Replace this file with prentcsmacro.sty for your meeting,
or with entcsmacro.sty for your meeting. Both can be
found at the ENTCS Macro Home Page.

From Bidirectional Model Transformation to Model Synchronization

Yingfei Xiong[1]

Department of Mathematical Informatics
The University of Tokyo, Tokyo, Japan

Song Hui[2]

Key Laboratory of High Confidence Software Technologies (Peking University)
Ministry of Education, Beijing, China

Zhenjiang Hu[3]

GRACE Center
National Institute of Informatics, Tokyo, Japan

Masato Takeichi[4]

Department of Mathematical Informatics
The University of Tokyo, Tokyo, Japan

Abstract

In model-driven engineering, it is common that there are several related models co-existing. When one model is updated or several models are updated at the same time, we need to propagate the updates across all models to make them consistent. This process is called synchronization. Bidirectional model transformation partially supports the synchronization of two models by updating one model according to the other models. However, it does not work when the two models are modified at the same time.
In this work we propose a new algorithm that wraps any bidirectional transformation into a synchronizer, and this synchronizer allows simultaneous updates on the two models. We propose a general algebraic framework for model synchronization, and prove that our algorithm can ensure the synchronization properties if the bidirectional transformation satisfies the correctness property and the hippocraticness property [7].

Keywords: bidirectional transformation, model synchronization, model transformation

[1] Email: xiong@ipl.t.u-tokyo.ac.jp
[2] Email: songhui06@sei.pku.edu.cn
[3] Email: hu@nii.ac.jp
[4] Email: takeichi@mist.i.u-tokyo.ac.jp

©2009 Published by Elsevier Science B. V.

1 Introduction

One central activity of model-driven software development is to transform high-level models into low-level models through model transformations. In an ideal situation, the target model is always obtained from the source model and never need to be modified. However, in reality, developers often need to modify the target model directly. In such cases, we need to reflect the updates on the target models back to the source models.

Bidirectional model transformation solves this maintenance problem by providing bidirectional model transformation languages, which describe the relation between two models symmetrically. Programs in these languages are able to not only transform models from one format into the other, but also update the other model automatically when one model is updated by users. Typical bidirectional model transformation languages include QVT [5] and TGG [4].

Perdita Stevens [7] formalizes 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 on the models. A bidirectional transformation consists of the following two functions:

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

Given a pair of models $(m, n) \in M \times N$, the function $\overrightarrow{R}$ changes n to be consistent to m. Similarly, $\overleftarrow{R}$ changes m in accordance with n.

However, in some cases the the model m and n may both be updated before bidirectional transformation can be applied. For example, a designer is working on the design model and a programmer is working on the implementation model at the same time. Applying the transformation of any direction will result in the loss of updates on the target side.

To solve this problem, we need a synchronizer to propagate the updates on each model to the other model at the same time. In this paper we consider such a synchronizer as a partial function

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

that takes two original models in the consistency relation R, two updated models and produces two synchronized models. The output model should be close to the original models, and also contains the updates in the updated models and the updates propagated from the other sides. The function is partial because sometimes the updates on the two models may conflict and cannot be synchronized.

Given the large number of available bidirectional model transformation languages, there are relative few ready-to-use synchronization languages. So one natural idea is to use bidirectional model transformation to support model synchronization. In this paper we carry out theoretical studies of how bidirectional model transformation can be used to support model synchronization. The main contributions of this paper can be summarized as follows:

- We extend our previous algebraic framework for model synchronization [8] to general cases. We consider the symmetrical cases where no model is necessarily

an abstraction of the other model and open door to free choice of updates.

- We propose an algorithm that wraps any bidirectional model transformation into a synchronizer, with the help of a three-way merger. We also discuss basic conflict-resolving support in the synchronizer.
- We prove that, for any bidirectional transformation satisfying the correctness and hippocraticness properties [7], the synchronizer satisfies the stability, preservation and consistency properties [8], ensuring a correct and predictable synchronization behavior.

This paper is organized as follows. Section 2 introduces our algebraic framework of model synchronization, including three properties to characterize the behavior of synchronization. Section 3 introduces the bidirectional model transformation properties introduced by Stevens [7]. Based on these properties, Section 4 introduces our algorithm and prove that bidirectional model transformation properties lead to model synchronization properties. Section 5 introduces our basic conflict-resolving strategy. Finally, Section 6 discusses two pieces of related work.

