Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case

Zinovy Diskin[a] Yingfei Xiong[a] Krzysztof Czarnecki[a]

a. Generative Software Development Lab, University of Waterloo, Canada

Abstract Existing bidirectional model transformation (BX) languages are mainly state-based: model alignment is hidden inside update propagating procedures, and model deltas are implicit. Weaving alignment with update propagation complicates the latter and makes it less predictable and less manageable. We propose to separate concerns and consider two distinct operations: delta discovery (alignment) and delta propagation. This architecture has several technological advantages, but requires a corresponding theoretical support.

The goal of the paper is to develop a delta-based algebraic framework for the case of *asymmetric* BX, where one model is a view of the other. In this framework, model spaces are categories (nodes are models and arrows are composable deltas), and delta propagation procedures are mappings between them. We call the corresponding algebras *delta lenses*, prove their several basic properties, and explore their relationships with ordinary lenses — well-known algebraic models for state-based asymmetric BX.

Keywords Model transformation, Bidirectional transformations, Lenses

1 Introduction

A bidirectional transformation (BX) synchronizes two models by propagating updates between them. An important class of BXs is when one of the models, say, B, is a view to the other, A, which abstracts away some information from A; we thus have a many-to-one relationship between the states of the models. That is, states of B are deterministically computed from A-states, but different A-states can have the same B-view. We call such BX *asymmetric*. The *symmetric* case of many-to-many relationship between states of the models is more complicated and considered in another paper [DXC+11].

Several asymmetric BX systems synchronizing different kinds of models have been developed [FGM+07, BFP+08, XLH+07, MHN+07, ACS09]. As a rule, they are

Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki. *From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case.* In Journal of Object Technology, vol. 10, 2011, pages 6:1–25.
doi:10.5381/jot.2011.10.1.a6

provided with algebraic models specifying behavior of propagating procedures. Indeed, understanding behavior of synchronization tools, particularly implementing BXs, is important for the user, who needs precise specifications in order to avoid surprises after automatic synchronization is done. Many algebraic models are variants of an algebraic structure called a *lens* [FGM+07].

A basic lens synchronizing a *source* model A and a *view* model B consists of two operations: unary ***get*** that computes (*gets*) the view, and binary ***put*** that computes the updated source if the view is updated (*puts the view update back*),

$$B = \textbf{\textit{get}}(A) \text{ and } A' = \textbf{\textit{put}}(B', A), \tag{1}$$

where unprimed letters denote original, and primed ones updated, states of the models. To simplify terminology, below we will often say *model* for a model state.

The intended behavior of a lens is specified by a number of equational laws the two operations must satisfy, e.g., $\textbf{\textit{get}}(\textbf{\textit{put}}(B', A)) = B'$. The more laws are stated, and the more transparent their semantics is, the more predictable is BX's behavior for the user.

Lenses, together with most of the existing BX tools, are *state-based*: propagation procedures take states of models before and after updates as the input, and do not require the information about how updates were actually done (normally recorded in edit logs). Loose coupling between synchronizers and applications provided by this architecture is an essential technological advantage [FGK+07]. However, it comes at a price.

Problems of the state-based synchronization. To propagate changes from model B' to model A, the tool must first *align* the two models, that is, map elements in B' to the corresponding elements in A, detect changed elements in B', and then update the respective elements in A. However, model alignment (also often called model *differencing*) is an expensive and complex operation. It is based on fine-grained heuristics, and depends on objects being aligned and some contextual information [XS05, TBWK07, LGJ07, AAAN+08]. Weaving model alignment with update propagation causes two types of problems in the use of synchronization tools.

The first one is related to interfaces. As neither heuristics is absolutely reliable, the user may want to control the alignment process, for example, choose an alignment strategy, pick-up a heuristics, or correct an automatically discovered delta. However, controlling alignment is impossible with state-based interfaces.

The problems of the second type are about semantics of synchronization procedures and their predictability. Complexities of alignment woven into update propagation significantly complicate the latter and make it less predictable. Even for simple string-based structures, operation ***put*** shows not quite manageable behavior [BFP+08], especially if reordering is involved; it is the more so for complex structures as models. Specifically, in the paper we will show that state-based BXs for model synchronization suffer from two serious problems:

(P1) ill-formed sequential composition of transformations, and

(P2) lack of reasonable laws regulating compatibility of update propagation with update composition.

Solutions: Deltas to the rescue. In the center of our approach is the notion of *(intermodel) delta* — a specification of commonalities and differences between two models. To achieve better modularity and separate concerns, we decompose update propagation into two operations: computing deltas (model alignment or differencing ***dif***), and propagating deltas (***dput***). In more detail, since alignment between the

source and the view is provided by the view definition, all that we need to align models B' and A is delta $\Delta_{BB'}$ between B and B'. Then a pure update propagation operation ***dput*** takes the delta and computes the updated source A' together with its alignment to the original source via delta $\Delta_{AA'}$:

$$A' = \Delta_{AA'}(A), \text{ where } \Delta_{AA'} = \textbf{\textit{dput}}(\Delta_{BB'}, A) \text{ and } \Delta_{BB'} = \textbf{\textit{dif}}_Y(B, B'), \qquad (2)$$

where Y is a parameter denoting an alignment strategy Y, and $\Delta_{AA'}(A)$ denotes application of the delta to the initial state. Synchronization architecture described by delta-based schema (2) has the following advantages over state-based schema (1).

1. Schema (2) is more flexible and can be adjusted to quite different user's needs.
   - If the user wants to control alignment and its results (the main mode of the architecture), then separation of delta discovery and propagation does allow the user to do that (e.g., the user can correct the results of automatic alignment $\Delta_{BB'}$, and pass to ***dput*** an improved delta $\Delta^!_{BB'}$).
   - Otherwise, the original state-based interface can be recovered by fixing a strategy Y and reconstructing ***put*** by composing $\textbf{\textit{dif}}_Y$ and ***dput***. Thus, schema (2) subsumes (1).
   - Finally, if the synchronizer can be tightly coupled with the application, deltas can be obtained by recording the user operations within the applications; alignment is not needed and operation $\textbf{\textit{dif}}_Y$ is skipped.
2. Deltas produced by update propagation ***dput*** can be passed to the input of other BX thus reducing the amount of expensive alignment invocations, and simultaneously ensuring correct composition (thus solving problem (P1)).
3. We will show that operation ***dput*** enjoys a cleaner and more manageable algebraic theory than ***put***, and problem (P2) can be significantly alleviated.

Technical goals of the paper. Synchronization architecture (2) converts update propagation into delta propagation, and changes the underlying algebraic framework. The main goal of the paper is to build a simple algebraic model of delta propagation in the asymmetric case, and specify its properties.

Making deltas explicit essentially changes the specification framework: model spaces become graphs whose nodes are models and arrows are deltas. Moreover, a fundamental feature of model deltas is their sequential compositionality, which enjoys associativity and units (identical deltas); it means that model spaces are categories rather than just graphs. In addition, operations of update propagation map identity deltas (idle updates) into identity deltas, and may be compatible with delta composition, that is, exhibit *functorial* properties well-known in category theory. Thus, category theory provides a natural mathematical foundation for delta-based model synchronization, and a secondary goal of the paper is to demonstrate adequacy and convenience of the categorical framework. We do not assume any knowledge of category theory from the reader and explain all categorical concepts we use.

The content. In Section 2 we illustrate two major deficiencies (P1,P2) of the state-based update propagation with simple examples. Section 3 shows how these deficiencies can be fixed or alleviated by introducing deltas into the framework; we also discuss possible implementation of deltas. In Section 4 we formalize the ideas of Section 3: we introduce *delta lenses* and prove their basic properties, particularly,

specifying their relations to state-based lenses.[1] Related work is discussed in Section 5, and Section 6 concludes. In Appendix we show how delta lenses can be applied to specifying synchronization of structured strings, and describe their relation to matching lenses.

2 Problems of State-based BXs

Recall that we use the terms "a model" and "a state of the model" interchangeably.

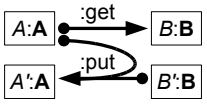
2.1 Background: Lenses, model synchronization, and deltas

We first remind the basic motivation and definition of lenses. Lenses are an asymmetric BX framework: in the pair of models being synchronized, one model (the view) is determined by the other (the source). We have a set of source models $\mathbf{A}$, a set of view models $\mathbf{B}$, and two propagation functions between them, ***get*** and ***put***, whose arities are shown in Fig. 1a. Function ***get*** takes a source model $A{\in}\mathbf{A}$ and computes its view $B{\in}\mathbf{B}$. Function ***put*** takes an updated view model $B'{\in}\mathbf{B}$ and the original source $A{\in}\mathbf{A}$ and computes an updated source $A'{\in}\mathbf{A}$.

Definition 1 (adapted from [FGM$^+$07])**.** A *well-behaved (wb) lens* is a tuple $l = (\mathbf{A}, \mathbf{B}, \boldsymbol{get}, \boldsymbol{put})$ with $\mathbf{A}$ and $\mathbf{B}$ sets called the *source* and the *view space* of the lens, and $\boldsymbol{get}\colon \mathbf{A} \to \mathbf{B}$ and $\boldsymbol{put}\colon \mathbf{B}{\times}\mathbf{A} \to \mathbf{A}$ are functions such that the laws GetPut and PutGet in Fig. 1b hold for any $A \in \mathbf{A}$ and $B' \in \mathbf{B}$. (We write $A.\boldsymbol{get}$ and $\boldsymbol{put}(B, A)$ for the values of ***get*** and ***put*** to ease readability.) We write $l\colon \mathbf{A} \rightleftarrows \mathbf{B}$ for a lens l with the source space $\mathbf{A}$ and the view space $\mathbf{B}$.

A wb lens is called *very well-behaved*, if for any $A \in \mathbf{A}$, $B', B'' \in \mathbf{B}$ the PutPut law holds as well.

