Karlsruhe Institute of Technology

Software Design and Quality

Translatability and Translation of Updated Views in ModelJoin

Master's Thesis of

Oliver Schneider

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr. Walter F. Tichy
Advisor:	Dr.-Ing. Erik Burger
Second advisor:	Dipl.-Inform. Jörg Henß

20. June 2015 – 19. December 2015

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 15. December 2015

. . . .

(Oliver Schneider)

Abstract

In the development process of modern software systems, different models are used to describe various system aspects and abstraction levels. The Vitruvius project uses a *view-centric* approach to combine information from multiple models into *flexible views*. To define these views easily, the new view definition language *ModelJoin* was developed. It allows to define both: the metamodel of the view and the model transformation for creating the view in an SQL-like syntax. ModelJoin allows to easily write custom, always up-to-date and consistent views of the whole software system. However the views could not be updated, because no *translation* for view updates to source model updates was defined.

In this master's thesis, the *View-Update-Problem* is studied for ModelJoin view definitions. Different translation strategies for view updates are proposed, and it is shown that model constraints can be used to decide the translatability of updated views. A scheme for deriving a set of OCL constraints from a ModelJoin view definition is provided. The derived OCL constraints can be used to check the translatability of an updated view.

A few view updates can be translated in multiple ways to source model updates. Alternative translation strategies are proposed for these updates and it is shown how the OCL constraints must be adapted to reflect the chosen strategy.

For some view updates, further view updates are necessary, to make the view fulfill the constraints for translatability. In certain situations, these updates can be automatically derived and performed before the translation. As part of the thesis, algorithms for automatically restoring constraints in common situations are given.

The constraints for the translatability test are evaluated for atomic update operations in general, and update sequences in two case study examples. The evaluation shows the applicability of the translation strategies, translatability check and algorithms for automatically restoring the translatability. Almost all consistent update sequences were able to be translated and all inconsistent updates rejected. The proposed algorithm to fix untranslatable views can also be used in there use cases. Using OCL constraints as validation specification for the translatability test allows the easy integration in existing tooling. However the derived constraints do not exactly characterize all translatable views. The fulfillment of the constraints is only a sufficient condition for the translatability of a view.

Zusammenfassung

Im Entwicklungsprozess moderner Software-System werden Modelle genutzt um unterschiedliche Systemaspekte und Abstraktionslevel zu beschreiben. Das Vitruvius-Projekt benutzt einen *Sicht-zentrierten Entwurf* um Informationen aus verschiedenen Modellen in *flexiblen Sichten* zu kombinieren. Um diese Sichten einfach definieren zu können, wurde die neue Sprache *ModelJoin* zur Spezifikation von Sichten entwickelt. Sie erlaubt es, sowohl das Metamodell der Sicht als auch die Modelltransformation zum erstellen der Sich in einem SQL ähnlichen Syntax zu definieren. Durch ModelJoin lassen sich so auf einfache Art und Weise maßgeschneiderte, immer aktuelle und konsistenten Sichten auf das gesamte Softwaresystem erstellen. Jedoch konnten diese Sicht nicht verändert werden, da die Übersetzung von Sichtänderungen zu Änderungen der Quellmodelle nicht definiert ist.

In dieser Masterarbeit wird das *Problem der Sichtänderung* für ModelJoin Sichtspezifikationen untersucht. Verschiedene Übersetzungsstrategien für Sichtänderungen werden vorgestellt. Es wird gezeigt, dass bestimmte Restriktionen genutzt werden können um die Übersetzbarkeit von veränderten Sichten festzustellen. Zu diesem Zweck wird eine Methode vorgestellt um eine Menge von OCL Restriktionen von einer ModelJoin Sichtdefinition abzuleiten. Die abgeleiteten OCL Restriktionen können dann genutzt werden, um die Übersetzbarkeit einer veränderten Sicht zu prüfen. Eine Änderung an der Sicht kann in einigen Fällen auf unterschiedliche Weise in Änderungen der Quellmodelle übersetzt werden. Verschiedene Übersetzungsstrategien werde für diese Fälle vorgestellt.

Für einige Sichtänderungen sind weiter Änderungen an der Sicht notwendig, damit die Sicht den Restriktion für die Übersetzbarkeit genügt. In bestimmten Situationen können diese nötigen Änderungen vor der Übersetzung automatisch abgeleitet und ausgeführt werden. In der Arbeit werden Algorithmen zur automatischen Wiederherstellung der Restriktion für übliche Situationen angegeben.

Die Restriktion zur Prüfung der Übersetzbarkeit werden für atomare Änderungen im allgemeinen und für Folgen von Änderungen mit zwei Beispielen aus Fallstudien evaluiert. Die Evaluation zeigt die Anwendbarkeit des erarbeiteten Ansatzes. Fast alle widerspruchsfreien Sichtänderungen konnten übersetzt werden und alle widersprüchlichen Änderungen wurden zurückgewiesen. Die vorgestellten Algorithmen zur Wiederherstellung der Übersetzbarkeit konnten in den gewünschten Situationen genutzt werden. Die Nutzung von OCL Restriktionen für die Prüfung der Übersetzbarkeit erlaubt eine einfache Integration in vorhandene Anwendungen. Jedoch charakterisieren die Restriktionen nicht genau alle übersetzbaren Sichten. Das erfüllen der Restriktionen ist nur eine hinreichende Bedingung für die Übersetzbarkeit einer Sicht.

Contents

1. Introduction

1.1. Motivation

In the development process of modern software systems multiple models are used to describe different system aspects and abstraction levels. For example, consider component models, class diagrams, performance and reliability models. Even the source code itself can be seen as a software model describing the implementation.

The Vitruvius [14] project uses a *view-centric* approach to combine information from different models into *flexible views*. To define these views easily, the new view definition language *ModelJoin* was developed. It allows to define both, the metamodel of the view type and the model transformation for creating the view in an SQL like syntax. Its goal is to allow for easily 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 the view be translated back to the underling models?

1.2. Goal

The View-Update-Problem is well studied in the context of relational databases [4, 7, 21, 38, 42, 45]. The SQL standard allows the update of views, if the effect to the underlying tables can easily be derived [37].

The goal of this master's thesis is to study the View-Update-Problem for ModelJoin views. This includes finding strategies to decide if an update operation on a view can be translated back to the source models and develop mechanisms to translate view updates to source model updates. For this purpose, the View-Update-Problem has to be formalized for the Ecore Metamodel [29]. Further 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 a formal abstract syntax is developed first. Then the translatability of updated target models are checked using OCL constraints. Therefore a scheme for deriving these constraints from the ModelJoin view definition is developed. To evaluate the applicability of the translatability check, the proposed algorithms are prototypically implemented and evaluated based on a case study.

1.3. Outline

Chapter 2 gives an introduction to view-based software development including the terms used in this context. It depicts the relationships between metamodels, models, views and view types. Further the basic concepts of ModelJoin and its use case in the Vitruvius project are illustrated. Then some fundamental findings about the View-Update-Problem in the context of relational databases are discussed.

As a foundation of our contribution, the View-Update-Problem and the semantic of update operation are defined for ModelJoin view definitions and Ecore Metamodels in chapter 3.

In chapter 4 we develop a scheme to derive a set of OCL constraints from a ModelJoin view definition. We show that the fulfillment of the derived constraints is a sufficient condition for the translatability of a updated target model. We further show, that an unmodified target model does fulfill all constraints. While we choose a fixed translation strategy in this chapter, we discuss different translation strategies in chapter 5. This includes different strategies for translating new target class instances and deleted target class instances.

In chapter 6 we propose algorithms for automatically restoring certain constraints after an update operation on the target model. These algorithms can be used in common situations to make a untranslatable target model translatable by executing further target model updates.

The findings are evaluated in chapter 7. First we evaluate under which conditions atomic update operations can be translated in general. Then concrete update sequences are translated for two case study examples. As examples the Common Component Modelling Example (CoCoME) [34] and the Media Store example from the upcoming Palladio Book [58] are used.

Finally in chapter 8 our findings are compared with other approaches to solve the View-Update-Problem in different domains. This includes update translation for relational and tree views, other linguistic approaches and specialized languages for Bi-Directional transformations.

Chapter 9 summarizes the developed approach and the result. Finally we give a conclusion.

2. Foundations

2.1. View-based Software Development

View-based Software Development is an established concept to break down the software development into different *viewpoints* [25]. A viewpoint is a conceptual point of view that satisfies a specific concern [31]. Having different views helps to manage the complexity of a software model, because stakeholders and developers can concentrate on smaller and simpler restricted system extracts. A *view* provides the needed extract by showing specific elements of the software system model. A view is an instance of a *view type*. The view type describes the elements that can be used to formulate the view. Unfortunately the terms view, view type and view point are not used consistently in the literature. We use the terminology from [32] shown in Figure 2.1. It is important to note, that a view type may represent parts of multiple different metamodels and therefore the view is capable to combine elements from heterogeneous models.

In the following sections the term view type is interpreted as the metamodel for an actual view. A view as an extract of the source models is then an instance of this metamodel and is called the *target model*.

