Softw Syst Model
DOI 10.1007/s10270-010-0187-3

SPECIAL SECTION PAPER

Synchronizing concurrent model updates based on bidirectional transformation

Yingfei Xiong · Hui Song · Zhenjiang Hu · Masato Takeichi

Received: 29 November 2009 / Revised: 21 October 2010 / Accepted: 15 December 2010

Abstract Model-driven software development often involves several related models. When models are updated, the updates need to be propagated across all models to make them consistent. A bidirectional model transformation keeps two models consistent by updating one model in accordance with the other. However, it does not work when the two models are modified at the same time. In this paper we first examine the requirements for synchronizing concurrent updates. We view a synchronizer for concurrent updates as a function taking the two original models and the two updated models as input, and producing two new models where the updates are synchronized. We argue that the synchronizer should satisfy three properties that we define to ensure a reasonable synchronization behavior. We then propose a new algorithm to wrap any bidirectional transformation into a synchronizer with the help of model difference approaches. We show that synchronizers produced by our algorithm are ensured to satisfy the three properties if the bidirectional transformation satisfies the correctness property and the hippocraticness property. We also show that the history ignorance property contributes to the symmetry of our algorithm. An implementation of our algorithm shows that it worked well in a practical runtime management framework.

Keywords Model synchronization · Bidirectional transformation · Concurrent updates · Model difference

Communicated by Richard Paige, Jeff Gray and Dang Van Hung.

This research was done while Y. Xiong was at the University of Tokyo.

M. Takeichi
Department of Mathematical Informatics,
The University of Tokyo, Tokyo, Japan
e-mail: takeichi@mist.i.u-tokyo.ac.jp

H. Song
Key Laboratory of High Confidence Software Technologies
(Peking University), Ministry of Education, Beijing, China
e-mail: songhui06@sei.pku.edu.cn

Z. Hu
GRACE Center, National Institute of Informatics, Tokyo, Japan
e-mail: hu@nii.ac.jp

Y. Xiong (✉)
Generative Software Development Lab, University of Waterloo,
Waterloo, Canada
e-mail: yingfei@gsd.uwaterloo.ca; xiong.yingfei@gmail.com

1 Introduction

One central activity of model-driven software development is to transform high-level models into low-level models through model transformation. For example, Fig. 1a shows a basic Unified Modeling Language (UML) model containing a `Book` class with two attributes. To implement this UML design, we can write a model transformation program to transform the model into a basic database model, as shown in Fig. 1b. Each UML class whose `persistent` feature is true is transformed into a database table of the same name. Each attribute belonging to a persistent class is transformed into a column with the same name. The database model also contains implementation-related information, the `owner` feature, and this feature is set with default value `"admin"`.

In an ideal situation, the target model is always obtained from a source model and never needs to be modified. In reality, however, developers often need to modify the target model directly. In such cases, the updates need to be reflected back to the source model.

Published online: 05 January 2011

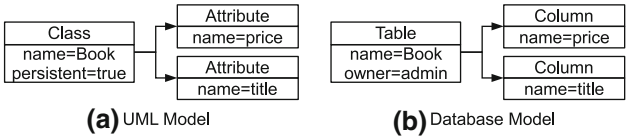


Fig. 1 Transforming a UML model into a database model

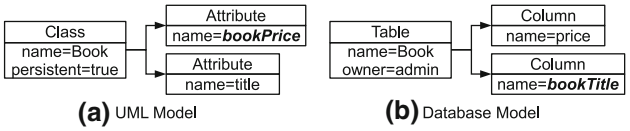


Fig. 2 Non-conflicting concurrent updates

Bidirectional model transformation [1,2] solves this maintenance problem by providing a bidirectional model transformation language, which is used to describe the relation between the two models symmetrically. Programs in these languages are used not only to transform models from one format into another, but also to update the other model automatically when a model is updated by users.

Stevens [3] formalizes a bidirectional model transformation as two functions. If M and N are meta models and $R \subseteq M \times N$ is the consistency relation to be established between them, a *bidirectional model transformation* consists of two functions:

$$\overrightarrow{R} : M \times N \to N$$
$$\overleftarrow{R} : M \times N \to M$$

Given a pair of models $(m, n) \in M \times N$, function $\overrightarrow{R}$ changes n to make it consistent with m. Similarly, $\overleftarrow{R}$ changes m in accordance with n. Many bidirectional model transformation languages (roughly) fall into this category; typical languages include Query/View/Transformation relations (QVT-R) [1] and TGGs [2].

However, in some cases, models m and n may be simultaneously updated before a bidirectional transformation can be applied. For example, an application designer could work on the UML model and change the `price` attribute into `"bookPrice"` at the same time a database designer changes the `title` column into `"bookTitle"` in the database model, as shown in Fig. 2. Applying the transformation in either direction will result in the loss of updates on the target side.

Because of the large number of available bidirectional transformation languages and existing transformation programs, it would be preferable if we could synchronize concurrent updates using existing bidirectional transformations. One basic idea is to sequentially apply the two updates and interleave them with two transformations. For the updates in Fig. 2, we can assume that the `title` column in the database model is changed first and perform a backward transformation to change the `title` attribute in the UML model. Then,

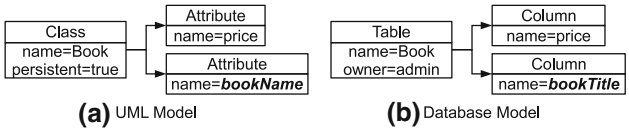


Fig. 3 Conflicting concurrent updates

we change the `price` attribute into `"bookPrice"` in the UML model and perform a forward transformation to change the `price` column in the database model.

However, there are two problems in implementing this idea. First, as with bidirectional transformation, we do not want to require users to track updates. We thus need to identify which part of the updated UML model was changed so that we can later apply the update to the result of the backward transformation. Second, the updates applied to the two models can sometimes conflict. Figure 3 shows an example of conflicting updates where the `title` attribute and the `title` column are changed to different values. If we transform backward and then go forward again, we will lose the update applied to the database model. A preferable synchronizer would detect such conflicts and warn the user.

In this paper we propose a new approach based on the idea of sequentially applying concurrent updates. We use commonly used model difference approaches [4–6] to solve the two problems mentioned above. We design an algorithm that uses model difference approaches to wrap any bidirectional transformation into a synchronizer for concurrent updates. The synchronizer takes the two original models and two updated models as input and produces two new models in which the updates are synchronized.

The main contributions of this work can be summarized as follows:

- We identify general requirements for synchronizing concurrent updates. The requirements mainly consist of three properties: consistency, stability and preservation. These properties are adapted from previous work [7] on non-symmetrical, language-specific synchronization. We significantly modify them to make them appropriate for more general and symmetrical synchronization.
- We propose an algorithm that can wrap any bidirectional model transformation and any model difference approach into a synchronizer supporting concurrent updates. It treats the bidirectional model transformation and the model difference approach as black boxes and does not require the user to write additional code. For any bidirectional transformation satisfying the correctness and hippocraticness properties [3], the synchronizer satisfies the consistency, stability, and preservation properties, ensuring correct and predictable synchronization behavior.
- We have implemented our algorithm and applied it to a runtime management application. The application

showed that our algorithm can be customized for specific cases and its performance is practically enough for mid-size models.

A previous version of the paper has been published in the Proceedings of the 2nd International Conference on Model Transformation [8]. Compared with the conference version, the present paper contains a completely new section (Sect. 6) discussing an important property of the algorithm: symmetry. We also include another bidirectional transformation property, history ignorance [9], for discussion and show that this property can lead to the symmetry of the algorithm by constructing constant complements on both sides (Appendix A). We added discussion about the research on optimistic replication and incremental bidirectional transformation in Sect. 8. Finally, many places of the conference version have been rewritten to improve readability. In particular, Sect. 7 has been completely rewritten to better address the goal and the result of the case study.

The rest of the paper is organized as follows: Section 2 describes the bidirectional model transformation properties introduced by Stevens [3]. Section 3 introduces our requirements for synchronizing concurrent updates. Section 4 describes model difference approaches in our context and introduces how we use a model difference approach to construct a three-way merger and a preservation tester, which are used in our algorithm. Section 5 introduces our algorithm and proves that bidirectional model transformation properties lead to model synchronization properties. Section 6 discusses an important property of the algorithm, symmetry, and gives a necessary condition to achieve symmetry. Section 7 describes its application and Sect. 8 discusses related work. Finally, Sect. 9 concludes the paper and discusses a possible future direction: conflict resolution.

