Supporting Parallel Updates with Bidirectional Model Transformations

Yingfei Xiong[1], Hui Song[2], Zhenjiang Hu[1,3], and Masato Takeichi[1]

[1] Department of Mathematical Informatics
University of Tokyo, Tokyo, Japan
{Yingfei_Xiong,takeichi}@mist.i.u-tokyo.ac.jp

[2] Key Laboratory of High Confidence Software Technologies (Peking University)
Ministry of Education, Beijing, China
songhui06@sei.pku.edu.cn

[3] GRACE Center
National Institute of Informatics, Tokyo, Japan
hu@nii.ac.jp

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 propose a new algorithm that wraps any bidirectional transformation into a synchronizer with the help of a model difference approach. The synchronizer enables parallel updates by taking the two original models, the two updated models as input and producing two new models where the updates are synchronized. We also examine the requirements for synchronizing parallel updates, and demonstrate that our algorithm satisfies the requirements if the bidirectional transformation satisfies the *correctness* property and the *hippocraticness* property. Implementation of our algorithm showed that it works well in a runtime management framework in practical cases.

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, Figure 1(a) 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 Figure 1(b). 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"`.

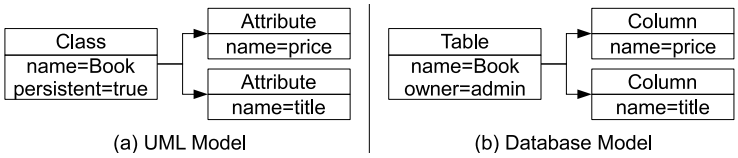

Fig. 1. Transforming a UML model into a database model

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.

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 fall into this model; 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, a designer could be working on the design model at the same time a programmer is working on the implementation model. 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 parallel updates using existing bidirectional transformations. One basic idea is to sequentially apply the two updates and interleave them with two transformations. For example, suppose a user changes the `price` attribute into `"bookPrice"` in the UML model and another user changes the `title` column into `"bookTitle"` in the database model at the same time, as shown in Figure 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, 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.

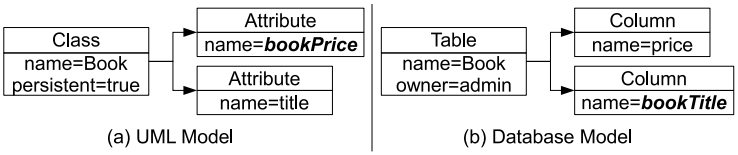

Fig. 2. Non-conflicting parallel updates

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 to the database model. A preferable synchronization procedure would detect such conflicts and advise the user.

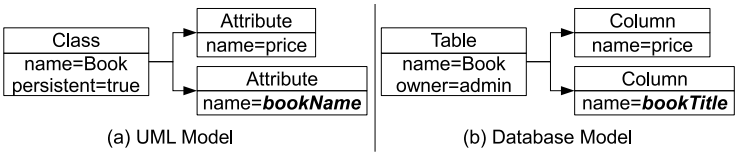

Fig. 3. Conflicting parallel updates

In this paper we propose a new approach based on the idea of sequentially applying parallel updates. We use commonly used model difference approaches [4–6] to solve the two problems above. We design an algorithm that use model difference approaches to wrap any bidirectional transformation into a synchronizer for parallel 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 parallel 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 parallel 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 framework. The application showed that our algorithm works well in practical cases.

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 parallel 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 describes its application and Section 7 discusses related work. Finally, Section 8 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. Stevens [3] proposes three 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 satisfy two of them (*correctness* and *hippocraticness*) because the last property, *undoability*, would prohibit many practical transformations.

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

Property 1 (Correctness).
$\forall m \in M, n \in N : \quad R(m, \overrightarrow{R}(m, n))$
$\forall m \in M, n \in N : \quad R(\overleftarrow{R}(m, n), n)$

The second property, *hippocraticness*, 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 modify neither model.

