# From State- to Delta-Based Bidirectional Model Transformations

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki

Generative Software Development Lab,
University of Waterloo, Canada
{zdiskin,yingfei,kczarnec}@gsd.uwaterloo.ca

**Abstract.** Existing bidirectional model transformation languages are mainly state-based: a transformation is considered composed from functions whose inputs and outputs only consist of original and updated models, but alignment relationships between the models are not specified. In the paper we identify and discuss three major problems caused by this under-specification. We then propose a novel formal framework based on a graphical language: models are nodes and updates are arrows, and show how the three problems can be fixed.

## 1 Introduction

A bidirectional transformation (*BX*) synchronizes two models by propagating updates between them. To date, there exist a large number of bidirectional transformation systems [1,2,3,4,5,6] synchronizing different kinds of models. Despite their diversity, BX systems are based on few underlying principles and enjoy several simple mathematical models [1,7,8] formulated within similar formal frameworks. The main idea is to consider BX composed from p*ropagating* functions ppg that take the updated model $B'$ on one side and, if necessary, the original model $A$ on the other side, and produce the updated model $A' = \mathsf{ppg}(B', A)$.

A fundamental feature of these frameworks is that they are *state-* rather than *update-based*. That is, propagation procedures take states of models before and after updates as input and ignore how updates were actually done. Freedom from operational details allows loose coupling between synchronizers and applications and is technologically beneficial [9]. However, it requires a mechanism for model alignment, i.e., relating objects in the before- and after-states of the updated model. For example, QVT [6], Boomerang [2] and FSML [10] use external keys: chosen sets of attributes such that objects having the same key values are considered to be the same.

We may model alignment by a binary operation dif that takes two models and produces a delta (an update) between them. Then a general schema of update propagation (roughly) takes the form

$$(\mathsf{DifPpg}) \qquad \mathsf{ppg} \overset{\text{def}}{=} \mathsf{dif} ; \mathsf{ppg}_{\Delta},$$

where ";" denotes sequential composition and $\mathsf{ppg}_{\Delta}$ is an operation that takes a delta between models and computes an updated model on the other side. However, existing state-based synchronization frameworks and tools "hide" DifPpg decomposition: the formalisms only specify the behavior of function ppg and correspondingly deltas do not occur into tools' interfaces.

Discovering updates (operation dif) and propagating updates (operation $\mathsf{ppg}_\Delta$) are two different tasks that must be treated differently and addressed separately. Mixing them in one state-based operation ppg leads to theoretical and practical problems. In this paper we will show that state-based framework have the following deficiencies: (a) inflexible interface for BX tools, (b) ill-formed transformation composition, and (c) an over-restrictive law regulating interaction of transformations with update composition.

To be concrete, we discuss these problems within the context of update propagation across views and the state-based framework of *lenses* [1], but basic ideas of our discussion can be generalized for symmetric synchronization frameworks [7,8] too.

To support the separation of dif and $\mathsf{ppg}_\Delta$ theoretically, we develop an algebraic theory of operation $\mathsf{ppg}_\Delta$, whose input and output are states of the models together with updates between them. Making updates explicit leads to a novel specification framework, in which model spaces are graphs: nodes are models and arrows are deltas. The latter can be interpreted either operationally (edit logs or directed deltas) or structurally (relations between models or symmetric deltas); the latter interpretation is central for the present paper. We will follow a common mathematical terminology and call a symmetric delta between the original and updated state of a model an *update mapping*. Thus, both models and update mappings are first-class citizens, and we call BXs specified in this framework update-based BXs, or more specifically, *update-based lenses*(*u-lenses* in short). We prove several basic results about u-lenses, and show that they present an essential generalization of ordinary lenses.

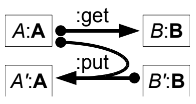
The structure of the paper is as follows. In the next section, we consider a simple example showing that different update mappings may lead to different synchronization results. We then identify and discuss three problems caused by the absence of update mappings in the lens framework. In Section 3, we introduce u-lenses and discuss their properties. In Section 4, we discuss how u-lenses relate to s-lenses and allow us to manage the three problems. Related work is discussed in Section 5.

## 2 Problems of State-Based Bidirectional Transformations

We interpret the term "model" as "a state of the model" and will use the two terms interchangeably.

### 2.1 Model Synchronization via Lenses: A Missing Link

We first remind the basic motivation and definition of lenses. Lenses are an asymmetric BX framework: in the two 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 $\mathsf{put}$, whose arities are shown in Fig. 1a. Function $\mathsf{get}$ (meaning "get the view") takes a source model $A\in\mathbf{A}$ and computes its view $B\in\mathbf{B}$. Function $\mathsf{put}$ takes an updated view model $B'\in\mathbf{B}$ and the original source $A\in\mathbf{A}$ and computes an updated source $A'\in\mathbf{A}$ ("puts the view back"). Function $\mathsf{put}$ is a special case of propagation function $\mathsf{ppg}$ (not $\mathsf{ppg}_\Delta$) discussed in Section 1.



