# Translatability and Translation of Updated Views in ModelJoin

Erik Burger and Oliver Schneider

Karlsruhe Institute of Technology, Karlsruhe, Germany
burger@kit.edu, oliver.schneider@student.kit.edu

**Abstract** The ModelJoin language offers the definition of views that combine information from heterogeneous models. These views are currently realised by unidirectional transformations. Thus, updates to the views are not translated back to the models. In this paper, we study the view-update problem for ModelJoin view definitions. We propose translation strategies for view updates, and show that generated model constraints can be used to decide whether updated views can be translated. We provide a transformation for deriving a set of OCL constraints to check for translatability. For untranslatable cases that can be made translatable with minor fixes to the view, we provide algorithms for automatic fixes. The constraints are evaluated in two case study examples. The evaluation shows the applicability of the translation strategies, and the algorithms for automatically checking and restoring the translatability. Most of the consistent update sequences could be translated, and all inconsistent updates could be identified.

**Keywords:** view-based modelling · view-update problem · editability of views on models

## 1 Introduction

In the development process of modern software systems, multiple models are used to describe different system aspects and abstraction levels, such as component models, class diagrams, performance and reliability models. Even programme code can be seen as a software model describing the implementation. *View-centric* approaches combine information from one or multiple models into *views*, which serve as the single mechanism for displaying and manipulating information. To define these views quickly, the view definition language *ModelJoin* offers an SQL-like syntax for the specification of both the metamodel of a view (the view type) and the model transformation for creating the view. ModelJoin's goal is to offer the easy creation of custom, always up-to-date and consistent views of the whole software system. However, the *View-Update-Problem* arises: How can updates to a view be translated back to the underlying models?

In this paper, we study the view-update problem ModelJoin. This includes finding strategies to decide if an update operation on a view can be translated back to the source models, and developing mechanisms to translate view updates

The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-319-42064-6_4

to source model updates. For this purpose, we formalize the view-update problem for the Ecore metamodel and ModelJoin. Properties for the *update translation* must be found, such that the update translation satisfies the users' expectations. For example, view updates should not have unexpected side effects, or change the view in an unwanted way. To specify the effects of model updates, we develop a formal abstract syntax first. Then, we check the translatability of updated target models using OCL constraints in the view metamodel, which are derived from a ModelJoin view definition. To evaluate the applicability of the translatability check, we have implemented the proposed algorithms prototypically and evaluated them based on two case studies in component-based software development.

This paper is structured as follows: In section 2, we present the foundations, most notably the ModelJoin language, and formulate the view-update problem for Ecore. Section 3 describes the scheme for deriving OCL constraints from a ModelJoin definition. In section 4, we propose algorithms for automatically restoring some of these constraints. The findings are evaluated in section 5. Section 6 contains related work. An outlook on future work and the conlusion (section 7) complete this paper.

## 2 Foundations

### 2.1 Set Notation for Ecore-based Metamodels and Models

We use the set notation of Ecore-based metamodels and their instances as introduced in our previous work [8], which is based on the set notation for EMOF as defined in the OCL standard [21]. We will only reproduce the parts here which are relevant for the remainder of this paper. A metamodel is a structure $M := (\textsc{Class}, \textsc{Att}, \textsc{Ref}, \mathit{associates}, \mathit{multiplicites}, \prec)$, consisting of the sets for classes, attributes, and references, the function *associates*, which maps references to the pair of classes between which the reference exists, the function *multiplicities*, which assigns multiplicities to features, and a generalization hierarchy $\prec$. $I(M)$ is the set of all possible instances of a metamodel $M$. An actual model is expressed as a *snapshot* $\sigma = (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}, \sigma_{\textsc{Ref}})$. These three functions describe the instances of classes and values of attributes and references. An instance of a class $c$ is written as $\underline{c}$.

### 2.2 ModelJoin

ModelJoin [8] is a DSL for the definition of views on heterogeneous models, i.e., models that are instances of different metamodels. It draws an analogy between metamodels and relational databases in the sense that it also offers several join operators and further operators for projection, selection, and aggregation. Similar to the way an SQL query defines the table schema of the result set as well as the contents of a table, a ModelJoin query defines a target metamodel, and the result set, which is an instance of the target metamodel. In this understanding, a view is just a special kind of target model which is defined by a query. The

semantics of ModelJoin have been defined formally [8], but semantics for updates on views have not been defined yet.

In this paper, we use the same notation as in the ModelJoin specification [7] with some minor enhancements and changes:

1. We write $M_s$ for the source metamodels and $M_t$ for the target metamodels.
2. We write $m_s \in I(M_s)$ for source models, and $m_t \in I(M_t)$ for the target model.

We introduce a *trace model* $M_\sim$, which is part of the target model. The trace model is non-editable and should form a explicit representation of the mapping relation between the source and target class instances. The function *mapsTo()* can be used to check whether two elements are mapped by an instance of the trace model.