Property 2 (Hippocraticness).
$R(m, n) \Longrightarrow \overrightarrow{R}(m, n) = n$
$R(m, n) \Longrightarrow \overleftarrow{R}(m, n) = m$

The last property, *undoability*, 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).

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

While *undoability* makes sense in some situations, here we do not require bidirectional transformations to satisfy this property because *undoability* imposes a strong requirement on the consistency relation, R, and prohibits many useful transformations. One example is the UML-to-database transformation we mentioned in Section 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. This problem cannot be solved from the transformation alone. To satisfy *undoability*, we must change the meta model of the database to store all deleted `owner` properties, which would be impossible and unnecessary in many cases.

3 Requirements of Synchronizing Parallel Updates

As discussed above, the interface of the bidirectional transformation functions do not allow parallel 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 *synchronization procedure* for parallel 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 synchronization procedure. 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 parallel updates. First, the synchronization procedure 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 these input.

However, like bidirectional transformations, this definition alone does not impose much constraint on the behavior of the synchronization. We introduce three

properties to ensure the synchronization procedure behaves in a reasonable way. These properties were first proposed in previous work [7], and are significantly modified for the synchronization of parallel updates.

Similar to the properties of bidirectional transformation, our first property, *consistency*[4], requires that the synchronization procedure to do something useful. It ensures that consistency relation R is established on the output models.

Property 4 (Consistency).
$sync(m, n, m', n')$ is defined $\Longrightarrow R(sync(m, n, m', n'))$

The second property, *stability*, prevents the synchronization procedure from doing something harmful. If neither of the two models has been updated, the synchronization procedure should update neither of them.

Property 5 (Stability).
$R(m, n) \Longrightarrow sync(m, n, m, n) = (m, n)$

The last property, *preservation*, is more interesting. Consider the updates shown in Figure 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 synchronization procedure should not change the attribute to any other value.

Formally, let $P_M \in M \times M \times M$ be a preservation relation over M, in the sense that $P_M(m_o, m_a, m_c)$ implies that the update from m_o to m_a is preserved in m_c. Similarly, let $P_N \in N \times N \times N$ be a preservation relation over N.

Property 6 (Preservation).
$sync(m, n, m', n') = (m'', n'') \Longrightarrow P_M(m, m', m'')$
$sync(m, n, m', n') = (m'', n'') \Longrightarrow P_N(n, n', n'')$

Note that we do not define a universal preservation relation for all meta models. Instead, we allow different preservation relations to be defined for one meta model, and a synchronization procedure should satisfy a specific preservation relation. This is because there may be multiple modification sequences from one model to another, and a different choice of modification sequences leads to a different preservation result.

For example, in Figure 3(a), 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

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

not conflict, and the model in Figure 4 preserves the updates. As a result, the preservation relation depends on what update operations we consider and how we recover updates from models. In the next section we will define a preservation relation from a model difference approach.

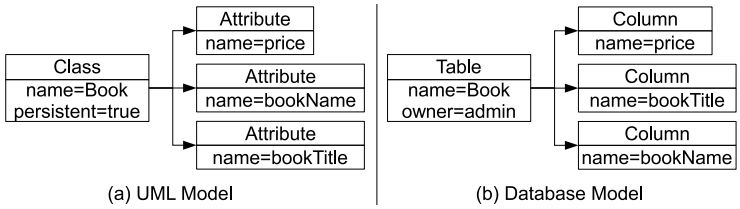

Fig. 4. Updates to both models are preserved

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

4 Model Difference, Three-Way Merger and Preservation

As introduced in Section 1, we use model difference approaches [4–6] to identify updates and detect conflicts. In this section we describe model differences in our context. We will also show how we use a model difference approach to define a three-way merger and a preservation relation, which will be used in our algorithm.

4.1 Model Difference

Following the definitions of Diskin [8], 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 $\delta.post$). There may be different updates leading from one model to another, so the graph is a multigraph, 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 [9, 10] 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. The first operation is used to identify the updates in two models.