In Fig. 1 and below, we use the following notation. Given a function $f\colon X \to Y$, we call a pair (x, y) with $y{=}f(x)$ an *application instance* of f and write $(x, y){:}f$ or $x \overset{:f}{\bullet\!\!\longrightarrow} y$. For a binary $f\colon X_1{\times}X_2 \to Y$, an application instance is a triple (x_1, x_2, y) with $y{=}f(x_1, x_2)$; we will denote it by an arrow with two tails with bullets. Thus, tuples (A, B) and (B, A, A') in Fig. 1 are application instances of operations ***get*** and ***put*** resp.



(a) Operations

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

(b) Equational laws

Figure 1 – Lens operations and laws

Figure 2 shows a transformation instance in the lens framework. The source model specifies Person-objects with their first and last names and birth dates. Each person belongs to a department, which has a department name. Function get extracts a view containing the persons from the "Marketing" department and omits their birth dates. The put function maps view updates back to the source.

Note the change from Melinda French ($p1$) in state B to Melinda Gates ($p1'$) in state B'. This change can be interpreted as the result of two different updates:

(u1) person $p1$ is renamed, or

(u2) person $p1$ is deleted from the model and another person $p1'$ is inserted.

[1] In our conference paper for ICMT'10, delta lenses were called update-based or u-lenses

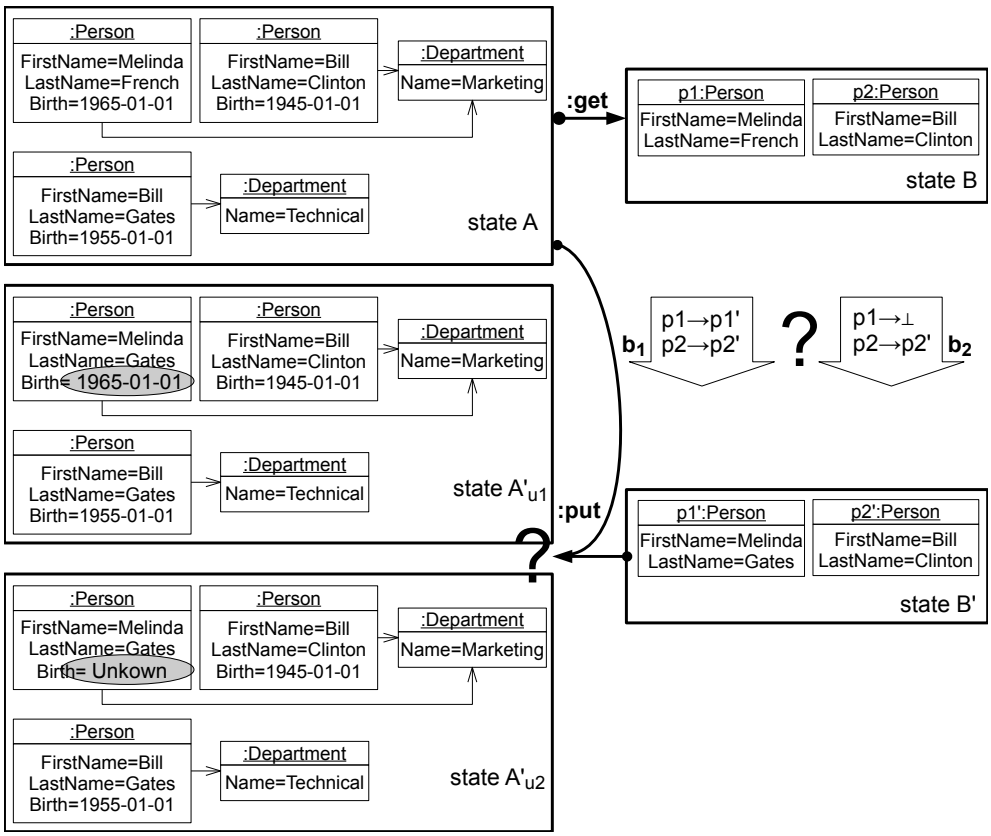


Figure 2 – Deltas do matter

These updates can be specified by a partially defined mapping $b\colon B \Rightarrow B'$ from elements of model B to those of B'. For update (u1), mapping b is defined by setting $b = b_1 = \{p_1 \bullet\!\!\rightarrow p'_1, p_2 \bullet\!\!\rightarrow p'_2\}$; for update (u2), $b = b_2 = \{p_1 \bullet\!\!\rightarrow \bot, p_2 \bullet\!\!\rightarrow p'_2\}$ with symbol $\bot$ meaning *undefined*. We call such mappings *deltas* from B to B'.

A reasonable ***put***-function should translate update (u1) into renaming of Melinda French in the source, whereas (u2) is translated into deletion of Melinda French from the source followed by insertion of a new person Melinda Gates with attribute Birth set to Unknown. Thus, the results of translation A' should be different, $A'_{\mathrm{u1}} \neq A'_{\mathrm{u2}}$, despite the same argument states (B', A). The difference may be more serious than just in the attribute values. Suppose that the model also specifies Cars to be owned by Persons, and in the source model there was a Car object with a single owner Melinda French. Then the backward translation of update (u2) must remove this Car-object as well, and models A'_{u1} and A'_{u2} will have different sets of objects.

Thus, in order to compute the updated source correctly, update propagation operation ***put*** must include delta discovery, i.e., discover an update mapping $b\colon B \Rightarrow B'$ in some or another way. However, apart of compromised modularity discussed in the introduction, hiding delta discovery inside operation ***put*** creates two problems discussed in the next two subsections.

2.2 Problem (P1): ill-formed sequential composition

Compositionality is at the heart of lenses' applications to practical problems. Writing correct bidirectional transformation for complex views is laborious and error-prone. To manage the problem, a complex view is decomposed into a sequence of simple components, say, model $B = B_n$ is a view of B_{n-1}, which is a view of B_{n-2},..., which is a view of $B_0 = A$, such that for each component view a correct lens can be found in a repository. A lens-based language provides the programmer with a number of operators of lens composition. Sequential composition is one of the most important operators, and a fundamental result states that sequential composition of wb lenses is also wb.

Definition 2 (Lens' composition [FGM+07])**.** Given lenses $l\colon \mathbf{A} \rightleftarrows \mathbf{C}$ and $k\colon \mathbf{C} \rightleftarrows \mathbf{B}$, their *sequential composition* $(l;k)\colon \mathbf{A} \rightleftarrows \mathbf{B}$ is defined as follows. For any $A \in \mathbf{A}$, $A.\boldsymbol{get}^{(l;k)} = A.\boldsymbol{get}^{l}.\boldsymbol{get}^{k}$, and for any pair $(B', A) \in \mathbf{B}\times\mathbf{A}$, $\boldsymbol{put}^{(l;k)}(B', A) = \boldsymbol{put}^{l}(C', A)$ where C' stands for $\boldsymbol{put}^{k}(B', A.\boldsymbol{get}^{l})$.

Theorem 1 ([FGM+07])**.** Sequential composition $(l;k)$ is a (very) well-behaved lens as soon as both lenses l and k are such.

For example, transformation in Fig. 2 can be implemented by composing two transformations as shown in Figure 3. The first one (transformation l) removes attribute Birth, and the second one (transformation k) extracts a list of persons from Marketing department. In the backward propagation, both transformations have to rely on model alignment to recover updates from models. Suppose both procedures use keys (sets of objects attributes providing their unique identification): $\boldsymbol{put}^{l}$ uses the key {FirstName, LastName}, and $\boldsymbol{put}^{k}$ uses a smaller key {FirstName}, which, nevertheless, works well for the Marketing Department.

However, sequential composition of these transformations can be incorrect. Suppose Melinda French has married and become Melinda Gates in state B' in Fig. 3. Transformation k will successfully discover this update, and modify the last name of Melinda to Gates in model C'. However, when transformation l compares C and C', it

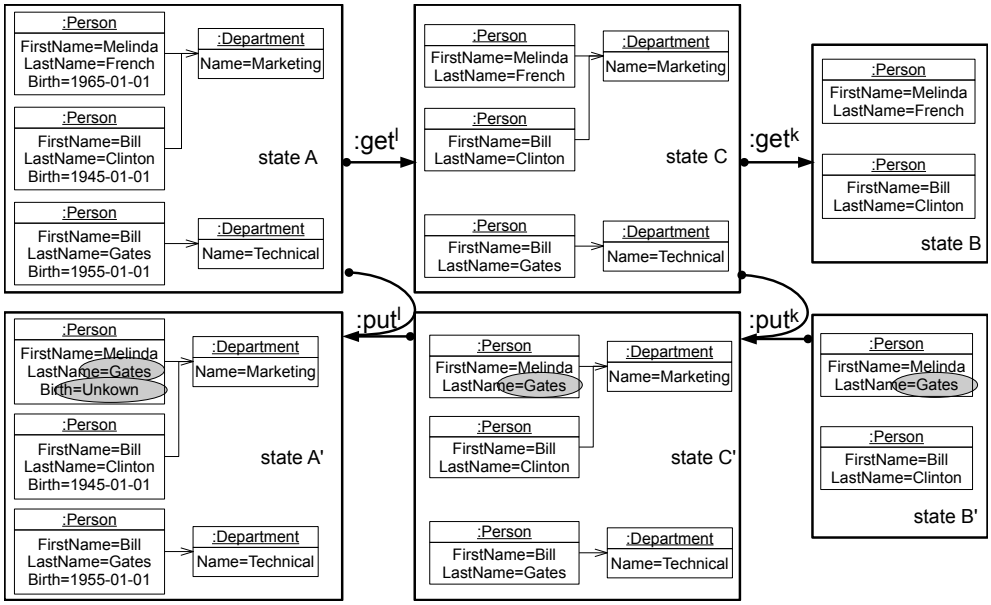


Figure 3 – Incorrect sequential composition of state-based BXs

will consider Melinda Gates as a new person because her last name is different. Then $\boldsymbol{put}^l$ will delete Melinda French in the source model A and insert a new person with an unknown birthday thus coming to state A'.

The result is wrong because the update produced by transformation k and the update discovered by transformation l are different, and hence the two transformations should not be composed. The situation is schematically specified in Fig. 4, where semi-round block arrows denote deltas produced by the respective BXs. The schema suggests a fundamental requirement for sequential composition of BX: two transformations are composable only when they coordinate on both states and deltas. However, this requirement is never specified (and is difficult to specify) in the state-based frameworks because deltas are not explicit. We need a different algebraic model, in which deltas are explicit and occur into inputs and outputs of the operations.