**Definition 1 (Trace model).** *We divide the target metamodel into a view metamodel $M_v$ and a* trace metamodel *$M_\sim$ with $M_t = M_v \cup M_\sim \wedge M_v \cap M_\sim = \emptyset$ The class instances in the models are divided into a view model $m_v$ and a* trace model *$m_\sim$ according to their metamodel membership. For a given target model $m_t \in I(M_t)$ we use the following notation: $m_v = [m_t]_v, m_\sim = [m_t]_\sim$*

### 2.3 The View-Update Problem for ModelJoin

The view-update problem [2] has been studied extensively for relational databases. In metamodelling, the modifications that can be applied to metamodels and models can also be described using the standard CRUD operations [6]. To formally define the semantics of these operations, we adopt the GetPut and PutGet properties by Foster et al. [13] and Diskin [11].

**Definition 2 (View-Update-Problem).** *The* View-Update-Problem *VUP($Q$) for a given ModelJoin view definition $Q \in M_s \times M_t$ is to decide if there exists a translation $\overleftarrow{q} : I(M_t) \times I(M_s) \to I(M_s)$ such that the following two properties hold for all views in $V = q\,[I(M_s)]$:*

(i) *Translating an unmodified target model, does not change the source model:*

$$\forall m_s \in I(M_s) \colon \overleftarrow{q}\,(q(m_s), m_s) = m_s \qquad \text{(GetPut)}$$

(ii) *Translating a modified target model and querying the result, yields the translated modified target model.*

$$\forall m_s \in I(M_s), \forall m_t \in V \colon q(\overleftarrow{q}\,(m_t, m_s)) = m_t \qquad \text{(PutGet)}$$

A model $m_t \in I(M_t)$ is called *translatable* if there exists a translation $\overleftarrow{q}$ that satisfies the properties GetPut and PutGet. In the case of ModelJoin, a translation $\overleftarrow{q}$ should reflect the semantics of its query function $q$. If each ModelJoin operator has a fixed translation semantics, the set of translatable target models forms only a subset of all obtainable target models. Fixing the

semantics of the translation function for each ModelJoin operation makes the translation predictable and comprehensible for the user. Therefore, we want to formulate the View-Update-Problem for a restricted set of translatable target models.

**Definition 3 (Restricted View-Update-Problem).** *The* restricted View-Update-Problem *(rVUP(Q)) for a given ModelJoin view definition Q is to find a restricted subset* $V_r : I(M_s) \to \mathbb{P}(I(M_t))$ *with the following properties:*

(i) *A translation* $\overleftarrow{q} : V_r\,[I(M_s)] \times I(M_s) \to I(M_s)$ *for q exists:*

$$\langle m_t, m_s \rangle \to m'_s \qquad \text{(EXISTENCE)}$$

(ii) $V_r$ *contains all unmodified target models:*

$$\forall m_s \in I(M_s) \colon q(m_s) \in V_r(m_s) \qquad \text{(TOTALITY)}$$

(iii) $\overleftarrow{q}$ *conforms to the* GetPut*-Property:*

$$\forall m_s \in I(M_s) \colon \overleftarrow{q}\,(q(m_s), m_s) = m_s \qquad \text{(GetPut)}$$

(iv) $\overleftarrow{q}$ *conforms to the* PutGet*-Property for all views in* $V_r$*:*

$$\forall m_s \in I(M_s), \forall m_t \in V_r(m_s) \colon \left[q(\overleftarrow{q}\,(m_t, m_s))\right]_v = m_t \qquad \text{(PutGet)}$$

*A set* $V_r$ *together with a translation* $\overleftarrow{q}$*, which solves rVUP(Q) is called a solution of the problem.*

The *rVUP*(*Q*) is solvable for all $Q$ because the set $V_r(m_s) = \{q(m_s)\}$, together with the translation $\overleftarrow{q}\,(m_t, m_s) = m_s$, is a trivial solution. This solution does, however, not allow any updates to the target model. Thus, we present a solution that allows useful target model updates and reflects the semantics of the ModelJoin operators. To verify these properties, we evaluate our solution in a case study in section 5.

## 3 Constraints for Translatable Views

In this section, we will present a scheme to derive a set of OCL constraints from a ModelJoin view definition, such that the possible instances in the target model are limited to those that can be translated to a source model. We show that the fulfilment of these constraints is a sufficient condition for the translatability of an updated target model. We further show that an unmodified target model fulfils all constraints.

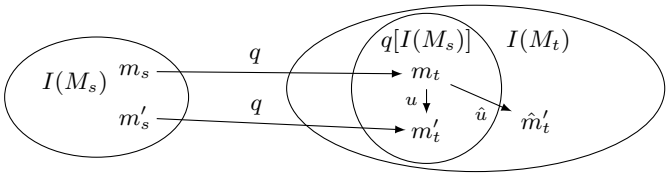


**Figure 1.** The Update operation $u$ is translatable because for $m'_t$ a corresponding source model $m'_s$ exists. The update operation $\hat{u}$ is, however, untranslatable, since $\hat{m}'_t$ has no corresponding source model.