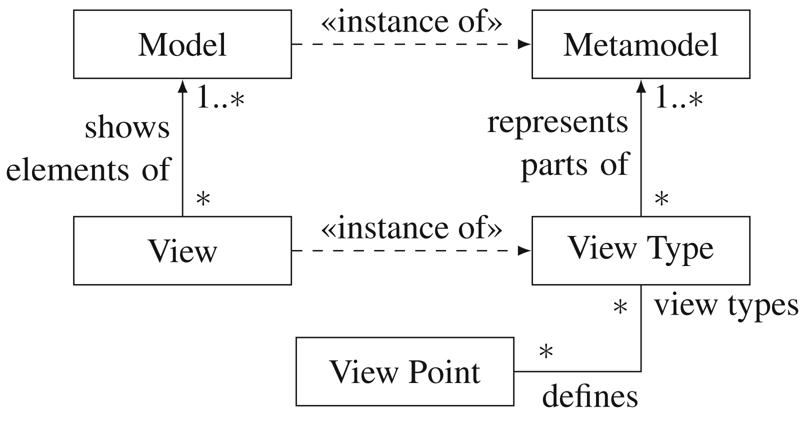


Figure 2.1.: Terminology for view-based modeling (adapted version in [17] from [32])

2.2. Meta-Object Facility (MOF)

The Meta-Object Facility (MOF) specification [54] is an industry-standard of the Object Management Group (OMG) for model-driven engineering. It provides a type system for defining models. Itself is defined by UML. However it provides the formal base for UML, too.

The MOF standard allows the definition of models in a layered metamodel architecture. For example the MOF standard for UML defines four metalevels (M3-M0). The M3 level is the top layer called the meta-metamodel. It is the language to describe the elements of the metamodel (M2). In fact the M3 level not only describes the lower level M2, but also itself. Therefore the metamodeling architecture of MOF is called closed. A prominent example for an M2 metamodel is the metamodel that describes UML. The M2 metamodel describes the elements of the user model. This may, e.g., be an UML diagram. The M1 model in turn describes the entities of the actual problem domain, the M0 layer or data layer. An example MOF-Hierarchy for UML is shown on the left side in Figure 2.2 .

Most important in our context are the M1 and M2 levels. Our metamodels are in the M2 layer and thus describe the elements of the models to transform. The models are residents of the M1 layer describing the system.

2.3. The Ecore Metamodel

The Ecore Metamodel [29] is a part of the Eclipse Modeling Framework (EMF) [28] for model-driven development and is based on a subset of the old MOF 1.4 standard. It was originally developed as an alternative to the MOF standard, but has influenced the development of the new Essential MOF (EMOF) standard in its current 2.0 version. EMOF is a more compact alternative to the complete MOF standard. The Ecore implementation is almost analog to EMOF, but has some minor differences. Ecore has no associations as first class elements. It uses references instead, which are bounded to a class. Furthermore Ecore models must be organized in an hierarchy with a single root element. ModelJoin uses the Ecore Metamodel for practical reasons, because this allows for the integration of ModelJoin into the Vitruvius project, which uses the Eclipse Modeling Framework.

2.4. ModelJoin and Vitruvius

ModelJoin [17] is a recently developed view definition language for rapid creation of views across instances of different metamodels. It allows to easily and declaratively define custom views without having to write model transformations. It is using a human-readable textual SQL-like DSL that describes both the resulting view type and the selection criteria.

It is particularity useful in the Vitruvius (VIew-cenTRic engineering Using a VIrtual Underlying Single model) approach [14]. Vitruvius does not use a monolithic single underlying metamodel (sum). Instead, it uses legacy metamodels, which are combined to a virtual underlying single model. This enables the usage of different software engineering tools that do not operate on instances of the same metamodel.

ModelJoin operators describe a relation between the source models and the target model (view). There are four kinds of operators that can be used to define a view with its view type:

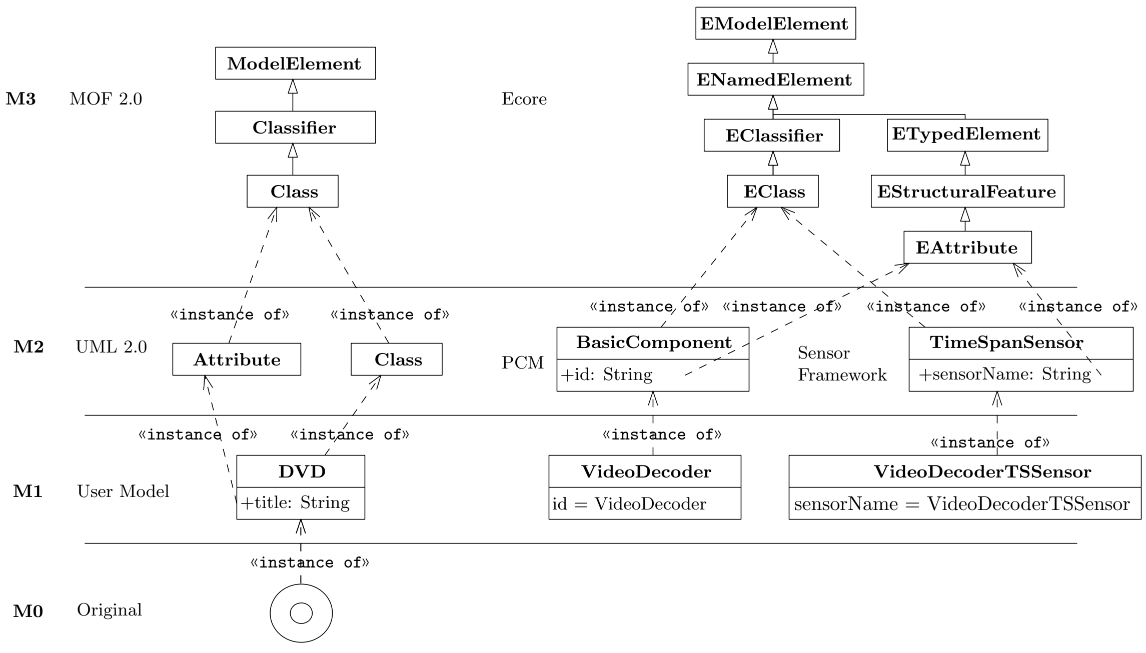


Figure 2.2.: MOF hierarchy examples: On the left side a simplified example of the MOF hierarchy for UML. On the right side a simplified example for Ecore and the Palladio Component Model and Sensor Framework.

Join expressions can be used to map two classes of source models to a new class in the target model. When two classes are joined by a natural join, a new target class that contains only the common attributes is created. For all pairs of instances of the the joined source classes with equal values of common attribute, an instance of the target class is created. Is it also possible to create outer joins. An outer join creates an instance for every left or right source class instance even if nor matching join partner instance exists. A generalization of natural joins are theta joins. In a theta join an arbitrary logical expression can be used to define the pairs of source class instances for which a new target class instance should be created.

Keep expressions allow the copying of source model attributes or references into the target model. Keep attribute can be used to copy renamed attributes from the source model into the target model. Keep reference expressions can be similarly used to copy references. The copied attributes and references are added to the target class and the values of the source class instances are used for the target class instances.

All created target class instances will get the attribute values and links from the corresponding source class instances. Further calculated attributes can be used to calculate the value of target model attributes using arbitrary source model attributes. This may include the usage of aggregate functions like sum and average.

Further *selection expressions* can be used to discard class instances. A selection expression is defined by a logical expression that decides if a class instance should be kept in the target model.

Finally *rename expressions* allow to rename classes, attributes and references in the target model.

An example for a ModelJoin view definition can be found in Listing 2.1. A more detailed and formal specification of the ModelJoin operators can be found in [17].

```
theta join classifiers.Interface with uml.Interface
where "classifiers.Interface.name = uml.Interface.name" as jointarget.Interface {
    keep attributes commons.NamedElement.name
    keep calculated attribute "classifiers.Interface.name.substring(1, classifiers.Interface
    .name.size() - 2).concat('Impl')" as jointarget.Interface.implName : String
    keep outgoing members.MemberContainer.members of type members.InterfaceMethod as type
    jointarget.Operation {
        keep attributes commons.NamedElement.name
        keep outgoing parameters.Parametrizable.parameters
             of type parameters.OrdinaryParameter as type jointarget.Parameter {
            keep attributes commons.NamedElement.name
        }
    }
}
```

Listing 2.1: Example ModelJoin view definition involving a theta join and keep attributes, reference and calculated attribute expressions.

2.5. The View-Update-Problem

An update to a view is the execution of an update operation (insert, delete, replace) on a view. Since a view is just the result of a model transformation, it is unclear how the source models of this transformation must be updated. The source model update in question is called an update translation. The translation should comply with the intension of the view update and thus meet the users expectations. The View-Update-Problem is to find a in some sense correct update translation for a given view and update operation.