(a) Operations

$$\begin{aligned}
&(\mathsf{GetPut})\ A = \mathsf{put}(A.\mathsf{get}, A)\\
&(\mathsf{PutGet})\ (\mathsf{put}(B', A)).\mathsf{get} = B'\\
&(\mathsf{PutPut})\ \mathsf{put}(B'', \mathsf{put}(B', A))\\
&\qquad\qquad = \mathsf{put}(B'', A)
\end{aligned}$$

(b) Equational laws

**Fig. 1.** Lens operations and laws

**Definition 1 (adapted from [1]) .** A *well-behaved (wb) lens* is a tuple $l = (\mathbf{A}, \mathbf{B}, \mathsf{get}, \mathsf{put})$ with $\mathbf{A}$ and $\mathbf{B}$ sets called the *source* and the *view space* of the lens, and $\mathsf{get}\colon \mathbf{A} \to \mathbf{B}$ and $\mathsf{put}\colon \mathbf{B}\times\mathbf{A} \to \mathbf{A}$ are functions such that the laws $\mathsf{GetPut}$ and $\mathsf{PutGet}$ in Fig. 1b hold for any $A \in \mathbf{A}$ and $B' \in \mathbf{B}$. (We write $A.\mathsf{get}$ and $\mathsf{put}(B, A)$ for the values of $\mathsf{get}$ and $\mathsf{put}$ 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 $\mathsf{PutPut}$ 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 $x \xrightarrow{:f} y$. 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 also often write a pair $(x, y)$ as $xy$.

Figure 2 shows a transformation instance in the lens framework. The source model specifies Person-objects with their first and last names and birthdates. Each person belongs to a department, which has a department name. The $\mathsf{get}$ function extracts a view containing the persons from the "Marketing" department and omits their birthdate. The $\mathsf{put}$ function reflects the updates on the view back into 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. These updates can be described structurally (rather than operationally) by specifying a binary *sameness* relation $R_\simeq$ between the states. For update (u1), $R_\simeq = \{p_1p_1', p_2p_2'\}$, and for (u2), $R_\simeq = \{p_2p_2'\}$, where $xy$ denotes pair $(x, y)$. We will call sameness relations or similar correspondence specifications *update mappings*. The latter allow us to specify updates lightly without involving full update operation logs.

A reasonable $\mathsf{put}$-function should translate (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, $A'(u1) \neq A'(u2)$, 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

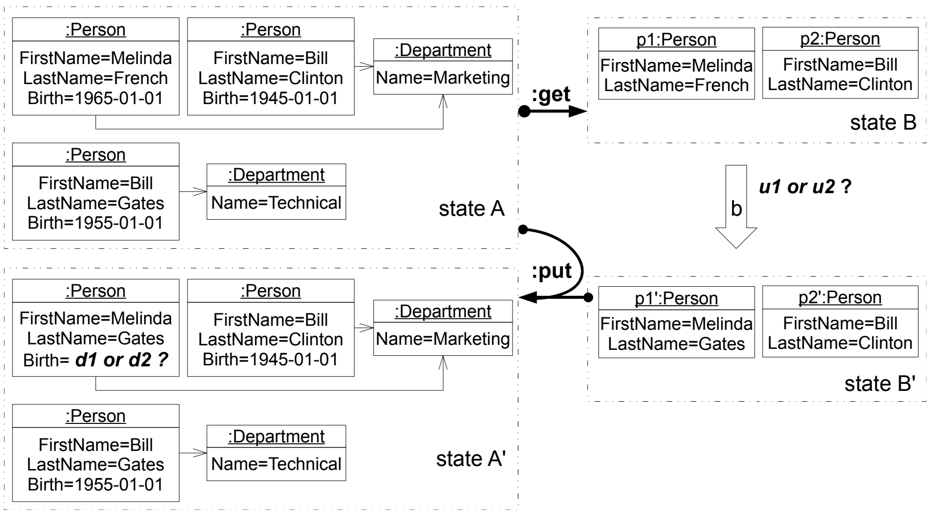


**Fig. 2.** Running Example

was a Car object with a single owner Melinda French. Then the backward translation of update (u2) must remove this Car-object as well. Thus, states $A'(u1)$ and $A'(u2)$ will have different sets of objects.

## 2.2 Inflexible Interface

At first glance, a state-based BX looks more flexible than an update-based, as a state-based BX frees users from specifying update mappings. However, since different update mappings lead to different synchronization results, we may want to control how to discover updates from two states or from an operation log. In these cases, we need the interface of update-based BX.

If we have an update-based procedure $\mathsf{ppg}_\Delta$, we can construct a state-based BX by composing it with operation $\mathsf{dif}$, where $\mathsf{dif}$ can be a manually-implemented procedure or one of existing model difference tools [11,12,13]. In contrast, given a state-based $\mathsf{ppg}$, we cannot easily build an update-based BX because it is not easy to extract $\mathsf{ppg}_\Delta$ from the whole $\mathsf{ppg}$ (unless decomposition $\mathsf{DifPpg}$is given explicitly). As a result, state-based BXs are in fact less flexible.