### 3.1 Motivating Example

The set of target models $q\,[I(M_s)]$ of a query function $q$ can be a real subset of all possible target class instances $I(M_t)$. A target model $m_t \in I(M_t) \setminus q\,[I(M_s)]$ is not translatable, since no source model $m_s \in I(M_s)$ with $m_t = q(m_s)$ exists (see Figure 1).

An example for a ModelJoin view definition $Q$ with a real subset relationship $q\,[I(M_s)] \subset I(M_t)$ is given in Listing 1. In this example, the attribute commons.NamedElement.name of the source class gets mapped to two different target class attributes: name and alias. The update operation given in Listing 2 cannot be translated, since no source model exists for such a target model.

```
1 theta join classifiers.Interface with uml.Interface
2 where "classifiers.Interface.name = uml.Interface.name" as jointarget.
      Interface {
3     keep attributes commons.NamedElement.name as name
4     keep attributes commons.NamedElement.name as alias
5 }
```

**Listing 1.** Example where the views are a real subset of all possible target models.

```
1 create jointarget.Interface { name: "StoreIf", alias: "Store" }
```

**Listing 2.** Untranslatable update operation for the view definition in Listing 1

Such untranslatable update operations shall be forbidden. Therefore, the metamodel needs to be extended by the constraint $\sigma_{\text{ATT}}(\mathsf{name})(\underline{c}) = \sigma_{\text{ATT}}(\mathsf{alias})(\underline{c})$ for all instances $\underline{c} \in I(\mathsf{jointarget.Interface})$. We will formulate such constraints in OCL. Ideally, the constraints characterize exactly the set $q[I(M_s)]$, which contains all translatable views. Since we want to fix the translation semantics for each ModelJoin operator in $Q$, we restrict the set $q[I(M_s)]$ further to a set $V_r$ like in the definition for $rVUP(Q)$.

### 3.2 Constraints Creation

**3.2.1 Meta Variables** The OCL expressions that characterize the translatable view instances depend not only on the values of attributes in the view itself, but

also to values in the source classes. In the aforementioned example in Listing 1, the identity of source.name and target.name should be formulated. Since the source models are not updateable by any operation, a simple OCL formulation would fix the value of target.name to the original source value and make it unchangeable. To avoid this issue, we introduce *meta variable expressions*, which will be replaced with a given definition during the execution of the query.

**Definition 4 (meta variable for attributes).** *Let* $c \in$ Class *be a class and* $a : t_{c'} \to t \in \text{Att}_c^*$ *be an attribute, then the initial value of the* meta variable $var_a : \mathit{Expr}_{t_c} \to \mathit{Expr}_t$ *is defined as* $var_a(\alpha) = \alpha.a$

*In addition, we define the interpretation function* $\underline{var}_a : I(c) \to I(t)$ *of* $var_a$ *as* $\underline{var}_a^\sigma(\underline{c}) = I\,[[var_a(\mathfrak{v})]]\,(\langle\sigma, \{\mathfrak{v} \to \underline{c}\}\rangle)$.

A meta variable is called *correct* if it evaluates to the same value as the corresponding attribute of a target model (Get-Equality), and after the translation to a source model, the corresponding attribute has the same value as the meta variable (Put-Equality). Meta variables for references are defined analogously.

**3.2.2 OCL Expression Rewriting** Some ModelJoin operations, such as calculate attributes or theta joins, can use OCL expressions to describe the values or instances of the target model. These expressions depend on values of source model elements. If we want to reason about the value of this expressions after the translation without performing the translation, the expressions have to be rewritten to depend on the values of the corresponding target model elements. We have defined rewriting rules for updating instances that are already mapped to source model instances, and for new instances. We have shown that the rewritten expressions fulfil Get-Equality and Put-Equality if the used meta variables fulfil these properties. For a given OCL expression $\theta$ we write $var_\theta$ for the rewritten expression. If the class type of the free variable $v$ in $\theta$ changes from $c$ to $c_t$ in the rewritten expression, we write $var_{\theta,v}^{c \to c_t}$. The complete definitions and proofs of these properties are omitted here for brevity, but can be found in [22, sect 4.3].

**3.2.3 Example: Theta Join** In Listing 3, a generated OCL constraint is shown for the theta join operator.

```
1 context c_⋈
2 inv keepMappingPairs:
3 Instances_c1 -> forAll(left | Instances_c2 -> forAll(right|θ(left, right) = var_θ(left,right)))
```

**Listing 3.** OCL constraint for keeping mapping pairs

The constraint in Listing 3 ensures that the mapping of source to target instances does not change when the target model is updated. Therefore, the join condition $\theta$ should hold after the update exactly for those instances for which it did before the update. Deleted instances must however be excluded.