The View-Update-Problem was studied intensively by database researchers in the late ‘70s and ‘80s. For relational databases a view is defined by a database query, instead of a model transformation. The database itself corresponds to the source models in the view-based software development context. A database scheme can be seen as the metamodel of a database. Dayal and Bernstein [22] have formalized the notion of an update translation and derived conditions for correct update translations in the context of relational databases. They have defined two properties an correct update translation should satisfy: It should “exactly perform” an update and should “preserve semantic”. A translation exactly performs an update, if getting the view after the translation yields the same view as if the update operation would have been applied to the unmodified database (see Figure 2.3). A translation preserve semantics, if the result of the translation on the database satisfies all constraints defined in the database scheme. Since more then one translation that satisfies these two properties can exist, its unclear which one to choose, if no further information is provided.

Therefore, Bancilhon and Spyratos [7] introduced the concept of “translation under constant complement”. For a given view, they define a “complementary” view, such that the database could be computed from the view and its complement. In general there exist multiple complements for a view. The choice of the complement determines the update policy. Hence for a fixed complement the translation is defined in a way, such that the complement remains invariant. The main result of their work is that the problem of translating view updates is equivalent to finding a suitable component.

Recently Lechtenbörger [42] showed that the translation of view updates under constant complement exist precisely, if the view update can be undone by using further view updates. He also argues, that particularly in the case of complex views, the existence of a translation under constant complement cannot be guaranteed in general.

The current SQL standard allows the update of views, if the effect to the underlying tables can easily be derived [37]. For example the Microsoft SQL Server 2014 allows the updates of a view under the following conditions [50]:

- Only columns from one base table are referenced in the update statement.
- All modified columns must directly reference a column of the underlying table. No aggregate function or computations are allowed.
- The modified columns are not affected by `GROUP BY`, `HAVING` or `DISTINCT` clauses.
- `TOP` cannot be used in the select statement together with the `WITH CHECK OPTION` clause.

Other database systems like Oracle MySQL [57] have similar restrictions. Some database systems like DB2 support additionally triggers, which determine the effected tables [36].

$$\begin{tikzcd}
v = q(s) \arrow[r, "u"] & u(v) \overset{!}{=} q(s') \\
s \arrow[u, "q"] \arrow[r, "T_u"'] & s' = T_u(s) \arrow[u, "q"']
\end{tikzcd}$$

Figure 2.3.: The “exactly perform” an update property of [22] states that running a view query q and updating the resulting view v with u should yield the same result as when applying the view translation T_u to the database state s and running the query q on the updated state s'.

3. The View-Update-Problem for ModelJoin

In this chapter the View-Update-Problem is formalized for ModelJoin view definitions. It forms the foundation for the rest of the thesis. We first formalize the query function of a ModelJoin view definitions and define the update operations for Ecore models. Next we develop a set of properties for a valid update translation based on the work of Foster et al. [27]. Finally, we define the View-Update-Problem for ModelJoin view definitions and a restricted version, which we will address in the further chapters of the thesis.

We use the same notation as in the ModelJoin specification [17] 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 $m_s \in I(M_s),\ m_t \in I(M_t)$ for a source respectively target model.
3. We do not explicitly distinguish between source and target model classes, references and attributes. We use Class, Ref and Att for classes, references and attributes defined in either M_s or M_t.

3.1. ModelJoin View Definitions and Update Operations

For the View-Update-Problem we interpret the relation between the source and the target model induced by the ModelJoin definition as query function.

Definition 1. A *ModelJoin view definition* is a relation $Q \in M_s \times M_t$ with an associated *query function*

$$q\colon I(M_s) \to I(M_t)$$

that maps a source model (source class instances with attribute values and links) to a target model. q is obtained from Q by applying the definition of the ModelJoin expression on model level and returning the target model elements.

An update operation does not modify the metamodel, it modifies either the source or target model.

Definition 2. An *update operation* u for a metamodel M is a function:

$$u\colon I(M) \to I(M)$$

that maps an instance of the metamodel to another instance of the same metamodel.

3.2. Update Operations for Ecore Models

In this section a set of update operations for Ecore models will be defined. We use the notation for MOF-Metamodels [54] here. The update operations describe how the model elements of the source and target model can be updated. We use the snapshot notation to describe the state of a model.

Definition 3. A *snapshot* $\sigma = (\sigma_{\text{Class}}, \sigma_{\text{Att}}, \sigma_{\text{Ref}})$ describes a model by three functions:

1. $\sigma_{\text{Class}} : \text{Class} \rightarrow I(\text{Class})$ is a function that returns for every class $c \in \text{Class}$ the set of instances.

2. $\sigma_{\text{Att}} : \text{Att} \rightarrow I(\text{Class}) \rightarrow \mathcal{J}$ is a function that returns for every attribute $a : t_c \rightarrow t \in \text{Att}$ a function that provides the value of a for a given instance $\underline{c} \in \sigma_{\text{Class}}(c)$.

3. $\sigma_{\text{Ref}} : \text{Ref} \rightarrow I(\text{Class}) \times I(\text{Class})$ is a function, which returns for each reference $r \in \text{Ref}$ the set of tuples of class instances that are linked by this reference in the model.

We want to describe the effect of update operations by their effect to the snapshot functions.

3.2.1. Updates for Attribute Values

An update of an attribute value changes the value of a certain attribute for a given class instance.

Definition 4 (Update attribute value). Let $a \in \text{Att}_c : t_c \rightarrow t$ be an attribute of class $c \in \text{Class}$ and v a value of type t.
An *update attribute operation* $\text{update}_{\text{Att}(a)}$ changes the value of the attribute a of an instance $\underline{c} \in I(c)$ to the value v by mapping σ_{Att} to σ'_{Att}:

$$\text{update}_{\text{Att}(a)}(\underline{c}, v) : \sigma_{\text{Att}} \mapsto \sigma'_{\text{Att}}$$

with

$$\sigma'_{\text{Att}}(a)(\underline{c}') = \begin{cases} v, & \underline{c}' = \underline{c} \\ \sigma_{\text{Att}}(a)(\underline{c}'), & \text{else} \end{cases}$$

3.2.2. Creation and Deletion of References

A *create link operation* creates a link of a certain reference between two given class instances.

Definition 5 (Create link). Let $r \in \text{Ref}$ be a reference between classes $c, \hat{c} \in \text{Class}$ and $\underline{c} \in I(c), \underline{\hat{c}} \in I(\hat{c})$ be two class instances.
A *create link operation* $\text{create}_{\text{Ref}(r)}$ creates a new link between $\underline{c} \in I(c)$ and $\underline{\hat{c}} \in I(\hat{c})$ by mapping σ_{Ref} to σ'_{Ref}:

$$\text{create}_{\text{Ref}(r)}(\underline{c}, \underline{\hat{c}}) : \sigma_{\text{Ref}} \mapsto \sigma'_{\text{Ref}}$$

with

$$\sigma'_{\text{Ref}}(r') = \begin{cases} \sigma_{\text{Ref}}(r') \cup \{(\underline{c}, \underline{\hat{c}})\}, & r' = r \\ \sigma_{\text{Ref}}(r'), & \text{else} \end{cases}$$

In the same fashion a *delete link operation*, removes a link of a certain reference between to given class instances.

Definition 6 (Delete link). Let $r \in \textsc{Ref}$ be a reference between classes $c, \hat{c} \in \textsc{Class}$ and $\underline{c} \in I(c), \underline{\hat{c}} \in I(\hat{c})$ be two class instances.
A *delete link operation* $\textsc{delete}_{\textsc{Ref}(r)}$ deletes a link between $\underline{c} \in I(c)$ and $\underline{\hat{c}} \in I(\hat{c})$ by mapping $\sigma_{\textsc{Ref}}$ to $\sigma'_{\textsc{Ref}}$:

$$\textsc{delete}_{\textsc{Ref}(r)}(\underline{c}, \underline{\hat{c}}) : \sigma_{\textsc{Ref}} \mapsto \sigma'_{\textsc{Ref}}$$

with

$$\sigma'_{\textsc{Ref}}(r') = \begin{cases} \sigma_{\textsc{Ref}}(r') \setminus \{(\underline{c}, \underline{\hat{c}})\}, & r' = r \\ \sigma_{\textsc{Ref}}(r'), & \text{else} \end{cases}$$

3.2.3. Creation and Deletion of Class Instances

A *create class instance operation* creates a new class instance of a certain class with a given set of attribute values.

Definition 7 (Create class instance). Let $c \in \textsc{Class}$ be a class, $V = \{v_a \in I(t) \mid a \in \textsc{Att}_c\}$ be a set of attribute values for each attribute of c. A *create class instance operation* $\textsc{create}_{\textsc{Class}(c)}(V)$ creates a new instance of c with the attribute values V by mapping $\sigma_{\textsc{Class}}$ to $\sigma'_{\textsc{Class}}$ and $\sigma_{\textsc{Att}}$ to $\sigma'_{\textsc{Att}}$:

$$\textsc{create}_{\textsc{Class}(c)}(V) : (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}) \mapsto (\sigma'_{\textsc{Class}}, \sigma'_{\textsc{Att}})$$

with

$$\sigma'_{\textsc{Class}}(c') = \begin{cases} \sigma_{\textsc{Class}}(c') \cup \{\underline{c}\}, & c' = c \\ \sigma_{\textsc{Class}}(c'), & \text{else} \end{cases}$$

$$\sigma'_{\textsc{Att}}(a)(\underline{c}') = \begin{cases} v_a, & \underline{c}' = \underline{c} \\ \sigma_{\textsc{Att}}(a)(\underline{c}'), & \text{else} \end{cases}$$

for a new instance $\underline{c} \in I(c)$

A *delete class instance operation* deletes a given instance of a certain class.

