adjective "mathematical" and consider CT as a mathematical theory of structures in a very broad sense. Then CT becomes directly applicable to SE as shown in Fig. 3(b). Moreover, CT has actually changed the way of building mathematical structures and thinking about them, and found extensive and deep applications in theoretical computer science. Hence, CT can be considered as a common theoretical framework for all modeling stages in the chain (and be placed at the center). In this way, CT provides a remarkable unification for modeling activities in SE.

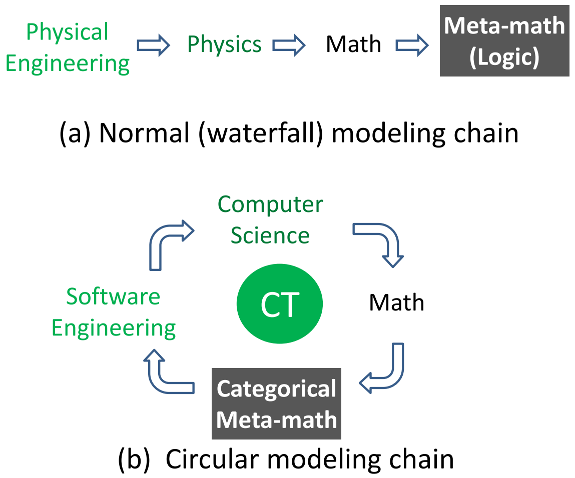

Figure 3: Modeling chains

The circular, non linear nature of the figure also illustrates an important point about the role of CT in SE. Because software artifacts are conceptual rather than physical entities, there is potential for feedback between SE and Mathematics in a way that is not possible in traditional scientific and engineering disciplines. Design patterns employed in SE can be, and have been, influenced by mathematical model of software and the way we develop them.

3 MDE and CT: an overall sketch

We will begin with a rough general schema of the MDE approach to building software (Section 3.1), and then will arrange this schema in categorical terms (Section 3.2).

3.1 MDE in a nutshell

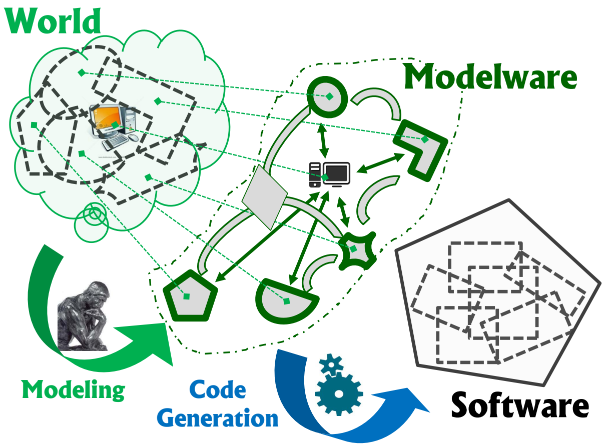

Figure 4: MDE, schematically

The upper-left corner of Fig. 4 shows a general goal of software design: building software that correctly interacts with different subsystems of the world (shown by figures of different shapes). For example, software embedded in a car interacts with its mechanical, electrical and electronic subsystems, with the driver and passengers, and with other cars on the road in future car designs. These components interact between themselves, which is schematically shown by overlaps of the respective shapes. The lower-right corner of Fig. 4 shows software modularized in parallel to the physical world it should interact with. The passage from the left to the right is highly non-trivial, and this is what makes SE larger and more challenging than mere programming. An effective means to facilitate the transition is to use models — a system of syntactical objects (as a rule, diagrammatic) that serve as abstractions of the "world entities" as shown in the figure (note the links from pieces of World to the respective parts of Modelware). These abstractions are gradually developed and refined until finally transformed into code. The modelware universe actually consists of a series of "modelwares" — systems of requirement, analysis, and design models, with each consecutive member in the list refining the previous one, and in its own turn encompassing several internal refinement chains. Modelware development consumes intelligence and time, but still easier and more natural for a human than writing code; the latter is generated automatically. The main idea of MDE is that human intelligence should be used for building models rather than code.

Of course, models have been used for building software long before the MDE vision appeared in the market. That time, however, after the first version of a software product had been released, its maintenance and further evolution had been conducted mainly through code, so that models had quickly become outdated, degraded and finally became useless. In contrast, MDE assumes that maintenance and evolution should also go through models. No doubts that some changes in the real world are much easier to incorporate immediately in the code rather than via models, but then MDE prescribes to update the models to keep them in sync with code. In fact, code becomes just a specific model, whose only essential distinction from other models in the modelware universe is its final position in the refinement chain. Thus, the Modelware boundary in Fig. 4 should be extended to encompass the Software region too.

3.2 Modelware categorically

Consider a modelware snapshot in Fig. 4. Notice that models as such are separated whereas their referents are overlapped, that is, interact between themselves. This interaction is a fundamental feature of the real world, and to make the model universe adequate to the world, intermodel correspondences/relations must be precisely specified. (For example, the figure shows three binary relations, and one ternary relation visualized as a ternary span with a diamond head.) With reasonable modeling techniques, intermodel relations should be compatible with model structures. The modelware universe then appears as a collection of structured objects and structure-compatible mappings between them, that is, as a categorical phenomenon. In more detail, a rough categorical arrangement could be as follows.

The base universe. Models are multi-sorted structures whose theories are called *metamodels*. The latter can be seen as generalized sketches [39, 20], that is, pairs $M = (G_M, C_M)$ with G_M a graph (or, more generally, an object of an apiori fixed presheaf topos $\mathbf{G}$), and C_M a set of *constraints* (i.e., diagram predicates) declared over G_M. An *instance* of metamodel M is a pair $A = (G_A, t_A)$ with G_A another graph (an object in $\mathbf{G}$) and $t_A\colon G_A \to G_M$ a mapping (arrow in $\mathbf{G}$) to be thought of as *typing*, which satisfy the constraints, $A \models C_M$ (see [20] for details). An *instance mapping* $A \to B$ is a graph mapping $f\colon G_A \to G_B$ commuting with typing: $f;t_B = t_A$. This defines a category $\boldsymbol{Mod}(M) \subset \mathbf{G}/G_M$ of M-instances.

To deal with the heterogeneous situation of models over different metamodels, we first introduce metamodel morphisms $m\colon M \to N$ as sketch morphisms, i.e., graph mappings $m\colon G_M \to G_N$ compatible with constraints. This gives us a category of metamodels $\boldsymbol{MMod}$. Now we can merge all categories $\boldsymbol{Mod}(M)$ into one category $\boldsymbol{Mod}$, whose objects are instances (= $\mathbf{G}$-arrows) $t_A\colon G_A \to G_{M(A)}$, $t_B\colon G_B \to G_{M(B)}$ *etc*, each having its metamodel, and morphisms $f\colon A \to B$ are pairs $f_{\text{data}}\colon G_A \to G_B$, $f_{\text{meta}}\colon M(A) \to M(B)$ such that $f_{\text{data}};t_B = t_A;f_{\text{meta}}$, i.e., commutative squares in $\mathbf{G}$. Thus, $\boldsymbol{Mod}$ is a subcategory of the arrow category $\mathbf{G}^{\cdot\to\cdot}$.

It can be shown that pulling back a legal instance $t_B\colon G_B \to G_N$ of metamodel N along a sketch morphism $m\colon M \to N$ results in a legal instance of M [20]. We thus have a fibration $\boldsymbol{p}\colon \boldsymbol{Mod} \to \boldsymbol{MMod}$, whose Cartesian lifting is given by pullbacks.

Intermodel relations and queries. A typical intermodeling situation is when an element of one model corresponds to an element that is not immediately present in another model, but can be derived from other elements of that model by a suitable operation (a query, in the database jargon) [19]. Query facilities can be modeled by a pair of monads $(\mathsf{Q}_{\text{def}}, \mathsf{Q})$ over categories $\boldsymbol{MMod}$ and $\boldsymbol{Mod}$, resp. The first monad describes the syntax (query definitions), and the second one provides the semantics (query execution).