In case that the meta variable $var_\theta$ is not defined or not well typed, a constraint must be created that forbids new instances of the respective class. If the target rewrite variables $var^{c_1 \to c_t}_{\theta(\mathsf{self},\mathsf{right})}$ and $var^{c_2 \to c_t}_{\theta(\mathsf{self},\mathsf{right})}$ are defined and well-typed, new target classes must satisfy the constraints in Listing 4. For simplicity, the placeholder *isNew(c)* denotes an OCL expression checking if the instance $c$ was newly created in the view, and the placeholder $Instances_c$ denotes an OCL expression returning all the instances of $c$ that exist after the update.

```
1  context c_t
2  inv newTargetInstancesLeft:
3  isNew(self) implies Instances_c1->forAll(left|not var^{c2->ct}_{θ(left,self),self})
4  inv newTargetInstancesRight:
5  isNew(self) implies Instances_c2->forAll(right|not var^{c1->ct}_{θ(self,right),self})
6  inv newTargetInstances: isNew(self) implies c_t.allInstances->select(other |
7      c_⋈.allInstances()->select(tj|tj.target = other)->isEmpty() and other <> self
8  )->forAll(other|not var^{c1->ct,c2->ct}_{θ(self,other),self,other})
9  inv isJoinConforming: isNew(self) implies let self2 = self in var^{c1->ct,c2->ct}_{θ(self,self2),self,self2}
10 -- New instances may only be created if the source class is not mapped elsewhere:
11 inv noConflictsWithOtherMappings:
12 ∃c'_t ∈ Class(c'_t ≠ c_t ∧ (c_1 ~⋈ c'_t ∨ c_1 ~⋈ c'_t) ⇒ c_⋈.allInstances()->
13 select(oj|oj.target = self )->notEmpty())
```

**Listing 4.** OCL constraint for target instances

### 3.3 Deciding Translatability

A ModelJoin expression is a composition of different subexpressions. We can show by induction over all subexpressions that two implications are valid: If the OCL-constraints hold for an target model, then it is translatable. Furthermore, the OCL-constraints hold for a target metamodel obtained from a query.

The definitions and proofs for all ModelJoin expressions are omitted here for the sake of brevity and can be found in [22]. We demonstrate at the example of the theta join operator, the most general operator, how translatability is proven.

**Definition 5 (Translation for theta join).** *Let* $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$ *be a theta join operator and* $\underline{c}_t \in \sigma_{\text{Class}}(c_t)$ *be a target class instance. The instance* $\underline{c}_t$ *should be translated according the following rule:*

(i) *If there exists no trace class instance* $\underline{c}_\bowtie \in \sigma_{\text{Class}}(c_\bowtie)$ *with* $L(\mathsf{target})(\underline{c}_\bowtie) = \underline{c}_t$*, then the create class instance operations* $\text{CREATE}_{\text{Class}(c_1)}(V_1)$*,* $\text{CREATE}_{\text{Class}(c_2)}(V_2)$ *with* $V_1 = \emptyset$ *and* $V_2 = \emptyset$ *are emitted.*

(ii) *For each* $\underline{c}_\bowtie \in \sigma_{\text{Class}}(c_\bowtie)$ *with* $L(\mathsf{target})(\underline{c}_\bowtie) = \emptyset$*, the following delete class instance operations* $\text{DELETE}_{\text{Class}(c)}(\underline{c}_1)$ *and* $\text{DELETE}_{\text{Class}(c)}(\underline{c}_2)$ *where* $\underline{c}_1 \in L(\mathsf{left})(\underline{c}_\bowtie)$ *and* $\underline{c}_2 \in L(\mathsf{right})(\underline{c}_\bowtie)$ *are emitted. If the deleted class instance* $\underline{c}_1$ *was linked by a reference* $r = \langle c, c_1 \rangle \in \text{Ref}$ *of instance* $\underline{c}$ *then the update operation* $\text{DELETE}_{\text{Ref}(r)}(\underline{c}, \underline{c}_1)$ *should be emitted. The corresponding delete operation is emitted for deleted instances of class* $c_2$*.*

**Theorem 1 (Induction for theta join)** *Let $q : I(M_{source}) \to I(M_{target})$ be a ModelJoin expression with a set of OCL-Constraints $C$ for which the Induction statement holds. If $q$ is extended by an arbitrary theta join expression to $q' : I(M_{source}) \to I(M'_{target})$ with the set of OCL-Constraints $C'$, then the Induction statement holds for $q'$ and $C'$. More precisely:*

(i) *For all source model instances $m_s$ the resulting target model instance $m_t = q'(m_s)$ satisfies all OCL constraints in $C'$.*
(ii) *For all pairs $\langle m_s, m'_t \rangle \in I(M_{source}) \times I(M'_{target})$ a target model instance $m'_t$, that satisfies $C'$, is translatable.*

*Proof.* Let $c_1, c_2 \in$ Class be the source classes and $c_t \in$ Class be the target class of the theta join operator: $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$.

We first prove (i). Let $m_s \in I(M_{source})$ be a source model instance. $m_t = q'(m_s)$ satisfies all constraints in $C$ because all constraints in $C$ do not depend on instances of $c_t$ and all instances of other classes then $c_t$ satisfy $C$ according to the premises. So it just has to be shown that all new constraints in $C' \setminus C$ are satisfied.