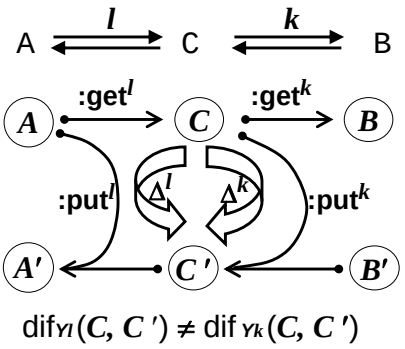


Figure 4 – Schema of ill-formed sequential composition of state-based BXs

2.3 Problem (P2): over-restrictive state-based PutPutlaw

The most controversial law of the basic lens framework is PutPut (Fig. 1b). It says that an updated view state B'' leads to the same updated source A'' regardless of whether the update is performed in one step from B to B'' or with a pair of smaller steps $B{-}B'{-}B''$ through an intermediate state B'. This seems to be a natural requirement for a reasonable backward propagation $\boldsymbol{put}$, but many practically interesting BXs fail to satisfy the law.

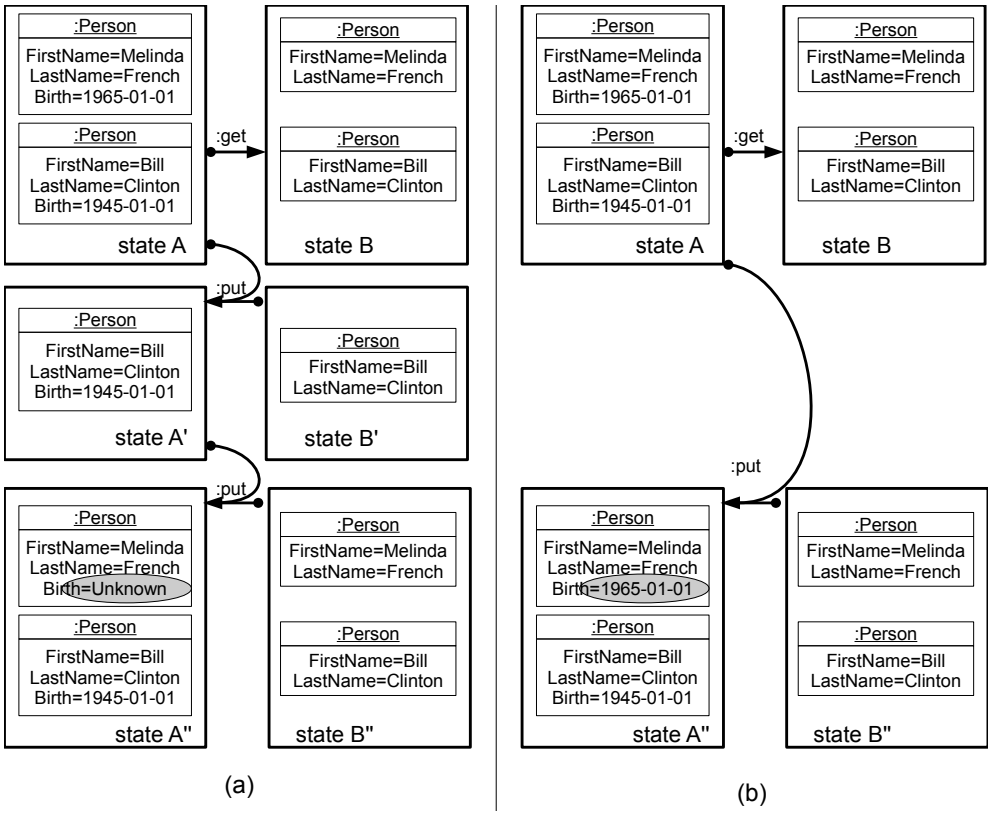


Figure 5 – Violation of state-based PutPut

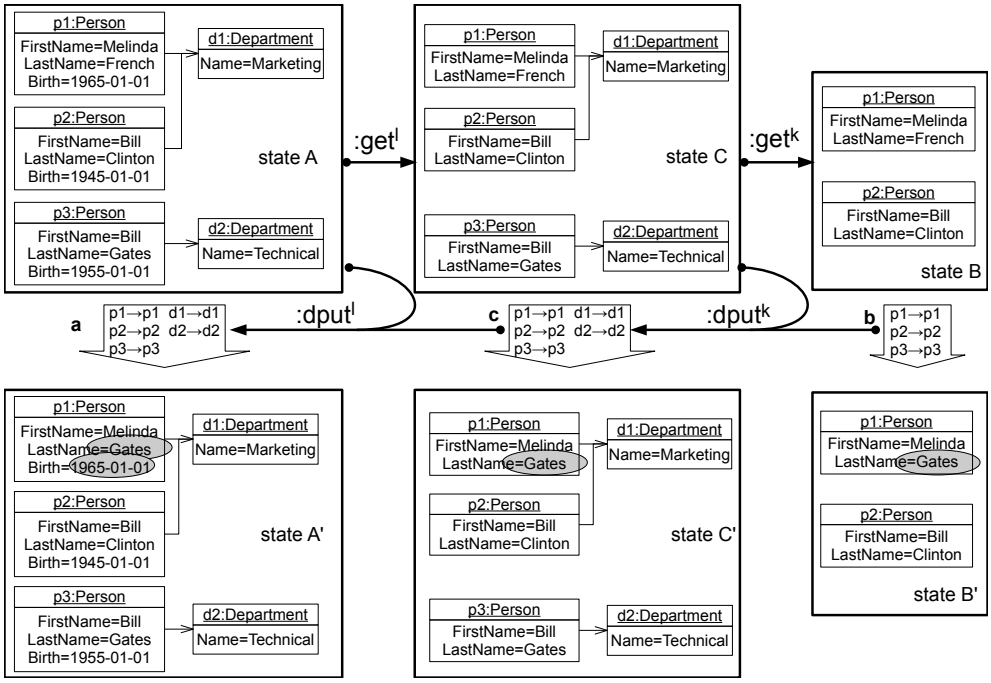


Figure 6 – Fixing sequential BX composition via deltas

Consider our running example. Suppose that in a view B (Fig. 5) the user deletes Melinda French and comes to state B'. The ***put***-function deletes Melinda in the source as well and results in state A'. If later the user inserts back exactly the same person into the view (state B''), the ***put***-function will insert this new person in the source and set attribute Birth to Unknown (state A'' in the figure); indeed, the birthdate of Melinda was lost in state A' and cannot be recovered in A''. However, since the states B and B'' are equal, GetPut-law prescribes the states A and A'' be also equal. Hence, $\boldsymbol{put}(B'', A) \neq \boldsymbol{put}(B', A)$ and PutPut-law fails for a quite reasonable transformation. Yet removing PutPut from the list of laws is also not a good solution because it frees BXs from any obligations to respect somehow update composition.

3 BXs based on delta propagation

3.1 Fixing the problems by deltas

Figure 6 demonstrates how the problem of state-based BX sequential composition can be fixed with the delta-propagation framework. In more detail, we require that deltas occur into both the input and the output of the backward propagation procedure ***dput***. Then two BXs are composed by passing the delta produced by the first BX to the input of the second. Delta discovery is only required for the very first transformation in the chain.

In a similar way, making deltas explicit allows us to fix violation of PutPut-law in the example of Fig. 5. We present operation ***put*** as sequential composition of two operations, $\boldsymbol{dif}_Y$ and ***dput***, where $\boldsymbol{dif}_Y$ computes deltas (with a fixed alignment strategy Y), and ***dput*** propagates them as we discussed above.

Let $b = \boldsymbol{dif}_Y(B, B')$, $b' = \boldsymbol{dif}_Y(B', B'')$ be two deltas shown in Fig. 7. Formally,

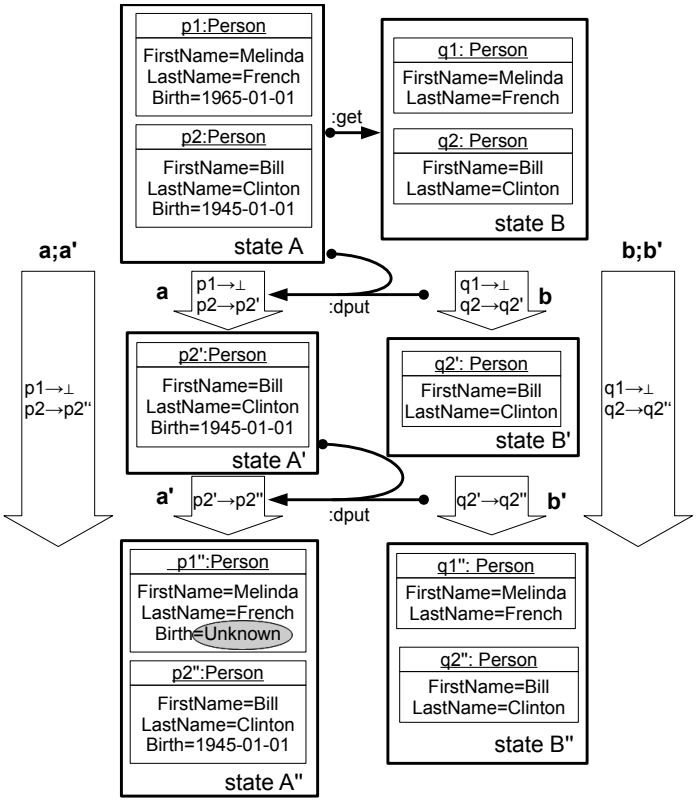


Figure 7 – Fixing PutPut via deltas

by specifying deltas by mappings, we have $b = \{q_1 \bullet\!\!\longrightarrow \bot, q_2 \bullet\!\!\longrightarrow q_2'\}$ and $b' = \{q_2' \bullet\!\!\longrightarrow q_2''\}$. Now we consider the PutPut requirements separately for $\boldsymbol{dif}_Y$ and $\boldsymbol{dput}$. For delta propagation $\boldsymbol{dput}$, PutPut-law should require preservation of delta composition:

$$\boldsymbol{dput}(b; b', A) = \boldsymbol{dput}(b, A); \boldsymbol{dput}(b', A') \tag{3}$$

that is, if we first propagate b and then propagate b', we should get the same result as if we propagate the composed delta $b; b'$. And indeed, this constraint holds in the example: if $a = \boldsymbol{dput}(b, A) = \{p_1 \bullet\!\!\longrightarrow \bot, p_2 \bullet\!\!\longrightarrow p_2'\}$ and $a' = \boldsymbol{dput}(b', A') = \{p_2' \bullet\!\!\longrightarrow p_2''\}$, then sequential composition $a; a' = \{p_2 \bullet\!\!\longrightarrow p_2''\}$ is exactly $\boldsymbol{dput}(b; b', A)$, where $b; b' = \{q_2 \bullet\!\!\longrightarrow q_2''\}$. Note that $b; b'$ is different from the identity update $\{q_1 \bullet\!\!\longrightarrow q_1'', q_2 \bullet\!\!\longrightarrow q_2''\}$ (indeed, information that objects q_1 and q_1'' are the same is beyond updates b and b'). Hence, update $a; a'$ is also not identity.