A fundamental property of queries is that the original data are not affected: queries compute new data but do not change the original. Mathematical modeling of this property results in a number of equations, which can be summarized by saying that monad Q is $\boldsymbol{p}$-Cartesian, ı.e., the Cartesian and the monad structure work in sync. If can be shown [19] that a query language $(\mathsf{Q}, \mathsf{Q}_{\text{def}})$ gives rise to a fibration $\boldsymbol{p}_{\mathsf{Q}}\colon \boldsymbol{Mod}_{\mathsf{Q}} \to \boldsymbol{MMod}_{\mathsf{Q}_{\text{def}}}$ between the corresponding Kleisli categories. These Kleisli categories have immediate practical interpretations. Morphisms in $\boldsymbol{MMod}_{\mathsf{Q}_{\text{def}}}$ are nothing but view definitions: they map elements of the source metamodel to queries against the target one. Correspondingly, morphisms in $\boldsymbol{Mod}_{\mathsf{Q}}$ are view executions composed from query execution followed by retyping. The fact that projection $\boldsymbol{p}_{\mathsf{Q}}$ is fibration implies that the view execution mechanism is compositional: execution of a composed view equals the composition of executions.

Now a correspondence between models A, B over metamodels M, N can be specified by data shown in Fig. 5; these data consist of three components.. (1) span $(m\colon N \Leftarrow MN, n\colon MN \Rightarrow N)$ (whose legs are Kleisli mappings) specifies a common view MN between the two metamodels. (2) trapezoids (arrows in $\boldsymbol{Mod}_{\mathsf{Q}}$) are produced by $\boldsymbol{p}_{\mathsf{Q}}$-Cartesian “lifting”, i.e., by executing views m and n for models A and B resp., which results in models $A\restriction_m$ and $B\restriction_n$ (here and below we use the following notation: computed nodes are not framed, and computed arrows are dashed). (3) span $(p\colon A\restriction_m \leftarrow AB,\ q\colon AB \rightarrow B\restriction_n)$ specifies a correspondence between the views. Note that this span is an independent modelware component and cannot be derived from models A,B.

Spans like in Fig. 5 integrate a collection of models into a holistic system, which we will refer to as a *multimodel*. Examples, details, and a precise definition of a multimodel's consistency can be found in [21].

It is tempting to encapsulate spans in Fig. 5 as composable arrows and work with the corresponding (bi)categories of metamodels and models. Unfortunately, it would not work out because, in general, Kleisli categories are not closed under pullbacks, and it is not clear how to compose Kleisli spans. It is an important problem to overcome this obstacle and find a workable approach to Kleisli spans,

$$\begin{tikzcd}
\boxed{M} & & \boxed{MN} \arrow[ll, Rightarrow, "m"'] \arrow[rr, Rightarrow, "n"] & & \boxed{N} \\
\boxed{G_M} & & \boxed{G_{MN}} \arrow[ll, Rightarrow, "m"'] \arrow[rr, Rightarrow, "n"] & & \boxed{G_N} \\
\boxed{G_A} \arrow[u, "t_A"] & G_A\restriction_m \arrow[l, Rightarrow, dashed, "\overline{m_A}"] \arrow[ur, dashed, "t_A\restriction_m"] & \boxed{G_{AB}} \arrow[l, "p"] \arrow[r, "q"'] \arrow[u, "t_{AB}"] & G_B\restriction_n \arrow[r, Rightarrow, dashed, "\overline{n_B}"'] \arrow[ul, dashed, "t_B\restriction_n"'] & \boxed{G_B} \arrow[u, "t_B"]
\end{tikzcd}$$

Figure 5: Correspondences between heterogeneous models

Until the problem above is solved, our working universe is the Kleisli category of heterogeneous models fibred over the Kleisli category of metamodels. This universe is a carrier of different operations and predicates over models, and a stage on which different modeling scenarios are played. Classification and specification of these operations and predicates, and their understanding in conventional mathematical terms, is a major task of building mathematical foundations for MDE. Algebraic patterns appear here quite naturally, and then model management scenarios can be seen as algebraic terms composed from diagram-algebra operations over models and model mappings.[1] The next section provides examples of such algebraic arrangements.

4 Model management (MMt) and algebra: Two examples

We will consider two examples of algebraic modeling of MMt scenarios. A simple one — model merging, and a more complex and challenging — bidirectional update propagation (BX).

4.1 Model merge via colimit

Merging several interrelated models without data redundancy and loss is an important MDE scenario. Models are merged (virtually rather than physically) to check their consistency, or to extract an integrated information about the system. A general schema is shown in Fig. 6. Consider first the case of several homogeneous models $A,B,C...$ to be merged. The first step is to specify correspondences/relations between models via Kleisli spans $R1,R2,...$, or perhaps direct mappings like $r3$. The intuition of merging without data loss and redundancy (duplication of correspondent data) is precisely captured by the universal property of colimits, that is, it is reasonable to define merge as the colimit of a diagram of models and model mappings specifying intermodel correspondences.

[1] Note, however, that a proper categorical treatment of these operations in terms of universal constructions can be not straightforward.

If models are heterogeneous, their relations are specified as in Fig. 5. To merge, we first merge metamodels modulo metamodel spans. Then we can consider all models and heads of the correspondence spans as instances of the merged metamodel, and merge models by taking the colimit of the entire diagram in the category of instances of the merged metamodel.

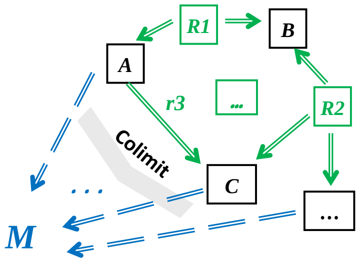

Figure 6: Model merge

An important feature of viewing model merge as described above is a clear separation of two stages of the merge process: (i) discovery and specifying intermodel correspondences (often called model matching), and (ii) merging models modulo these correspondences. The first stage is inherently heuristic and context dependent. It can be assisted by tools based on AI-technologies, but in general a user input is required for final adjustment of the match (and of course to define the heuristics used by the tool). The second stage is pure algebra (colimit) and can be performed automatically. The first step may heavily depend on the domain and the application, while the second one is domain and application independent. However, a majority of model merge tools combine the two stages into a holistic merge algorithm, which first somehow relates models based on a specification of conflicts between them, and then proceeds accordingly to merging. Such an approach complicates merge algorithms, and makes a taxonomy of conflicts their crucial component; typical examples are [49, 42].

The cause of this deficiency is that tool builders rely on a very simple notion of model matching, which amounts to linking *the-same-semantics* elements in the models to be matched. However, as discussed above in Section 3.2, for an element e in model A, the-same-semantics B-element e' can only be indirectly present in B, i.e., e' can be derived from other elements of B with a suitable operation (query) over B rather than being an immediate element of B. With complex (Kleisli) matching that allows one to link basic elements in one model with derived elements in another model, the algebraic nature of merge as such (via the colimit operation) can be restored. Indeed, it is shown in [9] that all conflicts considered in [42] can be managed via complex matching, that is, described via Kleisli spans with a suitable choice of queries, afterwards merge is computed via colimit.

4.2 Bidirectional update propagation (BX)

Keeping a system of models mutually consistent (model synchronization) is vital for model-driven engineering. In a typical scenario, given a pair of inter-related models, changes in either of them are to be propagated to the other to restore consistency. This setting is often referred to as bidirectional model transformation (BX) [6].

4.2.1 BX via tile algebra

A simple BX-scenario is presented in Fig. 7(a). Two models, A and B, are interrelated by some *correspondence specification* r (think of a span in a suitable category, or an object in a suitable comma category, see [21] for examples). We will often refer to them as *horizontal deltas* between models. In addition, there is a notion of delta *consistency* (extensionally, a class of *consistent* deltas), and if r is consistent, we call models A and B synchronized.

Now suppose that (the state of) model B has changed: the updated (state of the) model is B', and arrow b denotes the correspondence between B and B' (*a vertical delta*). The reader may think of a span, whose head consists of unchanged elements and the legs are injections so that B's elements beyond the range of the upper leg are deleted, and B''s elements beyond the range of the lower leg are inserted. Although update spans are denoted by bidirectional arrows, the upper node is always the source, and the lower is the target.

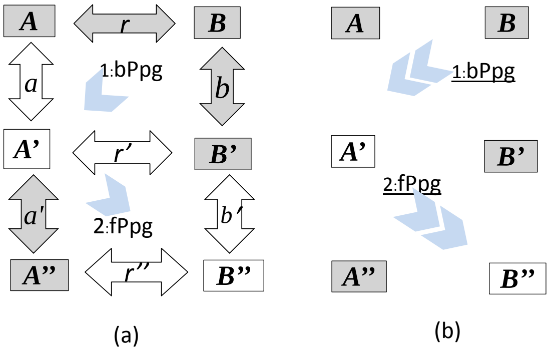

Figure 7: BX scenario specified in (a) delta-based and (b) state-based way

Suppose that we can re-align models A and B' and compute new horizontal delta $r * b$ (think of a span composition). If this new delta is not consistent, we need to update model A so that the updated model A' would be in sync with B'. More accurately, we are looking for an update $a\colon A \leftrightarrow A'$ such that the triple (A', r', B') is consistent. Of course, we want to find a minimal update a (with the biggest head) that does the job.