The keepMappingPairs invariant is true, because of the Get-Equality of $var_{\theta(\mathsf{left,right})}$. All invariants in the context of $c_t$ are immediately true, because the set $c_\bowtie$`.allInstances()->select(j | j.target = self)` in *isNew*(`self`) cannot be empty, since all instances $\underline{c}_t \in \sigma_{\text{Class}}(c_t)$ are a target of a join trace instance $\underline{c}_\bowtie \in \sigma_{\text{Class}}(c_\bowtie)$.

We show (ii) by constructing a translation $\overleftarrow{q'}$ and show that $c_t$ is translatable. Let $\underline{c}_t$ be an instance of $c_t$ in $m'_t$. Since $c_t$ has no attribute values by default, no attributes can be changed and the invariant keepMappingPairs ensure that the mapping between the existing source class instances and the target class instance does not change by other updates. This follows directly from the Put-Equality of $var_{\theta(\mathsf{left,right})}$.

Consider the case that $\underline{c}_t$ has no corresponding mapping instance $\underline{c}_\bowtie$, which means it was newly created. For new instances, at least one of the following update operations should be created: $\text{CREATE}_{\text{Class}(c_1)}(V_1)$, $\text{CREATE}_{\text{Class}(c_2)}(V_2)$ according to Definition 5. Because of the isJoinConforming invariant and the Put-Equality of $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\mathsf{self,self2}),\mathsf{self,self2}}$, we have $\underline{\theta(\mathsf{left,right})}^{\sigma}_{\mathsf{left,right}}(\underline{c}_1, \underline{c}_2)$ for the newly created instances by the create operations, leading to $\underline{c}_1 \sim_{\underline{\bowtie}} \underline{c}_t$ and $\underline{c}_2 \sim_{\underline{\bowtie}} \underline{c}_t$. Because of the newTargetInstancesLeft and newTargetInstancesRight invariant and the Put-Equality of $var^{c_1 \to c_t}_{\theta(\mathsf{self,right}),\mathsf{self}}$, there exists no class instance $\underline{c}'_1$ with $\underline{\theta(\mathsf{left,right})}^{\sigma}_{\mathsf{left,right}}(\underline{c}'_1, \underline{c}_2)$, and no new other joining pairs are created. The same argument can be given for other class instances $\underline{c}'_2$.

## 4 Automatic Fixes for Untranslatable Views

Executing an update operation on the target model can lead to an untranslatable target model if constraints are violated. Further updates are required to restore translatability. Consider the example in Figure 2: An update operation $\hat{u}$ creates

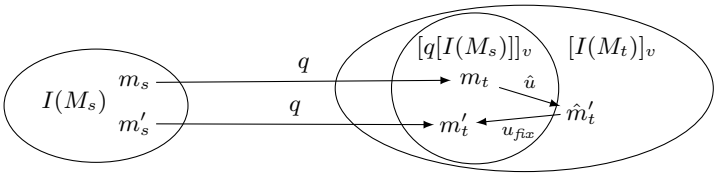


**Figure 2.** Fixing an untranslatable target model

a new target class instance. The translation of the new instance would lead to further new target elements, which are not present in the view. The corresponding constraints are violated, and $\hat{m}'_t$ is an untranslatable model. These missing target class instances could be created automatically before translation. Such an automatic fix $u_{fix}$ could be proposed to the user after the update operation or before the translation. The user could accept the fix, repair the target model manually or undo his change, if it was unintended, and gain a translatable model $m'_t$. The proposed automatic fix should not undo the update operation, but apply a minimal set of necessary changes instead to reflect the update in a translatable target model. In the example, the automatic fix should not delete the new target class instance, but should create the missing target class instance.

### 4.1 Automatic Creation of Target Class Instances

The PutGet-Property states that for each source class instance pair, for which the join condition holds, a corresponding target class instance exists after translation. In other words, querying the unmodified source model after a translation does not create any new target class instances. However, the missing target class instances could be created automatically at update translation, because these can be derived from the corresponding source class instances.

We will show this again at the example of the theta join. Algorithm 1 fulfils this purpose: We check for each new target instance if the translation would lead to new left or right source instances. This is the case if there are no source instances that map to the new instance. If this is the case, we check if all target class instances for the new source class exist. In createMissingJoinTargetsForNewLeft, we first check for each existing right source instance, for which the join condition with the new left source instance holds, if there is a new target instance. If this is not the case, we create a new target instance according to the join definition from the updated attribute values and links obtained from the meta variables. Not only existing right source instances could form new join pairs with the new left source class, also new created right source instances could be join partners. Therefore, we collect all new target instances for which the join condition holds for its right source class in matchingRightTarget. For each instance in matchingRightTarget, we check if a target instance exists. If not, we create the missing target class instance with the updated attribute values and links obtained from the meta variables. The procedure createMissingJoinTargetsForNewRight behaves symmetrically.