We call it the *difference operation*. Formally, a difference operation is a function, $\mathit{diff} \in M \times M \to \Delta_M$, that takes two models, m and m', and produces update δ, where $\delta.\mathit{pre} = m$ and $\delta.\mathit{post} = m'$. 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 [9], 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 $\delta_1.\mathit{pre} = \delta_2.\mathit{pre}$, we denote their union as $\delta_1 + \delta_2$, where $(\delta_1 + \delta_2).\mathit{pre} = \delta_1.\mathit{pre} = \delta_2.\mathit{pre}$ and $(\delta_1 + \delta_2).\mathit{post}$ 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 conflict, $\delta_1 + \delta_2$ is undefined. The techniques to implement this operation can be found in existing approaches [4, 9].

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

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 7 (Stability of Model Difference).
$\forall \delta \in \Delta_M : \quad \delta + \mathit{diff}(\delta.\mathit{pre}, \delta.\mathit{pre}) = \delta$

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 [11] 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.

$$\mathit{merge}(m_o, m_a, m_b) = (\mathit{diff}(m_o, m_a) + \mathit{diff}(m_o, m_b)).\mathit{post}$$

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

4.3 Preservation

In Section 3 we have mentioned that there are multiple preservation relations for one meta model if there are multiple updates from a pair of models. As model difference approaches identify an update 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 where $(\mathit{diff}(m_o, m_a) + \delta).\mathit{post} = m_c$.

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

Theorem 1. *If $m_c = \mathit{merge}(m_o, m_a, m_b)$, 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 $(\mathit{diff}(m_o, m_a) + \mathit{diff}(m_o, m_b)).\mathit{post} = m_c$. From the commutativity of +, we get $(\mathit{diff}(m_o, m_b) + \mathit{diff}(m_o, m_a)).\mathit{post} = m_c$. Because there exists $\mathit{diff}(m_o, m_b)$, 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

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 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)

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).

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 $\delta_{oa} = \mathit{diff}(m_o, m_a)$. 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 Figure 1(a), the `bookPrice` attribute in Figure 2(a), and the `price` attribute in Figure 3(a) share UUID e_p. The difference of Figure 1(a) and Figure 2(a) is thus an update containing one update operation: set(e_p, `name`, `"price"`, `"bookPrice"`). This update is not preserved in Figure 3(a) 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 Figure 3(a).

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 parallel 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 Figure 5.

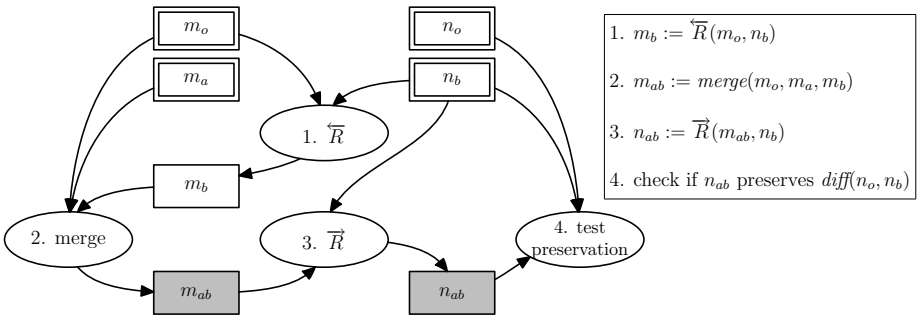

Fig. 5. Synchronization algorithm

We explain the algorithm using the example in Section 1. Initially, we have the two models in Figure 1, which correspond to m_o and n_o in our algorithm. Users modify the two models into the models in Figure 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.