Unfortunately, in a majority of practically interesting situations, the minimality condition is not strong enough to provide uniqueness of a. To achieve uniqueness, some update propagation policy is to be chosen, and then we have an algebraic operation bPpg ('b' stands for 'backward'), which, from a given a pair of arrows (b, r) connected as shown in the figure, computes another pair (a, r') connected with (b, r) as shown in the figure. Thus, a propagation policy is algebraically modeled by a diagram operation of arity specified by the upper square in Fig. 7(a): shaded elements denote the input data, whereas blank ones are the output. Analogously, choosing a forward update propagation policy (from the A-side to the B-side) provides a forward operation fPpg as shown by the lower square.

The entire scenario is a composition of two operations: a part of the input for operation application 2:fPpg is provided by the output of 1:bPpg. In general, composition of diagram operations, i.e., operations acting upon configurations of arrows (diagrams), amounts to their *tiling*, as shown in the figure; then complex synchronization scenarios become *tiled* structures. Details, precise definitions and examples can be found in [15].

Different diagram operations involved in model synchronization are not independent and their interaction must satisfy certain conditions. These conditions capture the semantics of synchronization procedures, and their understanding is important for the user of synchronization tools: it helps to avoid surprises when automatic synchronization steps in. Fortunately, principal conditions (synchronization laws) can be formulated as universally valid equations between diagrammatic terms — a tile algebra counterpart of universal algebraic *identities*. In this way BX becomes based on an algebraic theory: a signature of diagram operations and a number of equational laws they must satisfy. The appendix presents one such theory — the notion of a *symmetric delta lens*, which is currently an area of active research from both a practical and a theoretical perspective.

4.2.2 BX: delta-based vs. state-based

As mentioned above, understanding the semantics of model synchronization procedures is important, both theoretically and practically. Synchronization tools are normally built on some underlying algebraic theory [28, 53, 40, 4, 1, 41, 30], and many such tools (the first five amongst those cited above) use algebraic theories based on state-based rather than delta-based operations. The state-based version of the propagation scenario in Fig. 7(a) is described in Fig. 7(b). The backward propagation operation takes models A, B, B', computes necessary relations between them (r and b on the adjacent diagram), and then computes an updated model A'. The two-chevron symbol reminds us that the operation actually consists of two stages: model alignment (computing r and b) and update propagation as such.

The state-based frameworks, although they may look simpler, actually hides several serious deficiencies. Model alignment is a difficult task that requires contextual information about models. It can be facilitated by intelligent AI-based tools, or even be automated, but the user should have an option to step in and administer corrections. In this sense, model alignment is similar to model matching preceding model merge.[2] Weaving alignment (delta discovery) into update (delta) propagation essentially complicates the semantics of the latter, and correspondingly complicates the algebraic theory. In addition, the user does not have an access to alignment results and cannot correct them.

Two other serious problems of the state-based frameworks and architectures are related to operation composition. The scenario described in Fig. 7(a) assumes that the model correspondence (delta) used for update propagation 2:fPpg is the delta computed by operation 1:bPpg; this is explicitly specified in the tile algebra specification of the scenario. In contrast, the state-based framework cannot capture this requirement. A similar problem appears when we sequentially compose a BX program synchronizing models A and B and another program synchronizing models B and C: composition amounts to horizontal composition of propagation operations as shown in Fig. 8, and again continuity, $b_1 = b_2$, cannot be specified in the state-based framework. A detailed discussion of delta- vs. state-based synchronization can be found in [22, 10].

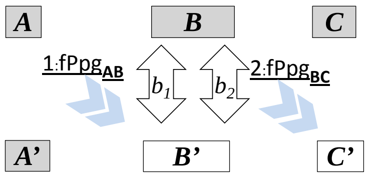

Figure 8: State-based BX: erroneous horizontal composition

4.2.3 Assembling model transformations

Suppose M,N are two metamodels, and we need to transform M-instances (models) into N-ones. Such a transformation makes sense if metamodels are somehow related, and we suppose that their relationship is specified by a span $(m\colon M \Leftarrow MN,\ n\colon MN \Rightarrow N)$ (Fig. 9), whose legs are Kleisli mappings of the respective query monad.

Now N-translation of an M-model A can be done in two steps. First, view m is executed (via its Cartesian lifting actually going down in the figure), and we obtain Kleilsi arrow $\overline{m_A}\colon A \Leftarrow R$ (with $R = A\!\upharpoonright_m$). Next we need to find an N-model B such that its view along n, $B\!\upharpoonright_n$, is equal to R. In other words, given a view, we are looking for a source providing this view. There are many such sources, and to achieve uniqueness, we need to choose some policy. Afterwards, we compute model B related to A by span $(\overline{m_A},\overline{n_B})$.

If model A is updated to A', it is reasonable to compute a corresponding update $b\colon B \leftrightarrow B'$ rather than recompute B' from scratch (recall that models can contain thousands elements). Computing b again consists of two steps shown in the figure. Operations Get^m and Put^n are similar to fPpg and bPpg considered above, but work in the asymmetric situation when mappings m and n are total (Kleisli) functions and hence view R contains nothing new wrt. M and N. Because of asymmetry, operations Get (‘get’ the view update) and Put (’put’ it back to

$$\begin{tikzcd}
\boxed{M} & \boxed{MN} \arrow[l, Leftarrow, "m"'] \arrow[r, Rightarrow, "n"] & \boxed{N} \\
\boxed{A} \arrow[d, leftrightarrow, "a"'] \arrow[dr, phantom, "\searrow;\mathsf{Get}^m"] & \boxed{R} \arrow[l, Leftarrow, "\overline{m_A}"'] \arrow[r, Rightarrow, "\overline{n_B}"] \arrow[d, dashed, leftrightarrow] \arrow[dr, phantom, "\searrow;\mathsf{Put}^n"] & \boxed{B} \arrow[d, dashed, leftrightarrow, "b"] \\
\boxed{A'} & R' \arrow[l, Leftarrow, dashed, "\overline{m_A}'"'] \arrow[r, Rightarrow, dashed, "\overline{n_B}'"] & B'
\end{tikzcd}$$

Figure 9: Model transformation via GetPut-decomposition

[2] A difference is that model matching usually refers to relating independently developed models, while models to be aligned are often connected by a given transformation.

the source) are different. Get^m is uniquely determined by the view definition m. Put^n needs, in addition to n, some update propagation policy. After the latter is chosen, we can realize transformation from M to N incrementally by composition $\mathsf{fPpg} = \mathsf{Get}^m;\mathsf{Put}^n$ — this is an imprecise linear notation for tiling (composition of diagram operations) specified in Fig. 9.

Note that the initial transformation from M to N sending, first, an M-instance A to its view $R = A\restriction_m$, and then finding an N-instance $B \in N$ such that $B\restriction_n = R$, can be also captured by Get and Put. For this, we need to postulate initial objects Ω_M and Ω_N in categories of M- and N-instances, so that for any A over M and B over N there are unique updates $0_A : \Omega_M \to A$ and $0_B : \Omega_N \to B$. Moreover, there is a unique span $(m_\Omega : \Omega_M \Leftarrow \Omega_{MN},\ n_\Omega : \Omega_{MN} \Rightarrow \Omega_N)$ relating these initial objects. Now, given a model A, model B can be computed as B' in Fig. 9 with the upper span being (m_Ω, n_Ω), and the update a being $0_A : \Omega_M \to A$.

The backward transformation is defined similarly by swapping the roles of m and n:

$$\mathsf{bPpg} = \mathsf{Get}^n;\mathsf{Put}^m.$$

The schema described above can be seen as a general pattern for defining model transformation declaratively with all benefits (and all pains) of having a precise specification before the implementation is approached (and must obey). Moreover, this schema can provide some semantic guarantees in the following way. Within the tile algebra framework, laws for operations Get and Put, and their interaction (invertibility), can be precisely specified [22] (see also the discussion in Section 5.1); algebras of this theory are called *delta lenses*. Then we can deduce the laws for the composed operations fPpg and bPpg from the delta lens laws. Also, operations Get^m, Put^m can themselves be composed from smaller blocks, if the view m is composed: $m = m_1;m_2;...;m_k$, via sequential lens composition. In this way, a complex model transformation is assembled from elementary transformation blocks, and its important semantic properties are guaranteed. More examples and details can be found in [15].