## 2.3 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

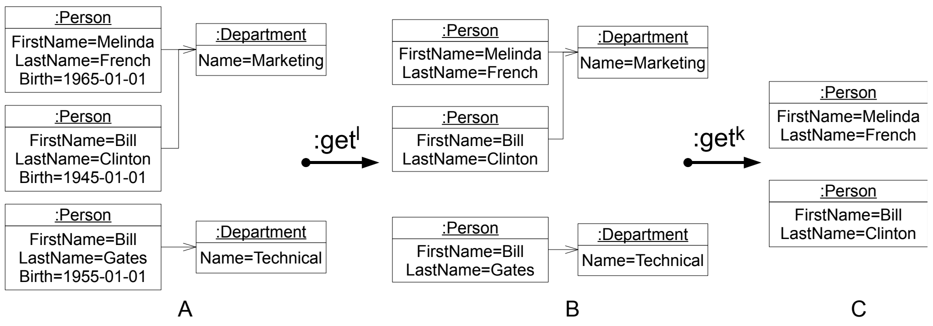


**Fig. 3.** Sequentially composed transformation

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

**Theorem 1 ([1]).** *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 in Figure 3. The first one (transformation $l$) removes the attribute Birth, and the second one (transformation $k$) extracts a list of persons from the “Marketing” department. In the backward propagation, both transformations rely on their $\mathsf{dif}$ component to recover updates from models. Suppose both $\mathsf{dif}$ procedures use key to recover update. Procedure $\mathsf{dif}^{l}$ uses the key {FirstName, LastName}, and $\mathsf{dif}^{k}$ uses a smaller key {FirstName}, which, nevertheless, works well for the Marketing Department (cf. Fig. 3C).

Ignoring updates may lead to incorrect BX composition. Suppose Melinda French has married and become Melinda Gates as shown in Figure 4C’. Transformation $k$ will successfully discover this update, and modify the last name of Melinda to Gates in model $B'$. However, when transformation $l$ compares $B$ and $B'$, $\mathsf{dif}^{k}$ will consider Melinda Gates as a new person because her last name is different. Then $\mathsf{ppg}^{k}_{\Delta}$ 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 $k$ and the update discovered by $l$ are different, and hence the two transformations should not be composed.

This example shows a fundamental requirement for sequential composition: two transformations are composable only when they agree on update mappings on the common intermediate model ($B$ in our case). However, this requirement is never specified (and is difficult to specify) in existing state-based frameworks.

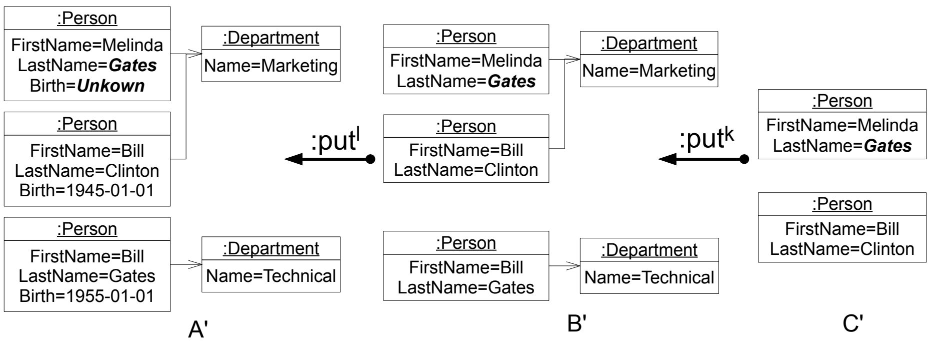


**Fig. 4.** A wrong putting back

## 2.4 PutPut: Over-Restrictive State-Based Version

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 put, but many practically interesting BXs fail to satisfy the law.

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 in the view, the put-function will insert this new person in the source and set attribute Birth to Unknown (state $A''$ in the figure). However, since the states $B$ and $B''$ are equal, PutPut-law prescribes the states $A$ and $A''$ be also equal. However, the birthdate of Melinda was lost in state $A'$ and cannot be recovered in $A''$. Hence, for a quite reasonable transformation shown in Fig. 5, PutPut fails.

To analyze the problem, we present function put as composition dif;put$_\Delta$ and consider the effect of PutPut componentwise.