Definition 8 (Delete class instance). Let $c \in \textsc{Class}$ be a class and $\underline{c} \in I(c)$ be a class instance. A *delete class instance operation* $\textsc{delete}_{\textsc{Class}(c)}(\underline{c})$ deletes the instance $\underline{c}$ by mapping $\sigma_{\textsc{Class}}$ to $\sigma'_{\textsc{Class}}$ and $\sigma_{\textsc{Att}}$ to $\sigma'_{\textsc{Att}}$:

$$\textsc{delete}_{\textsc{Class}(c)}(\underline{c}) : (\sigma_{\textsc{Class}}, \sigma_{\textsc{Att}}) \mapsto (\sigma'_{\textsc{Class}}, \sigma'_{\textsc{Att}})$$

with

$$\sigma'_{\textsc{Class}}(c') = \begin{cases} \sigma_{\textsc{Class}}(c') \setminus \{\underline{c}\}, & c' = c \\ \sigma_{\textsc{Class}}(c'), & \text{else} \end{cases}$$

$$\sigma'_{\textsc{Att}}(a)(\underline{c}') = \begin{cases} \perp, & \underline{c}' = \underline{c} \\ \sigma_{\textsc{Att}}(a)(\underline{c}'), & \text{else} \end{cases}$$

3.3. The View-Update-Problem Definition for ModelJoin

3.3.1. State-based vs. Delta-based Approaches

In the literature we find two fundamental different approaches to formulate the View-Update-Problem. A *delta-based approach* (such as in [22, 7, 42]) or a *state-based approach* (such as in [27, 49]).

- In the delta-based approach the translation function translates an update operation on the target model to an update operation on the source model.
- In the state-based approach the translation function takes the updated target model and original source model and returns the updated source model.

The delta-based approach has the advantage, that the knowledge about the changed elements is explicitly given by the update operation and does not need to be inferred from the update target model. However the tool used to perform the updates must emit the update operation or the update operation must be recovered from the updated target model by comparing it with the original one. If an external tool is used to update the target model and it does not emit the update operations, the updates must be obtained by doing the comparison, which is a complex operation, requiring heuristic algorithms [39, 43]. To keep the translatability check and translation independent of the tool used to update the target model, we will use a state-based approach in our work. In our approach the changed parts of the target model can be derived from a trace model, which will be introduced in this section.

3.3.2. Properties of a Valid Translation

Next we want to formulate properties, which characterize a valid update translation. Foster et al. [27] proposed two fundamental properties for the translation of tree like structures. We adopt their GetPut- and PutGet-Property for ModelJoin view definitions and Ecore models here.

Definition 9 (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 exist a translation $q^{-1} : I(M_t) \times I(M_s) \to I(M_s)$ 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:

$$\forall m_s \in I(M_s) \colon q^{-1}(q(m_s), m_s) = m_s \qquad \text{(GetPut)}$$

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

$$\forall m_s \in I(M_s), \forall m_t \in V \colon q(q^{-1}(m_t, m_s)) = m_t \qquad \text{(PutGet)}$$

In this case we call the function q^{-1} a *translation.*

Note, that the query function of a ModelJoin view definition is not a total function in general. More specifically, $q\,[I(M_s)]$ is a real subset of $I(M_t)$ and thus not all possible target models can be translated by q^{-1}. Update operations can lead to a target model $m_t \in I(M_t) \setminus q\,[I(M_s)]$. In this case m_t is not translatable, because no source model $m_s \in I(M_s)$ with $m_t = q(m_s)$ exists. An example for an untranslatable target model is given in the introduction of chapter 4.

In practice we are more interested in a finding the translation function, not only its existence.

3.4. The Restricted View-Update-Problem

In the case of ModelJoin, the translation function q^{-1} should reflect the semantic of its query function q. Therefore it should have a fixed translation semantic for each operation in the ModelJoin view definition Q. The translation function q^{-1} will be defined inductive over all ModelJoin operators in Q, similar to the query function q itself. If each ModelJoin operator has a fixed translation semantic, the set of translatable target models $V_r \subseteq I(M_t)$ forms only a subset of all obtainable target models in $q\,[I(M_s)]$. Fixing the semantic of the translation function for each ModelJoin operation in Q makes the translation predictable and comprehensible for the user.

Therefore we want to formulate the View-Update-Problem for a restricted set $V_r \subseteq q\,[I(M_s)]$ of translatable target models. In addition we want to introduce a *trace model* $M_\sim$, which is part of the target model. The trace model should be non-editable and should form a explicit representation of the mapping relation between the source and target class instances.

Definition 10 (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:

$$m_v = [m_t]_v$$
$$m_\sim = [m_t]_\sim$$

We now define the *restricted View-Update-Problem*, which will be addressed in the further section of this thesis.

Definition 11 (Restricted View-Update-Problem). The *restricted View-Update-Problem* (*rVUP*(Q)) for a given ModelJoin view definition Q is to find a restricted subset $V_r : I(M_s) \to \mathcal{P}(I(M_t))$ with the following properties:

(i) A translation $q^{-1} : V_r\,[I(M_s)] \times I(M_s) \to I(M_s)$ 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) q^{-1} conforms to the GetPut-Property:

$$\forall m_s \in I(M_s) \colon q^{-1}(q(m_s), m_s) = m_s \qquad \text{(GetPut)}$$

(iv) q^{-1} conforms to the GetPut properties for all views in V_r:

$$\forall m_s \in I(M_s), \forall m_t \in V_r(m_s) \colon \left[q(q^{-1}(m_t, m_s))\right]_v = [m_t]_v \qquad \text{(PutGet)}$$

A set V_r together with a translation q^{-1}, which solves *rVUP*(Q) is called a solution of the problem.

The TOTALITY-Property ensures that all unmodified target models are translatable. It is a requirement for the GetPut-Property to be well-defined, because the translation q^{-1} must be defined for all unmodified target models. The GetPut-Property ensures that translating an unmodified target model does not change the source model.

An update to the view model may lead to new or removed mappings between source class instances and target class instances. These mappings can not be present in the trace model before the translation. Therefore we restrict the GetPut-Property to the view model only, so that the trace model can be different before and after the query.

The *rVUP*(Q) is solvable for all Q because the set $V_r(m_s) = \{q(m_s)\}$ together with the translation $q^{-1}(m_t, m_s) = m_s$ is a trivial solution.

Even if the trivial solution solves *rVUP*(Q) for every Q, it is not very interesting because it does not allow any updates to the target model. We want to find a solution, which allows useful target model updates and reflects the semantic of the ModelJoin operators used in Q. To verify these properties we evaluate our solution in a case study in chapter 7.

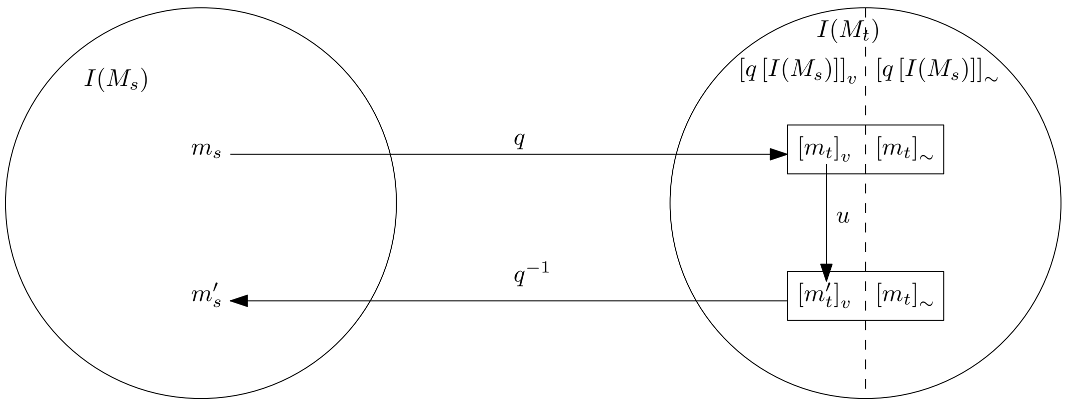


Figure 3.1.: We extend the target model of a ModelView view definition to contain a non-editable trace model in additional to the view. An update operation, only updates the view. The trace model stays the same and contains the relationship between source class instances and target class instances.

4. Constraints for Translatable Views

We have seen, that 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, because no source model $m_s \in I(M_s)$ with $m_t = q(m_s)$ exists (see Figure 4.1).