5 Applying CT to MDE: Examples and Discussion.

We will try to exemplify and discuss three ways in which CT can be applied in MDE. The first one — gaining a deeper understanding of an engineering problem — is standard, and appears as a particular instantiation of the general case of CT's employment in applied domains. The other two are specific to SE: structural patterns provided by categorical models of the software system to be built can directly influence the design. We will use models of BX as our main benchmark; other examples will be also used when appropriate.

5.1 Deeper understanding. As mentioned in Sect. 4.2, stating algebraic laws that BX procedures must obey is practically important as it provides semantic guaranties for synchronization procedures. Moreover, formulation of these laws should be semantically transparent and concise as the user of synchronization tools needs a clear understanding of propagation semantics. The original state-based theory of asymmetric BX [28] considered two groups of laws: invertibility (or round-tripping) laws, GetPut and PutGet, and history ignorance, PutPut. Two former laws say that two propagation operations, Get and Put, are mutually inverse. The PutPut law says that if a complex update is decomposed into consecutive pieces, it can be propagated incrementally, one piece after the other. A two-sorted algebra comprising two operations, Get and Put, satisfying the laws, is called a *well-behaved lens*.

Even an immediate arrow-based generalization of lenses to delta lenses (treated in elementary terms via tile algebra [15, 22]) revealed that the GetPut law is a simple law of identity propagation, IdPut, rather than of round-tripping. The benefits of renaming GetPut as IdPut are not exhausted by clarification of semantics: as soon as we understand that the original GetPut is about identity propagation, we at once ask what the *real round-tripping* law GetPut should be, and at once see that operation Put is not the inverse of Get. We only have the weaker 1.5-round-tripping GetPutGet law (or *weak* invertibility; see the Appendix, where the laws in question are named IdPpg and fbfPpg and bfbPpg). It is interesting (and remarkable) that papers [14, 31], in which symmetric lenses are studied in the state-based setting, mistakenly consider identity propagation laws as round-tripping laws, and correspondingly analyze a rather poor BX-structure without real round-tripping laws at all.

The tile algebra formulation of the PutPut law clarified its meaning as a composition preservation law [15, 22], but did not solve the enigmatic PutPut problem. The point is that PutPut does not hold in numerous practically interesting situations, but its entire removal from the list of BX laws is also not satisfactory, as it leaves propagation procedures without any constraints on their compositionality. The problem was solved, or at least essentially advanced, by a truly categorical analysis performed by Michael Johnson et al [35, 34]. They have shown that an asymmetric well-behaved lens is an algebra for some KZ monad, and PutPut is nothing but the basic associativity condition for this algebra. Hence, as Johnson and Rosebrugh write in [34], the status of the PutPut changes from being (a) "some law that may have arisen from some special applications and should be discarded immediately if it seems not to apply in a new application" to (b) a basic requirement of an otherwise adequate and general mathematical model. And indeed, Johnson and Rosebrugh have found a weaker — *monotonic* — version of PutPut (see Fig. 13 in the Appendix), which holds in a majority of practical applications, including those where the original (non-monotonic or mixed) PutPut fails. Hopefully, this categorical analysis can be generalized for the symmetric lens case, thus stating solid mathematical foundations for BX.

5.2 Design patterns for specific problems.

Recalling Figure 3, Figure 10 presents a rough illustration of how mathematical models can reshape our view of a domain or construct X. Building a well-structured mathematical model M of X, and then reinterpreting it back to X, can change our view of the latter as schematically shown in the figure with the reshaped construct X'. Note the discrepancy between the reshaped X' and model M: the upper-left block is missing from X'. If X is a piece of reality (think of mathematical modeling of physical phenomena), this discrepancy means, most probably, that the model is not adequate (or, perhaps, some piece of X is not observable). If X is a piece of software, the discrepancy may point to a deficiency of the design, which can be fixed by redesigning the software. Even better to base software design on a well-structured model from the very beginning. Then we say that model M provides a design pattern for X.

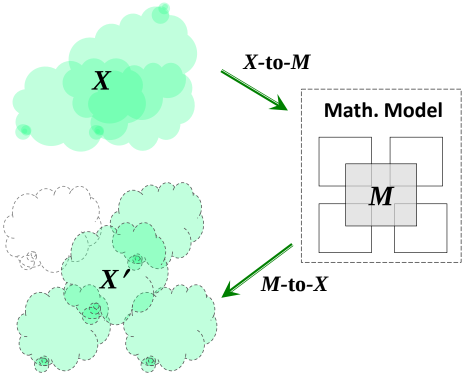

Figure 10: From mathematical models to design patterns

We have found several such cases in our work with categorical modeling of MDE-constructs. For example, the notion of a jointly-monic n-ary arrow span turns out to be crucial for modeling associations between object classes, and their correct implementation as well [17]. It is interesting to observe how a simple arrow arrangement allows one to clean the UML metamodel and essentially simplify notation [12, 8]. Another example is modeling intermodel mappings by Kleisli morphisms, which provide a universal pattern for model matching (a.k.a alignment) and greatly simplify model merge as discussed in Sect. 4.1. In addition, the Kleisli view of model mappings provides a design pattern for mapping composition — a problem considered to be difficult in the model management literature [3]. Sequential composition of symmetric delta lenses is also not evident; considering such lenses as algebras whose carriers are profunctors (see Appendix) suggests a precise pattern to be checked (this work is now in progress). Decomposition of a model transformation into Cartesian lifting (view execution) followed by the inverse operation of Cartesian lifting completion (view updating) as described in Section 4.2.3 provides a useful guidance for model transformation design, known to be laborious and error-prone. In particular, it immediately provides bidirectionality.

The graph transformation community also developed several general patterns applicable to MDE (with models considered as typed attributed graphs, see [23] for details). In particular, an industrial standard for model transformation, QVT [41], was essentially influenced by triple-graph grammars (TGGs). Some applications of TGGs to model synchronization (and further references) can be found in [30].

5.3 Diagrammatic modeling culture and tool architecture.

The design patterns mentioned above are based on the respective categorical machinery (monads, fibrations, profunctors). A software engineer not familiar with these patterns would hardly recognize them in the arrays of implementation details. Even less probable is that he will abstract away his implementation concerns and reinvent such patterns from scratch; distillation of these structures by the CT community took a good amount of time. In contrast, simple arrow diagrams, like in Fig. 7(a) (see also the Appendix), do not actually need any knowledge of CT: all that is required is making intermodel relations explicit, and denoting them by arcs (directed or undirected) connecting the respective objects. To a lesser extent, this also holds for the model transformation decomposition in Fig. 9 and the model merge pattern in Fig. 6. We refer to a lesser extent because the former pattern still needs familiarity with the relations-are-spans idea, and the latter needs an understanding of what colimit is (but, seemingly, it should be enough to understand it roughly as some algebraic procedure of “merging things”).

The importance of mappings between models/software artifacts is now well recognized in many communities within SE, and graphical notations have been employed in SE for a long time. Nevertheless, a majority of model management tools neglect the primary status of model mappings: in their architecture, model matching and alignment are hidden inside (implementations of) algebraic routines, thus complicating both semantics and implementation of the latter; concerns are intricately mixed rather than separated. As all SE textbooks and authorities claim separation of concerns to be a fundamental principle of software design, an evident violation of the principle in the cases mentioned above is an empirical fact that puzzles us. It is not clear why a BX-tool designer working on tool architecture does not consider simple arrow diagrams like in Fig. 7(a), and prefers discrete diagrams (b). The latter are, of course, simpler but their simplicity is deceiving in an almost evident way.

The only explanation we have found is that understanding the deceiving simplicity of discrete diagrams (b), and, simultaneously, manageability of arrow diagrams (a), needs a special diagrammatic modeling culture that a software engineer normally does not possess. This is the culture of elementary arrow thinking, which covers the most basic aspects of manipulating and using arrow diagrams. It appears that even elementary arrow thinking habits are not cultivated in the current SE curriculum, the corresponding high-level specification patterns are missing from the software designer toolkit, and software is often structured and modularized according to the implementation rather than specification concerns.