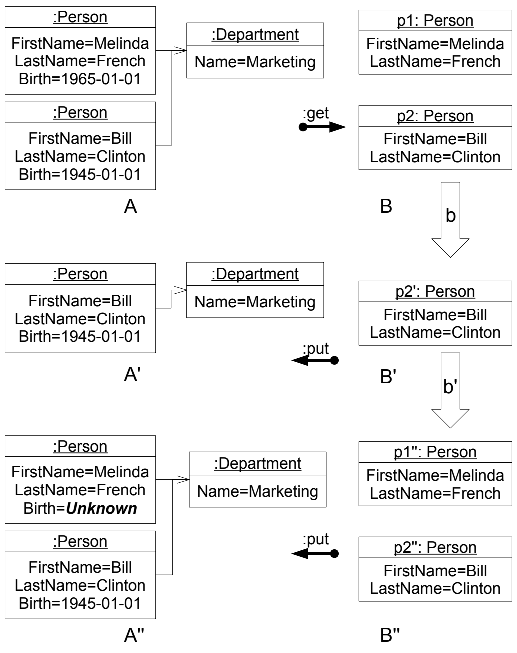


**Fig. 5.** Violation of PutPut

For pure propagation $\mathsf{put}_\Delta$, $\mathsf{PutPut}$ requires preservation of update composition: If we first propagate $b$ and then propagate $b'$, we should get the same result of propagating $b; b'$, where ";" indicates sequential composition of updates. For alignment $\mathsf{dif}$, $\mathsf{PutPut}$ requires $\mathsf{dif}(B, B'') = \mathsf{dif}(B, B'); \mathsf{dif}(B', B'')$. Although the constraint on $\mathsf{put}_\Delta$ holds well for the example, its counterpart on $\mathsf{dif}$ is unrealistic. For two identical models, $B$ and $B''$, it is quite reasonable that $\mathsf{dif}$ returns the identity update. However, update $b$ is a deletion and update $b'$ is an insertion. Their composition $b; b'$ is different from the identity update. Formally, by specifying updates with sameness relations we have $R_\simeq = \{p_2 p_2'\}$ for $b$ and $R'_\simeq = \{p_2' p_2''\}$ for $b'$. Sequential composition of $R_\simeq$ and $R'_\simeq$ as binary relations is relation $R_\simeq \otimes R'_\simeq = \{p_2 p_2''\}$ that is smaller than the identity update $\{p_1 p_1'', p_2 p_2''\}$ (and indeed, information that objects $p_1$ and $p_1'$ are the same is beyond updates $b$ and $b'$). As $b; b'$ is not identity, we cannot expect that the corresponding composition on the source $a; a'$ is identity, even though propagation $\mathsf{ppg}_\Delta$ preserves composition.

This example shows a serious defect of the $\mathsf{PutPut}$-law. In fact, it constrains both the alignment ($\mathsf{dif}$) and the propagation ($\mathsf{put}_\Delta$). Although the latter requirement is not too restrictive in practice, the former is rarely satisfied. Hence, $\mathsf{PutPut}$ often fails due to $\mathsf{dif}$ rather than $\mathsf{put}_\Delta$. To improve $\mathsf{PutPut}$, we should only constrain pure propagation $\mathsf{put}_\Delta$ to be defined in an update-based framework for BX.

## 3 U-Lenses: Update-Based Bidirectional Transformations

We build our formal framework for update-based BX in an incremental way. First we define updates abstractly as arrows between models, and thus come to the notion of a model space as a graph. Then we define an *update-based* lens (*u-lens*) as a triple of functions $(\mathsf{get}_0, \mathsf{get}_1, \mathsf{put})$ satisfying certain equational laws. Functions $\mathsf{get}_0$ and $\mathsf{get}_1$ send nodes (models) and arrows (updates) in the source space to, respectively, nodes and arrows in the view space. Function $\mathsf{put}$ formalizes the $\mathsf{ppg}_\Delta$ component of $\mathsf{put}$ in state-based lenses, mapping the arrows in the view space back to the arrows in the source space, and this mapping is parameterized by nodes (original models) in the source. To distinguish u-lenses from ordinary lenses working only with states, we call the latter *s-lenses*.

Since updates can be composed, our model spaces are graphs with composable arrows, or *categories*. Mappings between categories are normally compatible with composition and exhibit remarkable algebraic properties that have been studied in category theory. Not surprisingly, the u-lens framework is essentially categorical. Nevertheless, our presentation will mainly be arranged in set- rather than category-theoretical way to make comparison with s-lenses easier. Yet we employ simple categorical notions where their usage provides especially compact and transparent view of the formalities. Explanations of these categorical notions are displayed in boxes marked as "Background".

## 3.1 Building the Definition: Models and Updates

Representations of model updates can be classified into two main groups: *directed deltas* and *symmetric deltas* [14]. In the former, an update is basically a sequence of edit operations (add, change, delete). In the latter, an update is a specification of the common part of the before- and after-states of the model (while deleted and added parts are given by "subtracting" the common part from the before- and after-states). A typical representative of the second group is our *update mappings*, that is, triples $a = (A, R_{\simeq}, A')$ with $A$ and $A'$ being states of the model and $R_{\simeq}$ a sameness relation (the common part) between them. More details can be found in [15].

Whatever representation is used, updates have the following properties.

1. An update has a source model and a target model.
2. There may be multiple updates between two models.
3. A model can be updated to any model conforming to the same metamodel.
4. Updates can be composed sequentially. For the operational representation, if an edit sequence $a$ updates model $A$ to $A'$, and another edit sequence $a'$ updates $A'$ to $A''$, then the composed update from $A$ to $A''$ is concatenation of $a$ and $a'$. For the structural representation, composition of $a = (A, R_{\simeq}, A')$ and $a' = (A', R'_{\simeq}, A'')$ is $(A, R_{\simeq} \otimes R'_{\simeq}, A'')$ with $\otimes$ denoting relational composition. We write $a; a'$ for the composition of updates $a$ and $a'$.
5. For any model $A$, there is a "no-change" update, which we call *idle* and denote by $\mathbf{1}_A$. Operationally, an idle update is given by the empty edit sequence. Structurally, the idle update $\mathbf{1}_A$ is the identity mapping $(A, \{(e, e) : e \in A\}, A)$.

These considerations suggest to abstract updates as arrows between models, which can be composed. Hence, a model universe appears as a graph (points 1,2) with composable arrows (4), which is connected (3) and reflexive (5) — see Background below for precise definitions (where we write "a set X of *widgets*" instead of "a set $X$ of abstract elements called *widgets*"). Identity loops should be required to be neutral wrt. composition since idle updates are such. In addition, we require arrow composition in our abstract model to be associative (both concatenation of edit sequences and relational composition are associative).