2 Properties of Model Synchronization

We have seen the basic definition of a synchronizer: it takes two original models, two updated models and produces two synchronized models. However, this definition only characterize the input and output types of the synchronizer, and does not say much about the synchronization behavior. In this section we propose several properties to characterize the behavior of the synchronizer.

As the first step of charactering the behavior, let us define the updates on the models. In our definition, the synchronizer only takes models and produces new models, and one may ask: why do we need to consider updates? This is because we need to detect updates and merge simultaneous updates in synchronization. If we consider different sets of updates, the synchronization may lead to different results.

For example, let us consider the meta model M as a power set of some alphabet set Σ. Suppose two users made two different updates on one model, respectively, and their updated results are as follows.

the original model	$M_0 : \{a, b, c\}$
the first updated result	$M_1 : \{a, d, c\}$
the second updated result	$M_2 : \{a, e, c\}$

If we consider M_1 is created by replacing b by d, and M_2 is created by replacing b by e, the two updates will conflict. However, if we consider M_1 is created by deleting b and adding d, while M_1 is created by deleting b and adding e, this two updates are compatible and we can merge them as one model: $\{a, d, e, c\}$.

From these different results we can see that the synchronization behavior depends on what updates we choose. To clear characterize the behavior, we need to first be clear about which set of updates we will consider during the synchronization. First we give the definition of update: An *update* u defined on some meta-model M is an idempotent function $u \in M \rightarrow M$. We consider only the idempotent function because idempotence allows us to tell whether the update has been preserved in

a model. If we apply an update to a model and the model remain constant, the update has been preserved in the model.

Example 2.1 The meta model M is a power set of some alphabet set Σ. Suppose we have the following functions:

- $add[\![a]\!](A) = A \cup \{a\}$
- $remove[\![a]\!](A) = A \backslash \{a\}$
- $replace[\![a, b]\!](A) = \begin{cases} A\backslash\{a\} \cup \{b\} & a \in A \\ A & \text{otherwise} \end{cases}$

Then for any $a, b \in \Sigma$, we have $add[\![a]\!]$, $remove[\![a]\!]$ and $replace[\![a, b]\!]$ are updates.

After we define updates as functions, the relationship between updates can be defined through function composition. Two updates u_1, u_2 *conflict* iff $u_1 \circ u_2 \neq u_2 \circ u_1$. We write $u_1 \ominus u_2$ if u_1 and u_2 do not conflict.

Corollary 2.2 *$\ominus$ is commutative.*

Proof. By the definition. □

Corollary 2.3 *If $a \ominus b$, we have that $a \circ b$ is an update.*

Proof. Because $a \circ b = b \circ a$, we have $(a \circ b) \circ (a \circ b) = (a \circ a) \circ (b \circ b) = a \circ b$. □

Corollary 2.4 *If $b \circ c$ is an update and $a \ominus b$, $a \ominus c$, we have $a \ominus (b \circ c)$.*

Proof. Because $a \ominus b$, we have $a \circ b = b \circ a$. Putting together $a \ominus c$, we have $a \circ (b \circ c) = b \circ a \circ c = b \circ (a \circ c) = b \circ c \circ a$. □

Another relation we consider is whether an update is included in another update. An update u_1 is a *sub update* of another update u_2 iff $u_1 \circ u_2 = u_2 \circ u_1 = u_2$, denoted as $u_1 \sqsubseteq u_2$.

Corollary 2.5 *$\sqsubseteq$ is a partial order over any set of updates.*

Proof. By definitions. □

Proof. We need to show $\sqsubseteq$ is reflexive, antisymmetric and transitive.

Reflexive $a \circ a = a$

Antisymmetry If $a \sqsubseteq b$ and $b \sqsubseteq a$, we have $a \circ b = a$ and $a \circ b = b$, and then we have $a = b$.