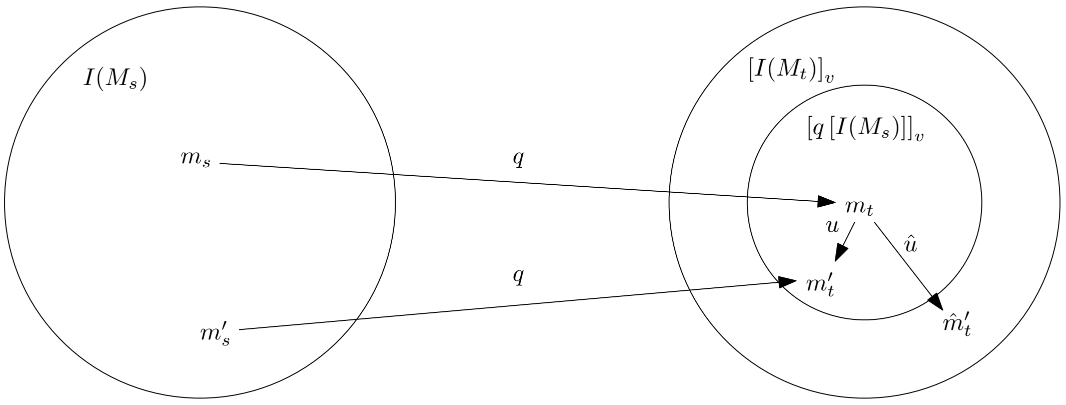


Figure 4.1.: The Update operation u is translatable because for m'_t a corresponding source model m'_s exists. However the update operation $\hat{u}$ is untranslatable since $\hat{m}'_s$ has no corresponding source model.

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 4.1. In this example, the commons.NamedElement.name attribute of the source class gets mapped to two different target class attributes. It gets mapped to an attribute named name and an attribute named alias. The update operation given in Listing 4.1 cannot be translated. Let $m'_t \in I(M_t)$ be the updated target model, then no source model $m'_s \in I(M_s)$ with $m'_t = q(m'_s)$ exists. This is the case, since all instances of the class jointarget.Interface in $q\,[I(M_s)]$ have the same attribute values for the attributes name and alias.

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

Listing 4.1: Example ModelJoin view definition where all obtainable views are a real subset of all possible target model.

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

Listing 4.2: Untranslatable update operation for the ModelJoin view definition in Listing 4.1

Such untranslatable update operations shall be forbidden. For this purpose the target model needs to be designed in a way, such that all valid instances can be translated. 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 for Ecore metamodels using the Object Constraint Language (OCL) [55]. A OCL expression for this case could be:

```
context jointarget.Interface
inv: self.name = self.alias
```

If such OCL-constraints can be derived from the ModelJoin view definition, we can easily decide if a given update operation is translatable for an updated target model. We can simply check if the OCL-constraints hold for the updated target metamodel. If the OCL-constraints hold, then the updated view is translatable. Ideally the constraints characterize exactly the set $q[I(M_s)]$, which contains all translatable views. However since we want to fix the translation semantic 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 $rVUP(Q)$.

4.1. OCL Expressions

We write OCL expressions in their concrete syntax [55]. Further we use the following Notations:

- Expr_t is the set of all OCL expressions of type t.
- Var_t is the set of variables of type t.
- $\pi = \langle \sigma, \beta \rangle$ is an environment for the evaluation of an expression. It contains the system state σ and a variable assignment $\beta : \mathsf{Var}_t \to I(t)$.
- $I[[\alpha]] : \mathsf{Env} \to I(t)$ is the interpretation of the OCL expression $\alpha \in \mathsf{Expr}_t$. Thus $I[[\alpha]](\pi)$ is the value of α in the environment π.
- $\mathsf{free} : \mathsf{Expr} \to \mathsf{Var}$ is the function, which returns all free variables in an OCL expression.
- For an OCL expression $\phi \in \mathsf{Expr}$ with $\mathsf{free}(\phi) = \{v_1, \dots, v_n\}$ we write for the interpretation of ϕ in the environment σ and variable assignment $v_1 \to \underline{c}_1, \dots, v_n \to \underline{c}_n$ the short form: $\underline{\phi}^{\sigma}_{v_1,\dots,v_n}(\underline{c}_1, \dots, \underline{c}_n)$ for $I[[\phi]](\langle \sigma, v_1 \to \underline{c}_1, \dots, v_n \to \underline{c}_n \rangle)$.

- We use = for both: mathematically equality and =for the equality operator of OCL. However it should be clear from the context which equality is meant.
- If we write $\alpha \hat{=} \gamma$ for two OCL expressions $\alpha, \gamma \in \mathsf{Expr}_t$, we mean structural equality, respectively the equality of the abstract syntax. The textual equality of the concrete syntax, should be ignored. For example notation shorthands are equivalent to their long forms, if they parse to the same abstract syntax.

The formal definition of the interpretation function I for all standard OCL expressions can be found in [55].

4.2. Meta Notations for OCL

The OCL constraints should use the source models in addition to the target model. This, e.g., allows the formulation of OCL-constraints that describe precisely, if an attribute is allowed to be updated. Let name be an attribute, source be a class instance in the source model, alias be an attribute and target be a class instance in the target model, then the constraint

```
source.name = target.alias
```

could be used to forbid updates of the name attribute.

Since the OCL constraints should be derived from the ModelJoin expression, a meta language will be used to describe the resulting OCL expressions:

For each class symbol $c \in \textsc{Class}$ and attribute symbol $a \in \textsc{Att}$ the class or respectively attribute name should be used in the resulting OCL expression. So for c_1 = Interface and a_1 = name the OCL meta expression:

c_1 .allInstances()->forAll(ac | ac.a_1 = “StoreIf”)

results in

```
Interface.allInstances()->forAll(ac | ac.name = “StoreIf”)
```

Each binary operator with subscript should be expanded to an expression joining the elements of the subscript expression by the binary operator. For example the expression

$\mathbf{AND}_{a \in \{a_1, a_2, a_3\}}$(self.$a$ = “StoreIf”)

should be expanded to:

self.a_1 = “StoreIf” **and** self.a_2 = “StoreIf” **and** self.a_3 = “StoreIf”

Further macro functions will be used to indicate expressions defined in other contexts. For example

```
self.target.alias = $var_a$(self.left)
```

with the definition

```
$var_a(\underline{c}) := \underline{c}.a$
```

should be expanded to:

```
self.target.alias = self.left.$a$
```

4.2.1. Meta Variables

Considering the view definition in Listing 4.1, it would be desirable to describe the relation of source model attributes and target model attributes like for example in

```
target.$a_t$ = left.$a_1$
```

However since the source model attributes are not updateable by an update operation on the target model, this kind of formulation would fix the value of target.a_t to the original source value and make it unchangeable. To avoid this issue, source model attribute values should not be used directly. Instead a *meta variable* expression is introduced. It describes the updated value of the source model attribute like for example in:

```
target.$a_t$ = $var_{a_1}$(left)
```

The meta variable expression will then be replaced with its definition. The definition is used as a canonical attribute that maps to the source attribute.

Definition 12 (meta variable for attributes)**.** Let $c \in \textsc{Class}$ be a class and $a : t_{c'} \to t \in \textsc{Att}^*_c$ be an attribute, then the initial value of the *meta variable* $var_a : \mathsf{Expr}_{t_c} \to \mathsf{Expr}_t$ 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

$$\underline{var}^{\sigma}_a(\underline{c}) = I\,[[var_a(\mathsf{v})]]\,(\langle\sigma, \{\mathsf{v} \to \underline{c}\}\rangle).$$

For the initial definition of var_a the value of $\underline{var}^{\sigma}_a$ is

$$\underline{var}^{\sigma}_a(\underline{c}) = I\,[[var_a(\mathsf{c})]]\,(\langle\sigma, \{\mathsf{c} \to \underline{c}\}\rangle) = \sigma_{\textsc{Att}}(a)(I\,[[\mathsf{c}]]\,(\langle\sigma, \{\mathsf{c} \to \underline{c}\}\rangle)) = \sigma_{\textsc{Att}}(a)(\underline{c}).$$

If there is an attribute a_t of a target model class c_t that maps to the source attribute a, the definition of var_a will be extended to use the attribute a_t as canonical attribute for an instance $\underline{c} \in I(c)$ with a mapping $\underline{c} \sim_{\bowtie} \underline{c}_t$ to a target instance $\underline{c}_t \in I(c_t)$.