**Definition 3.** A *model space* is a connected category, whose nodes are called *models* and arrows are *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 $\mathbf{1} \colon \mathbf{G}_1 \to \mathbf{G}_0$ that assigns to each node $A$ a special arrow-loop $\mathbf{1}_A \colon A \to A$ called *identity*.

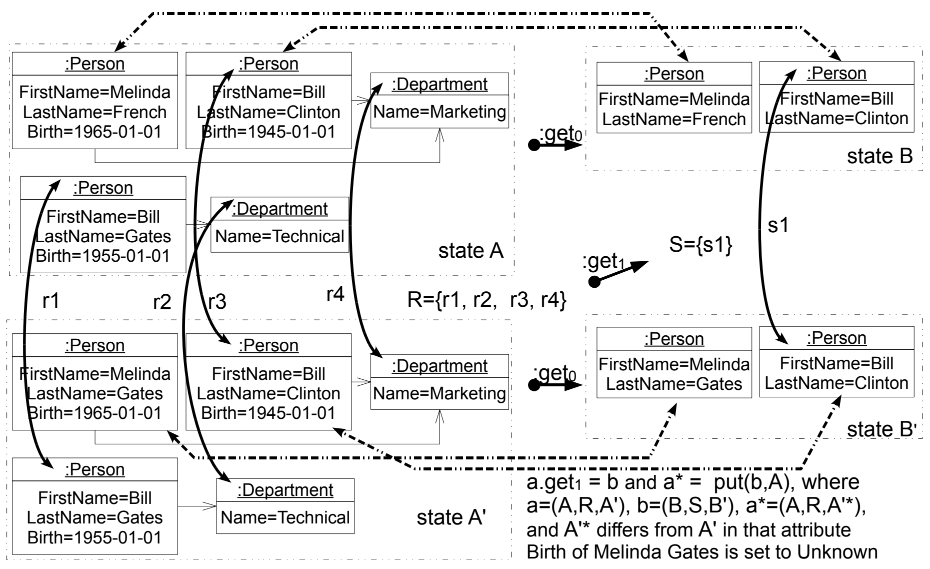


**Fig. 6.** Translation of update mappings

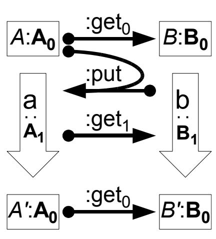
**Background: Categories.** A *category* is a reflexive graph with well-behaved composition of arrows. In detail, a category is a pair $\mathbf{C}{=}(|\mathbf{C}|, ;)$ with $|\mathbf{C}|$ a reflexive graph and ; a binary operation of arrow composition, which assigns to arrows $a\colon A \to A'$ and $a''\colon A' \to A''$ 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 $\mathbf{1}_A;a = a = a;\mathbf{1}_{A'}$ 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 $a{\in}|\mathbf{C}|_1$.

## 3.2 Building the Definition Cont'd: Views and Update Translation

**Background: Functors.** Let $\mathbf{A}$ and $\mathbf{B}$ be categories. A *semi-functor* $\mathsf{f}\colon \mathbf{A} \to \mathbf{B}$ is a pair $(\mathsf{f}_0, \mathsf{f}_1)$ of functions $\mathsf{f}_i\colon \mathbf{A}_i \to \mathbf{B}_i$, $(i{=}0{,}1)$ that preserves 1) the incidence relations between nodes and arrows, $\partial_{\mathsf{x}} f_1(a){=}f_0(\partial_{\mathsf{x}} a)$, $\mathsf{x} = \mathsf{s}, \mathsf{t}$, and 2) identities, $f_1(\mathbf{1}_A) = \mathbf{1}_{f_0(A)}$. A semi-functor is called a *functor* if 3) composition is also preserved: $\mathsf{f}_1(a;a') = \mathsf{f}_1(a);\mathsf{f}_1(a')$.

In the state-based framework, model spaces are sets and a view is a function $get\colon \mathbf{A} \to \mathbf{B}$ between these sets. In the update-based framework, model spaces are graphs and a view $\mathsf{get}$ consists of two components: a function on nodes $\mathsf{get}_0\colon \mathbf{A}_0 \to \mathbf{B}_0$ computing views of source models, and a function on arrows $\mathsf{get}_1\colon \mathbf{A}_1 \to \mathbf{B}_1$ translating updates in the source space to updates in the view space. The idea of function $\mathsf{get}_1$ is illustrated by Fig. 6, in which vertical arrows denote pairs of elements from the sameness relations, and horizontal arrows denote traceability links. Roughly, function $\mathsf{get}_1$ maps the links in the update



| Equational laws | | Their meaning |
|---|---|---|
| (GetInc) | $\partial_{\mathsf{x}}(a.\mathsf{get}_1) = (\partial_{\mathsf{x}}a).\mathsf{get}_0$, $\mathsf{x} = \mathsf{s}, \mathsf{t}$ | Preservation of incidence relations |
| (PutInc$_1$) | $\mathsf{put}(b, A)$ is defined iff $A.\mathsf{get}_0 = \partial_{\mathsf{s}}b$ | |
| (PutInc$_2$) | $\partial_{\mathsf{s}}\mathsf{put}(b, A) = A$ | |
| (GetIdl) | $\mathbf{1}_A.\mathsf{get}_1 = \mathbf{1}_B$ with $B \overset{\text{def}}{=} A.\mathsf{get}_0$ | Preservation of idle updates |
| (PutIdl) | $\mathbf{1}_A = \mathsf{put}(\mathbf{1}_B, A)$ with $B \overset{\text{def}}{=} A.\mathsf{get}_0$ | |
| (PutGet) | $(\mathsf{put}(b, A)).\mathsf{get}_1 = b$ | Correctness |
| (GetGet) | $(a; a').\mathsf{get}_1 = (a.\mathsf{get}_1); (a'.\mathsf{get}_1)$ | Preservation of composition |
| (PutPut) | $\mathsf{put}(b; b', A) = \mathsf{put}(b, A); \mathsf{put}(b', A')$ with $A' \overset{\text{def}}{=} \partial_{\mathsf{t}}\mathsf{put}(b, A)$ | |