For differencing operation $\boldsymbol{dif}_Y$, PutPut would require

$$\boldsymbol{dif}_Y(B, B'') = \boldsymbol{dif}_Y(B, B'); \boldsymbol{dif}_Y(B', B''). \tag{4}$$

However, this is an evidently misleading requirement. For two identical models, B and B'', it is quite reasonable that $\boldsymbol{dif}_Y$ returns the identity update distinct from $b; b'$. Thus, although PutPut-law holds for delta propagation $\boldsymbol{dput}$, PutPut fails for the state-based propagation $\boldsymbol{put}$ because it fails for differencing $\boldsymbol{dif}_Y$.

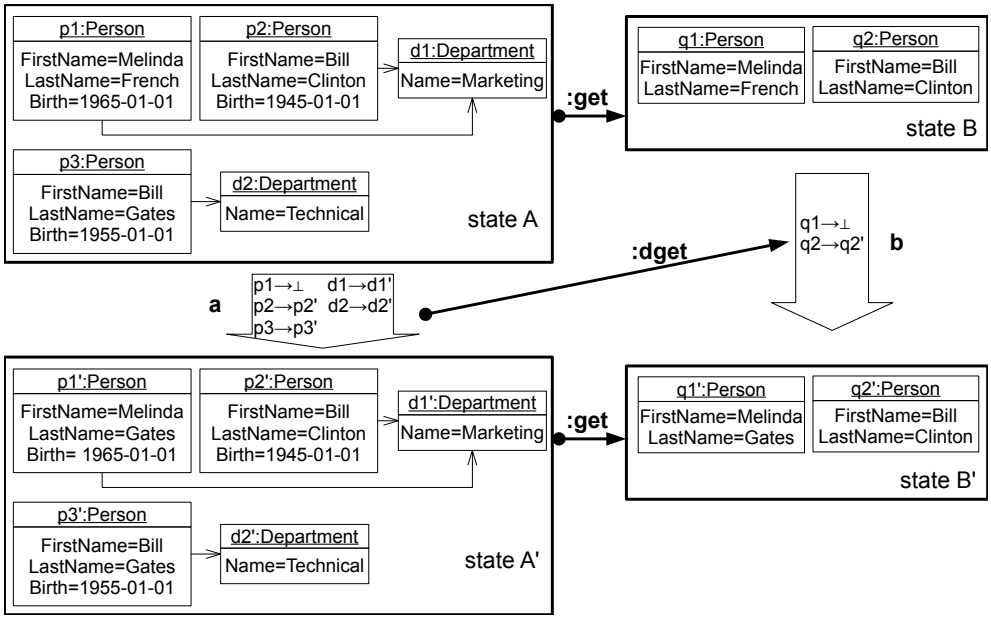


Figure 8 – Forward update propagation

3.2 Forward delta propagation

The PutGet law (see Fig. 1b) is crucial for specifying semantics of state-based lenses: it states that backward update propagation must be consistent with the view definition. In the delta-based framework, we must formulate a similar law for operation ***dput*** and require that if delta $a\colon A \to A'$ is the result of backward propagation of delta $b\colon B \to B'$, $a = \boldsymbol{dput}(b, A)$, then $\boldsymbol{dget}(a) = b$, where ***dget*** denotes operation of forward delta propagation, and is often derivable from the view definition.

Figure 8 illustrates how to derive ***dget*** in our example. Note that each object q_i in the view can be traced back to the corresponding object $p_i = \boldsymbol{back}(q_i)$ in the source ($i = 1, 2$), and we define $b(q_i) = q_i'$ iff $\boldsymbol{back}(q_i') = a(p_i)$. This is a general idea: when we execute a (forward) transformation program (the view definition), normally we not only get a view but also a set of traceability links between objects in the view and those in the source [Jou05, XLH$^+$07], which allow us to propagate deltas and derive function ***dget***. The mechanism works well for so called *monotonic* views (see [Dis09] for details), and it is known from the relational database theory that views defined by Select-Project-Join queries are monotonic. This class of queries covers many practically important cases, and we thus have a simple forward propagation mechanism for a wide class of transformations on relational-like data. However, there are situations in which ***dget*** cannot be derived from the view definition. For example, this is often the case for complex views on graph-based hierarchical data; nevertheless, a reasonable operation ***dget*** can be still specified separately *in addition* to the view definition [AMR$^+$98].

3.3 Implementation of deltas

So far we have considered deltas as partially defined mappings between models. Given such a mapping $a\colon A \to A'$, those elements of A for which a is undefined are considered to be deleted, elements of A' beyond the range of a are those newly created, and

if an A-element is preserved but one or more of its attributes change their values, the element is modified. Thus, the mapping representation allows us to restore the changes unambiguously, but it is very uneconomic. Indeed, normally updates only involve small parts of models, and hence update mappings would be mostly identities with much of information duplicated. We will consider two practically implementable representation of deltas.

Overriding. A reasonable idea is to specify directly what is changed in model A. Such changes can be often seen as a new small model Δ_A so that the updated model is $A' = A \overrightarrow{\cup} \Delta_A$, where $\overrightarrow{\cup}$ denotes a non-commutative *overriding union (o-union)* operation. It works as follows: fresh elements from Δ_A are added to A, whereas elements from $A \cap \Delta_A$ are overridden by Δ_A; if an element from A is to be deleted, it occurs in Δ_A and is labeled with a special "killing" value $\perp$ (or `Null`). Thus, a pair $a = (A, \Delta_A)$ uniquely determines the updated model A' and we may write $a\colon A \to A'$; we will call this representation of deltas *overriding*. It is easy to see that the mapping and the overriding representations of deltas are mutually convertible.

Overriding representation is well-known in the database literature on the incremental view maintenance [GM95]. A typical instance of overriding in the model transformation area is the approach used in Beanbag [XHZ+09]. Models are converted into *dictionaries*. For example, model B in Fig. 2 can be represented as the following dictionary.

```
{p1 -> {fName -> Melinda,
        lName -> French},
 p2 -> {fName -> Bill,
        lName -> Clinton}}
```

The outermost dictionary maps the IDs of the objects to the objects themselves, which are also represented as dictionaries. A dictionary of an object maps the property names to the property values.

Updates are also encoded by dictionaries. For example, update ($u1$) renaming Melinda French into Melinda Gates is represented by the following dictionary.

```
{p1 -> {lName -> French}}
```

Update $u2$ that deletes Melinda French and inserts Melinda Gates is given by dictionary:

```
{p1 -> Null,
 p1'-> {fName -> Melinda,
        lName -> Gates}}
```

Thus, a delta is a pair of dictionaries $b = (B, \Delta_B)$ representing the source model and the update resp. To obtain the updated model $B' = B \overrightarrow{\cup} \Delta_B$, we simply override the corresponding mappings in B with those in Δ_B. Importantly, so defined o-union is associative.

Overriding delta representations can be sequentially composed. Having $a\colon A \to A'$ with $A' = A \overrightarrow{\cup} \Delta_A$, and $a'\colon A' \to A''$ with $A'' = A' \overrightarrow{\cup} \Delta_{A'}$, we define $a; a' = (A, \Delta_A \overrightarrow{\cup} \Delta_{A'})$. Since

$$A \overrightarrow{\cup} (\Delta_A \overrightarrow{\cup} \Delta_{A'}) = (A \overrightarrow{\cup} \Delta_A) \overrightarrow{\cup} \Delta_{A'} = A' \overrightarrow{\cup} \Delta_{A'} = A'',$$

delta $a; a'$ is indeed a delta from A to A'' as required. This composition is associative (since o-union is associative), and the empty delta $\varnothing$ is the neutral unit:

$$(A, \varnothing); a = a = a; (A', \varnothing)$$

because $\varnothing \overrightarrow{\cup} \Delta_A = \Delta_A = \Delta_A \overrightarrow{\cup} \varnothing$.

Forward propagation of overriding deltas takes the following form. A source delta $a\colon A \to A'$ with $A' = A \overrightarrow{\cup} \Delta_A$ is propagated into view delta $b = \boldsymbol{dget}(a)\colon B \to B'$, where $B = \boldsymbol{get}(A)$ and $B' = B \overrightarrow{\cup} \Delta_B$ with Δ_B being the result of propagating Δ_A to the view side. We want this propagation to be compatible with the view definition (be *forward correct*), and so require $B' = \boldsymbol{get}(A)$.

In many practically interesting cases, we can treat Δ_B and Δ_A as models and directly use ***get*** to produce Δ_B from Δ_A: $\Delta_B = \boldsymbol{get}(\Delta_A)$. Then forward correctness amounts to distributivity of view computation wrt. o-union: $\boldsymbol{get}(A \overrightarrow{\cup} \Delta_A) = \boldsymbol{get}(A) \overrightarrow{\cup} \boldsymbol{get}(\Delta_A)$, and operation ***dget*** is derived from the view definition. In more complex cases of non-distributive views, we may still assume that a correct delta propagation operation is defined separately and augments the view definition; a discussion and references can be found in [ESRM06]. Backward delta propagation is defined separately from the view definition anyway.