**Algorithm 1** Create missing target instances for new left source instance

**procedure** createMissingJoinTargetsForNewLeft(left : $c_t$)
  matchingRight ← $c_2$.allInstances()->select(right | $var^{c_2 \to c_t}_{\theta(\mathsf{right},\mathsf{right}),\mathsf{right}}$)
  **for** right ∈ matchingRight **do**
    mappingTargets ← $c_t$.allInstances()->select(target | *isNew*(target))
    ->select(target | $mapsTo^{c_t,\theta}_{\langle c_1,c_t\rangle}(\mathsf{left},\mathsf{target})$ **and** $mapsTo^{\theta}_{\langle c_2,c_t\rangle}(\mathsf{right},\mathsf{target})$)
    **if** mappingTargets->isEmpty() **then** createNewTargetClassInstance
    **end if**
  **end for**
  matchingRightTarget ← $c_t$.allInstances()->select(right |
    *isNew*(right) **and** $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\mathsf{left},\mathsf{right}),\mathsf{left},\mathsf{right}}$)
  **for** right ∈ matchingRightTarget **do**
    mappingTargets ← $c_t$.allInstances()->select(target | *isNew*(target))
    ->select(target | $mapsTo^{c_t,\theta}_{\langle c_1,c_t\rangle}(\mathsf{left},\mathsf{target})$ **and** $mapsTo^{c_t,\theta}_{\langle c_2,c_t\rangle}(\mathsf{right},\mathsf{target})$)
    **if** mappingTargets->isEmpty() **then** createNewTargetClassInstance
    **end if**
  **end for**
**end procedure**
create new target class instance with source attributes $V_1$, $V_2$ and links $L_1$, $L_2$ according to the definition of $c_t$.
**procedure** createNewTargetClassInstance
  $V_1 \leftarrow \{v_a \mid a \in \text{Att}^*_{c_1},\ v_a = var^{c_t}_a(\mathsf{left})\}$
  $V_2 \leftarrow \{v_a \mid a \in \text{Att}^*_{c_1},\ v_a = var^{c_t}_a(\mathsf{right})\}$
  $L_1 \leftarrow \{l_r \mid r = \langle c_1, \hat{c}\rangle \in \text{Ref},\ l_r = var^{c_t}_r(\mathsf{left})\}$
  $L_2 \leftarrow \{l_r \mid r = \langle c_2, \hat{c}\rangle \in \text{Ref},\ l_r = var^{c_t}_r(\mathsf{right})\}$
**end procedure**

### 4.2 Further Fixes

In addition to the creation of new target instances, *derived model elements* can also be re-calculated automatically. Therefore, they have to be made non-editable for the user and re-evaluated every time a view update is translated. To preserve the PutGet property, the fixes are applied before the execution of the translation through the rewrite mechanisms and meta variables.

Furthermore, changes to *attributes* in the view can affect further target model elements that depend on the same source attributes. These updates can be proposed to the user automatically, so that they need not be executed manually.

## 5 Evaluation

For the evaluation of the approach presented in this paper, we have extended the ModelJoin prototype.[1] Constraints are generated automatically for the target metamodel. We implemented this functionality as Xpand templates. Using the standard OCL engines of Eclipse, the translatability of views can be checked by invoking an OCL validation on the respective model.

[1] `https://sdqweb.ipd.kit.edu/wiki/ModelJoin`

### 5.1 Case Studies

Since ModelJoin is not used in industrial applications yet, we use general modeling examples instead. For the case study, we have therefore chosen the Common Component Modelling Example (CoCoME) [15] and the Media Store example from the upcoming Palladio Book [3].

**5.1.1 CoCoME** CoCoME describes a trading system for supermarkets. The system architecture is described as UML component models, the implementation in Java [15]. Using the Java Model Parser and Printer (JaMoPP), the Java code converted from textual representation to a model representation and vice versa.

With ModelJoin, a UML view of the trading system can therefore be extended with implementation details using the model representation of the source code. Furthermore, translated updates cannot only update the UML models, but also the Java source code. We assume the software engineer wants to create a view containing the interfaces with method signatures from the Java source code and the providing compontents from the UML component model, and therefore creates the ModelJoin view definition. The following actions were evaluated: changing the name of an interface, adding an interface, and deleting an interface.

**5.1.2 MediaStore** As a further example, we use the media store example from the upcoming Palladio Book [3]. The Media Store example describes a file hosting system for audio files. There is an example project for Palladio with System, Execution Environment, Component Allocation and Usage Models.

For the evaluation, we have used the database cache example from the Palladio Media Store example project. We assume that the caching behavior was modeled by a UML activity diagram and then the corresponding SEFF was derived from it. While both models describe the same behavior, they contain different information. The SEFF contains the branch probabilities, random variables and hardware resource demands needed for the performance analysis. The activity diagram contains requirements and details about the behavior to implement. The following actions were evaluated: renaming an action, deleting an edge, and creating a new action.

### 5.2 Conclusion of the Case Studies

For each of the actions, we formulated an expected behaviour and compared it with the translation generated by our implementation. The case study examples have confirmed that most of the target model elements are updateable, if the view definitions are designed with updatability in mind. All update operation on the view were translatable in the CoCoME case. The constraint checking prevented the translation of inconsistent or ambiguous updates. In five of the six cases the actual translation result meets our intuitive expectation.