Transitivity If $a \sqsubseteq b$ and $b \sqsubseteq c$, we have $a \circ b = b \circ a = b$ and $b \circ c = c \circ b = c$, then we have $a \circ c = a \circ (b \circ c) = (a \circ b) \circ c = b \circ c = c$. Similarly ,we can have $c \circ a = c$. □

Corollary 2.6
$\forall a, b \in \Sigma : a \neq b \Rightarrow add[\![a]\!] \ominus add[\![b]\!]$
$\forall a, b \in \Sigma : a \neq b \Rightarrow remove[\![a]\!] \ominus remove[\![b]\!]$

$\forall a, b \in \Sigma : a \neq b \Rightarrow add[\![a]\!] \ominus remove[\![b]\!]$

As we have discussed, to clearly characterize the synchronization behavior of a synchronizer, we need to know what kinds of updates can be applied to a model. We define the updates that can be applied to models in M through a *proper update set* U_M. A proper update set U_M defined on a meta model M is a set of updates that satisfies:

- U_M is closed on composition,
- the identity function $id \in U_M$, and
- for any $m, n \in M$, the set $\{u \in U_M | u(m) = n\}$ has a least element.

The first condition requires the property update set to be complete, so that we can freely apply a sequence of updates in the set without worrying whether we are applying a “proper update”. The second condition allows us to keep the model unmodified. The third condition implies two things: 1) any model can be applied to any other model, and 2) given two models, we can always find a unique update that is least among all updates.

If u is the least element in the set $\{u \in U_M | u(m) = n\}$ for any $m, n \in M$, we say u is the least update from m to m'.

To give an example of proper update set, let us define a function which construct a set of functions by composing another set of function with a predefined set: $compose[\![F]\!](H) = \{f \circ h \mid \forall f \in F, h \in H\}$

Lemma 2.7
$\forall a \in \Sigma : add[\![a]\!] \circ remove[\![a]\!] = add[\![a]\!]$
$\forall a \in \Sigma : remove[\![a]\!] \circ add[\![a]\!] = remove[\![a]\!]$

Example 2.8 Let
$B_1 = \{add[\![a]\!] \mid \forall a \in \Sigma\} \cup \{remove[\![a]\!] \mid \forall a \in \Sigma\}$,
$M_1 = \bigcup_{n=0}^{\infty}(compose[\![B_1]\!])^n(\{id\})$,
we have M_1 is a proper update set.

Proof. First, every element in M_1 is an update. Every element in M_1 can be written as a sequence of composition $oper[\![a_0]\!] \circ oper[\![a_1]\!] \ldots oper[\![a_n]\!]$ where $oper = add$ or $remove$ and $a_i \neq a_j$ for any $i \neq j$. This can be proved by mathematical induction. Furhter because of Corollary 2.7, Corollary 2.3 and Corollary 2.4, every element is an update.

Second, similarly, we have M_1 is closed on composition.

Third, by definition, id is in M_1.

Forth, consider two element m_0 and m_1 in M. Let $set_1 = \{add[\![a]\!] \mid a \in m_0 \wedge a \notin m_1\}$ and $set_2 = \{remove[\![a]\!] \mid a \in m_1 \wedge a \notin m_0\}$. The least update from m_0 to m_1 is a composition of all element in set_1 and set_2. We can easily prove this is a sub update of any other updates. □

Example 2.9 Let
$B_2 = B_1 \cup \{replace[\![a, b]\!] \mid \forall a, b \in \Sigma\}$,

$M_2 = \bigcup_{n=0}^{\infty}(compose[\![B_2]\!])^n(\{id\})$,
we have M_2 is not a proper update set.

Proof. Given two sets $m_1 = \{a, b\}$ and $m_2 = \{a, c\}$, both $add[\![c]\!] \circ remove[\![b]\!]$ and $replace[\![b, c]\!]$ can update m_1 to m_2, but none is a sub update of the other. □

With updates clearly defined, we are ready to move to define the properties. We consider stability, preservation and consistency defined in [8] and leave the composability property, which is arguably too strong. The three properties are previously defined in the case where the target model is an abstraction of the source model, here we adapt the definitions to symmetrical cases.

Stability says that if no model is updated, the synchronizer should update no model.

Property 1 (Stability)
$R(m, n) \Rightarrow sync(m, n, m, n) = (m, n)$

Preservation requires user updates should be preserved during synchronization. In other words, when users modify a data item to some specific valuess, the synchronizer should not modify the data item to any other value.

Property 2 (Preservation)
If $sync(m, n, m', n') = (m'', n'')$, u_m *is the least update from* m *to* m', *we have* $u_m(m'') = m''$

If $sync(m, n, m', n') = (m'', n'')$, u_n *is the least update from* n *to* n', *we have* $u_n(n'') = n''$

Consistency requires the synchronizer to produce consistent result.