Operational representation. Another typical way to implement deltas is to present them operationally as sequences of edit operations [CRP07], and some differencing tools work in this mode [AP03, AAAN+08].

A typical set of operations include:

- *create*(o) Create an object with ID o.
- *delete*(o) Delete an object with ID o.
- *change*(o, p, v) Change property p of object o to value v.

Accordingly, a delta is a pair $a = (A, s)$ where A is a model and s is a sequence of operation instances to be applied to A. An operation instance is an operation from the above list, whose formal parameters are bound by elements from A; the result is an updated model $A' = s(A)$ and we may write $s\colon A \to A'$.

Operationally represented deltas can be sequentially composed by concatenating their edit sequences. Having $a = (A, s)$ with $A' = s(A)$, and $a' = (A', s')$ with $A'' = s'(A')$, we define $a; a' = (A, s'')$ where $s'' = s; s'$. It is easy to see that $s''(A) = (s; s')(A) = s'(s(A)) = s'(A') = A''$, so that we indeed have sequential composition. It is associative with the empty sequence e being the unit.

An advantage of the operational representation is that we can easily add domain-specific operations. For example, if the model is a sequence of objects, we may add an operation *move* as an abbreviation of a sequence of property change operations. Operations like ***dget*** and ***dput*** could treat *move* in a special way to improve performance. On the other hand, operational representation always require an independent implementation of delta propagation, since we cannot directly apply ***get*** to a sequence of edit operations.

4 Delta lenses: Delta-based BX, algebraically

In this section we specify delta-based constructions described above in an abstract formal way. In Section 4.1 we define model spaces as categories, thus summarizing

our preceding discussion of delta implementation. Then we define BX based on delta propagation (Section 4.2) and their sequential composition (Section 4.3). In Section 4.4 we show that state-based lenses can be indeed presented as delta lenses equipped with a model differencing operation. Section 4.4 is a brief discussion of the delta lens formalism.

4.1 Abstract deltas: Models spaces as categories

In our examples, we have considered concrete representations of models and deltas. Now we abstract away their internal structure, and make them indivisible nodes and arrows resp. Our discussion of deltas in Section 3.3 shows that irrespectively of representation, deltas have the following properties.

1. A delta has a source model and a target model.
2. There may be multiple deltas between two models.
3. There is a delta between any two models conforming to the same metamodel.
4. Deltas can be composed sequentially: given deltas $a\colon A \to A'$ and $a'\colon A' \to A''$, their *composition*, delta $(a;a')\colon A \to A''$, is uniquely defined.
5. For any model A, there is a special *identity* delta $\mathsf{id}_A\colon A \to A$ specifying an idle (empty) update of model A resulting in the identity alignment of A to itself.

Hence, a model universe appears as a graph (points 1,2) with composable arrows (4), which is connected (3) and reflexive (5) — see Background box below for precise definitions (where we write "a set X of *widgets*" instead of "a set X of abstract elements called *widgets*"). Moreover, discussion in Section 3.2 suggests to require composition to be associative with identity loops being its neutral units. All these requirements can be concisely summarized as follows.

Definition 3. A *model space* is a connected category $\mathbf{A}$, whose nodes are called *models* and arrows are *(model) deltas*. (See the Background box for the definition of category.) Sometimes we will call deltas *updates*.

Background: Graphs. A *graph* $\mathbf{G}$ consists of a set of *nodes* $\mathbf{G}_0$ and a set of *arrows* $\mathbf{G}_1$ together with two functions $\partial_{\mathsf{s}}\colon \mathbf{G}_1 \to \mathbf{G}_0$ and $\partial_{\mathsf{t}}\colon \mathbf{G}_1 \to \mathbf{G}_0$. For an arrow $a{\in}\mathbf{G}_1$, we write $a\colon A \to A'$ if $\partial_{\mathsf{s}}a = A$ and $\partial_{\mathsf{t}}a = A'$ and call nodes A the *source* and A' the *target* of a. A graph is *connected*, if for any pair of nodes A, A' there is at least one arrow $a\colon A \to A'$.

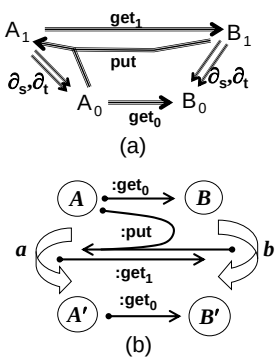
A *reflexive* graph is additionally equipped with operation $\mathsf{id}\colon \mathbf{G}_0 \to \mathbf{G}_1 2$ that assigns to each node A a special arrow $\mathsf{id}_A\colon A \to A$ called *identity*.

Background: Categories. A *category* is a reflexive graph with well-behaved composition of arrows. In detail, a category is a pair $\mathbf{C}{=}(|\mathbf{C}|,;)$ with $|\mathbf{C}|$ a reflexive graph and ; a binary operation of arrow composition, which assigns to arrows $a\colon A \to A'$ and $a''\colon A' \to A''$ an arrow $a;a'\colon A \to A''$ such that the following two laws hold: $a;(a';a'') = (a;a');a''$ for any triple of composable arrows (Associativity), and $\mathsf{id}_A;a = a = a;\mathsf{id}_{A'}$ (Neutrality) of identity wrt. composition.

We write $A{\in}\mathbf{C}$ for a node $A{\in}|\mathbf{C}|_0$, and $a{\in}\mathbf{C}$ for an arrow $a{\in}|\mathbf{C}|_1$.

4.2 Delta lenses

In the state-based framework, model spaces are sets and a view is a function between these sets. In the delta-based framework, model spaces are graphs, and a view definition



Equational laws		
(GetInc)	$\partial_{\mathsf{x}}(a.\mathsf{get}_1) = (\partial_{\mathsf{x}}a).\mathsf{get}_0$, $\mathsf{x} = \mathsf{s}, \mathsf{t}$	(1)
(PutInc_1)	$\mathsf{put}(b, A)$ is defined iff $A.\mathsf{get}_0 = \partial_{\mathsf{s}} b$	
(PutInc_2)	$\partial_{\mathsf{s}}\mathsf{put}(b, A) = A$	
(GetId)	$\mathsf{id}_A.\mathsf{get}_1 = \mathsf{id}_B$ with $B \stackrel{\text{def}}{=} A.\mathsf{get}_0$	(2)
(PutId)	$\mathsf{id}_A = \mathsf{put}(\mathsf{id}_B, A)$ with $B \stackrel{\text{def}}{=} A.\mathsf{get}_0$	
(PutGet)	$(\mathsf{put}(b, A)).\mathsf{get}_1 = b$	(3)
(GetGet)	$(a; a').\mathsf{get}_1 = (a.\mathsf{get}_1); (a'.\mathsf{get}_1)$	(4)
(PutPut)	$\mathsf{put}(b; b', A) = \mathsf{put}(b, A); \mathsf{put}(b', A')$ with $A' \stackrel{\text{def}}{=} \partial_{\mathsf{t}}\mathsf{put}(b, A)$	

Figure 9 – Delta lens operations and laws. Diagram (a) specifies arities of functions involved, and (b) presents general application instances. Laws (1),(2) and (4) require preservation of incidence relationships, identity deltas and delta composition resp.; law (3) is *backward correctness* of update propagation.

provides two components: a function on nodes $\boldsymbol{get}\colon \mathbf{A}_0 \to \mathbf{B}_0$ computing views of source models, and a function on arrows $\boldsymbol{dget}\colon \mathbf{A}_1 \to \mathbf{B}_1$ translating deltas in the source space (in whatever representation) to deltas in the view space (in the same representation; cf. our discussion in Section 3.3).

Evidently, the pair ($\boldsymbol{get}$,$\boldsymbol{dget}$) should preserve the incidence between models and updates (as prescribed by GetInc-law in Fig. 9), and hence constitute a graph morphism $\mathsf{get}\colon \mathbf{A} \to \mathbf{B}$ (see the Background box) with $\mathsf{get}_0 = \boldsymbol{get}$ and $\mathsf{get}_1 = \boldsymbol{dget}$. Note that for overriding deltas, this requirement amounts to forward correctness discussed in Section 3.3.

Another reasonable requirement is that identity deltas (idle updates) on the source are mapped to identity deltas on the view as prescribed by (GetId-law) in Fig. 9, group (2). In other words, forward update translation get is required to be a semi-functor.

Background: Graph morphisms. Let $\mathbf{A}$ and $\mathbf{B}$ be graphs. A *graph morphism* $\boldsymbol{g}\colon \mathbf{A} \to \mathbf{B}$ is a pair of functions $\boldsymbol{g}_i\colon \mathbf{A}_i \to \mathbf{B}_i$, ($i$=0,1) that preserves the incidence relations between nodes and arrows: $\partial_{\mathsf{x}} f_1(a) = f_0(\partial_{\mathsf{x}} a)$, $\mathsf{x} = \mathsf{s}, \mathsf{t}$.

Background: Functors. Let $\mathbf{A}$ and $\mathbf{B}$ be categories. A *semi-functor* $\boldsymbol{f}\colon \mathbf{A} \to \mathbf{B}$ is a graph morphism $\boldsymbol{f}\colon |\mathbf{A}| \to |\mathbf{B}|$ that preserves identities: $f_1(\mathsf{id}_A) = \mathsf{id}_{f_0(A)}$.

A semi-functor is called a *functor* if composition is also preserved: $\boldsymbol{f}_1(a; a') = \boldsymbol{f}_1(a); \boldsymbol{f}_1(a')$.

Backward update translation is given by a function $\mathsf{put}\colon \mathbf{B}_1 \times \mathbf{A}_0 \to \mathbf{A}_1$, which takes a delta in the view space and produces a delta in the source. Similarly to ordinary lenses, it also takes the initial state of the source model as its second argument to recover information missing in the view. Examples of put can be found in Section 3 (where it was denoted by $\boldsymbol{dput}$).

The backward translation must satisfy the following three technical laws ensuring that the formal model is adequate to the intuition.