2 Background: properties of bidirectional model transformation

The definition of bidirectional transformation describes only the input and output of a transformation; it does not constrain the behavior of the transformation. Existing work [3, 9] proposes four properties that a bidirectional transformation should satisfy to ensure that models are transformed in a reasonable way. In this paper, however, we require only that a bidirectional transformation satisfies two of them (correctness and hippocraticness) because the other two properties (undoability and history ignorance) would prohibit many practical transformations.

The first property, correctness [3], ensures that a bidirectional transformation does something useful. Given two models, m and n, the forward and backward transformations must establish a consistency relation R between them.

Property 1 (Correctness)

$$\forall m \in M, n \in N : R(m, \overrightarrow{R}(m, n))$$
$$\forall m \in M, n \in N : R(\overleftarrow{R}(m, n), n)$$

The second property, hippocraticness [3], prevents a bidirectional transformation from doing something harmful. Given two consistent models m and n, if neither model is modified, the forward and backward transformations should not modify any model.

Property 2 (Hippocraticness)

$$R(m, n) \Longrightarrow \overrightarrow{R}(m, n) = n$$
$$R(m, n) \Longrightarrow \overleftarrow{R}(m, n) = m$$

The third property, undoability [3], means that a performed transformation can be undone. Suppose there are two consistent models, m and n. A user, working on the M side, updates m to m' and performs a forward transformation to propagate the updates to the N side. Immediately after the transformation, he realizes that the update is a mistake. He modifies m' back to m and performs the forward transformation again. If the bidirectional transformation satisfies undoability, the second transformation will produce the exact n to cancel the previous modification on the N side.

Property 3 (Undoability)

$$\forall m' \in M : R(m, n) \Longrightarrow \overrightarrow{R}(m, \overrightarrow{R}(m', n)) = n$$
$$\forall n' \in N : R(m, n) \Longrightarrow \overleftarrow{R}(\overleftarrow{R}(m, n'), n) = m$$

The last property, history ignorance [9], means that the result of a transformation does not depend on whether we have executed a previous transformation or not. In other words, if a user modifies a model, transforms it to the other side, then modifies the model again and transforms again, the result should be equal to the result of performing the two modifications together and transforming just once.

Property 4 (History Ignorance)

$$\overrightarrow{R}(m, \overrightarrow{R}(m', n)) = \overrightarrow{R}(m, n)$$
$$\overleftarrow{R}(\overleftarrow{R}(m, n'), n) = \overleftarrow{R}(m, n)$$

Although undoability and history ignorance make sense in some situations, they are considered too strong and prohibit many useful transformations. One example is the UML-to-database transformation we mentioned in Sect. 1. If we change the `persistent` property of a class to `false` in the UML model, a forward transformation will delete the corresponding table in the database model. However, if we modify the property back to `true`, it is not possible for the forward transformation to recover the original table because the value of the `owner` property has been lost. In this case,

both undoability and history ignorance are violated. This problem cannot be solved from the transformation alone. To satisfy those two properties, we must change the meta model of the database to store all deleted `owner` properties, which would be impossible and unnecessary in many cases. As a result, we do not require bidirectional transformations to satisfy the two properties in this paper. However, these properties do make sense in some situations. In Sect. 6 and Appendix A we will see how history ignorance leads to the symmetry of the synchronization algorithm.

3 Requirements of synchronizing concurrent updates

As discussed above, the interface of the bidirectional transformation functions do not allow concurrent updates and we need a new interface. Suppose M and N are meta models and $R \subseteq M \times N$ is the consistency relation to be established. A *synchronizer* for concurrent updates is a partial function of the following type:

$$sync : R \times (M \times N) \to M \times N$$

This definition describes the input and output of the synchronizer. The input includes four models: the two original models satisfying consistency relation R, and the two updated models. The output is two new models for which the updates are synchronized.

This definition already implies some requirements for synchronizing concurrent updates. First, the synchronizer is a function, which means that this procedure must be deterministic. Second, the function is partial, which implies detection of conflicts in updates. If the updates to the two models conflict, the function should be undefined for the input.

However, like bidirectional transformations, this definition alone confines little the behavior of the synchronizer. We introduce three properties to ensure the synchronizer behaves in a reasonable way. These properties were first proposed in previous work [7] and are significantly modified for the synchronization of concurrent updates.

Similar to the properties of bidirectional transformation, our first property, consistency[1], requires that the synchronizer to do something useful. It ensures that consistency relation R is established on the output models.

Property 5 (Consistency)

$$sync(m, n, m', n') \text{ is defined} \implies R(sync(m, n, m', n'))$$

The second property, stability, prevents the synchronizer from doing something harmful. If neither of the two models has been updated, the synchronizer should not modify any model.

[1] This was called propagation in the previous publication [7].

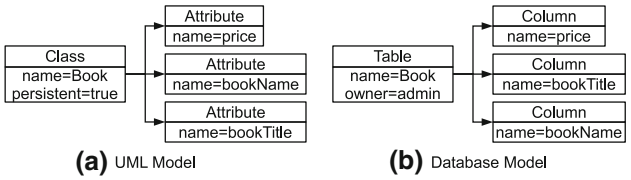


Fig. 4 Updates to both models are preserved

Property 6 (Stability)

$$R(m, n) \implies sync(m, n, m, n) = (m, n)$$

The last property, preservation, is more interesting. Consider the updates shown in Fig. 2. The easiest way to achieve consistency is to change the attribute name from `"bookPrice"` back to `"price"` and change `"bookTitle"` back to `"title"`. However, this is not the behavior we want. What we want is that the updates are propagated from the modified parts to the unmodified parts, rather than changing back the modified parts. To prevent the unwanted behavior, we require that the user updates be preserved in the output models. If the user changes the name of the price attribute to `"bookPrice"`, the synchronizer should not change the attribute to any other value.

However, as the synchronizer only deals with the original models and the updated models, it is often unclear what update is applied to the models. Given two models, there usually exist multiple updates that can change one into the other, and choosing a different update may lead to a different preservation result. For example, in Fig. 3a, we change the `name` feature of the `Attribute` object from `"title"` to `"bookName"`. However, we can also consider the update as deleting the `Attribute` element `"title"` and adding a new `Attribute` element `"bookName"`. The same dilemma applies to the database model. As a result, if we adopt the feature-changing update, the updates on the two models conflict and we cannot find a consistent model that preserves both updates. However, if we adopt the object-deleting-adding update, the updates to the two models do not conflict, and the model in Fig. 4 preserves the updates. How to choose an update from all possible updates is often application-specific and we should leave the option to users. To allow such an application-specific option, we assume that there is an update preservation relation $P_M \in M \times M \times M$ for any model M, where $P_M(m_o, m_a, m_c)$ implies that the update from m_o to m_a is preserved in m_c. Users can specify different preservation requirements by defining different preservation relations. In the next section we will see how to define a preservation relation from a model difference approach.

Given the preservation relations, we can define the preservation property. Formally, let $P_M \in M \times M \times M$ be a preservation relation over M, and $P_N \in N \times N \times N$ be a preservation relation over N.

Property 7 (Preservation)

$$sync(m, n, m', n') = (m'', n'') \Longrightarrow P_M(m, m', m'')$$
$$sync(m, n, m', n') = (m'', n'') \Longrightarrow P_N(n, n', n'')$$

The previous work [7] also introduces a fourth property: composability. However, this property has the same problem as undoability and history ignorance: it constrains the consistency relation too much and prohibits many useful transformations. Therefore, we do not require the synchronizer to satisfy this property.

4 Model difference, three-way merger and preservation

Model difference approaches [4–6,10] play an important role in our approach. First, they are used in our algorithm to identify updates and detect conflicts, as mentioned in Sect. 1. Second, they are used to define the preservation relation, as mentioned in Sect. 3. In this section we describe model difference approaches in our context.

4.1 Model difference

Following the definitions of Diskin [9], we consider the space of models in the meta model M as a directed graph; its nodes are models and its arrows are updates. We call the starting node of update δ the *pre-model* of δ (denoted as $\delta.pre$) and the end node of δ the *post-model* (denoted as $\delta.post$), where δ is considered to update $\delta.pre$ into $\delta.post$. There may be different updates leading from one model to another, so the graph is a multi-graph, meaning that there can be more than one arrow between two nodes. In addition, any model in M should be updatable to any model, so the graph is a complete graph. This definition is different from that in other work [11,12] in which updates are considered to be functions. In our definition, each update has only one associated pre-model and only one associated post-model, and cannot be directly applied to other models. We use Δ_M to denote the set of updates in the model space of M.