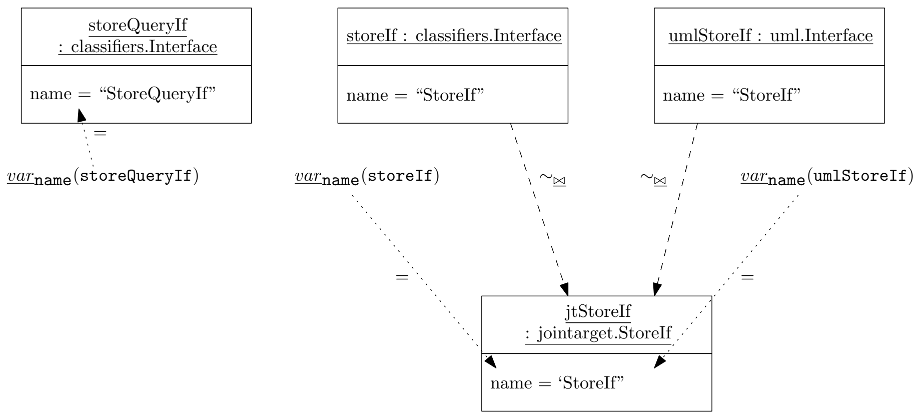

storeQueryIf : classifiers.Interface
name = "StoreQueryIf"
storeIf : classifiers.Interface
name = "StoreIf"
umlStoreIf : uml.Interface
name = "StoreIf"
=
$\underline{var}_{\mathtt{name}}$(storeQueryIf)
$\underline{var}_{\mathtt{name}}$(storeIf)
$\sim_{\bowtie}$
$\sim_{\bowtie}$
$\underline{var}_{\mathtt{name}}$(umlStoreIf)
=
=
jtStoreIf : jointarget.StoreIf
name = 'StoreIf"

Figure 4.2.: Initially the meta variables will have the same value as the attribute of the given class instance (left). However if the attributes get mapped to target model attributes, the meta variable definition will be extended to use the corresponding attribute of target class instances that are related to the given source class instance (right).

Theorem 1 (correctness of meta variables for attributes). *A meta variable var_a for an attribute $a : t_c \to t \in \textsc{Att}$ is called correct, if*

(i) *For an unmodified target model obtained from a query, it has the same value as the corresponding attribute:*

$$\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target}) \qquad \text{(Get-Equality)}$$
$$(m_t = q(m_s) \to \forall \underline{c} \in \sigma^{m_s}_{\textsc{Class}}(c)(\sigma^{m_s}_{\textsc{Att}}(a)(\underline{c}) = \underline{var}_a^{\sigma^{m_s \cup m_t}}(\underline{c})))$$

(ii) *After the translation of a target model back to a source model, the corresponding attribute has the same value as the meta variable:*

$$\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target}) \qquad \text{(Put-Equality)}$$
$$(m'_s = q^{-1}(m_t, m_s) \to \forall \underline{c} \in \sigma^{m'_s}_{\textsc{Class}}(c)(\sigma^{m'_s}_{\textsc{Att}}(a)(\underline{c}) = \underline{var}_a^{\sigma^{m_s \cup m_t}}(\underline{c})))$$

Similarly, meta variables are defined for references.

Definition 13 (meta variable for references). Let $c, \hat{c} \in \textsc{Class}$ be classes and $r \in \textsc{Ref}$ be a reference with *associates*$(r) = \langle c, \hat{c} \rangle$, then the initial value of the meta variable $var_r : \mathsf{Expr}_{t_c} \to \mathsf{Expr}_{\mathsf{Set}(t_{\hat{c}})}$ is:

$$var_r(\alpha) := \alpha.r$$

In addition we define the interpretation function $\underline{var}_r : I(c) \to \mathcal{P}(I(\hat{c}))$ of var_r as

$$\underline{var}_r^{\sigma}(\underline{c}) = I\,[[var_r(\mathsf{v})]]\,(\langle \sigma, \{\mathsf{v} \to \underline{c}\} \rangle).$$

For the initial definition of var_r the value of $\underline{var}_r^\sigma$ is

$$\underline{var}_r^\sigma(\underline{c}) = I\,[[var_r(\mathsf{c})]]\,(\langle\sigma, \{\mathsf{c} \to \underline{c}\}\rangle) = L(r)(I\,[[\mathsf{c}]]\,(\langle\sigma, \{\mathsf{c} \to \underline{c}\}\rangle)) = L(r)(\underline{c}).$$

Theorem 2 (correctness of meta variables for references). *A meta variable var_r for a reference $r \in$ REF with $associates(r) = \langle c, \hat{c}\rangle$ is called correct, if*

(i) *For a unmodified target model obtained from a query, it has the same links as the corresponding reference:*

$$\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target}) \qquad \text{(GET-EQUALITY)}$$
$$(m_t = q(m_s) \to \forall \underline{c} \in \sigma_{CLASS}^{m_s}(c)(L(r)(\underline{c}) = \underline{var}_r^{\sigma^{m_s \cup m_t}}(\underline{c})))$$

(ii) *After the translation of a target model back to a source model, the corresponding reference in the resulting source model has the same links as the meta variable:*

$$\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target}) \qquad \text{(PUT-EQUALITY)}$$
$$(m_s' = q^{-1}(m_t, m_s) \to \forall \underline{c} \in \sigma_{CLASS}^{m_s'}(c)(L(r)(\underline{c}) = \underline{var}_r^{\sigma^{m_s \cup m_t}}(\underline{c})))$$

The defined meta variables will be used as a placeholder in the resulting OCL expressions. We will see that if the GET-EQUALITY holds for all meta variable definitions, then we can show that the OCL expressions are satisfied for a target model directly obtained from a query. Further if the OCL expressions are satisfied for a modified target model and the translation is chosen in a way, such that the PUT-EQUALITY also holds for all meta variables, then we can show that the PUTGET-Property holds.

To check the validity of the OCL expressions, the meta variables are replaced with their definition after all OCL expressions are derived from the ModelJoin-Expression. The resulting OCL expression then does not contain any meta variables anymore and can be checked without modifications by existing tools.

4.3. 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.

For example, consider the ModelJoin expression in Listing 4.3. The OCL expression $\phi(\alpha)$ = α.name.substring(1, α.name.size() - 2) for the calculated attribute implementationName depends on the source element name. If we described the relation of implementationName and interfaceName by using the common source attribute name we would not be able to update the value of implementationName, because name is a source model element and hence cannot be changed directly. It would be desired to rewrite the OCL expression $\phi(\alpha)$, so that it does not depend on source model elements anymore. There are several ways the expression could be rewritten.

```
theta join classifiers.Interface with uml.Interface
where "classifiers.Interface.name = uml.Interface.name" as jointarget.Interface {
    keep attributes commons.NamedElement.name as interfaceName
    keep calculated attribute commons.NamedElement.name.substring(
        1, commons.NamedElement.name.size() - 2
    ) as implementationName
}
```

Listing 4.3: Example ModelJoin view definition with a calculate attribute, which strips off the “If” prefix of the interface names.

4.3.1. Rewriting for Existing Source Instances

In case we only want to update target class instances, which are already mapped to source model class instances, the OCL expression could be rewritten by using the meta variables. To save space, we abbreviate commons.NamedElement.name with name and jointarget.Interface with Interface. The rewrite of $\phi(v)$ is $\phi'(v) = var_{\text{name}}(v)$.substring(1, $var_{\text{name}}(v)$.size()) and the derived OCL-constraints with v = left is:

target.alias = var_{name}(right).substring(1, var_{name}(right).size())

with the meta variable var_{name} mapping self.right to the value of a canonical attribute like self.target.name.

Similarly navigation calls can be handled. Consider the following example expression returning the number of interface methods throwing exceptions:

$\phi(v)$ = v.members->select(m |
  m.oclAsType(members.Method).exceptions->size() > 0
)->size()

It can be replaced with

$\phi(v)$ = $var_{\text{members}}(v)$->select(m |
  m.oclAsType(members.Method).exceptions->size() > 0
)->size()

where members is an abbreviation for the reference members.MemberContainer.members.

The problem with these simple rewrites is, that the creation of new target class instances cannot be allowed. For these, the expressions cannot be evaluated, because there is no mapping to source class instance. However the rewrite is possible for all expressions.

Definition 14 (Rewritten expression)**.** Let $\phi \in \mathsf{Expr}_t$ be an OCL expression, then the *rewrite function* $R : \mathsf{Expr}_t \to \mathsf{Expr}_t$ is defined as

$$R[\phi] = \begin{cases} var_a(R[\alpha_c]), & \text{if } \phi \mathrel{\hat{=}} \alpha_c.a, \\ & \text{with } \alpha_c \in \mathsf{Expr}_{t_c},\ a \in \textsc{Att}^*_c \\[1em] var_r(R[\alpha_c]), & \text{if } \phi \mathrel{\hat{=}} \alpha_c.r, \\ & \text{with } \alpha_c \in \mathsf{Expr}_{t_c},\ r \in \textsc{Ref} \\[1em] & \text{if } \phi \mathrel{\hat{=}} \gamma(\alpha_1, \dots, \alpha_n), \\ \gamma(R[\alpha_1], \dots, R[\alpha_n]), & \text{with } \gamma \text{ an OCL expression with} \\ & \text{subexpressions } \alpha_1, \dots, \alpha_n \end{cases}$$

and the result of applying R to ϕ should be called var_ϕ:

$$var_\phi = R[\phi]$$

Further let $\{v_{c_1}, \dots v_{c_n}\} = \mathrm{free}(var_\phi)$ be the set of free variables in ϕ with types t_{c_i} for $i = 1, \dots n$. The interpretation function $\underline{var}^\sigma_\phi : I(c_1) \times \dots \times I(c_n) \to I(t)$ is defined as:

$$\underline{var}^\sigma_\phi(\underline{c}_1, \dots, \underline{c}_n) = I\left[\left[var_\phi\right]\right](\langle\sigma, \{v_{c_1} \to \underline{c}_0, \dots, v_{c_n} \to \underline{c}_n\}\rangle).$$