6 Related work

First applications of CT in computer science, and the general claim of CT's extreme usefulness for computer applications should be, of course, attributed to Joseph Goguen [29]. The shift from modeling semantics of computation (behavior) to modeling structures of software programs is emphasized by José Fiadeiro in the introduction to his book [36], where he refers to a common "social" nature of both domains. The ideas put forward by Fiadeiro were directly derived from joint work with Tom Maibaum on what has become known as component based design and software architecture [26, 27, 24]. A clear visualization of these ideas by Fig. 2 (with M standing for Fiadeiro's "social") seems to be new. The idea of round-tripping modeling chain Fig. 3 appears to be novel, its origin can be traced to [11].

Don Batory makes an explicit call to using CT in MDE in his invited lecture for MoDELS'2008 [2], but he employs the very basic categorical means, in fact, arrow composition only. In our paper we refer to much more advanced categorical means: sketches, fibrations, Cartesian monads, Kleisli categories.

Generalized sketches (graphs with diagram predicates) as a universal syntactical machinery for formalizing different kinds of models were proposed by Diskin *et al*, [18]. Their application to special MDE problems can be found in [17, 38] and in the work of Rutle *et al*, see [46], [43] and references therein. A specific kind of sketches, ER-sketches, is employed for a number of problems in the database context by Johnson *et al* [32]. Considering models as typed attributed graphs with applications to MDE has been extensively put forward by the graph transformation (GT) community [23]; their work is much more operationally oriented than our concerns in the present paper. On the other hand, in contrast to the generalized sketches framework, constraints seem to be not the first-class citizens in the GT world.

The shift from functorial to fibrational semantics for sketches to capture the metamodeling foundations of MDE was proposed in [13] and formalized in [20]. This semantics is heavily used in [15], and in the work of Rutle *et al* mentioned above. Comparison of the two semantic approaches, functorial and fibrational, and the challenges of proving their equivalence, are discussed in [52].

The idea of modeling query languages by monads, and metamodel (or data schema) mappings by Kleisli mappings, within the functorial semantics approach, was proposed in [16], and independently by Johnson and Rosebrugh in their work on ER-sketches [32]. Reformulation of the idea for fibrational semantics was developed and used for specifying important MDE constructs in [15, 21]. An accurate formalization via Cartesian monads can be found in [19].

Algebraic foundations for BX is now an area of active research. Basics of the state-based algebraic framework (lenses) were developed by Pierce with coauthors [28]; their application to MDE is due to Stevens [50]. Delta-lenses [22, 10] is a step towards categorical foundations, but they have been described in elementary terms using tile algebra [15]. A categorical approach to the view update problem has been developed by Johnson and Rosebrugh *et al*[33]; and extended to categorical foundations for lenses based on KZ-monads in [35, 34]. The notion of symmetric delta lens in Appendix is new; it results from incorporating the monotonic PutPut-law idea of Johnson and Rosebrugh into the earlier notion of symmetric delta lens [10]. Assembling synchronization procedures from elementary blocks is discussed in [15].

7 Conclusion

The paper claims that category theory is a good choice for building mathematical foundations for MDE. We first discuss two very general prerequisites that concepts and structures developed in category theory have to be well applicable for mathematical modeling of MDE-constructs. We then exemplify the arguments by sketching several categorical models, which range from general definitions of multimodeling and intermodeling to important model management scenarios of model merge and bidirectional update propagation. We briefly explain (and refer to other work for relevant details) that these categorical models provide useful design patterns and guidance for several problems considered to be difficult.

Moreover, even an elementary arrow arrangement of model merge and BX scenarios makes explicit a deficiency of the modern tools automating these scenarios. To wit: these tools' architecture weaves rather than separates such different concerns as (i) model matching and alignment based on heuristics and contextual information, and (ii) relatively simple algebraic routines of merging and update propagation. This weaving complicates both semantics and implementation of the algebraic procedures, does not allow the user to correct alignment if necessary, and makes tools much less flexible. It appears that even simple arrow patterns, and the corresponding structural decisions, may not be evident for a modern software engineer.

Introduction of CT courses into the SE curriculum, especially in the MDE context, would be the most natural approach to the problem: even elementary CT studies should cultivate arrow thinking, develop habits of diagrammatic reasoning and build a specific intuition of what is a healthy vs. ill-formed structure. We believe that such intuition, and the structural lessons one can learn from CT, are of direct relevance for many practical problems in MDE.

Acknowledgement. Ideas presented in the paper have been discussed at NECSIS seminars at McMaster and the University of Waterloo; we are grateful to all their participants, and especially to Michał Antkiewicz and Krzysztof Czarnecki, for useful feedback and stimulating criticism. We have also benefited greatly from discussions with Don Batory, Steve Easterbrook, José Fiadeiro, Michael Healy, Michael Johnson, Ralf Lämmel, Bran Selic and Uwe Wolter. Thanks also go to anonymous referees for comments.

Financial support was provided with the NECSIS project funded by the Automotive Partnership Canada.

References

[1] Michal Antkiewicz, Krzysztof Czarnecki & Matthew Stephan (2009): *Engineering of Framework-Specific Modeling Languages*. *IEEE Trans. Software Eng.* 35(6), pp. 795–824. Available at `http://doi.ieeecomputersociety.org/10.1109/TSE.2009.30`.

[2] Don S. Batory, Maider Azanza & João Saraiva (2008): *The Objects and Arrows of Computational Design*. In Czarnecki et al. [7], pp. 1–20. Available at `http://dx.doi.org/10.1007/978-3-540-87875-9_1`.

[3] Philip A. Bernstein (2003): *Applying Model Management to Classical Meta Data Problems*. In: *CIDR*. Available at `http://www-db.cs.wisc.edu/cidr/cidr2003/program/p19.pdf`.

[4] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz & Alan Schmitt (2008): *Boomerang: resourceful lenses for string data*. In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, ACM, New York, NY, USA, pp. 407–419. Available at `http://dx.doi.org/10.1145/1328438.1328487`.

[5] Artur Boronat, Alexander Knapp, José Meseguer & Martin Wirsing (2008): *What Is a Multi-modeling Language?* In Andrea Corradini & Ugo Montanari, editors: *WADT*, *Lecture Notes in Computer Science* 5486, Springer, pp. 71–87. Available at `http://dx.doi.org/10.1007/978-3-642-03429-9_6`.

[6] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr & James F. Terwilliger (2009): *Bidirectional Transformations: A Cross-Discipline Perspective*. In Richard F. Paige, editor: *ICMT*, *Lecture Notes in Computer Science* 5563, Springer, pp. 260–283. Available at `http://dx.doi.org/10.1007/978-3-642-02408-5_19`.

[7] Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl & Markus Völter, editors (2008): *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*. Lecture Notes in Computer Science 5301, Springer. Available at `http://dx.doi.org/10.1007/978-3-540-87875-9`.

[8] Z. Diskin (2007): *Mappings, maps, atlases and tables: A formal semantics for associations in UML2*. Technical Report CSRG-566, University of Toronto. `http://ftp.cs.toronto.edu/pub/reports/csrg/566/TR-566-umlAssons.pdf`.

[9] Z. Diskin, S. Easterbrook & R. Miller (2008): *Integrating schema integration frameworks, algebraically*. Technical Report CSRG-583, University of Toronto. `http://ftp.cs.toronto.edu/pub/reports/csrg/583/TR-583-schemaIntegr.pdf`.

[10] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann & F. Orejas (2011): *From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case*. In Whittle et al. [51], pp. 304–318. Available at `http://dx.doi.org/10.1007/978-3-642-24485-8_22`.

[11] Zinovy Diskin (2001): *On Modeling, Mathematics, Category Theory and RM-ODP*. In José A. Moinhos Cordeiro & Haim Kilov, editors: WOODPECKER, ICEIS Press, pp. 38–54.

[12] Zinovy Diskin (2002): *Visualization vs. Specification in Diagrammatic Notations: A Case Study with the UML*. In Mary Hegarty, Bernd Meyer & N. Hari Narayanan, editors: Diagrams, Lecture Notes in Computer Science 2317, Springer, pp. 112–115. Available at `http://dx.doi.org/10.1007/3-540-46037-3_15`.

[13] Zinovy Diskin (2005): *Mathematics of Generic Specifications for Model Management*. In Laura C. Rivero, Jorge Horacio Doorn & Viviana E. Ferraggine, editors: Encyclopedia of Database Technologies and Applications, Idea Group, pp. 351–366.

[14] Zinovy Diskin (2008): *Algebraic Models for Bidirectional Model Synchronization*. In Czarnecki et al. [7], pp. 21–36. Available at `http://dx.doi.org/10.1007/978-3-540-87875-9_2`.