We consider that a model difference approach should provide at least two operations for every meta model. The first operation is used to identify the updates in two models. We call it the *difference operation*. Formally, a difference operation over M is a function, $diff \in M \times M \rightarrow \Delta_M$, that takes two models, m and m', and produces update δ, where $\delta.pre = m$ and $\delta.post = m'$. We define a difference operation as a function to require the procedure to be deterministic. A difference operation should choose one update from all possible updates using predefined criteria. For example, in Alanen et al.'s approach [4], the result is a set of insertions and deletions that preserve the longest common subsequence when comparing two ordered features.

The second operation, *the union operation*, also known as "parallel composition" in some publications [11], is used to merge different updates to be applied to the same model. This operation is useful in distributed development environments where several developers may simultaneously work on the same model, and their updates need to be merged. Given updates δ_1 and δ_2 where $\delta_1.pre = \delta_2.pre$, we denote their union as $\delta_1 + \delta_2$, where $(\delta_1 + \delta_2).pre = \delta_1.pre = \delta_2.pre$ and $(\delta_1 + \delta_2).post$ is a model that is considered to have both δ_1 and δ_2 applied. The union operation should be commutative, that is, $\delta_1 + \delta_2 = \delta_2 + \delta_1$. In addition, we do not require the union operation to be total. If δ_1 and δ_2 are in conflict, $\delta_1 + \delta_2$ is undefined. The techniques to implement this operation can be found in existing approaches [4,11].

For example, given the model in Fig. 1a and the model in Fig. 2a, a difference operation may return the update (intuitively) "change the `price` attribute in Fig. 1a to `bookPrice`". Similarly, for Figs. 1a and 3a it may return "change the `title` attribute in Fig. 1a to `bookName`". The union of the two updates may be a new update that changes both attributes in Fig. 1a.

One special case in the model difference function and the union operation is the identity update, which means nothing is changed. We require that the difference operation always returns the identity update when comparing two identical models and that computing the union of arbitrary update δ with the identity update results in δ. Formally, we require that the *diff* function and the "+" operator satisfy the following property.

Property 8 (Stability of Model Difference)
Let $id_m = diff(m, m)$, we have $\forall \delta \in \Delta_M : \delta + id_{\delta.pre} = \delta$

4.2 Three-way merger

With the model difference function and the union operator, we can construct a three-way merger of models. A *three-way merger* takes one original model and two independently updated copies of the model and produces a new model in which the updates to the two copies are merged. Three-way mergers are widely used in many distributed systems, like the Concurrent Versions System (CVS), and in the `diff3` command [13] in Unix. Given an original model m_o and two independently modified copies, m_a and m_b, a three-way merger is a partial function defined as the following.

$$merge(m_o, m_a, m_b) = (diff(m_o, m_a) + diff(m_o, m_b)).post$$

If $(diff(m_o, m_a) + diff(m_o, m_b))$ is not defined, *merge* is not defined, indicting there are conflicts between m_a and m_b.

4.3 Preservation

In Sect. 3 we have presented the semantics of update preservation as a preservation relation, but we have not shown how to define a preservation relation. To define a preservation relation, we need to decide how to choose an update from all possible updates. As model difference approaches identify an update from two models using certain criteria, we can define a preservation relation in accordance with the semantics of a model difference approach.

Definition 1 Given a difference operation *diff* and a union operator "+", we say m_c *preserves* the update from m_o to m_a if and only if there exists an update δ from m_o to a model m_b where $(diff(m_o, m_a) + \delta).post = m_c$.

One natural result is that a three-way merger will always preserve the updates in both models.

Lemma 1 *If $m_c = merge(m_o, m_a, m_b)$, then m_c preserves the update from m_o to m_a and the update from m_o to m_b.*

Proof From the definition of *merge* we get $(diff(m_o, m_a) + diff(m_o, m_b)).post = m_c$. From the commutativity of +, we get $(diff(m_o, m_b) + diff(m_o, m_a)).post = m_c$. Because there exists $diff(m_o, m_b)$, from the first formula, we have that m_c preserves the update from m_o to m_a. Similarly, from the second formula, we have that m_c preserves the update from m_o to m_b. □

This definition of preservation gives us a basic method for testing whether three models (m_o, m_a, and m_c) satisfy the preservation relation. However, to actually test it, we must iterate all possible updates starting from m_o, which is not possible in practice. What we need is an efficient procedure for quickly testing the preservation of three models. Such an efficient testing procedure is difficult to find in general. However, given a specific model difference approach, it is often possible to define an efficient testing procedure in accordance with the update operations considered in the difference approach. In the following, we show how to efficiently test preservation for Alanen et al.'s [4] model difference approach as an example.

Testing preservation in Alanen et al.'s approach

Alanen et al. consider an update as a sequence of update operations, and they define seven types of operations, as shown in Table 1. In their work, they assume that each element has a universally unique identifier (UUID) that does not change across versions. Under this assumption, we can easily identify and match model elements in different versions of objects. In addition, they consider limited types of features on the models. Features can be classified as attributes that store primitive values and references that store links to other model elements. They assume that all attributes are single features (can contain only one value) and that all references are multiple features (can contain more than one feature, either ordered or unordered).

Table 1 Modification Operations

Operation	Description
new(e, t)	Create a new element e of type t
delete(e, t)	Delete element e of type t
set(e, f, v_o, v_n)	Set an attribute f of element e from v_o to v_n
insert(e, f, e_t)	Add a link from $e.f$ to e_t for an unordered reference f
remove(e, f, e_t)	Remove a link from $e.f$ to e_t for an unordered reference f
insertAt(e, f, e_t, i)	Add a link from $e.f$ to e_t at index i for an ordered reference f
removeAt(e, f, e_t, i)	Remove a link from $e.f$ to e_t at index i for an ordered reference f

Table 2 Testing of Preservation

Operation in δ_{oa}	Preservation condition
new(e, t)	e exists in m_c, and all features of e are the same as those in m_a
delete(e, t)	e does not exist in m_c
set(e, f, v_o, v_n)	e exists in m_c, and $e.f$ is the same value as v_n
insert(e, f, e_t)	e exists in m_c, and a link to e_t exists in $e.f$
remove(e, f, e_t)	e does not exist in m_c, or a link to e_t does not exist in $e.f$
insertAt(e, f, e_t, i)	e exists in m_c, a link to e_t exists in $e.f$, and the inserted links have their order in m_a preserved in m_c for all insertAt operations on the feature
removeAt(e, f, e_t, i)	Always preserved (as deleted links can be inserted back)

To test whether an update from m_o to m_a is preserved in m_c, we first use the difference operation to get the update $\delta_{oa} = diff(m_o, m_a)$. Then we examine m_c for each update operation in δ_{oa}. If we find that an operation such that the union of any operation and this operation cannot reach m_c from m_o, we report a violation of preservation. The detailed rules for examining the update operations can be found in Table 2.

For example, suppose the `price` attribute in Fig. 1a, the `bookPrice` attribute in Fig. 2a, and the `price` attribute

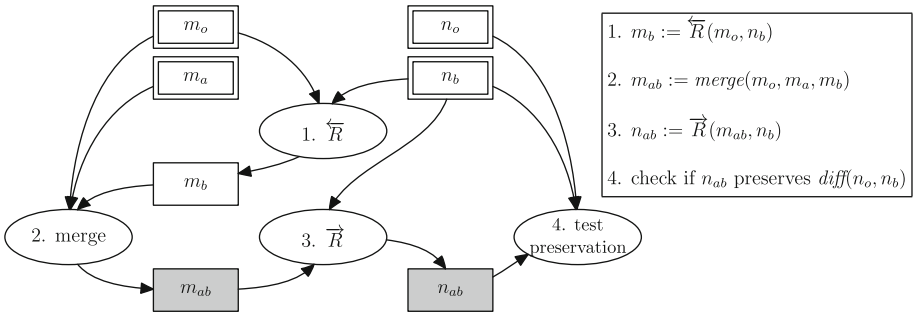


Fig. 5 Synchronization algorithm

in Fig. 3a share UUID e_p. The difference of Figs. 1a and 2a is thus an update containing one update operation: set(e_p, `name`, `"price"`, `"bookPrice"`). This update is not preserved in Fig. 3a because the rule for `set`(e, f, v_o, v_n) is violated: e_p.`name` has a value of `"price"` and is different from `"bookPrice"` in Fig. 3a.