Theorem 3 (Well typed attribute call rewrites)**.** *Let $c \in \textsc{Class}$ be a class, $a : t_c \to t \in \textsc{Att}^*_c$ be an attribute and $\alpha \in \mathit{Expr}_t$ be an OCL expression with $\alpha = \alpha_c.a$ for an OCL expression $\alpha_c \in \mathit{Expr}_{t_c}$, then the expression $R[\alpha]$ is well typed, if $R[\alpha_c]$ is well typed.*

Proof. It is $R[\alpha] = var_a(R[\alpha_c])$, according to the definition of R. For all $a \in \textsc{Att}^*_c$ the meta variable $var_a : t_c \to t$ is defined and has the same return type t as $\alpha_c.a$. Therefore the expression $var_a(R[\alpha_c])$ is defined and has the correct type. □

Theorem 4 (Well typed navigation call rewrites)**.** *Let $c, \hat{c} \in \textsc{Class}$ be classes, $r \in \textsc{Ref}$ be a reference with $r = (c, \hat{c})$ and $\alpha \in \mathit{Expr}_t$ be an OCL expression with $\alpha = \alpha_c.r$ for an OCL expression $\alpha_c \in \mathit{Expr}_{t_c}$, then the expression $R[\alpha]$ is well typed, if $R[\alpha_c]$ is well typed.*

Proof. $R[\alpha] = var_r(R[\alpha_c])$ according to the definition of R. For all $r \in \textsc{Ref}$ with $r = (c, \hat{c})$ the meta variable $var_r : t_c \to \mathsf{Set}(t_{\hat{c}})$ is defined and has the same return type $\mathsf{Set}(t_{\hat{c}})$ as $\alpha_c.r$. Therefore the expression $var_r(R[\alpha_c])$ is defined and has the correct type. □

Theorem 5 (Well typed rewrites)**.** *Let $\phi \in \mathit{Expr}$ an OCL expression, then the expression $R[\phi]$ is well typed.*

Proof. The result of R is well typed in all cases of the definition: Theorem 3 and Theorem 4 show that the result is well typed in the the first and second case. In the third third case the expression $\gamma(R[\alpha_1], \dots, R[\alpha_n])$ is well typed for an OCL expression γ with subexpressions $\alpha_1, \dots, \alpha_n$, because $R[\alpha_i]$ and α_i have the same types, for $i = 1, \dots, n$. □

Theorem 6 (Correctness of var_ϕ)**.** *Let $c_0, \ldots, c_n \in \textsc{Class}$ be classes, t be a type and $\phi \in Expr_t$ be an OCL expression with $\underline{var}^{\sigma}_{\phi} : I(c_0) \times \ldots \times I(c_n) \to I(t)$ and* $\mathrm{free}(\phi) = \{v_1, \ldots, v_n\}$. *The rewrite var_ϕ is correct, that is*

(i) *If the* Get-Equality *and* Put-Equality *hold for all meta variables var_r and var_a, then the* Get-Equality *and* Put-Equality *hold for var_ϕ*

$$\begin{aligned}
&\forall c \in \textsc{Class}\, \forall \underline{c} \in \sigma_{\textsc{Class}}(c)\\
&\quad (\forall a \in \textsc{Att}^*_c(\underline{var}^{\sigma}_a(\underline{c}) = \sigma_{\textsc{Att}}(a)(\underline{c}))\\
&\quad \wedge \forall \hat{c} \in \textsc{Class}\, \forall r = \langle c, \hat{c}\rangle \in \textsc{Ref}\ (\underline{var}^{\sigma}_r(\underline{c}) = L(r)(\underline{c})))\\
&\qquad \Longrightarrow \forall \underline{c}_0 \in \sigma_{\textsc{Class}}(c_0) \ldots \forall \underline{c}_n \in \sigma_{\textsc{Class}}(c_n)(\underline{\phi}^{\sigma}_{v_1,\ldots,v_n}(\underline{c}_0, \ldots, \underline{c}_n) = \underline{var}^{\sigma}_{\phi}(\underline{c}_0, \ldots, \underline{c}_n))
\end{aligned}$$

(ii) *There are no direct property or navigation calls in var_ϕ.*

Proof. The correctness follows immediately from the definition. □

Theorem 7 (Get-Equality and Put-Equality for var_ϕ)**.** *If var_ϕ is correct and for all attributes $a \in \textsc{Att}$ and classes $c \in \textsc{Class}$ the meta variables var_c and var_r are correct, then var_ϕ satisfies the* Get-Equality *and* Put-Equality*. This means:*

(i) *For a unmodified target model directly obtained from a query, var_ϕ has the same value as ϕ:*

$$\begin{aligned}
&\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target})(m_t = q(m_s) \to \qquad\qquad (\textsc{Get-Equality})\\
&\forall \underline{c}_0 \in \sigma^{m_s}_{\textsc{Class}}(c_0) \ldots \forall \underline{c}_n \in \sigma^{m_s}_{\textsc{Class}}(c_n)(\underline{\phi}^{\sigma^{m_s}}_{v_1,\ldots,v_n}(\underline{c}_1, \ldots, \underline{c}_n) = \underline{var}^{\sigma^{m_s \cup m_t}}_{\phi}(\underline{c}_1, \ldots, \underline{c}_n))
\end{aligned}$$

(ii) *After the translation of a target model back to a source model, var_ϕ has the same value as ϕ:*

$$\begin{aligned}
&\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target})(m'_s = q^{-1}(m_t, m_s) \to \qquad\qquad (\textsc{Put-Equality})\\
&\forall \underline{c}_0 \in \sigma^{m'_s}_{\textsc{Class}}(c_0) \ldots \forall \underline{c}_n \in \sigma^{m'_s}_{\textsc{Class}}(c_n)(\underline{\phi}^{\sigma^{m'_s}}_{v_1,\ldots,v_n}(\underline{c}_1, \ldots, \underline{c}_n) = \underline{var}^{\sigma^{m_s \cup m_t}}_{\phi}(\underline{c}_1, \ldots, \underline{c}_n)))
\end{aligned}$$

Proof. The Get-Equality and Put-Equality for var_ϕ follows directly from the correctness of var_ϕ and the Get-Equality and Put-Equality of var_c and var_r. □

4.3.2. Rewriting for New Source Instances

Next we want to rewrite an expression $\phi(\underline{c})$, such that it does not depend on an instance $\underline{c}$ of a source class anymore. For this purpose new meta variables are introduced.

Definition 15 (target meta variable for attributes)**.** Let $c, c_t \in$ Class be classes with $c \sim_{\bowtie} c_t$ and $a, a_t \in \textsc{Att}^*_c$ be attributes with $a \sim_{\bowtie} a_t$, then the initial value of the *target meta variable* $var_a^{c_t} : \mathsf{Expr}_{t_{c_t}} \rightarrow \mathsf{Expr}_t$ is defined as

$$var_a^{c_t}(\alpha) := \alpha.a_t$$

In addition we define the interpretation function $\underline{var}_a : I(c_t) \rightarrow I(t)$ of $var_a^{c_t}$ as

$$\underline{var}_a^{c_t,\sigma}(\underline{c}_t) = I\,[[var_a^{c_t}(\mathsf{v})]]\,(\langle\sigma, \{\mathsf{v} \rightarrow \underline{c}_t\}\rangle).$$

Theorem 8 (correctness of target meta variables for attributes)**.** *A target meta variable $var_a^{c_t}$ for an attribute $a : t_c \rightarrow t \in \textsc{Att}$ is called correct, if*

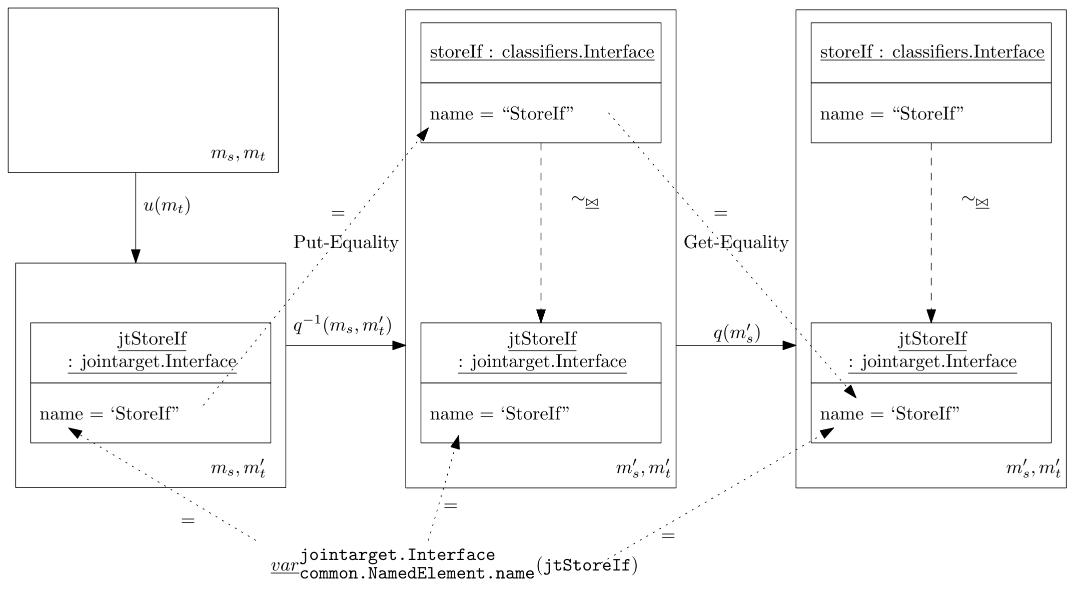
(i) *For a unmodified target model directly obtained from a query, it has the same value as the corresponding attribute:*