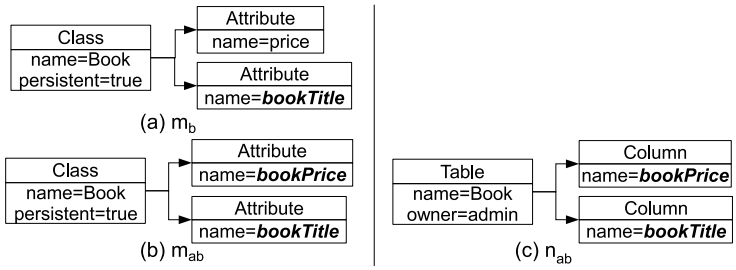

Fig. 6. Execution of algorithm

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. The result is shown in Figure 6(a). 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 synchronized model m_{ab} on the M side. The result is shown in Figure 6(b). 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 synchronized model n_{ab} on the N side. The result is shown in Figure 6(c). 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 Figure 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.

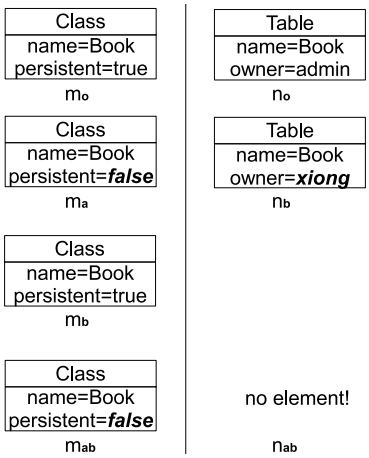

Fig. 7. Violating *preservation*

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.

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

The models used in Figure 6 and Figure 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 above execution. Nevertheless, whatever bidirectional transformation and model difference approach we choose, our algorithm ensures the three synchronization properties: *consistency*, *stability*, and *preservation*.

Theorem 2. *If the bidirectional transformation satisfies correctness, the synchronization algorithm satisfies consistency.*

Proof. Because $\overrightarrow{R}(m_{ab}, n_b) = n_{ab}$, we have $R(m_{ab}, n_{ab})$.

Theorem 3. *If the bidirectional transformation satisfies hippocraticness and the model difference approach satisfies stability of model difference, the synchronization algorithm 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, $n_{ab} = \overrightarrow{R}(m_{ab}, n_b) = \overrightarrow{R}(m_o, n_o) = n_o$, and the preservation testing always passes because of the existence of identity update.

Theorem 4. *The synchronization algorithm always satisfies preservation.*

Proof. Because of Theorem 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 bidirectional transformation languages in practice 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*).

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. Implementing the algorithm in both directions can guarantee the three properties. However, due to the heterogeneity of M and N, it is possible that different directions may produce different results for some input. The difference is related to the specific bidirectional transformation approach and the difference approach used in the algorithm, and we do not discuss it in this paper.

6 Application

We implemented our algorithm in a runtime management framework [12], as shown in Figure 8. We used our algorithm to wrap a QVT-R program [1] (executed in mediniQVT [13]) and a Beanbag-based model difference approach [10] into a synchronizer for parallel updates, and used our synchronizer to synchronize a runtime management user interface and a running system.

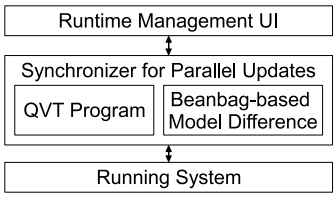

Fig. 8. Structure of runtime management system

A high-level management user interface (UI) is often provided in a runtime management system for monitoring the state of the running system and for reconfiguring it. Because the high-level management UI often abstracts away many low-level details, the high-level UI and the running system

are heterogeneous and need to be synchronized. Because the system state is constantly changing during runtime, any modification to the management UI will cause parallel updates. In our implementation, we captured both the management UI and the running system as models and used our algorithm to synchronize them.

The bidirectional transformation used to synchronize the two models is a QVT-R program. The QVT-R language [1] enables rapid development of bidirectional transformations. However, it does not always guarantee *correctness* and *hippocraticness*. If a program has complex interaction with the constraints on the meta models, it may produce inconsistent result. In our implementation, we manually check the consistency of our program and the constraints on the meta models to ensure *correctness* and *hippocraticness*.