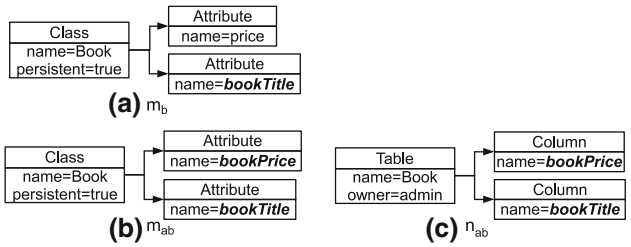


Fig. 6 Execution of algorithm

5 Algorithm

Now we have a three-way merger and can test the preservation of updates. Let us use them to wrap a bidirectional transformation into a synchronizer for concurrent updates. The basic idea is to first convert the model from the N side to the M side using backward transformation, then use the three-way merger to reconcile the updates, and transform back using the forward transformation. The detailed algorithm is shown in Fig. 5.

We explain the algorithm using the example in Sect. 1. Initially, we have the two models in Fig. 1, which correspond to m_o and n_o in our algorithm. Users modify the two models into the models in Fig. 2, which correspond to m_a and n_b in our algorithm. We use different subscripts to show different updates, where a represents the update on m_o and b represents the update on n_o. The four models together comprise the algorithm input.

The first step of our algorithm is to invoke backward transformation $\overleftarrow{R}$ to propagate the updates made to n_b to m_o, resulting in m_b[2]. The result is shown in Fig. 6a. The attribute name is changed from `"title"` to `"bookTitle"`.

Now we have model m_a containing update a and model m_b containing update b. The second step is to use the three-way merger we constructed in the last section to merge the two updates and produce a synchronized model, m_{ab}, on the M side. The result is shown in Fig. 6b. The model has both attributes changed; i.e., it contains updates from both sides. If the updates to the two models conflict, the three-way merger detects the conflict and reports an error.

The third step is to use forward transformation $\overrightarrow{R}$ to produce a synchronized model, n_{ab}, on the N side. The result is shown in Fig. 6c. This model also contains updates from both sides, with both columns changed.

Now we have two synchronized models to which the updates have propagated. It looks as if we have performed enough steps to finish the algorithm. However, the above steps do not ensure the detection of all conflicts and may lead to violation of preservation due to the heterogeneousness of the two models.

To see how this can happen, let us consider the example in Fig. 7. Initially we have only one class and one table, and they are consistent. Then suppose that a user changes the `persistent` feature of the class to `false` and changes the owner of the table to `"xiong"`. Because the `owner` feature is not related to the UML model, the backward transformation changes nothing, and m_b is the same as m_o. The three-way merger detects no updates in m_b and produces a model that is the same as m_a. Finally, we perform the forward transformation, and the table is deleted because of the change to the `persistent` feature. However, as the user has modified a feature of the table, so he or she will expect to see the existence of the table in the final result. The input models contain conflicting updates, but the synchronization process does not detect them.

This kind of violation is caused by the heterogeneity of M and N. Due to the heterogeneity, not all updates to N are visible on the M side. As the three-way merger only works on the M side, it cannot detect such invisible conflicts.

[2] We may also implement the algorithm in an opposite direction by staring with a forward transformation. More discussion about this can be found in Sect. 6. Here we randomly choose a direction to illustrate the algorithm.

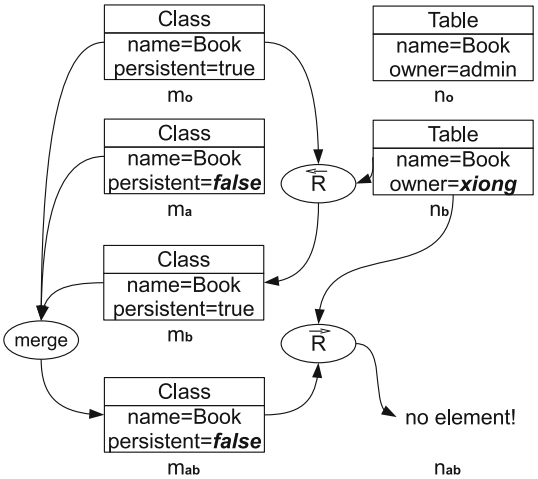


Fig. 7 Violating preservation

To capture such a conflict, we add an additional step, preservation testing, to the end of the algorithm. It is shown as the fourth step in Fig. 5. This step uses the preservation testing procedure described in Sect. 4 and checks whether the update from n_o to n_b is preserved in n_{ab}. If not, the algorithm reports an error.

To facilitate further discussion, we denote the algorithm as a high-level function SYNC that takes a bidirectional transformation $(\overrightarrow{R}, \overleftarrow{R})$ over $M \times N$, a model difference operation *diff* and a union operator $+$, and produces a synchronizer $\mathsf{SYNC}[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ to synchronize concurrent updates over M and N.

The models used in Figs. 6 and 7 are simply examples. The actual execution depends on the bidirectional transformation and the model difference approach used in the synchronization and may differ from the aforementioned execution. Nevertheless, whatever bidirectional transformation and model difference approach we choose, our algorithm ensures the three synchronization properties: consistency, stability, and preservation.

Theorem 1 *If bidirectional transformation* $(\overrightarrow{R}, \overleftarrow{R})$ *satisfies correctness, synchronizer* $\mathsf{SYNC}[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ *satisfies consistency for any model difference approach* $(diff, +)$.

Proof Consider the last two steps of the algorithm. Because $\overrightarrow{R}(m_{ab}, n_b) = n_{ab}$, we have $R(m_{ab}, n_{ab})$. □

Theorem 2 *If bidirectional transformation* $(\overrightarrow{R}, \overleftarrow{R})$ *satisfies hippocraticness and model difference approach* $(diff, +)$ *satisfies stability of model difference, synchronizer* $\mathsf{SYNC}[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ *satisfies stability.*

Proof If we have $m_o = m_a$ and $n_o = n_b$, we have $R(m_o, n_b)$. Because of hippocraticness, we have $m_b = \overleftarrow{R}(m_o, n_b) = m_o$. Because of stability of model difference, $m_{ab} = merge(m_o, m_a, m_b) = (diff(m_o, m_a) + diff(m_o, m_b)).post = (diff(m_o, m_o) + diff(m_o, m_o)).post = m_o$. On the other hand, we have $n_{ab} = \overrightarrow{R}(m_{ab}, n_b) = \overrightarrow{R}(m_o, n_o) = n_o$, and the preservation testing always passes because of the existence of the identity update. □

Theorem 3 *The synchronizer* $\mathsf{SYNC}[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ *always satisfies preservation for any bidirectional transformation* $(\overrightarrow{R}, \overleftarrow{R})$ *and any model difference approach* $(diff, +)$.

Proof Because of Lemma 1, the update on the M side is preserved. Because of the last preservation test, the update on the N side is preserved. □

It is worth noting that our algorithm works even if the bidirectional transformation does not satisfy correctness or hippocraticness. This has practical value because many existing bidirectional transformation languages do not guarantee the properties [3]. In such cases, the algorithm still produces output but does not guarantee the corresponding synchronization properties (consistency or stability).

6 Algorithm symmetry

Bidirectional transformations are symmetrical, so we can also implement this algorithm in the opposite direction. We can start a forward transformation first, merge models on the N side, perform a backward transformation, and check preservation on the M side. We denote the algorithm in the opposite direction as $\mathsf{SYNC}^{\triangleleft}$ where $\mathsf{SYNC}^{\triangleleft}[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ is a synchronizer over M and N for a bidirectional transformation $(\overrightarrow{R}, \overleftarrow{R})$ over $M \times N$, a model difference operation *diff* and a union operator $+$. Here comes one question: do SYNC and $\mathsf{SYNC}^{\triangleleft}$ always produce the same synchronizer? In other words, is the algorithm symmetric?

Unfortunately, symmetry is not ensured by our algorithm. Let us suppose that in the UML-to-database example the backward transformation $\overleftarrow{R}$ will delete the corresponding class in the UML model when a table is deleted in the database model. Consider the execution described in Fig. 8. Initially we have a class and a table. Then users change the `persistent` feature of the class to `false` and delete the table. When we first transform backwardly, as shown in Fig. 8a, the backward transformation will delete the class and there will be a conflict. However, if we start with the forward transformation, as shown in Fig. 8b, the change of `persistent` will lead to the deletion of table and there will be no conflict.