[15] Zinovy Diskin (2009): *Model Synchronization: Mappings, Tiles, and Categories*. In João M. Fernandes, Ralf Lämmel, Joost Visser & João Saraiva, editors: GTTSE, Lecture Notes in Computer Science 6491, Springer, pp. 92–165. Available at `http://dx.doi.org/10.1007/978-3-642-18023-1_3`.

[16] Zinovy Diskin & Boris Cadish (1997): *A Graphical Yet Formalized Framework for Specifying View Systems*. In: ADBIS, Nevsky Dialect, pp. 123–132. Available at `http://www.bcs.org/upload/pdf/ewic_ad97_paper17.pdf`.

[17] Zinovy Diskin, Steve M. Easterbrook & Jürgen Dingel (2008): *Engineering Associations: From Models to Code and Back through Semantics*. In Richard F. Paige & Bertrand Meyer, editors: TOOLS (46), Lecture Notes in Business Information Processing 11, Springer, pp. 336–355. Available at `http://dx.doi.org/10.1007/978-3-540-69824-1_19`.

[18] Zinovy Diskin, Boris Kadish, Frank Piessens & Michael Johnson (2000): *Universal Arrow Foundations for Visual Modeling*. In Michael Anderson, Peter Cheng & Volker Haarslev, editors: Diagrams, Lecture Notes in Computer Science 1889, Springer, pp. 345–360. Available at `http://link.springer.de/link/service/series/0558/bibs/1889/18890345.htm`.

[19] Zinovy Diskin, Tom Maibaum & Krzysztof Czarnecki (2012): *Intermodeling, Queries, and Kleisli Categories*. In Juan de Lara & Andrea Zisman, editors: FASE, Lecture Notes in Computer Science 7212, Springer, pp. 163–177. Available at `http://dx.doi.org/10.1007/978-3-642-28872-2_12`.

[20] Zinovy Diskin & Uwe Wolter (2008): *A Diagrammatic Logic for Object-Oriented Visual Modeling*. Electr. Notes Theor. Comput. Sci. 203(6), pp. 19–41. Available at `http://dx.doi.org/10.1016/j.entcs.2008.10.041`.

[21] Zinovy Diskin, Yingfei Xiong & Krzysztof Czarnecki (2010): *Specifying Overlaps of Heterogeneous Models for Global Consistency Checking*. In: MoDELS Workshops, Lecture Notes in Computer Science 6627, Springer, pp. 165–179. Available at `http://dx.doi.org/10.1007/978-3-642-21210-9_16`.

[22] Zinovy Diskin, Yingfei Xiong & Krzysztof Czarnecki (2011): *From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case*. Journal of Object Technology 10, pp. 6: 1–25. Available at `http://dx.doi.org/10.5381/jot.2011.10.1.a6`.

[23] H. Ehrig, K. Ehrig, U. Prange & G. Taenzer (2006): *Fundamentals of Algebraic Graph Transformation*.

[24] J. L. Fiadeiro & T. S. E. Maibaum (1995): *A Mathematical Toolbox for the Software Architect*. In J. Kramer & A. Wolf, editors: 8th Int. Workshop on Software Specification and Design, IEEE CS Press, pp. 46–55.

[25] José Luiz Fiadeiro (2004): *Software Services: Scientific Challenge or Industrial Hype?* In Zhiming Liu & Keijiro Araki, editors: ICTAC, Lecture Notes in Computer Science 3407, Springer, pp. 1–13. Available at `http://dx.doi.org/10.1007/978-3-540-31862-0_1`.

[26] José Luiz Fiadeiro & T. S. E. Maibaum (1992): *Temporal Theories as Modularisation Units for Concurrent System Specification*. Formal Asp. Comput. 4(3), pp. 239–272. Available at `http://dx.doi.org/10.1007/BF01212304`.

[27] José Luiz Fiadeiro & T. S. E. Maibaum (1995): *Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality*. In: SIGSOFT FSE, pp. 72–80. Available at `http://doi.acm.org/10.1145/222124.222141`.

[28] J. N. Foster, M. Greenwald, J. Moore, B. Pierce & A. Schmitt (2007): *Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem*. ACM Trans. Program. Lang. Syst. 29(3), doi:10.1145/1232420.1232424.

[29] Joseph A. Goguen (1991): *A Categorical Manifesto*. Mathematical Structures in Computer Science 1(1), pp. 49–67. Available at `http://dx.doi.org/10.1017/S0960129500000050`.

[30] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin & Yingfei Xiong (2011): *Correctness of Model Synchronization Based on Triple Graph Grammars*. In Whittle et al. [51], pp. 668–682. Available at `http://dx.doi.org/10.1007/978-3-642-24485-8_49`.

[31] M. Hofmann, B. Pierce & D. Wagner (2011): *Symmetric Lenses*. In: POPL. Available at `http://doi.acm.org/10.1145/1328438.1328487`.

[32] M. Johnson, R. Rosebrugh & R. Wood (2002): *Entity-relationship-attribute designs and sketches*. Theory and Applications of Categories 10(3), pp. 94–112.

[33] Michael Johnson & Robert D. Rosebrugh (2007): *Fibrations and universal view updatability*. Theor. Comput. Sci. 388(1-3), pp. 109–129. Available at `http://dx.doi.org/10.1016/j.tcs.2007.06.004`.

[34] Michael Johnson & Robert D. Rosebrugh (2012): *Lens put-put laws: monotonic and mixed*. To appear. http://www.easst.org/eceasst.

[35] Michael Johnson, Robert D. Rosebrugh & Richard J. Wood (2010): *Algebras and Update Strategies*. J. UCS 16(5), pp. 729–748. Available at `http://dx.doi.org/10.3217/jucs-016-05-0729`.

[36] José Fiadeiro (2004): *Categories for Software Engineering*. Springer.

[37] Stefan Jurack & Gabriele Taentzer (2009): *Towards Composite Model Transformations Using Distributed Graph Transformation Concepts*. In Andy Schürr & Bran Selic, editors: MoDELS, Lecture Notes in Computer Science 5795, Springer, pp. 226–240. Available at `http://dx.doi.org/10.1007/978-3-642-04425-0_17`.

[38] Hongzhi Liang, Zinovy Diskin, Jürgen Dingel & Ernesto Posse (2008): *A General Approach for Scenario Integration*. In Czarnecki et al. [7], pp. 204–218. Available at `http://dx.doi.org/10.1007/978-3-540-87875-9_15`.

[39] M. Makkai (1997): *Generalized Sketches as a Framework for Completeness Theorems*. Journal of Pure and Applied Algebra 115, pp. 49–79, 179–212, 214–274.

[40] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana & Masato Takeichi (2007): *Bidirectionalization transformation based on automatic derivation of view complement functions*. In Ralf Hinze & Norman Ramsey, editors: ICFP, ACM, pp. 47–58. Available at `http://dx.doi.org/10.1145/1291151.1291162`.

[41] Object Management Group (2008): *MOF Query / Views / Transformations Specification 1.0*. http://www.omg.org/docs/formal/08-04-03.pdf.

[42] Rachel Pottinger & Philip A. Bernstein (2003): *Merging Models Based on Given Correspondences*. In: VLDB, pp. 826–873. Available at http://www.vldb.org/conf/2003/papers/S26P01.pdf.

[43] Alessandro Rossini, , Juan de Lara, Esther Guerra, Adrian Rutle & Yngve Lamo (2012): *A Graph Transformation-based Semantics for Deep Metamodelling*. In: AGTIVE 2012. Available at http://dx.doi.org/10.1016/j.jlap.2009.10.003. To appear.

[44] Alessandro Rossini, Adrian Rutle, Yngve Lamo & Uwe Wolter (2010): *A formalisation of the copy-modify-merge approach to version control in MDE*. J. Log. Algebr. Program. 79(7), pp. 636–658. Available at http://dx.doi.org/10.1016/j.jlap.2009.10.003.

[45] Adrian Rutle, Alessandro Rossini, Yngve Lamo & Uwe Wolter (2010): *A Formalisation of Constraint-Aware Model Transformations*. In David S. Rosenblum & Gabriele Taentzer, editors: FASE, Lecture Notes in Computer Science 6013, Springer, pp. 13–28. Available at http://dx.doi.org/10.1007/978-3-642-12029-9_2.