The model difference approach we used is extracted from the Beanbag system [10]. We first convert models into the Beanbag data types and then use operations provided by Beanbag to merge the models. In the conversion, we assume each model element has a unique identifier and do not consider ordered multiple features. The situation is simpler than those considered by most model difference approaches, but it is sufficient for models in runtime management. The details of the Beanbag data types and model conversion can be found in the technical report of Beanbag [10].

When there is a conflict between updates to the running system and those to the management UI, our synchronization algorithm reports an error and halts. The user needs to manually resolve the conflict and resynchronize again. However, as the system is constantly changing, it is often impossible for users to resolve all conflicts. We solve this problem by giving precedence to the updates made to the management UI. In a runtime management system, updates to the management UI are in fact control operations that the user want to perform on the system, so it is always safe to overwrite an update made to the running system with one made to the management UI. 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 performed a set of experiments using our runtime management framework, and the results showed that our algorithm works well. The details of the runtime management framework and the experiments can be found else where [12].

7 Related Work

Several other approaches also target synchronizing parallel updates on heterogeneous data. Typical ones include Harmony [14] and Beanbag [10].

The goal of Harmony is similar to ours: use bidirectional transformations to construct synchronizers for parallel updates. Compared to 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 parallel 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.

Another related branch of research is detecting and fixing inconsistencies in models [15, 16]. The methods developed can also be used to synchronize parallel updates but from a different perspective: only the updated models are examined, and the inconsistencies are resolved by human intervention or heuristic rules. This is very different from our objective of fully automatic, predictable synchronization behavior. Compared to them, our approach is fully automatic, and the synchronization behavior is predicable through the three properties.

Some researchers build frameworks for classifying synchronization approaches. Antkiewicz and Czarnecki [17] classifies 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 [8] 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 parallel 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.

8 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 parallel 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*.

Currently, our approach only reports the existence of conflicts; it does not support conflict resolution. A preferable synchronization procedure 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.

References

1. Object Management Group: MOF query / views / transformations specification 1.0. http://www.omg.org/docs/formal/08-04-03.pdf (2008)
2. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Proc. of the 4th International Conference on Graph Transformation. (2008) 411–425
3. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Proc. 10th MoDELS. (2007) 1–15
4. Alanen, M., Porres, I.: Difference and union of models. In: Proc. 6th UML. (2003) 2–17
5. Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design. In: Proc. 20th ASE. (2005) 204–213
6. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D.: Differencing and merging of architectural views. In: Proc. 21st ASE. (2006) 47–58
7. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Proc. 22nd ASE. (2007) 164–173
8. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: Proc. 11th MoDELS. (2008) 21–36
9. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: Managing model conflicts in distributed development. In: Proc. 11th MoDELS, Springer (2008) 311–325
10. Xiong, Y., Hu, Z., Zhao, H., Takeichi, M., Hui, S., Mei, H.: Beanbag: Operation-based synchronization with intra-relations. Technical Report GRACE-TR-2008-04, GRACE Center, National Institute of Informatics, Japan (December 2008)
11. Khanna, S., Kunal, K., Pierce, B.C.: A formal investigation of diff3. In Arvind, Prasad, eds.: Foundations of Software Technology and Theoretical Computer Science (FSTTCS). (December 2007) 485–496
12. 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 (December 2008)
13. ikv++ technologies: medini QVT homepage. http://projects.ikv.de/qvt
14. 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)
15. Egyed, A.: Fixing inconsistencies in UML design models. In: Proc. 29th ICSE. (2007) 292–301
16. Kolovos, D., Paige, R., Polack, F.: Detecting and repairing inconsistencies across heterogeneous models. In: ICST '08: Proceedings of the International Conference on Software Testing, Verification, and Validation. (2008) 356–364
17. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: Proc. 2nd GTTSE. (2007) 3–46