Asymmetry is an unwanted property. Since bidirectional transformations are symmetrical, the two domains M and N are interchangeable; from a bidirectional transformation over $M \times N$, we can obtain a bidirectional transformation over

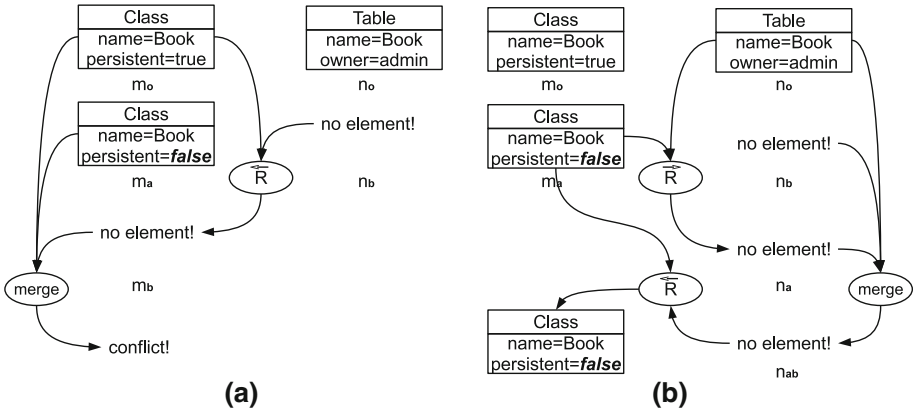


Fig. 8 An asymmetrical execution

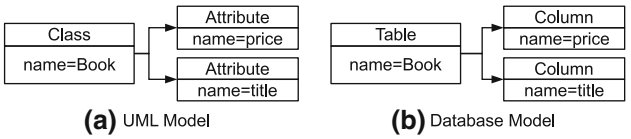


Fig. 9 Bijective transformation

$N \times M$ by simply swapping the forward transformation and the backward transformation. However, as our algorithm is asymmetrical, the two domains are no longer interchangeable and users must pay attention to the order of the two domains. This is sometimes very confusing. In an ideal setting, SYNC and SYNC$^{\triangleleft}$ should always produce the same synchronizer. In this section we try to find a necessary condition for such a setting.

Let us review the above example. The two synchronizers produce different results because the two models do not contain symmetrical information. A UML model containing no class and a UML model containing non-persistent classes are not distinguishable on the database side, because they both correspond to a database model containing no table. If the two models contain symmetrical information, i.e., the consistency relation is bijective, the two synchronizers could possibly be the same. Figure 9 shows an example of two models in a bijective relation. Compared with our running example, this example omits the `persistent` feature on UML classes and the `owner` feature on database tables. Now each UML class corresponds to a database table, and each UML attribute corresponds to a database column. Any modification on one side can be reflected to the other side without information loss.

Bijective relations alone do not ensure the symmetry of the algorithm because the algorithm still contains a "merge" step where a model difference approach is used to find and merge updates on models. We need to further ensure the model difference approach works consistently with the bijective relation on the two meta models. If we find and merge updates on one side, we should get the same result if we find and merge updates on the other side. This is a natural requirement when the two meta-models have a one-to-one correspondence. Formally, we require that the model difference approach $(diff, +)$ conforms to the consistency relation R, as defined below.

Property 9 (Conformance to Relation R) If we have

$$R(m_o, n_o),$$
$$R(m_a, n_a), \quad \text{and}$$
$$R(m_b, n_b),$$

we have either

$$R(merge(m_o, m_a, m_b), merge(n_o, n_a, n_b))$$

or both $merge(m_o, m_a, m_b)$ and $merge(n_o, n_a, n_b)$ are not defined.

When the consistency relation is bijective and the model difference approach conforms to the consistency relation, we can ensure that the synchronization algorithm is symmetrical.

Theorem 4 *If the consistency relation R is bijective, the bidirectional transformation $(\overrightarrow{R}, \overleftarrow{R})$ satisfies correctness, and the model difference approach $(diff, +)$ conforms to R, we have that* SYNC$[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ *and* SYNC$^{\triangleleft}[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ *are the same synchronizer.*

Proof To show that the two synchronizers are the same, we need to show that SYNC$[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ and SYNC$^{\triangleleft}[\![\overrightarrow{R}, \overleftarrow{R}, diff, +]\!]$ produce the same result for any input m_o, n_o, m_a, n_b. In the execution of SYNC, we have

$$m_b := \overleftarrow{R}(m_o, n_b)$$
$$m_{ab} := merge(m_o, m_a, m_b)$$
$$n_{ab} := \overrightarrow{R}(m_{ab}, n_b).$$

In the execution of SYNC$^{\triangleleft}$, we have

$$n_a := \overrightarrow{R}(m_a, n_o)$$
$$n'_{ab} := merge(n_o, n_b, n_a)$$
$$m'_{ab} := \overleftarrow{R}(m_a, n'_{ab}).$$

We need to show either $m_{ab} = m'_{ab}$ and $n_{ab} = n'_{ab}$, or both executions fail.

From the definition of the synchronizer, we have $R(m_o, n_o)$. From correctness, we have $R(m_a, n_a)$ and $R(m_b, n_b)$. Because the model difference approach conforms to R, we have either $R(m_{ab}, n'_{ab})$, or both executions fail. If the executions do not fail, we have $R(m_{ab}, n_{ab})$ and $R(m'_{ab}, n'_{ab})$ from correctness. Because the relation is bijective, we have $m_{ab} = m'_{ab}$ and $n_{ab} = n'_{ab}$.

There is also a preservation testing step, and this testing will always pass if the merge step does not fail. In the execution of SYNC, we have $n_{ab} = n'_{ab} = merge(n_o, n_b, n_a) = diff(n_o, n_b) + diff(n_o, n_a)$, and thus the update from n_o to n_b is preserved in n_{ab}. Similarly, in the execution of SYNC$^{\triangleleft}$ the update from m_o to m_a is preserved in m'_{ab}. □

This theorem seems to have little practical use at first glance, because in most cases the consistency relations between two domains are not bijective [3]. However, using the constant complement techniques [14,15], many consistency relations can be lifted as a bijective relation. Interested readers may refer to Appendix A for more information. In Appendix A we also show that for any bidirectional transformation satisfying history ignorance, there exists a constant complement to lift its consistency relation into a bijective relation [16].

7 Case study

Our algorithm greatly reduces the complexity of developing synchronizers, but may not have an optimal performance. To perform a synchronization, we need to cache the old versions of models and perform two difference operations, two transformations, and a preservation testing. A manually implemented synchronizer can potentially achieve much higher performance by recording update operations and propagating updates incrementally. Therefore, one issue to figure out is how our algorithm performs in a practical setting, so that we can decide whether the reduced development effort outweighs the increased execution time.

Another issue about our approach is that we develop the three properties purely from a theoretical perspective, but it is unclear whether the three properties always make sense in a practical setting.

In this section we discuss these issues through a case study. We have used our algorithm to implement the runtime management feature of ArchStudio [17], in which a C2-style software architecture model and a runtime system model are synchronized. This case study is also part of a bigger project implementing a runtime management framework. More details of this framework can be found in a technical report [18].

7.1 Case description

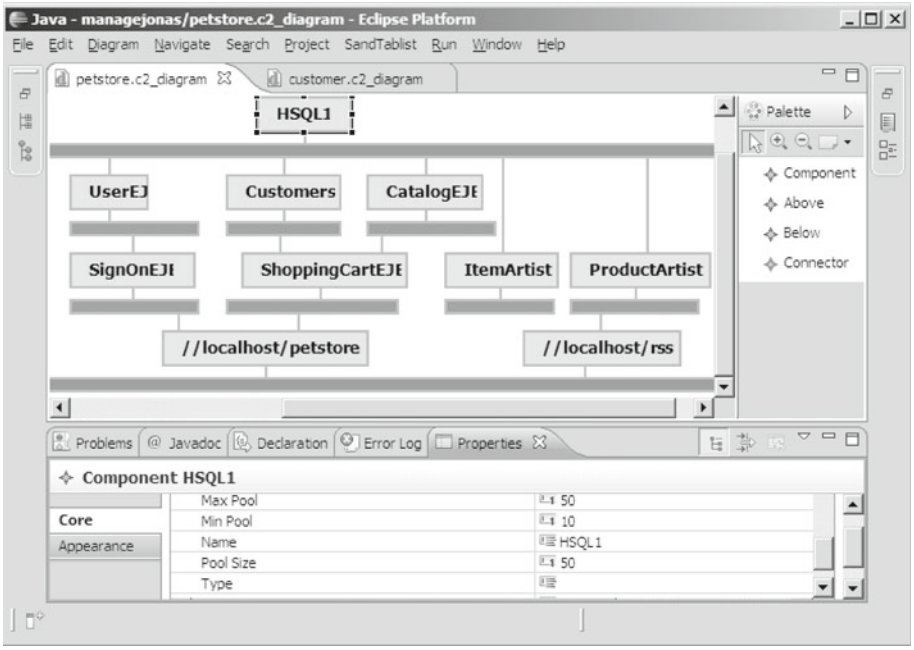
Oreizy et al. [17] propose an architecture-based runtime management tool, ArchStudio, which allows users to monitor and reconfigure a running system through a C2-style architecture model. In our case study, we re-implement this tool to monitor and reconfigure any Java application supporting the JMX interface [19]. Figure 10 shows a snapshot of our tool, in which the runtime state of a running web application, Java Pet Store (JPS) [20] is shown as a C2-style architecture model. In the model, labeled rectangles are components currently running in the system, including Enterprise JavaBeans (EJBs), web modules, and database connections. Solid bars connecting components are called connectors, representing the communication channels between components. Each component has a set of attributes, which are shown in the "Properties" view when the component is selected. These attributes reflect the runtime state of the component. For example, each EJB component has a property "Pool Size" showing the current size of its object pool.