For the CoCoME case, the translation of the actions leads to the following results: Changing the name of the interface was translated to change of the

corresponding source attributes. Creating a interface with method signatures creates the component interface and the corresponding interface in the Java model after translation. It forms, however, no complete compilation unit, and has to be completed by the user. Deleting the interface and the referenced model elements was translated to a deletion of the component interface and the Java model interface, as well as the removing of the links to the corresponding referenced source model elements. The ModelJoin keep reference operation is problematic: A keep reference operation only creates target model instances for class instances included in a given source model reference. Unreferenced class instance are not present in the target model, so no unreferenced target class instance can exist in the target model. To fulfill the PutGet-Property, it is not possible to remove the links to a target class instance without deleting the linked target class as well. The translation however only removed the linked class instance, but does not delete the corresponding source class instance.

In the Palladio Media Store example case, the rename and deletion action led to similar results as in the CoCoME case. The creation of a new action showed that the translatability of new target class instances is problematic if a source class is used in multiple join conditions, and not all elements in join conditions have target elements in the view. Source attribute expressions, as proposed in [22], are one way to supply the value for missing target attributes, however there is no construct for references yet. If a join expression uses a value from a referenced class instance, the expression cannot be rewritten if the reference itself is not mapped to a target model reference. In this case, the value for the join expression after the translation cannot be derived, and so the mapping for the new target class instance to source target class instances is undetermined. The case studies are described in detail in [22].

### 5.3 Limitations and Validity

Since ModelJoin is a proposal for a view definition language and there is just an experimental implementation, it is not yet used in real world cases. Therefore, the ModelJoin view definitions used in the case study are created specifically for this case study. We tried to model practical scenarios and therefore used common model examples. It is however unclear whether the results of this evaluation fulfil real-world requirements. This requires further evaluation in a real ModelJoin use case. The expected translation results, used to value the actual translation results, are not empirical researched and are chosen in an intuitive way by the authors of this paper.

## 6 Related Work

EMF views [5] implements a similar approach as ModelJoin for the definition of queries in a SQL-like manner. Updates to the views can only be translated for primitive attribute value changes. The EMF-IncQuery approach also supports

virtual views that can be derived and synchronized incrementally [10]. These approaches are however limited to views on homogeneous models.

The View-Update-Problem is well studied in the context of relational databases [2, 9, 20]. An approach for the View-Update-Problem is to target it at the syntax level of the language used to declare views. Different forms of bi-directional model transformation approaches have been developped for this purpose. Foster et al. [13] identify three major classes for theses languages: *bi-directional languages*, *bijective languages*, and *reversible languages*. ModelJoin with the translation extensions presented in this paper falls into the first category. A more comprehensive overview is given in [16]. According to this classification, ModelJoin is an approach in the technical space of MDE, which is forward, but not backward functional, has total target coverage, is not turing-complete, and has a state-based change representation. An investigation of well-behavedness properties of ModelJoin is subject of future work. The *lenses* approach by Foster et al. has also been applied to metamodel-based structures [12]. They have also been extended to support editability [17, 18]. Triple Graph Grammars can also be used to define bi-directional model transformations [14, 1]. These theoretically founded approaches could be applied to metamodels if a suited theoretical foundation for MOF and Ecore were provided, as suggested in [23]. For changes to views, change-driven approaches [4, 17, 19] could serve as a basis, since the updates are triggered by well-defined change events.

## 7 Conclusion

We have formulated the view-update problem for ModelJoin views. To check if an update target model satisfies these properties, we have chosen OCL as a validation language. We have shown that the target model is translatable if all constraints are fulfilled, and that an unmodified target model satisfies all constraints. Additionally, we have proposed specialized algorithms for fixing untranslatable target models. These algorithms can be used to make a target model translatable after applying an update operation by adapting dependent model elements. Finally, we have evaluated our approach in two case study examples. All but one chosen update operations were translatable in the case study example cases.

Our approach introduces a translation semantics for the editability of ModelJoin views. In some cases, it is possible to decide for a given metamodel and update operation, if the update operation can be translated independent of the model. It has to be studied if the generated OCL constraints can be used to show the translatability in general. We have seen that not all updated target models that fulfil the GetPut-Property can be translated. The generated constraints are too restrictive in some cases. In future work, the constraints could be relaxed more by improving the rewrite function for target class instances and introducing concepts for changing source references, similar to source attributes at update translation.