**Fig. 7.** U-lens operations and laws

mapping $a\colon A \to A'$ accordingly to how function $\mathsf{get}_0$ maps the elements in the models $A$ and $A'$. Thus, given a view mechanism, the same view definition determines both get-functions.[1]

Evidently, the pair ($\mathsf{get}_0$,$\mathsf{get}_1$) should preserve the incidence between models and updates (as prescribed by GetInc-law in Fig. 7), and map idle updates to idle updates (GetIdl-law). In other words, it is reasonable to assume that update translation is a semi-functor.

A backward translation is given by a function $\mathsf{put}\colon \mathbf{B}_1 \times \mathbf{A}_0 \to \mathbf{A}_1$, which takes an update in the view space and produces an update in the source. Similarly to ordinary lenses, it also takes the original source model as its second argument to recover information missing in the view. An example of put is implicit in Fig. 6, where the the update $a^* = \mathsf{put}(b, A)$ is not shown to avoid clutter but specified in the right lower corner of the figure. Note that $a^* \neq a$ because birthdates are lost in the view and update mapping $S$ specifies Melinda Gates as a new object.

The backward translation must satisfy three technical laws ensuring that the formal model is adequate to the intuition : PutInc$_1$, PutInc$_2$ and PutIdl. Applying put to a view update $b\colon B \to B'$ and a source $A$, we of course assume that $B$ is the view of $A$, i.e., $A.\mathsf{get}_0 = B$ as prescribed by PutInc$_1$-law in Fig. 7. Thus, though function put is partially defined, this partiality is technical and ensures right incidence between updates and models: this gives us the "only if" half of the law PutInc$_1$. On the other hand, we require that for any update $b$ that holds the required incidence of $A$, 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 [1]. Similarly, we must require that the result of put is to be an update of model $A$, hence, the PutInc$_2$ law. In addition, it is reasonable to require that idle updates in the view space are translated into idle updates in the source. as stated by PutIdl-law.

Five laws introduced above state the most basic properties of get and put operations, which ensure that the formal model is adequate to its intended

[1] Action of a view definition on update mappings can be shown by categorical arguments with a precise formal definition of queries [16]. How it can be done for XML schemas as models and relational deltas as updates is shown in [17].

meaning. Other laws specified in Fig. 7 provide more interesting properties of update propagation, which really constrain the transformational behavior.

The PutGet-law ensures the correctness of backward propagation. It is similar to the corresponding s-lens law, but is defined on updates rather than states. The incidence laws allow us to deduce PutGet for states from PutGet for updates. We do not require GetPut-law because some information contained in update $a\colon A \to A'$ is missing from its view $\mathsf{get}_1(a)$ and cannot be recovered from $a$'s source $A$. As for the s-lens' law GetPut, its actual meaning is given by ours PutIdl-law, and the name GetPut for this law may be misleading.

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

**Definition 4 (u-lens).** An *update-based lens (u-lens)* is a tuple $l = (\mathbf{A}, \mathbf{B}, \mathsf{get}, \mathsf{put})$, in which $\mathbf{A}$ and $\mathbf{B}$ are model spaces called the *source* and *target* of the u-lens, $\mathsf{get}\colon \mathbf{A} \to \mathbf{B}$ is a semi-functor providing $\mathbf{B}$-views of $\mathbf{A}$-models and their updates, and $\mathsf{put}\colon \mathbf{B}_1 \times \mathbf{A}_0 \to \mathbf{A}_1$ is a function translating view updates back to the source so that laws $\mathsf{PutInc}_1$ and $\mathsf{PutInc}_2$ in Fig. 7 are respected.

An u-lens is called *well-behaved* (we will write wb) if it also satisfies PutIdl and PutGet laws. A wb u-lens is called *very well-behaved* if it satisfies GetGet and PutPut.

We will write $l\colon \mathbf{A} \rightleftarrows \mathbf{B}$ for a u-lens with source $\mathbf{A}$ and target $\mathbf{B}$, and denote the functions by $\mathsf{get}^l$ and $\mathsf{put}^l$. Note that for a very wb u-lens, $\mathsf{get}^l$ is a functor between categories $\mathbf{A}$ and $\mathbf{B}$.

## 3.3 Sequential Composition of U-Lenses

As we discussed in Section 2.3, sequential composition of lenses is extremely important for their practical applications. In the present section we will define sequential composition of u-lenses and prove that composition of two (very) wb u-lenses is also a (very) wb u-lens as soon as the components are such.