Property 3 (Consistency)
$sync(m, n, m', n')$ *is defined* $\Rightarrow R(sync(m, n, m', n'))$

3 Properties of Bidirectional Model Transformation

Perdita Stevens [7] also proposes three properties to ensure a predictable behavior of bidirectional model transformations. Two of the properties are correctness and hippocraticness. The third property, undoability, is also arguably too strong, and we do not consider it here.

Property 4 (Correctness)
$\forall m \in M, n \in N \qquad R(m, \overrightarrow{R}(m, n))$
$\forall m \in M, n \in N \qquad R(\overleftarrow{R}(m, n), n)$

Property 5 (Hippocraticness)
$R(m, n) \Rightarrow \overrightarrow{R}(m, n) = n$
$R(m, n) \Rightarrow \overleftarrow{R}(m, n) = m$

4 Algorithm

The basic idea of the algorithm is to first convert the model in one side to the other side using bidirectional transformation, then use a three-way merger [3] to reconcile

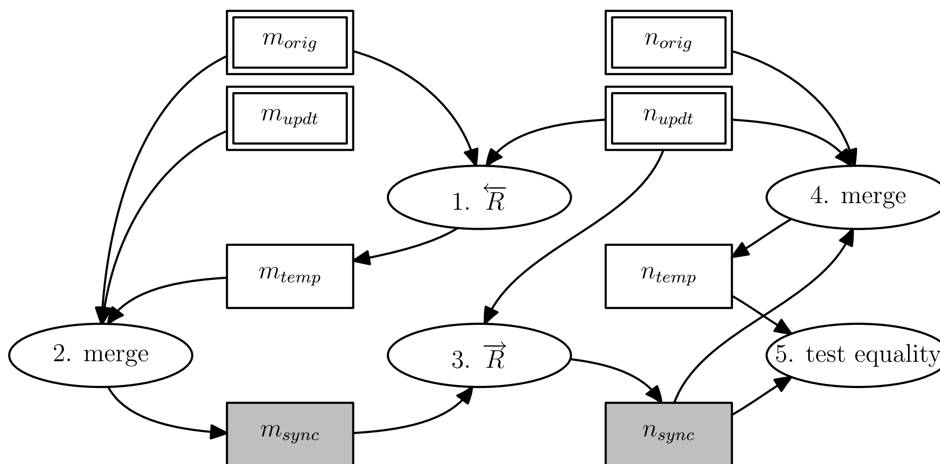

Fig. 1. The Synchronization Algorithm

the updates, and transform back using the opposite transformation. The detailed algorithm is shown in Figure 1.

Initially, we have the original models m_{orig}, n_{orig} and the updated models m_{updt}, n_{updt}. First we use $\overleftarrow{R}$ to propagate the updates on n_{updt} to m_{orig} and we get m_{temp}. Then we invoke a three-way merger to merge m_{orig}, m_{updt} and m_{temp}.

A three-way merger is a partial function $merge \in M \times M \times M \to M$ that takes a reference model m_o and two updated models m_a and m_b diverged from m_o, and produced a new model m'_o where the updates in m_a and m_b are reconciled. Suppose u_a is the least update from m_o to m_a and u_b is the least update from m_o to m_b, the *merge* function will ensure the following:

- $merge(m_o, m_a, m_b)$ is not defined iff u_a and u_b conflict.
- $merge(m_o, m_a, m_b) = m'_o \Rightarrow (u_a \circ u_b)(m_o) = m'_o$.

Back to our algorithm, here m_{updt} contains the update on m_{orig} and m_{temp} contains the update transformed from n_{orig}. After we merge them using m_{orig} as a reference model, we can get m_{sync} that contains updates from both sides.

When we have a synchronized model m_{sync} on M side, we can perform $\overrightarrow{R}$ to get a synchronized model n_{sync} on N side, and the n_{sync} should contains updates from both side.