1. Anthony Anjorin et al.: Efficient Model Synchronization with View Triple Graph Grammars. In: ECMFA 2014. Ed. by Jordi Cabot and Julia Rubin. York, UK: Springer International Publishing, July 2014, pp. 1–17. URL: http://dx.doi.org/10.1007/978-3-319-09195-2_1.
2. François Bancilhon and Nicolas Spyratos: Update semantics of relational views. In: ACM Trans. Database Syst. 6.4 (Dec. 1981), pp. 557–575.
3. Steffen Becker et al.: Modeling and Simulating Software Architectures – The Palladio Approach. Ed. by Ralf H. Reussner et al. to appear. Cambridge, MA: MIT Press, 2016.
4. Gábor Bergmann et al.: Change-driven model transformations. In: Software & Systems Modeling 11.3 (2012), pp. 431–461. URL: http://dx.doi.org/10.1007/s10270-011-0197-9.
5. Hugo Brunelière et al.: EMF Views: A View Mechanism for Integrating Heterogeneous Models. In: 34th International Conference on Conceptual Modeling (ER 2015). Stockholm, Sweden, Oct. 2015. URL: https://hal.inria.fr/hal-01159205.
6. Erik Burger and Boris Gruschko: A Change Metamodel for the Evolution of MOF-Based Metamodels. In: Proceedings of Modellierung 2010. Ed. by Gregor Engels, Dimitris Karagiannis and Heinrich C. Mayr. Vol. P-161. GI-LNI. Klagenfurt, Austria, Mar. 2010, pp. 285–300. URL: http://sdqweb.ipd.kit.edu/publications/pdfs/burger2010a.pdf.
7. Erik Burger et al.: ModelJoin. A Textual Domain-Specific Language for the Combination of Heterogeneous Models. Tech. rep. 1. Karlsruhe Institute of Technology, Faculty of Informatics, 2014. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000037908.
8. Erik Burger et al.: View-Based Model-Driven Software Development with ModelJoin. In: Software & Systems Modeling 15.2 (2014). Ed. by Robert France and Bernhard Rumpe, pp. 472–496.
9. Stavros S. Cosmadakis and Christos H. Papadimitriou: Updates of Relational Views. In: J. ACM 31.4 (Sept. 1984), pp. 742–760. URL: http://doi.acm.org/10.1145/1634.1887.
10. Csaba Debreceni et al.: Query-driven Incremental Synchronization of View Models. In: Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling. VAO '14. York, United Kingdom: ACM, 2014, 31:31–31:38. URL: http://doi.acm.org/10.1145/2631675.2631677.
11. Zinovy Diskin: Algebraic Models for Bidirectional Model Synchronization. In: MoDELS 2008, Toulouse, France, September 28 – October 3, 2008. Proceedings. Ed. by Krzysztof Czarnecki et al. Berlin, Heidelberg: Springer, 2008, pp. 21–36. URL: http://dx.doi.org/10.1007/978-3-540-87875-9_2.
12. Zinovy Diskin et al.: From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In: Model Driven Engineering Languages and Systems. Ed. by Jon Whittle, Tony Clark and Thomas Kühne. Vol. 6981.

Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 304–318.
13. J. Nathan Foster et al.: Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem. In: SIGPLAN Not. 40.1 (Jan. 2005), pp. 233–246. URL: http://doi.acm.org/10.1145/1047659.1040325.
14. Holger Giese and Robert Wagner: From model transformation to incremental bidirectional model synchronization. In: Software and Systems Modeling 8 (1 2009), pp. 21–43.
15. Sebastian Herold et al.: CoCoME-the common component modeling example. In: The Common Component Modeling Example. Springer, 2008, pp. 16–53.
16. Soichiro Hidaka et al.: Feature-based classification of bidirectional transformation approaches. In: Software & Systems Modeling (2015), pp. 1–22. URL: http://dx.doi.org/10.1007/s10270-014-0450-0.
17. Martin Hofmann, Benjamin Pierce and Daniel Wagner: Edit Lenses. In: SIGPLAN Not. 47.1 (Jan. 2012), pp. 495–508. URL: http://doi.acm.org/10.1145/2103621.2103715.
18. Michael Johnson and Robert D. Rosebrugh: Unifying Set-Based, Delta-Based and Edit-Based Lenses. In: Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016. Eindhoven, The Netherlands, Apr. 2016, pp. 1–13. URL: http://ceur-ws.org/Vol-1571/paper_13.pdf.
19. Max E. Kramer: A Generative Approach to Change-Driven Consistency in Multi-View Modeling. In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures. QoSA '15. 20th International Doctoral Symposium on Components and Architecture (WCOP '15). Montréal, QC, Canada: ACM, 2015, pp. 129–134. URL: http://doi.acm.org/10.1145/2737182.2737194.
20. Jens Lechtenbörger: The impact of the constant complement approach towards view updating. In: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. PODS '03. New York, NY, USA: ACM, 2003, pp. 49–55.
21. Object Management Group (OMG): Object Constraint Language (OCL). http://www.omg.org/spec/OCL/2.4/. Version 2.4. 2014.
22. Oliver Schneider: Translatability and Translation of Updated Views in ModelJoin. Master's Thesis. Karlsruhe Institute of Technology, Dec. 2015. URL: http://sdqweb.ipd.kit.edu/publications/pdfs/schneider2015a.pdf.
23. Gabriele Taentzer et al.: A fundamental approach to model versioning based on graph modifications: from theory to implementation. In: Software and Systems Modeling 13.1 (2014), pp. 239–272. URL: http://dx.doi.org/10.1007/s10270-012-0248-x.