The key of the system is a menu item called "synchronization". If we make any change on the architecture model, e.g., adding/removing a component, changing the attribute of a component, or changing the communication channels between components, the changes are automatically applied to the running system after we click the "synchronization" menu item. At the same time, the architecture model is also updated to display the latest state of the system.

We implement the system using the structure in Fig. 11. We create two models for the architecture and the running system, respectively, and use our algorithm to synchronize the two models. On the architecture side, we create a C2-style architecture meta model, and use Eclipse Graphical Modeling Framework [21] to generate a visual editor for its model. On the system side, we use the technology proposed by Song et al. [22] to wrap the JMX interface as a dynamic runtime model, which redirects any read/write access to the corresponding JMX functions. The architecture model and the system model are heterogeneous and need to be synchronized. We write a QVT-R [1] program to synchronize the two models and use our algorithm to wrap this QVT program as a synchronizer for concurrent updates. The model difference approach we used is extracted from the Beanbag system [12]. Every time a user clicks the "synchronization" menu item, we invoke the synchronizer to perform a synchronization.

The system is not trivial, but using our algorithm, we spent only a little effort on developing the synchronization component. The meta model of the architecture model contains 4 classes and 25 attributes/references, while the meta model of the system model contains 21 classes and 157 attributes/references. Using our algorithm, we only wrote 207 lines of QVT code and the whole development took no more than one day.

Fig. 10 Tool snapshot



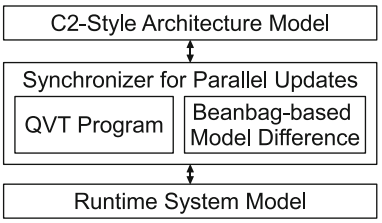


Fig. 11 Structure of our implementation

7.2 Discussion of properties

As we implement the synchronization component using our algorithm, the three properties of our algorithm affect the behavior of the whole system. Because of stability, nothing will change if there is no change on both sides when we click the "synchronization" menu item. Because of preservation, all changes to the architecture and the system since last synchronization are preserved during the synchronization. Because of consistency, the architecture model and the system model are ensured to be consistent after we click the "synchronization" menu item.

The effects of stability and consistencycorrespond to the basic requirements of runtime management systems [17]. The effect of preservation is more interesting. Because the running system is constantly changing, it is highly possible that a change on the architecture conflicts with a change on the running system. If we enforce preservation, users will often encounter conflicts when they update the architecture model, and can only solve the conflicts by canceling the update. On the other hand, updates on the architecture model are in fact control operations over the system, so it should be enforced to the system regardless of whether the system is changed or not. Based on the consideration, we loosen the preservation property to give precedence to the architecture model: the updates on the architecture models are preserved, while the updates on the system can be overwritten by the updates from the other side. To implement this, we change the difference algorithm so that it overwrites an update made to the running system with one made to the management UI if the two updates conflict. In addition, we remove the final preservation test. We also get an extra benefit from this change: all conflicts are now automatically solved by overwriting the system updates with the architecture updates.

The case study shows that the properties of synchronization are not always required. Depending on the types of the application, some properties may need to be loosened or canceled. On the other hand, since our algorithm is simple, it is possible to adjust the algorithm for specific cases.

Another interesting issue is the properties of bidirectional transformation. To ensure stability and consistency, the bidirectional transformation should satisfy hippocraticness and correctness. However, QVT-R does not always guarantee the two properties. If a program has complex interaction with the constraints on the meta models, it may produce inconsistent results. In our implementation, we manually check the consistency of our program and the constraints on the meta models to ensure correctness and hippocraticness.

7.3 Performance

To test the performance of our system, we perform four typical runtime management operations and record the time required for these operations. The four operations are (1) synchronize an empty architecture model to get a new

Table 3 Execution time of management operations

Operation	Time (s) in our tool	Time (s) in jonasAdmin
Get a new model	0.39	0.82[a]
Update an existing model	0.51	0.82[a]
Change an attribute	0.75	0.79
Add a component	1.37	1.20

[a]The two operations are in fact the same because the JOnAS administrative tool does not have an existing model to update

architecture model from the running system; (2) synchronize an unchanged architecture model to get the newest system state; (3) change an attribute of a component and synchronize; and (4) add a new component and synchronize. These operations are performed on a JPS application running on a JOnAS application server [23].

For comparison purposes, we perform the same operations using the JOnAS administrative tool (in WAR file "jonasAdmin"), a web-based tool for invoking JMX interface. Compared with the execution in our tool, executing the operations in the JOnAS administrative tool does not need to synchronize any model, but the tool needs to generate web pages and parse web requests in the background.

The experiments were performed on a computer with Intel Pentium 3.0 GHz process and 2.0 GB memory. The operating system was Windows XP and the web browser was Internet Explorer 7.0. The system model contains 41 model elements with approximately 300 attributes/references and the architecture model contains 82 model elements with approximately 700 attributes/references when we performed the operations. Based on our experience, these models are of mid-size in all models that we encounter in practice. All operations were performed ten times and the average time was calculated.

Table 3 summarizes the result of the experiment. In three out of four operations our tools are faster than the JOnAS administrative tool. This shows that on average the synchronization time of our algorithm is shorter than the background processing time of the JOnAS administrative tool for mid-size models. Since the JOnAS administrative tool is widely used by developers and no performance issue is reported as far as we know, the performance of our algorithm is acceptable for practical cases.

8 Related work

Several other approaches also target synchronizing concurrent updates on heterogeneous data. Typical ones include Harmony [24] and Beanbag [12].

The goal of Harmony is similar to ours: to use bidirectional transformations to construct synchronizers for concurrent updates. Compared with our approach, Harmony uses an asymmetrical form of bidirectional transformation, where the target is an abstract of the source. Users must design a common replica and write two transformation programs to map the replicas to be synchronized to the common replica. Our approach does not require users to design an extra model, so users can better reuse existing transformation programs. In addition, we adopt the symmetrical form of bidirectional transformation, which is more frequently used in the model transformation community.

Beanbag is a general language for synchronizing concurrent updates. Different from this paper, Beanbag uses an operation-based approach: users need to tell the synchronizer what update operations have been applied, and the synchronizer returns more update operations to make the data consistent. The approach in this paper is state-based: whole copies of models (the current states of models) are taken as input and new copies of these models are returned.

Research on optimistic replication [25] also aims at synchronizing concurrent updates. Different from our approach, this kind of research mainly deals with homogeneous data over distributed network and focuses on network communication, operation scheduling, etc. On the other hand, our approach focuses on heterogeneous data and simplifies other problems by assuming that all models are local and operations can be discovered and merged by model difference approaches. Our approach can potentially be combined with approaches of optimistic replications and synchronize heterogeneous data over distributed network.

Another related branch of research is detecting and fixing inconsistencies in models [26,27]. The methods developed can also be used to synchronize concurrent updates but they consider a different setting where only the updated models are available. In such cases we do not know what updates are applied to the models and usually cannot determine a unique way to synchronize updates. These approaches either generate a list of fixing actions for users to choose from or require users to provide extra operations for fixing. Compared with them, our approach utilizes the information of updates and requires neither user intervention nor extra operations.