Law (PutInc_1). Applying put to a view update $b\colon B \to B'$ and a source A, we assume that B is the view of A, i.e., $A.\mathsf{get}_0 = B$ as prescribed by PutInc_1-law in Fig. 9. Thus, though function put is partially defined, this partiality is technical and ensures right incidence between deltas and models: this gives us the "only if" half of the law PutInc_1. On the other hand, we require that for any pair (b, A) that satisfies

the required incidence, the backward translation is defined, which gives us the "if" half of the law. In this sense, PutInc_1 is analogous to Totality requirement in the lens framework [FGM$^+$07].

Law (PutInc_2) says that the result of $\mathsf{put}(b, A)$ is to be an update of model A.

Law (PutId) requires that identity deltas (idle updates) in the view space are translated into identity deltas (idle updates) in the source.

Five laws introduced above (groups (1) and (2) in Fig. 9) state the most basic properties of get and put operations, which ensure that the formal model is adequate to its intended meaning. The other three laws specified in Fig. 9 provide more advanced properties of update propagation, which further constrain the transformational behavior.

The PutGet-law ensures the correctness of backward propagation. It is similar to the corresponding state-based law in Fig. 1, but is defined on deltas rather than states. The incidence laws allow us to deduce PutGet for states from PutGet for deltas. Note that we do not require GetPut-law on deltas because some information contained in delta $a\colon A \to A'$ is missing from its view $\mathsf{get}_1(a)$ and cannot be recovered from a's source A. As for the state-based law GetPut, its actual meaning is identity preservation given by ours PutId-law.

Finally, GetGet and PutPut laws state compatibility of update propagation with update composition.

Definition 4 (delta lens)**.** A *delta lens* is a tuple $l = (\mathbf{A}, \mathbf{B}, \mathsf{get}, \mathsf{put})$, in which $\mathbf{A}$ and $\mathbf{B}$ are model spaces (i.e., connected categories) called the *source* and the *target* of the lens, $\mathsf{get}\colon \mathbf{A} \to \mathbf{B}$ is a graph morphism providing $\mathbf{B}$-views of $\mathbf{A}$-models and their deltas, and $\mathsf{put}\colon \mathbf{B}_1 \times \mathbf{A}_0 \to \mathbf{A}_1$ is a function translating view deltas back to the source so that laws PutInc_1 and PutInc_2 in Fig. 9 are respected.

A delta lens is called *well-behaved* (we will write wb) if it also satisfies GetId, PutId and PutGet laws. Particularly, GetId means that get is a semi-functor.

A wb delta lens is called *very well-behaved* if it satisfies GetGet and PutPut laws. Particularly, GetGet makes get a functor.

We will write $l\colon \mathbf{A} \rightleftarrows \mathbf{B}$ for a delta lens with source $\mathbf{A}$ and target $\mathbf{B}$, and denote the functions by get^l and put^l. We will often write *d-lens* for *delta lens*.

Remark. The terminology adopted in the definition above is chosen in parallel with state-based lenses. A slight difference is that the latter are simply pairs of functions without additional conditions while delta lens' get and put without incidence laws do not make sense. Therefore we have included them into the very notion of delta lens.

4.3 Sequential composition of delta lenses

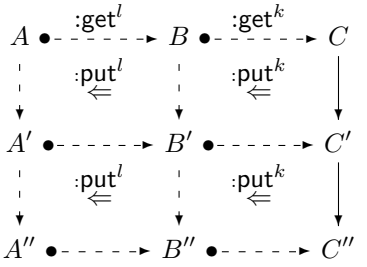
As we discussed in Section 2.2, sequential composition of lenses is crucial for their practical applications. In the present section we will define sequential composition of d-lenses and prove that composition of two (very) wb d-lenses is also a (very) wb d-lens.

Background: Functor composition. Given two semi-functors between categories, $\boldsymbol{f}\colon \mathbf{A} \to \mathbf{B}$ and $\boldsymbol{g}\colon \mathbf{B} \to \mathbf{C}$, their composition $\boldsymbol{f;g}\colon \mathbf{A} \to \mathbf{C}$ is defined componentwise via function composition: $(\boldsymbol{f;g})_i = \boldsymbol{f}_i; \boldsymbol{g}_i$, $i = 0, 1$. Evidently, $\boldsymbol{f;g}$ is a semi-functor again. Moreover, if $\boldsymbol{f}$ and $\boldsymbol{g}$ are functors, their composition is also a functor (the proof is straightforward).

Definition 5. Let $l\colon \mathbf{A} \rightleftarrows \mathbf{B}$ and $k\colon \mathbf{B} \rightleftarrows \mathbf{C}$ be two d-lenses. Their *sequential composition* is a d-lens $(l;k)\colon \mathbf{A} \rightleftarrows \mathbf{C}$ defined as follows. Forward propagation of $(l;k)$ is sequential composition of semi-functors, $\mathsf{get}^{(l;k)} \stackrel{\text{def}}{=} \mathsf{get}^l;\mathsf{get}^k$

Backward propagation is defined as follows. Let $c\colon C \to C'$ be an update in space $\mathbf{C}$, and $A \in \mathbf{A}$ a model such that $A.\mathsf{get}_0^{(l;k)} = C$, that is, $B.\mathsf{get}_0^k = C$ with B denoting $A.\mathsf{get}_0^l$. Then $\mathsf{put}^{(l;k)}(c, A) = \mathsf{put}^l(b, A)$ with $b = \mathsf{put}^k(c, B)$.

Theorem 2. Sequential composition of two (very) wb d-lenses is also a (very) wb d-lens as soon as the components are such.



Proof. The get-part of the theorem is evident: sequential composition of semi-functors (functors) is a semi-functor (functor). The put-part needs just an accurate unraveling of Definition 5 and straightforward checking of the laws. The inset diagram explains it all (dashed vertical arrows denote deltas computed by put-operations, and horizontal arrows show view computation). □

Another important result is that sequential composition of d-lenses is associative. It directly follows from associativity of function composition. Also, for any space $\mathbf{A}$, there is the identity d-lens $\mathsf{id}_{\mathbf{A}}\colon \mathbf{A} \rightleftarrows \mathbf{A}$ whose get and put functions are identities and hence $\mathsf{id}_{\mathbf{A}}$ is neutral wrt. lens composition. These two facts can be concisely stated as follows.

Theorem 3. Collection of model spaces as nodes and d-lenses as arrows is a category. We denote it by ***DLens***.

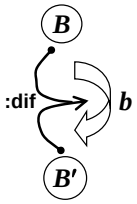
In addition, Theorem 2 together with the fact that any identity d-lens is trivially very wb imply the following.

Corollary 4. Collections of spaces as nodes and (very) wb d-lenses as arrows form categories $\boldsymbol{DLens}_{\mathsf{wb}}$ ($\boldsymbol{DLens}_{\mathsf{vwb}}$), respectively. Thus, we have inclusions $\boldsymbol{DLens}_{\mathsf{vwb}} \subset \boldsymbol{DLens}_{\mathsf{wb}} \subset \boldsymbol{DLens}$ of categories of d-lenses.

4.4 S-lenses = D-lenses + Differencing

If an alignment strategy Y is fixed, model differencing $\boldsymbol{dif}_Y$ becomes a binary operation that takes two models (from the same model space) and returns their delta.

Definition 6. *Differencing* over a model space $\mathbf{B}$ is a binary operation $\boldsymbol{dif}\colon \mathbf{B}_0 \times \mathbf{B}_0 \to \mathbf{B}_1$ satisfying the incidence law DifInc in Fig. 10. It is called *well-behaved (wb)* if the DifId-law holds.



(DifInc) $\partial_{\mathsf{s}}\boldsymbol{dif}(B, B') = B,\ \partial_{\mathsf{t}}\boldsymbol{dif}(B, B') = B'$

(DifId) $\boldsymbol{dif}(B, B) = \mathsf{id}_B$

Figure 10 – Differencing operation and its laws

Definition 7 (dd-lenses)**.** A *d-lens with differencing (dd-lens)* is a pair $\ell = (l, \boldsymbol{dif})$ with $l{=}(\mathsf{get}, \mathsf{put})$: $\mathbf{A} \rightleftarrows \mathbf{B}$ an d-lens and $\boldsymbol{dif}$ a differencing operation over $\mathbf{B}$. A dd-lens is called *well-behaved (wb)* if both d-lens l and operation $\boldsymbol{dif}$ are wb. A dd-lens is *very wb* if, an addition, l is very wb.

Theorem 5. Any well-behaved dd-lens ℓ gives rise to a well-behaved s-lens ℓ_0.

Proof. Given a dd-lens $\ell{=}(\mathsf{get}, \mathsf{put}, \boldsymbol{dif})$: $\mathbf{A} \rightleftarrows \mathbf{B}$, we define an s-lens ℓ_0: $\mathbf{A}_0 \rightleftarrows \mathbf{B}_0$ as follows. For any models $A \in \mathbf{A}_0$ and $B' \in \mathbf{B}_0$, we set $A.\mathsf{get}^{\ell_0} \stackrel{\text{def}}{=} A.\mathsf{get}^{\ell}_0$ and $\mathsf{put}^{\ell_0}(B', A) \stackrel{\text{def}}{=} \partial_{\mathsf{t}}\mathsf{put}^{\ell}(\boldsymbol{dif}_{\ell}(B, B'), A)$ where $B = A.\mathsf{get}^{\ell}_0$. It is easy to check that s-lens laws in Fig. 1b (page 4) follow from the differencing laws in Fig. 10 and d-lens laws in Fig. 9. In more detail, given the incidence laws for differencing and d-lenses, laws DifId and d-lens PutId imply s-lens GetPut, and d-lens PutGet ensures s-lens PutGet. □

The theorem shows that we do not loose expressiveness with unweaving propagation and differencing: an s-lens behavior can be recovered from a respectively restricted behavior of the component d-lens and $\boldsymbol{dif}$. Note also that s-lens GetPut is ensured by PutId, and hence is an identity propagation law rather than an invertibility requirement similar to PutGet.

Theorem 6. Any very well-behaved dd-lens satisfies the following conditional version of the state-based PutPut.