Now we have two synchronized models where the updates are propagated. It looks that we have performed enough steps to finish the algorithm. However, the above steps is not always able to detect 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 following example. Suppose M contains two constants $\{a, b\}$ and N contains two constants $\{x, y\}$. The consistency relation between them is

$$\left\{ \begin{array}{c} (a, x) \\ (a, y) \\ (b, x) \end{array} \right\},$$

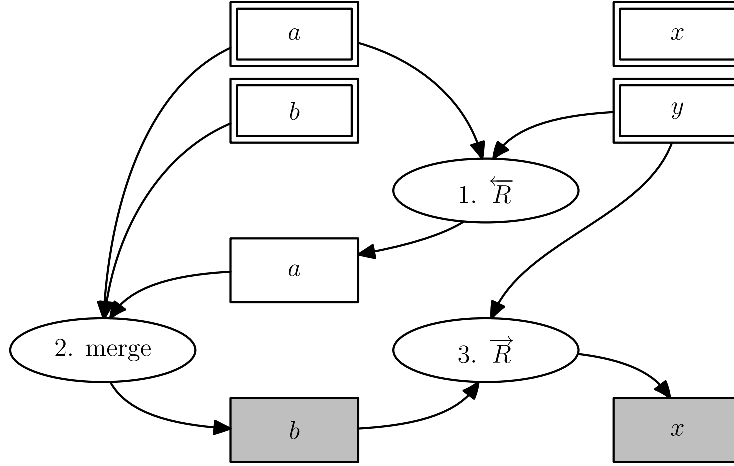

Fig. 2. An Example Violating Preservation

That is, a is related to x and y while x is also related to b. Suppose initially the two models are $m_{orig} = a$ and $n_{orig} = x$, and then m_{orig} is updated to b while n_{orig} is updated to y.

The process of this computation is shown in Figure 2. After computation, the algorithm produces b on the M side and x on the N side. However, if we check the update on the N side, we will find that x is updated to y and this updated is not preserved in the synchronized model. The property of preservation is violated.

The violation is caused by the asymmetry of M and N. Both x and y in N are related to the same element a in M. When n_{updt} is transformed to the M side, the update is not recognizable by the state-based three-way merger.

To capture such conflict, we add an additional preservation check at the end of the synchronization. As shown in the 4th and the 5th steps in Figure1. We first merge n_{updt} and n_{sync} with n_{orig} as a reference, and then compare whether the merged model n_{temp} is equal to n_{sync}. If the preserve property is satisfied, the two model should be equal, otherwise the algorithm will report an error message indicating there are conflicts.

Theorem 4.1 *If the bidirectional transformation* $(\overrightarrow{R}, \overleftarrow{R})$ *satisfies correctness and hippocraticness, we can ensure that the synchronization algorithm satisfy stability, consistency and preservation.*

Proof. Stability If we have $m_{orig} = m_{updt}$ and $n_{orig} = n_{updt}$, then we have $R(m_{orig}, n_{updt})$. Because of hippocraticness, $m_{temp} = \overleftarrow{R}(m_{orig}, n_{updt}) = m_{orig}$. Because the least update from m_{orig} to m_{updt} and to m_{temp} are both id, $merge$ will produce the same model, that is $m_{sync} = m_{orig}$. Similarly, $n_{sync} = n_{orig}$ and the preservation check always passes successfully.

Preservation On the M side, suppose u_m is the least update from m_{orig} to m_{updt}, because $merge(m_{orig}, m_{updt}, m_{temp}) = m_{sync}$, we have $u_m(m_{sync}) = m_{sync}$. Similarly, suppose u_n is the least update from n_{orig} to n_{updt}, because $merge(n_{orig}, n_{updt}, n_{sync}) = n_{temp} = n_{sync}$, we have $u_n(n_{sync}) = n_{sync}$.

Consistency Because $\overrightarrow{R}(m_{sync}, n_{updt}) = n_{sync}$, we have $R(m_{sync}, n_{sync})$. □

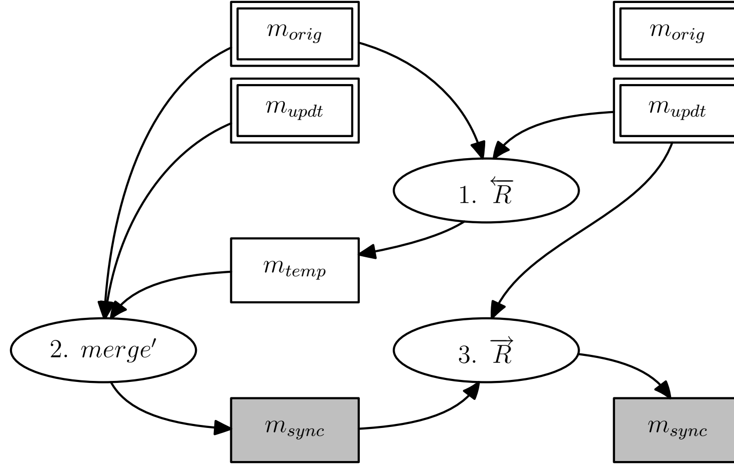

Fig. 3. The Synchronization Algorithm for Conflict Resolving

5 Conflict Resolving

One important issue in synchronization is how to resolve conflicts in the two updated models, automatically or with user intervention. A full discuss of conflict resolving relates to conflict presentation and interaction, which is beyond the scope of the paper. In this section we consider a simple automatic resolving strategy: overwriting all conflicting updates on one model with the updates on the other model.