[46] Adrian Rutle, Alessandro Rossini, Yngve Lamo & Uwe Wolter (2012): *A formal approach to the specification and transformation of constraints in MDE*. J. Log. Algebr. Program. 81(4), pp. 422–457. Available at http://dx.doi.org/10.1016/j.jlap.2012.03.006.

[47] Bran Selic (2008): *Personal reflections on automation, programming culture, and model-based software engineering*. Autom. Softw. Eng. 15(3-4), pp. 379–391. Available at http://dx.doi.org/10.1007/s10515-008-0035-7.

[48] M. Shaw (1996): *Three patterns that help explain the development of software engineering (position paper)*. In: Dagstuhl Workshop on Software Architecture.

[49] Stefano Spaccapietra & Christine Parent (1994): *View Integration: A Step Forward in Solving Structural Conflicts*. IEEE Trans. Knowl. Data Eng. 6(2), pp. 258–274. Available at http://doi.ieeecomputersociety.org/10.1109/69.277770.

[50] Perdita Stevens (2010): *Bidirectional model transformations in QVT: semantic issues and open questions*. Software and System Modeling 9(1), pp. 7–20. Available at http://dx.doi.org/10.1007/s10270-008-0109-9.

[51] Jon Whittle, Tony Clark & Thomas Kühne, editors (2011): *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*. Lecture Notes in Computer Science 6981, Springer. Available at http://dx.doi.org/10.1007/978-3-642-24485-8.

[52] Uwe Wolter & Zinovy Diskin: *From Indexed to Fibred Semantics The Generalized Sketch File*. Technical Report 361, Department of Informatics, University of Bergen, Norway. http://www.ii.uib.no/publikasjoner/texrap/pdf/2007-361.pdf, year = 2007,.

[53] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi & H. Mei (2007): *Towards automatic model synchronization from model transformations*. In: ASE, pp. 164–173. Available at http://doi.acm.org/10.1145/1321631.1321657.

A Appendix. Algebra of bidirectional update propagation

In Section 4.2, we considered operations of update propagation, but did not specify any laws they must satisfy. Such laws are crucial for capturing semantics, and the present section aims to specify algebraic laws for BX. We will do it in an elementary way using tile algebra (rather than categorically — it is non-trivial and left for a future work). We will begin with the notion of an *alignment framework* to formalize delta composition ($*$ in Section 4.2), and then proceed to algebraic structures modeling BX — *symmetric delta lenses*. (Note that the lenses we will introduce here are different from those defined in [10].)

$$\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, "a"'] \arrow[r, phantom, shift right=4ex, "\Searrow{:}\mathsf{fAln}"'] & \boxed{B} \arrow[dl, leftrightarrow, dashed, "a*r"] \\
\boxed{A'} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[dr, leftrightarrow, dashed, "r*b"'] \arrow[r, phantom, shift right=4ex, "\Swarrow{:}\mathsf{bAln}"'] & \boxed{B} \arrow[d, leftrightarrow, "b"] \\
 & \boxed{B'}
\end{tikzcd}$$

(a) arities

$$\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, "a"'] \arrow[dr, leftrightarrow, dashed, "r*b"] & \boxed{B} \arrow[d, leftrightarrow, "b"] \arrow[dl, leftrightarrow, dashed, "a*r"'] \\
\boxed{A'} \arrow[r, leftrightarrow, dashed, "r'"] & \boxed{B'}
\end{tikzcd}$$

(b) (fAln-bAln) law

Figure 11: Realignment operations and their laws

Definition 1 () An *alignment framework* is given by the following data.

(i) Two categories with pullbacks, $\mathbf{A}$ and $\mathbf{B}$, called *model spaces*. We will consider spans in these categories up to their equivalence via a head isomorphism commuting with legs. That is, we will work with equivalence classes of spans, and the term 'span' will refer to an equivalence class of spans. Then span composition (via pullbacks) is strictly associative, and we have categories (rather than bicategories) of spans, $\mathsf{Span}_1(\mathbf{A})$ and $\mathsf{Span}_1(\mathbf{B})$. Their subcategories consisting of spans with injective legs will be denoted by $\mathbf{A}^\bullet$ and $\mathbf{B}^\bullet$ resp.

Such spans are to be thought of as *(model) updates*. They will be depicted by vertical bi-directional arrows, for example, a and b in the diagrams Fig. 11(a). We will assume that the upper node of such an arrow is its formal source, and the lower one is the target; the source is the the original (state of the) model, and the target is the updated model. Thus, model evolution is directed down.

A span whose upper leg is identity (nothing deleted) is an *insert update*; it will be denoted by unidirectional arrows going down. Dually, a span with identity lower leg is a *delete update*; it will be denoted by a unidirectional arrow going up (but the formal source of such an arrow is still the upper node).

(ii) For any two objects, $A \in \mathbf{A}_0$ and $B \in \mathbf{B}_0$, there is a set $R(A,B)$ of *correspondences* (or *corrs* in short) from A to B. Elements of $R(A,B)$ will be depicted by bi-directional horizontal arrows, whose formal source is A and the target is B.

Updates and corrs will also be called *vertical* and *horizontal deltas*, resp.

(iii) Two diagram operations over corrs and updates called *forward* and *backward (re)alignment*. Their arities are shown in Fig. 11(a) (output arrows are dashed). We will also write $a * r$ for $\mathsf{fAln}(a,r)$ and $r * b$ for $\mathsf{bAln}(b,r)$. We will often skip the prefix 're' and say 'alignment' to ease terminology.

There are three laws regulating alignment. Identity updates do not actually need realignment:

$$(\mathsf{IdAln}) \qquad \mathsf{id}A * r = r = r * \mathsf{id}B$$

for any corr $r\colon A \leftrightarrow B$.

The result of applying a sequence of interleaving forward and backward alignments does not depend on the order of application as shown in Fig. 11(b):

$$(\mathsf{fAln}{-}\mathsf{bAln}) \qquad (a * r) * b = a * (r * b)$$

for any corr r and any updates a, b.

We will call diagrams like those shown in Fig. 11(a,b) *commutative* if the arrow at the respective operation output is indeed equal to that one computed by the operation. For example, diagram (b) is commutative if $r' = a * r * b$.

$$\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, "a"'] \arrow[dr, phantom, "\searrow\!:\mathsf{fPpg}"] & \boxed{B} \arrow[d, leftrightarrow, dashed, "b"] \\
\boxed{A'} \arrow[r, leftrightarrow, dashed, "r'"] & B'
\end{tikzcd}
\quad
\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, "\mathsf{id}A"'] \arrow[dr, phantom, "\searrow\!:\mathsf{fPpg}"] & \boxed{B} \arrow[d, leftrightarrow, dashed, "\mathsf{id}B"] \\
\boxed{A} \arrow[r, leftrightarrow, dashed, "r"] & B
\end{tikzcd}
\quad
\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, "a"'] \arrow[dr, phantom, "\searrow\!:\mathsf{fPpg}"] & \boxed{B} \arrow[d, dashed, "b"] \\
\boxed{A'} \arrow[r, leftrightarrow, dashed, "r'"] & B'
\end{tikzcd}
\quad
\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[dr, phantom, "\searrow\!:\mathsf{fPpg}"] & \boxed{B} \\
\boxed{A'} \arrow[u, "a"] \arrow[r, leftrightarrow, dashed, "r'"] & B' \arrow[u, dashed, "b"']
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, dashed, "a"] \arrow[dr, phantom, "\swarrow\!:\mathsf{bPpg}"] & \boxed{B} \arrow[d, leftrightarrow, "b"] \\
A' \arrow[r, leftrightarrow, dashed, "r'"] & \boxed{B'}
\end{tikzcd}
\quad
\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, dashed, "\mathsf{id}A"'] \arrow[dr, phantom, "\swarrow\!:\mathsf{bPpg}"] & \boxed{B} \arrow[d, leftrightarrow, "\mathsf{id}B"] \\
A \arrow[r, leftrightarrow, dashed, "r"] & \boxed{B}
\end{tikzcd}
\quad
\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, dashed, "a"'] \arrow[dr, phantom, "\swarrow\!:\mathsf{bPpg}"] & \boxed{B} \arrow[d, "b"] \\
A' \arrow[r, leftrightarrow, dashed, "r'"] & \boxed{B'}
\end{tikzcd}
\quad
\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[dr, phantom, "\swarrow\!:\mathsf{bPpg}"] & \boxed{B} \\
A' \arrow[u, dashed, "a"] \arrow[r, leftrightarrow, dashed, "r'"] & \boxed{B'} \arrow[u, "b"']
\end{tikzcd}$$

(a) arities (b) (IdPpg) laws (c) monotonicity laws

Figure 12: Operations of update propagation

Finally, alignment is compositional: for any consecutive updates $a\colon A \to A'$, $a'\colon A' \to A''$, $b\colon B \to B'$, $b'\colon B' \to B''$, the following holds:

$$(\mathsf{AlnAln}) \qquad a' * (a * r) = (a;a') * r \text{ and } (r * b) * b' = r * (b;b')$$

where ; denotes sequential span composition.