Let ***Difdif*** and ***Putput*** denote the following ternary predicates over model states: $\boldsymbol{Difdif}(B, B', B'')$ iff $\boldsymbol{dif}(B, B'); \boldsymbol{dif}(B', B'') = \boldsymbol{dif}(B, B'')$, and $\boldsymbol{Putput}(A, B', B'')$ iff s-lens PutPut holds for (A, B', B'') (see Fig. 1). Then for all $A{\in}\mathbf{A}$, $B', B''{\in}\mathbf{B}$ the following holds :

($\mathsf{PutPut}_{\mathrm{cond}}$) if $\boldsymbol{Difdif}(A.\mathsf{get}_0, B', B'')$ then $\boldsymbol{Putput}(A, B', B'')$

Proof. Given $\boldsymbol{Difdif}(B, B', B'')$ with $B = A.\mathsf{get}$, d-lens PutPut together with incidence laws provide $\boldsymbol{Putput}(A, B', B'')$. □

Note that like s-lens PutPut, law $\mathsf{PutPut}_{\mathrm{cond}}$ is also formulated for states but is weaker: it requires ***Putput*** not for all triples $(A, B'B'')$ but only for those satisfying the $\boldsymbol{Difdif}^{-}$premise. Hence, a BX that fails to satisfy the ordinary PutPut, may still satisfy the conditional $\mathsf{PutPut}_{\mathrm{cond}}$ (like in our example in section 3.1).

Unfortunately, in some situations, PutPut may fail even for delta propagation. To see that, we modify example considered in Fig. 5 in the following way. Suppose that the view also shows Persons' birth years. When putting the updated year back, function ***put*** uses the month and the day of the month in the original source to restore the whole birthdate. Now, if a Person's birthdate is 1984-02-29, changing the year to a non-leap one, say, 1985, in state B' will be propagated back to the date 1985-?-? in state A'. Changing the year in the view back to 1984 in state B'' gives us the identity update $b''{=}b; b'{=}\mathsf{id}_B : B \to B$ (as $B = B''$) to be propagated back into identity $a''{=}\mathsf{id}_A : A \to A$ (as $A = A''$). However, update $a' : A' \to A''$ cannot restore the lost date 02-29 and hence $a; a' \neq a''$. We leave this problem for future work.

4.5 Towards richer delta frameworks

We will outline several possible extensions of the delta lens framework, which we consider to be potentially useful theoretically or practically.

The abstract categorical foundations of delta lenses make them a really flexible algebraic model: abstract deltas (arrows between models) can be interpreted as

overriding deltas, or as edit logs, or as mappings between models. In Section 3.3 we outlined these interpretations in a semi-formal way; with precise formal definitions of overriding and edit-log deltas, these interpretations could be specified formally and proven to be formal refinements of the abstract delta lens theory. New important laws specific for a particular implementation may emerge in this way (for example, distributivity of view computation wrt. overriding deltas). We may also consider richer deltas that allow merging and splitting objects; their mapping representations then become binary relations rather than partially defined injections considered in the paper. Delta lenses may be also applicable to specifying synchronization of structured strings with reordering allowed. Appendix presents a first step in this direction; more could be done by rearranging the formalism of matching lenses in delta terms.

The delta framework can be extended with traceability mappings between the view and the source. Then propagation operations (forward and backward) take an (update) delta and a traceability mapping as their input, and produce a delta on the other side and an updated traceability mapping. The four operands are modeled by arrows and form a square (called a tile); composition of propagation operations then amounts to *tiling* (i.e., fitting tiles together by their sides) in the vertical and horizontal dimensions. This setting gives rise to a general tile algebra framework for model synchronization described and discussed in [Dis09]. Particularly, tiling of the asymmetric BX amounts to so called double categories [Dis11], which compactly encode complex compositional properties. On the other hand, applying the delta-centered approach to symmetric BX gives rise to an essentially different, and much richer, tile-based world [DXC+11].

Finally, we have also shown that model spaces as nodes and delta lenses as arrows themselves form a category. Hence, important constructions (combinators) on delta-lenses (e.g., parallel composition) should be readily defined by using standard categorical means. A rich set of combinators would support design of a programming language for writing delta-based asymmetric BX.

5 Related Work

A well-known idea is to use incremental model transformations operating with updates to speed up synchronization, e.g., [GW09]. However, semantics of these approaches is state-based and updates are only considered as an auxiliary technological means. Ráth et al. [RVV09] propose change-driven model transformations that map updates to updates, but they only concern uni-directional transformations, consider updates only operationally, and do not provide a formal framework.

Some researchers motivated by practical needs have realized the limitation of the state-based approach and introduced update modeling constructs into their frameworks. Xiong et al. enrich a state-based framework with updates to deal with situations where both models are modified at the same time [XLH+07, XSHT11]. Foster et al. augment basic lenses with a key-based mechanism to manage alignment of sets of strings, coming to the notion of *dictionary* lens [BFP+08]. However, even in these richer (and more complex) frameworks updates are still second-class entities. Transformations still produce only states of models, and the transformation composition problem persists. This problem is attacked in the framework of *matching* lenses [BCF+10], whose relation to delta lenses is discussed in Appendix. Note also that matching lenses assume that the chunk structure is fixed and the view function get cannot add, delete, or reorder chunks; hence the construct is not directly applicable to the area of model transformations, in which filtering objects is common for view creation.

On the theoretical side, a different delta-based algebraic model of asymmetric BX is proposed in [Dis09], in which traceability mappings explicitly occur in the inputs and outputs of operations; the paper also includes view (transformation) definitions into the framework, which are modeled as mappings between metamodels. Close to the present paper is work by Johnson and Rosebrugh, who consider the view update problem in the database context and employ category theory (see [JR08] and reference therein). For them, updates are also arrows, a model space is a category and a view is a functor. However, in contrast to the lens-based frameworks including ours, in which models are points without internal structure, for Johnson and Rosebrugh models are functors from a category considered as a database schema to the category of sets and functions. This setting is too detailed for our goals, complicates the machinery and makes it heavily categorical. Also, while we are interested in the laws of composition (of transformations and within a single transformation via PutPut), they focus on conditions ensuring existence of a unique update policy for a given view. Further, they do not consider relations between the update-based and state-based frameworks, which are our main concern in the paper.

6 Conclusion

The paper identifies several problems of state-based BXs caused by weaving alignment and update propagation: poor modularity, incorrect sequential composition of transformations, and ill-formed PutPut-law regulating interaction of update propagation with update composition. It is shown that these problems can be managed if update propagation procedures are decomposed into delta discovery (model diff) followed by a pure delta propagation. Importantly, the latter inputs and outputs not only models but also deltas between them. The corresponding algebraic framework of delta lenses is developed, in which model spaces are categories, and propagation procedures are mappings between them compatible with the categorical structure (arrow composition and identity arrows). Delta lenses form a proper extension of the ordinary lens formalism: we have shown that behavior of state-based lenses can be recovered within the delta lens framework. Nevertheless, delta lenses enjoy a semantically transparent, and technically manageable, algebraic theory. Perhaps most importantly, delta lenses support a modular architecture of BX tools, which can be adapted to different user's needs.

References

[AAAN+08] Marwan Abi-Antoun, Jonathan Aldrich, Nagi H. Nahas, Bradley R. Schmerl, and David Garlan. Differencing and merging of architectural views. *Autom. Softw. Eng.*, 15(1):35–74, 2008. doi:10.1007/s10515-007-0023-3.

[ACS09] Michal Antkiewicz, Krzysztof Czarnecki, and Matthew Stephan. Engineering of framework-specific modeling languages. *IEEE Transactions on Software Engineering*, 99(RapidPosts):795–824, 2009. doi:10.1109/TSE.2009.30.

[AMR+98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J.Wiener. Incremental maintenance for materialized views over semistructured data.

In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB*. Morgan Kaufmann, 1998.

[AP03] Marcus Alanen and Ivan Porres. Difference and union of models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *?UML? 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin / Heidelberg, 2003. doi:10.1007/978-3-540-45221-8_2.

[BCF+10] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *ICFP'10: Proceeding of the 15th ACM SIGPLAN international conference on Functional programming*, pages 193–204, 2010. doi:10.1145/1863543.1863572.

[BFP+08] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 407–419, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328487.

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007. doi:10.5381/jot.2007.6.9.a9.

[Dis09] Zinovy Diskin. Model synchronization: Mappings, tiles, and categories. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 6491 of *Lecture Notes in Computer Science*, pages 92–165. Springer, 2009. doi:10.1007/978-3-642-18023-1_3.

[Dis11] Z. Diskin. From lenses to tiles: Model synchronization via double categories. Technical Report GSDLab-TR 2011-06-10, University of Waterloo, 2011. http://gsd.uwaterloo.ca/node/351.

[DXC+11] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From state- to delta-based bidirectional model transformations: the symmetric case. Technical Report GSDLab-TR 2011-05-03, University of Waterloo, 2011. http://gsd.uwaterloo.ca/node/338.

[ESRM06] Maged El-Sayed, Elke A. Rundensteiner, and Murali Mani. Incremental maintenance of materialized XQuery views. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *ICDE*, page 129. IEEE Computer Society, 2006. doi:10.1109/ICDE.2006.80.

[FGK+07] J. N. Foster, M. Greenwald, C. Kirkegaard, B. Pierce, and A. Schmitt. Exploiting schemas in data synchronization. *J. Comput. Syst. Sci.*, 73(4):669–689, 2007. doi:10.1016/j.jcss.2006.10.024.

[FGM+07] J. N. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007. doi:10.1145/1232420.1232424.

[FPP08] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 383–396, New York, NY, USA, 2008. ACM. doi:10.1145/1411204.1411257.

[GM95] A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[GW09] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8:21–43, 2009. 10.1007/s10270-008-0089-9. doi:10.1007/s10270-008-0089-9.

[Jou05] Frederic Jouault. Loosely coupled traceability for ATL. In *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29–37, 2005.

[JR08] M. Johnson and R. Rosebrugh. Constant complements, reversibility and universal view updates. In José Meseguer and Grigore Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 238–252. Springer, 2008. doi:10.1007/978-3-540-79980-1_19.

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models. *European J. of Information Systems*, 16:349–361, 2007.

[MHN+07] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In Ralf Hinze and Norman Ramsey, editors, *ICFP*, pages 47–58. ACM, 2007. doi:10.1145/1291151.1291162.