Incremental bidirectional transformation [28] potentially can be combined with our approach to improve the performance of our approach. Instead of transforming models, incremental bidirectional transformation transforms an update on one model to an update on the other model. Because updates are usually much smaller than models, the transformation time is much shorter. So to improve the performance of our approach, we may replace the bidirectional transformation with an incremental bidirectional transformation. Furthermore, because the output of the transformation is also an update, we can ignore some model difference operations to further improve performance.

Some researchers build frameworks for classifying synchronization approaches. Antkiewicz and Czarnecki [29] classify synchronization approaches using different design decisions. Under their classification schema, our synchronization algorithm can be classified as a "bidirectional, non-incremental, and many-to-many synchronizer using artifact translation, homogeneous artifact comparison, and reconciliation with choice". Diskin [9] builds a more formal framework for bidirectional model synchronization, in which bidirectional transformation is classified into lenses, di-systems, and tri-systems on the basis of the relation between models and the number of input models. Our definition of a synchronizer for concurrent updates can be considered a supplement to his framework, where we add quadruple-systems, in the sense that our synchronizer takes four models as input.

9 Conclusion and future work

In this paper we have proposed an approach that wraps a bidirectional transformation program and a model difference approach into a synchronizer for concurrent updates. Our approach is general and predictable. It is general in the sense that it allows the use of any bidirectional transformation and any model difference approach, and it is predictable because it satisfies three model synchronization properties: consistency, stability and preservation. Our approach also shows the relation between the bidirectional transformation properties and the synchronization properties. Particularly, correctness leads to consistency, hippocraticness leads to stability, and history ignorance contributes to the symmetry of the synchronization algorithm through constant complement.

Currently, our approach only reports the existence of conflicts; it does not support conflict resolution. A preferable synchronizer would report the features and model elements involved in the conflicts and give a list of solutions for the user to choose from. However, such a resolution procedure is difficult to define in general because the reason for a conflict is related to the specific bidirectional transformation and the model difference approach used. We plan to design a resolution procedure based on a specific transformation language and a specific model difference approach. One idea is to use QVT-R as the transformation language and exploit the trace information recorded by QVT-R. This remains for future work.

A Converting history-ignorance transformation to bijective relation

In this section we show how to convert a history-ignorant bidirectional transformation into a bijective relation to ensure symmetrical synchronization. We achieve this through *constant complement* [14,15], a commonly used method for constructing bidirectional transformation. The original constant complement technique is defined for the asymmetrical form of bidirectional transformation [30]; here we extend it to the symmetrical form used in this paper.

A.1 Constant complement

Constructing bidirectional transformation is usually difficult because the consistency relation R is not bijective. If the relation is bijective, bidirectional transformation can be achieved by simply returning the only consistent model on the other side. A relation is not bijective because the two domains involved in the relation are not symmetrical; each may contain information that does not exist on the other side. If we augment the models with the lost information from the other side (called *complement*), the relation becomes bijective.

Formally, given a consistency relation R over two domains M and N, we can construct two *complement functions*,

$$l_m : M \to C_M$$
$$l_n : N \to C_N$$

such that the following relation is a bijective relation between $M \times C_N$ and $N \times C_M$:

$$S = \{((m, c_n), (n, c_m)) \mid c_m = l_m(m) \wedge c_n = l_n(n) \wedge R(m, n)\}$$

The complement function c_m maps a model in M to a complement containing the information lost on the N side. Similarly, c_n maps a model in N to a complement containing the information lost on the M side.

Since S is bijective, we can easily construct a bidirectional transformation over S as follows. In the definitions we use _ to denote the parameters we do not care about.

$$\overrightarrow{S}((m, c_n), (\_, \_)) = (n', l_m(m)) \quad \text{where } R((m, c_n), (n', l_m(m)))$$
$$\overleftarrow{S}((\_, \_), (n, c_m)) = (m', l_n(n)) \quad \text{where } R((m', l_n(n)), (n, c_m))$$

After we have the bidirectional transformation over S, we can construct the bidirectional transformation over R. This is achieved by constructing the complement before the transformation and discarding the complement after the transformation.

$$\overrightarrow{R}(m, n) = n' \quad \text{where } (n', \_) = \overrightarrow{S}(m, l_n(n))$$
$$\overleftarrow{R}(m, n) = m' \quad \text{where } (m', \_) = \overleftarrow{S}(n, l_m(m))$$

It is easy to see that this bidirectional transformation satisfies correctness and hippocraticness. Because the complement is

never modified by users, this technique is called constant complement.

A.2 Converting history-ignorance transformation

Now let us return to the problem of symmetrical synchronization. Given a bidirectional transformation $(\overrightarrow{R}, \overleftarrow{R})$ and two model difference operations $(diff, +)$, we would like to have SYNC and SYNC$^{\triangleleft}$ produce the same synchronizer. If R is bijective and $(diff, +)$ conforms to R, we can ensure SYNC and SYNC$^{\triangleleft}$ produce the same result. However, in many cases the consistency relation R is not bijective. On the other hand, using a pair of complement functions we can convert the consistency relation R into a bijective relation S. From S we can derive a bidirectional transformation $(\overrightarrow{S}, \overleftarrow{S})$ over S and a bidirectional transformation $(\overrightarrow{R'}, \overleftarrow{R'})$ over R, where $(\overrightarrow{S}, \overleftarrow{S})$ and $(\overrightarrow{R'}, \overleftarrow{R'})$ exhibit the same behavior and only differ in containing or not containing the complement. If we can find a pair of complement functions such that the derived transformation $(\overrightarrow{R'}, \overleftarrow{R'})$ is the same as $(\overrightarrow{R}, \overleftarrow{R})$, we can use $(\overrightarrow{S}, \overleftarrow{S})$ to construct a synchronizer $sync'$ over S. Since S is bijective, using a proper model difference approach, we can ensure the symmetry of the algorithm, and the synchronizer $sync'$ is ensured to be the same whether constructed by SYNC or by SYNC$^{\triangleleft}$. From $sync'$ we can obtain a synchronizer $sync$ over R by constructing and discarding the complement, as follows:

$$
\begin{aligned}
&sync(m, n, m', n') = (m'', n'') \\
&\quad \text{where } ((m'', \_), (n'', \_)) = \\
&\quad sync'((m, l_n(n)), (n, l_m(m)), (m', l_n(n')), (n', l_m(m')))
\end{aligned}
$$

However, it is unclear whether there exists such a pair of complement functions for a specific bidirectional transformation. Here we show that, at least for any bidirectional transformation $(\overrightarrow{R}, \overleftarrow{R})$ satisfying correctness, hippocraticness and history ignorance, there exists a pair of complement functions such that the derived bidirectional transformation is the same as $(\overrightarrow{R}, \overleftarrow{R})$. This result has been proved by Lechtenbörger [16] on the asymmetrical form of bidirectional transformation; here we prove it on the symmetrical form.

To show the existence of the complement functions, we can construct such a pair of complement functions from the bidirectional transformation $(\overrightarrow{R}, \overleftarrow{R})$. Suppose $(\overrightarrow{R}, \overleftarrow{R})$ is defined over the consistency relation $R \subseteq M \times N$. First let us define an equivalence relation on N.

$$n \sim n' \quad \text{iff } \exists m \in M.\, \overrightarrow{R}(m, n) = \overrightarrow{R}(m, n')$$

It is easy to show this relation is an equivalence relation.

Reflexitivity We have $n \sim n$ because $\exists m.\, \overrightarrow{R}(m, n) = \overrightarrow{R}(m, n)$

Symmetry If $n \sim n'$, we have $\exists m.\, \overrightarrow{R}(m, n) = \overrightarrow{R}(m, n')$, and thus $n' \sim n$.

Transitivity If $n \sim n'$ and $n' \sim n''$, we need to show $n \sim n''$. From $n \sim n'$ we have $\exists m.\, \overrightarrow{R}(m, n) = \overrightarrow{R}(m, n')$. From $n' \sim n''$ we have $\exists m'.\, \overrightarrow{R}(m', n') = \overrightarrow{R}(m', n'')$. Because of history ignorance, we have $\overrightarrow{R}(m', n) = \overrightarrow{R}(m', \overrightarrow{R}(m, n)) = \overrightarrow{R}(m', \overrightarrow{R}(m, n')) = \overrightarrow{R}(m', n') = \overrightarrow{R}(m', n'')$, and thus $n' \sim n''$.