**Background: Functor composition.** Given two semi-functors between categories, $\mathsf{f}\colon \mathbf{A} \to \mathbf{B}$ and $\mathsf{g}\colon \mathbf{B} \to \mathbf{C}$, their composition $\mathsf{f;g}\colon \mathbf{A} \to \mathbf{C}$ is defined componentwise via function composition: $(\mathsf{f;g})_i = \mathsf{f}_i; \mathsf{g}_i$, $i = 0, 1$. Evidently, f;g is a semi-functor again. Moreover, if f and 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 u-lenses. Their *sequential composition* is an u-lens $(l;k)\colon \mathbf{A} \rightleftarrows \mathbf{C}$ defined as follows. Forward propagation of $(l;k)$ is sequential composition of semi-functors, $\mathsf{get}^{(l;k)} \overset{\text{def}}{=} \mathsf{get}^l; \mathsf{get}^k$

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 u-lenses is also a (very) wb u-lens as soon as the components are such.*

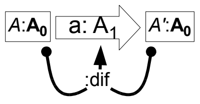
*Proof.* The get-part of the theorem is evident: sequential composition of semi-functors (functors) is a semi-functor (functor). The put-part needs just an accurate unraveling of Definition 5 and straightforward checking of the laws. □

## 4 U-Lenses with Alignment: Fixing the Problems

We will first define the notion of u-lens with alignment and show how it is related to the s-lens formalism. Then we will review the three main problems of state-based synchronization within the u-lens framework.

### 4.1 From U- to S-Lenses

**Definition 6 .** An *alignment* over a model space $\mathbf{A}$ is a binary operation $\mathsf{dif}\colon \mathbf{A}_0 \times \mathbf{A}_0 \to \mathbf{A}_1$ satisfying the incidence law DifInc in Fig. 8. Alignment is *well-behaved (wb)* if the DifId-law holds, and *very wb* if, in addition, DifDif holds too.



(DifInc) $\partial_{\mathsf{s}}\mathsf{dif}(A, A') = A,\ \partial_{\mathsf{t}}\mathsf{dif}(A, A') = A'$
(DifId) $\mathsf{dif}(A, A) = \mathbf{1}_A$
(DifDif) $\mathsf{dif}(A, A'); \mathsf{dif}(A', A'') = \mathsf{dif}(A, A'')$

**Fig. 8.** Alignment operations and laws

In most practically interesting cases, alignment is wb but not very wb; we have only introduced the latter notion for analysis of s-lens's PutPut.

**Definition 7 (ua-lenses) .** An *u-lens with alignment (ua-lens)* is a pair $\ell = (l, \mathsf{dif})$ with $l$=(get, put): $\mathbf{A} \rightleftarrows \mathbf{B}$ an u-lens and dif an alignment over $\mathbf{B}$. An ua-lens is called **(a)** *well-behaved (wb)* if its both component are such, **(b)** *very wb* if $l$ is very wb, and **(c)** *very-very wb* if alignment is also very wb.

**Theorem 3.** *Any ua-lens $\ell$ gives rise to an s-lens $\ell_0$. Moreover, if $\ell$ is (very-very) wb, then $\ell_0$ is also (very) wb.*

*Proof.* Given a ua-lens $\ell$=(get, put, dif): $\mathbf{A} \rightleftarrows \mathbf{B}$, we define an s-lens $\ell_0$: $\mathbf{A}_0 \rightleftarrows \mathbf{B}_0$ as follows. For any models $A \in \mathbf{A}_0$ and $B' \in \mathbf{B}_0$, we set $A.\mathsf{get}^{\ell_0} \overset{\text{def}}{=} A.\mathsf{get}^{\ell}_0$ and $\mathsf{put}^{\ell_0}(B', A) \overset{\text{def}}{=} \partial_{\mathsf{t}}\mathsf{put}^{\ell}(\mathsf{dif}^{\ell}(A.\mathsf{get}^{\ell}_0, B'), A)$. It is easy to check that s-lens laws in Fig. 1b follow from the alignment laws in Fig. 8 and u-lens laws in Fig. 7. In more detail, given the incidence laws for alignment and u-lenses, laws DifId and u-lens PutIdI imply s-lens GetPut, u-lens PutGet ensures s-lens PutGet, and DifDif together with u-lens PutPut provide s-lens PutPut. □

Note that to obtain a very wb s-lens, we need both very wb alignment and very wb u-lens. This is a formal explication of our arguments at the end of Section 2.4 that PutPut-law often fails due to non-very wb alignment rather than propagation procedures as such. Nevertheless, the following weakened version of PutPut holds.

**Theorem 4.** *Any very wb ua-lens satisfies the following conditional law:*

(PutPut!) if $\mathsf{difdif}(A.\mathsf{get}_0, B', B'')$ then $\mathsf{putput}(A, B', B'')$ for all $A\in\mathbf{A}$, $B', B''\in\mathbf{B}$,

*where difdif and putput denote the following ternary predicates:*
*difdif$(B, B', B'')$ holds iff dif$(B, B')$; dif$(B', B'')$ = dif$(B, B'')$, and*
*putput$(A, B', B'')$ holds iff s-lens PutPut (see Fig. 1) holds for $(A, B', B'')$.*

*Proof.* Given $\mathsf{difdif}(B, B', B'')$ with $B = A.\mathsf{get}$, the u-lens PutPut together with incidence laws provide $\mathsf{putput}(A, B', B'')$. □

Theorem 4 shows that the notion of very wb ua-lens is a reasonable formalization of BX compatible with updates, and Theorem 3 shows that this notion is stronger than wb s-lens but weaker than very wb s-lens.