It is easy to see that having an alignment framework amounts to having a functor $\alpha\colon \mathbf{A}^\bullet \times \mathbf{B}^\bullet \to \boldsymbol{Set}$.

Definition 2 () A *symmetric delta lens* (briefly, an sd-lens) is a triple $(\alpha, \mathsf{fPpg}, \mathsf{bPpg})$ with $\alpha\colon \mathbf{A}^\bullet \times \mathbf{B}^\bullet \to \boldsymbol{Set}$ an alignment framework, and fPpg, bPpg two diagram operations over corrs and updates (called forward and backward update propagation, resp.). The arities are specified in Fig. 12(a) with output arrows dashed and output nodes not framed. Sometimes we will use a linear notation and write $b = a.\mathsf{fPpg}(r)$ and $a = b.\mathsf{bPpg}(r)$ for the cases specified in the diagrams.

Each operation must satisfy the following laws.

Stability or IdPpg law: if nothing changes on one side, nothing happens on the other side as well, that is, identity mappings are propagated into identity mappings as shown by diagrams Fig. 12(b).

Monotonicity: Insert updates are propagated into inserts, and delete updates are propagated into deletes, as specified in Fig. 12(c).

Monotonic Compositionality or PpgPpg law: composition of two consecutive inserts is propagated into composition of propagations as shown by the left diagram in Fig. 13 (to be read as follows: if the two squares are fPpg, then the outer rectangle is fPpg as well). The right diagram specifies compositionality for deletes. The same laws are formulated for bPpg.

Note that we do not require compositionality for propagation of general span updates. The point is that interleaving inserts and deletes can annihilate, and lost information cannot be restored: see [28, 22, 10] for examples.

Commutativity: Diagrams Fig. 12(a) must be commutative in the sense that $a * r * b = r'$.

Finally, forward and backward propagation must be coordinated with each other by some ***invertibility*** law. Given a corr $r\colon A \to B$, an update $a\colon A \to A'$ is propagated into update $b = a.\mathsf{fPpg}(r)$, which can be propagated back to update $a' = b.\mathsf{bPpg}(r)$. For an ideal situation of *strong invertibility*, we should require $a' = a$. Unfortunately, this does not hold in general because the $\mathbf{A}^\bullet$-specific part of the information is lost

$$\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, "a"'] \arrow[dr, phantom, "\searrow{:}\mathsf{fPpg}"] & \boxed{B} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, dashed, "b"] \arrow[dr, phantom, "\searrow{:}\mathsf{bPpg}"] & \boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, dashed, "a_1"] \arrow[dr, phantom, "\searrow{:}\mathsf{fPpg}"] & \boxed{B} \arrow[d, leftrightarrow, dashed, "b"] \\
\boxed{A'} \arrow[r, leftrightarrow, dashed, "r'"] & B' \arrow[r, leftrightarrow, dashed, "r'_1"] & A'_1 \arrow[r, leftrightarrow, dashed, "r'_1"] & B'
\end{tikzcd}$$

(a) fbfPpg law

$$\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, dashed, "a"'] & \boxed{B} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, dashed, "b_1"] \arrow[dl, phantom, "\swarrow{:}\mathsf{bPpg}"] & \boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, leftrightarrow, dashed, "a"] \arrow[dl, phantom, "\swarrow{:}\mathsf{fPpg}"] & \boxed{B} \arrow[d, leftrightarrow, "b"] \arrow[dl, phantom, "\swarrow{:}\mathsf{bPpg}"] \\
A' \arrow[r, leftrightarrow, dashed, "r'_1"] & B'_1 \arrow[r, leftrightarrow, dashed, "r'_1"] & A' \arrow[r, leftrightarrow, dashed, "r'"] & \boxed{B'}
\end{tikzcd}$$

(b) bfbPpg law

Figure 14: Round-tripping laws. (Scenario in diagram (b) "runs" from the right to the left.)

in passing from a to b, and cannot be restored [10]. However, it makes sense to require the following *weak invertibility* specified in Fig. 14, which does hold in a majority of practically interesting situations, e.g., for BX determined by TGG-rules [30]. The law fbfPpg says that although $a_1 = a.\mathsf{fPpg}(r).\mathsf{bPpg}(r) \neq a$, a_1 is equivalent to a in the sense that $a_1.\mathsf{fPpg}(r) = a.\mathsf{fPpg}(r)$. Similarly for the bfbPpg law.

$$\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[d, "a"'] \arrow[dr, phantom, "\searrow{:}\mathsf{fPpg}"] & \boxed{B} \arrow[d, dashed, "b"] \\
\boxed{A'} \arrow[r, leftrightarrow, dashed, "r'"] \arrow[d, "a'"'] \arrow[dr, phantom, "\searrow{:}\mathsf{fPpg}"] & B' \arrow[d, dashed, "b'"] \\
\boxed{A''} \arrow[r, leftrightarrow, dashed, "r''"] & B''
\end{tikzcd}
\qquad
\begin{tikzcd}
\boxed{A} \arrow[r, leftrightarrow, "r"] \arrow[dr, phantom, "\searrow{:}\mathsf{fPpg}"] & \boxed{B} \\
\boxed{A'} \arrow[r, leftrightarrow, dashed, "r'"] \arrow[u, "a"] \arrow[dr, phantom, "\searrow{:}\mathsf{fPpg}"] & B' \arrow[u, dashed, "b"'] \\
\boxed{A''} \arrow[r, leftrightarrow, dashed, "r''"] \arrow[u, "a'"] & B'' \arrow[u, dashed, "b'"']
\end{tikzcd}$$

Figure 13: Monotonic (PpgPpg) laws

The notion of sd-lens is specified above in elementary terms using tile algebra. Its categorical underpinning is not evident, and we only present several brief remarks.

1) An alignment framework $\alpha\colon \mathbf{A}^\bullet \times \mathbf{B}^\bullet \to \boldsymbol{Set}$ can be seen as a profunctor, if $\mathbf{A}$-arrows will be considered directed up (i.e., the formal source of update a in diagram Fig. 11(a) is A', and the target is A). Then alignment amounts to a functor $\alpha\colon \mathbf{A}^{\bullet\mathsf{op}} \times \mathbf{B}^\bullet \to \boldsymbol{Set}$, that is, a profunctor $\alpha\colon \mathbf{B}^\bullet \nrightarrow \mathbf{A}^\bullet$. Note that reversing arrows in $\mathbf{A}^\bullet$ actually changes the arity of operation fAln: now its input is a pair (a,r) with a an update and r a corr from the target of a, and the output is a corr r' from the source of a, that is, realignment goes back in time.

2) Recall that operations fPpg and bPpg are functorial wrt. injective arrows in $\mathbf{A}$, $\mathbf{B}$, not wrt. arrows in $\mathbf{A}^\bullet$, $\mathbf{B}^\bullet$. However, if we try to resort to $\mathbf{A}$, $\mathbf{B}$ entirely and define alignment wrt. arrows in $\mathbf{A}$, $\mathbf{B}$, then we will need two fAln operations with different arities for inserts and deletes, and two bAln operations with different arities for inserts and deletes. We will then have four functors $\alpha_i\colon \mathbf{A} \times \mathbf{B} \to \boldsymbol{Set}$ with i ranging over four-element set $\{\mathit{insert}, \mathit{delete}\} \times \{\mathbf{A}, \mathbf{B}\}$.

3) The weak invertibility laws suggest that a Galois connection/adjunction is somehow hidden in sd-lenses.

4) Working with chosen spans and pullbacks rather than with their equivalence classes provides a more constructive setting (given we assume the axiom of choice), but then associativity of span composition only holds up to chosen natural isomorphisms, and $\mathbf{A}^\bullet$ and $\mathbf{B}^\bullet$ have to be considered bicategories rather than categories.

All in all, we hope that the categorical analysis of asymmetric delta lenses developed by Johnson *et al* [35, 34] could be extended to capture the symmetric case too.