Similarly, we can define an equivalence relation on M.

$$m \sim m' \quad \text{iff } \exists n \in M.\, \overleftarrow{R}(m, n) = \overleftarrow{R}(m', n)$$

Now we can define the two complement functions. We define the complement function as a function mapping a model to its equivalence class.

$$
\begin{aligned}
&l_m : M \to M/\sim \\
&l_m(m) = [m]^{\sim} \\
&l_n : N \to N/\sim \\
&l_n(n) = [n]^{\sim}
\end{aligned}
$$

We need to show that the relation S constructed by the two complement functions is bijective and the derived bidirectional transformation $(\overrightarrow{R}', \overleftarrow{R}')$ is the same as $(\overrightarrow{R}, \overleftarrow{R})$. We first prove S is bijective. To prove it, we need to show this relation is functional, injective, left-total, and right-total.

Functional Suppose we have $S((m, c_n), (n, c_m))$ and $S((m, c_n), (n', c'_m))$, we need to show that $n = n'$ and $c_m = c'_m$.

- From $S((m, c_n), (n, c_m))$ we have $R(m, n)$, and thus $\overrightarrow{R}(m, n) = n$. Similarly, we have $\overrightarrow{R}(m, n') = n'$. From $l_n(n) = c_n = l_n(n')$ we know that there exists m_0, $\overrightarrow{R}(m_0, n') = \overrightarrow{R}(m_0, n)$. Given history ignorance, we have $\overrightarrow{R}(m, n') = \overrightarrow{R}(m, \overrightarrow{R}(m_0, n')) = \overrightarrow{R}(m, \overrightarrow{R}(m_0, n)) = \overrightarrow{R}(m, n)$, and thus $n = n'$.
- From the definition of S we have $c_m = l_m(m) = c'_m$.

Injective Similar to the above.

Left-total Let (m, c_n) be an arbitrary pair in $M \times C_N$; we need to show that there exists $(n, c_m) \in N \times C_M$ where $S((m, c_n), (n, c_m))$. Let n_0 be an arbitrary element in c_n; we have $\overrightarrow{R}(m, n_0) = \overrightarrow{R}(m, \overrightarrow{R}(m, n_0))$ according to hippocraticness and correctness, and thus $n_0 \sim \overrightarrow{R}(m, n_0)$. Consequently $l_n(\overrightarrow{R}(m, n_0)) = c_n$. From correctness we also have $R(m, \overrightarrow{R}(m, n_0))$. Therefore, we have $S((m, c_n), (\overrightarrow{R}(m, n_0), l_m(m)))$. Let $n = \overrightarrow{R}(m, n_0)$ and $c_m = l_m(m)$, we have $S((m, c_n), (n, c_m))$.

Right-total Similar to the above.

Next we show that the derived bidirectional transformation $(\overrightarrow{R}', \overleftarrow{R}')$ is the same as $(\overrightarrow{R}, \overleftarrow{R})$. Given two arbitrary models m, n, we need to show $\overrightarrow{R}'(m, n) = \overrightarrow{R}(m, n)$ and

$\overleftarrow{R}'(m,n) = \overleftarrow{R}(m,n)$. Let $\overrightarrow{R}(m,n) = n'$. From hippocraticness we have $\overrightarrow{R}(m,n') = n' = \overrightarrow{R}(m,n)$, and thus $n \sim n'$. Consequently $l_n(n) = l_n(n')$. From correctness we have $R(m,n')$, and thus $S((m,l_n(n)),(n',l_m(m)))$. According to the definition of $\overrightarrow{R}'$, we have $\overrightarrow{R}'(m,n) = n'$. Similarly, we can prove $\overleftarrow{R}'(m,n) = \overleftarrow{R}(m,n)$.

To sum up, using constant complement we can convert a bidirectional transformation into a bijective relation to ensure the symmetry of the algorithm, and the synchronizer constructed can be applied to the previous meta models by constructing and discarding complements. The complement functions are ensured to exist for bidirectional transformation satisfying correctness, hippocraticness and history ignorance.

References

1. Object Management Group: MOF query/views/transformations specification 1.0. (2008). http://www.omg.org/docs/formal/08-04-03.pdf
2. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Proceedings of the 4th International Conference on Graph Transformation. Lecture Notes in Computer Science, vol. 5214, pp. 411–425. Springer, Berlin (2008)
3. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Softw. Syst. Model. **9**(1), 7–20 (2010)
4. Alanen, M., Porres, I.: Difference and union of models. In: UML'03: Proceedings of the 6th International Conference on the Unified Modeling Language. Lecture Notes in Computer Science, vol. 2863, pp. 2–17. Springer, Berlin (2003)
5. Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design. In: ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ACM, pp. 204–213 (2005)
6. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D.: Differencing and merging of architectural views. In: ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, IEEE Computer Society, pp. 47–58 (2006)
7. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ACM, pp. 164–173 (2007)
8. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting parallel updates with bidirectional model transformations. In: ICMT '09: Proceedings of the Second International Conference on Theory and Practice of Model Transformations. Lecture Notes in Computer Science, vol. 5563, pp. 213–228. Springer, Berlin (2009)
9. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: MoDELS '08: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 5301, pp. 21–36. Springer, Berlin (2008)
10. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ACM, pp. 54–65 (2005)
11. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: Managing model conflicts in distributed development. In: MoDELS '08: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 5301, pp. 311–325. Springer, Berlin (2008)
12. Xiong Y., Hu Z., Zhao H., Song H., Takeichi M., Mei H.: Supporting automatic model inconsistency fixing. In: ESEC/FSE '09: Proceedings of 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, New York, NY, USA, ACM, pp. 315–324 (2009)
13. Khanna, S., Kunal, K., Pierce, B.C.: A formal investigation of diff3. In: Arvind, P. (eds.) Foundations of Software Technology and Theoretical Computer Science (FSTTCS), pp. 485–496 (2007)
14. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. (TODS) **6**(4), 557–575 (1981)
15. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM, pp. 47–58 (2007)
16. Lechtenbörger, J.: The impact of the constant complement approach towards view updating. In: PODS '03: Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, New York, NY, USA, ACM, pp. 49–55 (2003)
17. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering (ICSE'98), pp. 177–186 (1998)
18. Song, H., Xiong, Y., Hu, Z., Huang, G., Mei, H.: A model-driven framework for constructing runtime architecture infrastructures. Technical Report GRACE-TR-2008-05, GRACE Center, National Institute of Informatics, Japan (2008)
19. Java Management Extensions http://www.jcp.org/en/jsr/detail?id=77
20. Java PetStore http://java.sun.com/developer/releases/petstore/
21. Eclipse Consortium: The Eclipse Graphical Modeling Framework (2008). http://www.eclipse.org/modeling/gmf/
22. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Hong, M.: Generating synchronization engines between running systems and their model-based views. In: Proceedings of the 4th International Workshop on Models@run.time, pp. 11–20 (2009)
23. JOnAS Project. Java Open Application Server. http://jonas.objectweb.org
24. Pierce, B.C., Schmitt, A., Greenwald, M.B.: Bringing Harmony to optimism: a synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania (2003)
25. Saito, Y., Shapiro, M.: Replication: optimistic approaches. Technical Report HPL-2002-33, HP Laboratories Palo Alto (2002)
26. Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in UML design models. In: ASE '08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, IEEE Computer Society, pp. 99–108 (2008)
27. Kolovos, D., Paige, R., Polack, F.: Detecting and repairing inconsistencies across heterogeneous models. In: ICST '08: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation. IEEE Computer Society, pp. 356–364 (2008)
28. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. **8**(1), 21–43 (2009)
29. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: GTTSE '07: Proceedings of the 2nd Summer School on Generative and Transformational Techniques in

Software Engineering. Lecture Notes in Computer Science, vol. 5235, pp. 3–46. Springer, Berlin (2007)
30. 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)

Author Biographies

Yingfei Xiong is a postdoctoral fellow at University of Waterloo. Before joining Unviersity of Waterloo, he got his PhD degree from the University of Tokyo in 2009. His research interst is in the change of software, including synchronization of software artifacts, inconsistency management and tracking changes in code.

Hui Song is a PhD student at the Institute of Software in Peking University. His research interests includes models at runtime, software architecture, and system management.

Zhenjiang Hu is a professor of National Institute of Informatics (NII) in Japan. He received his PhD degree from the University of Tokyo in 1996. His main interest is in programming languages and software engineering in general, and functional programming, program transformation and model driven software development in particular.

Masato Takeichi is a Professor of the University of Tokyo since 1993, and he is a member of the Science Council of Japan since 2003. His research concern includes bidirectional transformation and functional programming.