### 4.2 Review of the Three Problems

U-lenses allow us to manage the problems identified in Section 2 in the following way.

First, the possibility to build an s-lens from ua-lens (Theorem 3) shows the flexibility of u-lenses. Second, when the get and put functions of two u-lenses are composed (Definition 5), they must match on update mappings (arrows) rather than only on models (nodes) thus providing continuity of composition wrt. updates. Third, PutPut-law for u-lenses and based on it PutPut!-law for ua-lenses (Theorem 4) are much more adequate to the intended behavior of BXs than state-bases PutPut. Particularly, many useful BXs that satisfy the former would violate the latter (e.g., that one presented in Fig. 5). However, we can still find useful transformations that violate u-lens PutPut.

For instance, we can 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 $b''{=}b; b'{=}\mathbf{1}_B\colon B \to B$ (as $B = B''$) to be propagated back into identity $a''{=}\mathbf{1}_A\colon A \to A$ (as $A = A''$). However, update $a'\colon A' \to A''$ cannot restore the lost date 02-29 and hence $a; a' \neq a''$. Yet removing PutPut from the list of u-lens laws is also not a good solution because it frees BXs from any obligations to respect somehow update composition. We plan to attack this problem in a future work.

## 5 Related Work

A well-known idea is to use incremental model transformations operating with updates to speed up synchronization, e.g., [18]. However, semantics of these approaches is state-based and updates are only considered as an auxiliary technological means. Ráth et al. [19] 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.

Many BX systems conform to a formal framework. Focal is based on basic s-lenses [1], QVT [6] can be modeled in a "symmetric lens" framework [7], and MOFLON [20] is built upon TGGs [5]. These frameworks are state-based and thus have the problems we described.

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 [3,21]. 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 [2]. 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 managed in the recent framework of *matching* lenses [22], where alignment is explicitly separated from transformation. However, even in the novel frameworks addressing alignment, updates are still second-class entities and the BX laws are state-based. As a result, laws for updates have to be expressed through states, making the framework very restrictive. For example, if we build a model transformation as a matching lens, the source model and the view model must have the same number of objects for each object type.

Close to ours is work by Johnson and Rosebrugh, who consider the view update problem in the database context and employ category theory (see [16] 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 three major problems of the state-based bidirectional transformation (BX) frameworks: inflexible interfaces, ill-formed composition of transformations and over-restricting PutPut-law. It is shown that these problems can be managed (though the third one only partially) if propagation procedures are decomposed into alignment followed by pure update propagation. Importantly, the latter inputs and outputs not only models but also update mappings between them. We have developed the corresponding algebraic framework for an asymmetric BXs determined by a view, and called the basic structure *u-lens*.

In the u-lens framework, updates are arrows to be thought of as update mappings. They may be considered as light structural representation of updates: updates mappings do the alignment job yet keep us away from the complexities of full update operation logs. This shows that being non-state-based does not necessarily mean being operation-based: it means being update-arrow-based with an open possibility to interpret arrows as required by applications.

**Acknowledgements.** Thanks go to Michał Antkiewicz for his contributions to the initial version of the paper, and to anonymous referees for valuable comments. Financial support was provided by the Ontario Research Fund.

## References

1. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3), 17 (2007)
2. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: Proc. 35th POPL (2008)
3. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE, pp. 164–173 (2007)
4. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ICFP, pp. 47–58 (2007)
5. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: ICGT, pp. 411–425 (2008)
6. Object Management Group: MOF query / views / transformations specification 1.0 (2008), `http://www.omg.org/docs/formal/08-04-03.pdf`
7. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and System Modeling 9(1), 7–20 (2010)
8. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 21–36. Springer, Heidelberg (2008)
9. Foster, J.N., Greenwald, M., Kirkegaard, C., Pierce, B., Schmitt, A.: Exploiting schemas in data synchronization. J. Comput. Syst. Sci. 73(4), 669–689 (2007)
10. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE Transactions on Software Engineering 99 (RapidPosts), 795–824 (2009)
11. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
12. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE, pp. 54–65 (2005)
13. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D.: Differencing and merging of architectural views. In: ASE, pp. 47–58 (2006)
14. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Software Eng. 28(5), 449–462 (2002)
15. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: Algebraic foundations and the tile notation. In: ICSE 2009 Workshop on Comparison and Versioning of Software Models, pp. 7–12 (2009), doi:10.1109/CVSM.2009.5071715

16. Johnson, M., Rosebrugh, R.: Constant complements, reversibility and universal view updates. In: Meseguer, J., Ro*C*su, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 238–252. Springer, Heidelberg (2008)
17. Liefke, H., Davidson, S.: View maintenance for hierarchical semistructured data. In: Kambayashi, Y., Mohania, M., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, p. 114. Springer, Heidelberg (2000)
18. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1), 21–43 (2009)
19. Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 342–356. Springer, Heidelberg (2009)
20. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
21. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 213–228. Springer, Heidelberg (2009)
22. Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: Alignment and view update. Technical Report MS-CIS-10-01, University of Pennsylvania (2010)