[RVV09] István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2009. doi:10.1007/978-3-642-04425-0_26.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 295–304, New York, NY, USA, 2007. ACM. doi:10.1145/1287624.1287665.

[XHZ+09] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 315–324, New York, NY, USA, 2009. ACM. doi:10.1145/1595696.1595757.

[XLH+07] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 164–173, New York, NY, USA, 2007. ACM. doi:10.1145/1321631.1321657.

[XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 54–65, 2005. doi:10.1145/1101908.1101919.

[XSHT11] Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. Synchronizing concurrent model updates based on bidirectional transformation. *Software and Systems Modeling*, pages 1–16, 2011. 10.1007/s10270-010-0187-3. doi:10.1007/s10270-010-0187-3.

A Appendix. Delta lenses vs. reordering

Several lens formalisms aim to specify synchronization of structured strings [BFP+08, FPP08, BCF+10]. In many cases, a structured string can be considered as an (ordered) sequence of *chunks* (substrings with a certain structure). Chunks can be added, deleted or reordered. The ***get*** functions are supposed to work only inside the chunks, and do not delete, add or reorder chunks.

To deal with this kind of data, Barbosa et al. developed a construct of *matching* lens [BCF+10] aiming at separating alignment from update propagation. The construct is fairly complex: a matching lens consists of several functions subjected to a mixed set of state- and delta-based laws, which have (quoting the authors) "a somewhat low-level and operational feeling" [BCF+10, p.5]. In this section, we will show that the notion of matching lens can be readily simulated (and clarified) within the delta lens framework.

We start by defining the domain of structured strings in our terms. Assuming the space of chunks is defined as a model space (connected category) $\mathbf{A}$, we build a new model space/category $\mathbf{A}^*$ of model sequences as follows.

Definition 8 (matching deltas)**.** (i) $\mathbf{A}^*$-objects are sequences $A = (A_1...A_n)$ of $\mathbf{A}$-objects. We define $|A|$ to be the n-element set $\{1..n\}$ of numbers lesser than n. In [BCF+10], elements of $|A|$ are called *locations* and objects A_i *chunks*.

(ii) A $\mathbf{A}^*$-arrow $a\colon A \to A'$ is a pair $(|a|, a^{\S})$ with $|a|\colon |A'| \to |A|$ a partial function mapping locations to locations, and $a^{\S} = (a_i)_{i\in \mathsf{dom}|a|}$ a sequence of $\mathbf{A}$-arrows $a_i\colon A_{|a|(i)} \to A'_i$ as shown by the upper square in Fig. 11. The intuition is that updating a sequence of models (chunks) consists of two parts. The first one (given by function $|a|$) is reordering of chunks with, perhaps, adding new chunks at locations for which $|a|$ is not defined. The second part ($a^{\S}$) is a sequence of deltas between the chunks matched by $|a|$. We will call $\mathbf{A}^*$-arrows *matched deltas*.

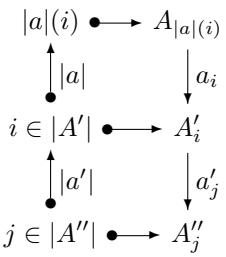


Figure 11 – Matched deltas and their composition

If the model space is a *thin* connected category (see Background box below), then the second component of a delta is trivial, and a matching delta is actually given by function $|a|$ matching locations. This is the situation considered in [BCF+10].

Background: Thin categories. A category $\mathbf{A}$ is called *thin* if there is at most one arrow $a\colon A \to A'$ for any given pair of nodes (A, A'). For example, a partially ordered set $(\mathbf{A}, \preceq)$ can be seen as a thin category, whose arrows are pairs (A, A') for which $A \preceq A'$.

For uniformity, and to simplify technicalities below, it is convenient to make function $|a|$ total and set $|a|(i) = 0$ when it is undefined. Correspondingly, for any sequence A we assume $|A| = \{0, 1, .., n\}$ and set $A_0 = \Omega$, where Ω denotes the minimally possible (initial) model in space $\mathbf{A}$ (for example, one may think of the empty model or the model with the only Root object); Background provides a precise definition. Hence, if A'_i is a new chunk (see the diagram above), then a_i is a trivial delta $\omega_{A'_i} : \Omega \to A'_i$. Thus, a matched delta $a : A \to A'$ appears as a tuple $(|a|, a_1, .., a_{|A'|})$ in which some components may be ω-arrows.

Background: Initial objects. Given a category $\mathbf{A}$, an object $\Omega \in \mathbf{A}_0$ is called *initial* if for any object $A \in \mathbf{A}_0$ there is one and only one arrow $\omega_A : \Omega \to A$

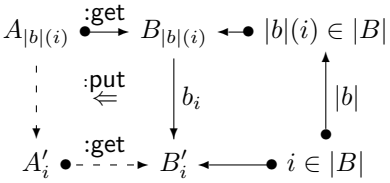
Theorem 7. Given a model space $\mathbf{A}$, the universe $\mathbf{A}^*$ of model sequences and matched deltas is a category.

Proof. Composition of matching deltas $a : A \to A'$ and $a' : A' \to A''$ is defined as shown by Fig. 11 with $i = |a'|(j))$, that is, $|a; a'| \stackrel{\text{def}}{=} |a|; |a'|$ and $(a; a')_j \stackrel{\text{def}}{=} (a_{|a'|(j)}); (a'_j)$. It is associative because functional composition (of $|a|$'s) and delta composition in $\mathbf{A}$ are such. Given an object $A \in \mathbf{A}^*$, the identity matching delta id_A is defined by setting $|\mathsf{id}_A| = \mathsf{id}_{|A|}$ and $(\mathsf{id}_A)_i = \mathsf{id}_{A_i}$. It is easy to see that it is indeed the identity wrt. composition.

Definition 9 (matching delta lens)**.** Let $l = (\mathsf{get}, \mathsf{put}) : \mathbf{A} \rightleftarrows \mathbf{B}$ be an delta lens. It gives rise to a *matching* lens $l^* = (\mathsf{get}^*, \mathsf{put}^*) : \mathbf{A}^* \rightleftarrows \mathbf{B}^*$ in the following way.

If $A = (A_i)_{i \in |A|}$ is an $\mathbf{A}^*$-object, then $|A.\mathsf{get}^*| \stackrel{\text{def}}{=} |A|$ and $(A.\mathsf{get}^*)_i \stackrel{\text{def}}{=} A_i.\mathsf{get}$ for each $i \in |A|$, which gives us a $\mathbf{B}^*$-object of length $|A|$.

If $a : A \to A'$ is an $\mathbf{A}^*$-arrow, then $a.\mathsf{get}^* : A.\mathsf{get}^* \to A'.\mathsf{get}^*$ is the following $\mathbf{B}^*$-arrow: $|a.\mathsf{get}^*| \stackrel{\text{def}}{=} |a| : |A'| \to |A|$ and for each $i \in |A'|$, $(a.\mathsf{get}^*)_i \stackrel{\text{def}}{=} a_i.\mathsf{get}$. It is easy to check that conditions of Definition 8(ii) are fulfilled, and hence $a.\mathsf{get}^*$ is a correct $\mathbf{B}^*$-arrow.



If $b : B \to B'$ is an arrow in $\mathbf{B}^*$, and A is an $\mathbf{A}^*$-object such that $A.\mathsf{get}^* = B$, then the matching delta $\mathsf{put}^*(b, A) : A \to A'$ is defined as illustrated by the inset figure, (where derived arrows are dashed), to wit: (i) $|\mathsf{put}^*(b, A)| \stackrel{\text{def}}{=} |b| : |B'| \to |B|$ (which also means that $|A'| = |B'|$); and (ii) $(\mathsf{put}^*(b, A))_i \stackrel{\text{def}}{=} \mathsf{put}(b_i, A_{|b|(i)})$ for each $i \in |B'|$. It is easy to check that all incidence conditions are respected, and so l^* is an delta lens indeed.

Note that by condition (ii), the i-th component (chunk) of the source update is computed by applying operation put^* to the i-th delta of the view update and $|b|(i)$-chunk of the source, that is, those chunk of the source that matched the i-th chunk before the update. Since operations get^* and put^* are defined componentwise, the following result is straightforward.

Theorem 8. Matching lens l^* is (very) well-behaved as soon as lens l is such.

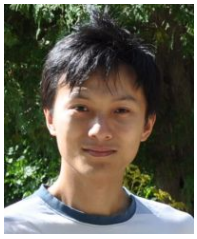
This theorem is a delta lens generalization of the Lowering lemma in [BCF+10].

In our definition of matching deltas and lenses, we have identified elements of spaces $\mathbf{A}^*$, $\mathbf{B}^*$ with sequences of chunks. The notion of matching lens introduced in

[BCF+10] is more complicated: elements of the source and target spaces are sequences plus some "framing" structure containing placeholders for chunks. They call this additional structure of "model" A the *skeleton* of A. Let's call a skeleton *trivial*, if it coincides with a bare list of locations. It is easy to see that if model spaces are thin categories, then our notion of matching delta lens coincides with matching lenses of [BCF+10] with trivial skeletons.

About the authors

Zinovy Diskin is Research Associate cross-appointed in the Department of Computing & Software at McMaster University and the Department of Electrical & Computer Engineering at the University of Waterloo. His area of expertise is formal semantics and algebraic models for constructs and concepts used in MDE, particularly metamodeling, multimodleing and model management. Contact him at `zdiskin@gsd.uwaterloo.ca`



Yingfei Xiong is a postdoctoral fellow in the Department of Electrical and Computer Engineering at the University of Waterloo. His research interests include bidirectional transformation, model synchronization, and inconsistency fixing. Contact him at `yingfei@gsd.uwaterloo.ca`

Krzysztof Czarnecki is Associate Professor in the Department of Electrical and Computer Engineering at the University of Waterloo and NSERC/Bank of Nova Scotia Industrial Research Chair in Requirements Engineering of Service-oriented Software Systems. His research interests include domain-specific modeling, variability modeling, and model synchronization. Contact him at `kczarnec@gsd.uwaterloo.ca`

Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki. *From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case.* In Journal of Object Technology, vol. 10, 2011, pages 6:1–25.
`doi:10.5381/jot.2011.10.1.a6`