Let us first consider the case where the updates in M take priority. Because bidirectional transformation describe the transformation symmetrically, we can just swap M and N when we need N to take priority.

Because of the updates in N may be overwritten by the updates in M, we need to loosen the preservation property to allow loss of updates on the N side. The loose preservation property only requires to preserve updates on the M side, as the following.

Property 6 (Loose Preservation for Conflict Resolving)
If u_m is the least update from m to m' and $sync(m, n, m', n') = (m'', n'')$, we have $u_m(m'') = m''$

Furthermore, we need an extended merge function $merge'$ which deals with conflicting updates and gives priority to the first model.

- $merge' : M \times M \times M \to M$ is a total function.
- $merge'(m_o, m_a, m_b) = m'_o \Rightarrow (u_a \circ u_b)(m_o) = m'_o$.

The algorithm for the updated model is shown in Figure 3. This algorithm is similar to the original one, except that we use $merge'$ instead of $merge$ and we do not post-check preservation. We can similarly prove that the algorithm ensure stability, consistency and the loose preservation if the bidirectional transformation satisfies correctness and hippocraticness.

6 Related Work

Pierce and et al. [6] propose the Harmony framework, which also addresses the issue of supporting synchronization from bidirectional transformations. Compare to our work, Harmony emphasizes on totality, but requires users to design middle model and two transformation which relates the middle model to the original two models respectively. In this way Harmony can achieve totality, but it requires more programming work to design the middle model and code the two transformations.

Antkiewicz and Czarnecki discuss various design decisions of synchronizers in their work[1]. Their work classifies synchronizers into different types using different design decisions. Use their classification, our synchronization algorithm can be classified as “bidirectional, non-incremental, and many-to-many synchronizer using artifact translation, homogeneous artifact comparison and reconciliation with choice”.

7 Conclusion

In this paper we propose an approach that wraps a bidirectional transformation program into a synchronizer for simultaneous updates. Our approach is *general*, in the sense that it allows any bidirectional transformation, and *predictable*, satisfying the model synchronization properties: consistency, stability and preservation.

Our approach is built upon idempotent updates. However, in the real world many updates cannot easily be presented as idempotent functions. For example, “inserting the item a into a list at index 2” is not an idempotent function. In our future work we plan to adopt more general definitions of updates (e.g., considering updates as arrows in a graph [2]), and extend our synchronization framework to more general cases.

References

[1] Antkiewicz, M. and K. Czarnecki, *Design space of heterogeneous synchronization*, in: *Proc. 2nd GTTSE*, 2007, pp. 3–46.

[2] Diskin, Z., *Algebraic models for bidirectional model synchronization*, in: *Proc. 11th MoDELS*, 2008, pp. 21–36.

[3] Khanna, S., K. Kunal and B. C. Pierce, *A formal investigation of diff3*, in: Arvind and Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2007.

[4] Kindler, E. and R. Wagner, *Triple graph grammars: Concepts, extensions, implementations, and application scenarios*, Technical Report tr-ri-07-284, University of Paderborn (2007).

[5] Object Management Group, *MOF QVT final adopted specification*, http://www.omg.org/docs/ptc/05-11-01.pdf (2005).

[6] Pierce, B. C., A. Schmitt and M. B. Greenwald, *Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data*, Technical Report MS-CIS-03-42, University of Pennsylvania (2003).

[7] Stevens, P., *Bidirectional model transformations in QVT: Semantic issues and open questions*, in: *Proc. 10th MoDELS*, 2007, pp. 1–15.

[8] Xiong, Y., D. Liu, Z. Hu, H. Zhao, M. Takeichi and H. Mei, *Towards automatic model synchronization from model transformations*, in: *Proc. 22nd ASE*, 2007, pp. 164–173.