$$\begin{aligned}&\forall m_s \in I(M_{source})\forall m_t \in I(M_{target})(m_t = q(m_s) \rightarrow && (\textsc{Get-Equality})\\ &\quad \forall \underline{c} \in \sigma^{m_s}_{\textsc{Class}}(c)\forall \underline{c}_t \in \sigma^{m_t}_{\textsc{Class}}(c_t)(\underline{c} \sim_{\underline{\bowtie}} \underline{c}_t \rightarrow \sigma^{m_s}_{\textsc{Att}}(a)(\underline{c}) = \underline{var}_a^{c_t,\sigma^{m_s \cup m_t}}(\underline{c}_t)))\end{aligned}$$

(ii) *And after the translation of a target model back to a source model, the corresponding attribute has the same value as the meta variable:*

$$\begin{aligned}&\forall m_s \in I(M_{source})\forall m_t \in I(M_{target})(m'_s = q^{-1}(m_t, m_s) \rightarrow && (\textsc{Put-Equality})\\ &\quad \forall \underline{c} \in \sigma^{m'_s}_{\textsc{Class}}(c)\forall \underline{c}_t \in \sigma^{m_t}_{\textsc{Class}}(c_t)(\underline{c} \sim_{\underline{\bowtie}} \underline{c}_t \rightarrow \sigma^{m'_s}_{\textsc{Att}}(a)(\underline{c}) = \underline{var}_a^{c_t,\sigma^{m_s \cup m_t}}(\underline{c}_t)))\end{aligned}$$

The meta variable $var_a^{c_t}$ is not defined for all target classes c_t and attributes a with $a \sim_{\bowtie} a_t$ may not exist. However if it is defined, we can reason about attribute values of source class instances, that do not exist yet and get created by the translation of the corresponding newly created target class instances. If the Put-Equality holds for a chosen translation, then the attribute of the created source instance will get the same value as the target meta variable. If further the Get-Equality holds we can show that the PutGet property holds, even if new instances are created (see Figure 4.3).



m_s, m_t
$u(m_t)$
jtStoreIf : jointarget.Interface
name = 'StoreIf"
m_s, m'_t
$q^{-1}(m_s, m'_t)$
storeIf : classifiers.Interface
name = "StoreIf"
$\sim_{\underline{\bowtie}}$
jtStoreIf : jointarget.Interface
name = 'StoreIf"
m'_s, m'_t
$q(m'_s)$
storeIf : classifiers.Interface
name = "StoreIf"
$\sim_{\underline{\bowtie}}$
jtStoreIf : jointarget.Interface
name = 'StoreIf"
m'_s, m'_t
=
Put-Equality
=
Get-Equality
=
=
=
$\underline{var}^{\texttt{jointarget.Interface}}_{\texttt{common.NamedElement.name}}(\texttt{jtStoreIf})$

Figure 4.3.: The Update operation u creates a new target model class instance jtStoreId $\in I$(jointarget.Interface). The Put-Equality ensures that for the corresponding source class instance storeIf created by the translation q^{-1} the value of the attribute name equals to the target meta variable. The Get-Equality ensures that after the query q the value does not change either.

In a similar way navigation calls can be rewritten.

Definition 16 (target meta variable for references)**.** Let $c, \hat{c}, c_t, \hat{c}_t \in$ Class be classes and $r, \hat{r} \in$ Ref be references with $associates(r) = \langle c, \hat{c} \rangle$, $associates(r_t) = \langle c_t, \hat{c}_t \rangle$ and $r \sim r_t$, then the initial value of the *target meta variable* $var_r^{c_t} : \mathsf{Expr}_{t_c} \to \mathsf{Expr}_{\mathsf{Set}(t_{\hat{c}_t})}$ is:

$$var_r^{c_t}(\alpha) := \alpha.r_t$$

In addition we define the interpretation function $\underline{var}_r : I(c) \to \mathcal{P}(I(\hat{c}))$ of var_r as

$$\underline{var}_r^{c_t,\sigma}(\underline{c}) = I\left[\left[var_r^{c_t}(\mathsf{v})\right]\right](\langle \sigma, \{\mathsf{v} \to \underline{c}\} \rangle).$$

Theorem 9 (correctness of target meta variables for references)**.** *A target meta variable $var_r^{c_t}$ for a reference $r \in$ Ref with $associates(r) = \langle c, \hat{c} \rangle$ is called correct, if*

(i) *For a unmodified target model obtained from a query, it has links to mapped instances of the corresponding links in the relation:*

$$\begin{aligned} &\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target})(m_t = q(m_s) \to \qquad \text{(Get-Equality)}\\ &\quad \forall \underline{c} \in \sigma^{m_s}_{CLASS}(c) \forall \underline{c}_t \in \sigma^{m_t}_{CLASS}(c_t)(\underline{c} \sim_{\underline{\bowtie}} \underline{c}_t \to\\ &\qquad (\forall \underline{\hat{c}} \in L(r)(\underline{c}) \to \exists \underline{\hat{c}}_t \in \underline{var}_r^{\sigma^{m_s \cup m_t}}(\underline{c}_t)(\underline{\hat{c}} \sim_{\underline{\bowtie}} \underline{\hat{c}}_t)\\ &\qquad \wedge (\forall \underline{\hat{c}}_t \in \underline{var}_r^{\sigma^{m_s \cup m_t}}(\underline{c}_t) \to \exists \underline{\hat{c}} \in L(r)(\underline{c})(\underline{\hat{c}} \sim_{\underline{\bowtie}} \underline{\hat{c}}_t))) \end{aligned}$$

(ii) *After the translation of a target model back to a source model, the corresponding relation has links to mapped instances of the links of the meta variable:*

$$\begin{aligned}&\forall m_s \in I(M_{source})\forall m_t \in I(M_{target})(m_s' = q^{-1}(m_t, m_s) \rightarrow \\ &\quad\forall \underline{c} \in \sigma^{m_s'}_{\textsc{Class}}(c)\forall \underline{c}_t \in \sigma^{m_t}_{\textsc{Class}}(c_t)(\underline{c} \sim_{\underline{\bowtie}} \underline{c}_t \rightarrow \\ &\quad\quad(\forall \underline{\hat{c}} \in L(r)(\underline{c}) \rightarrow \exists \underline{\hat{c}}_t \in \underline{var}_r^{\sigma^{m_s \cup m_t}}(\underline{c}_t)(\underline{\hat{c}} \sim_{\underline{\bowtie}} \underline{\hat{c}}_t) \\ &\quad\wedge(\forall \underline{\hat{c}}_t \in \underline{var}_r^{\sigma^{m_s \cup m_t}}(\underline{c}_t) \rightarrow \exists \underline{\hat{c}} \in L(r)(\underline{c})(\underline{\hat{c}} \sim_{\underline{\bowtie}} \underline{\hat{c}}_t)))\end{aligned} \qquad \text{(Put-Equality)}$$

Note that if a navigation call is rewritten, then the type of the expression changes. Therefore all expressions, that are using the result, have to be rewritten as well. For example the handling of iterator expressions over relations need special care. Consider the following OCL expression, which counts all interface methods named “onEvent”:

```
φ(v) = v.members->select(m | m.name = “onEvent”)->size()
```

Replacing v.members with $var^{\text{jointarget.Interface}}_{\text{members}}(v_t)$, where members is an abbreviation for the reference members.MemberContainer.members, changes the type in the body of the iterator. Therefore all usages of v inside the body must be replaced as well.

A proper rewrite could be:

```
φ(v_t) = var^{jointarget.Interface}_{members}(v_t)->select(m |
  var^{jointarget.Method}_{name}(m) = “onEvent”
)->size()
```

In the general case we can define an rewrite function $R_v^{c \rightarrow c_t}$ that changes the type of the free variable v inside the expression from c to c_t and rewrites all subexpressions to use the meta variables.

Definition 17 (Rewritten expression)**.** Let $\phi \in \mathsf{Expr}_t$ be an OCL expression and $c, c_t \in$ $\textsc{Class}$ be classes. Further let $v \in \text{free}(\phi)$ be a free variable in ϕ of type t_c, then the *target rewrite function* $R_v^{c \rightarrow c_t} : \mathsf{Expr}_t \rightarrow \mathsf{Expr}_{t'}$ is defined as

$$R_v^{c \to c_t}[\phi] = \begin{cases} v : c_t \ , & \text{if } \phi \mathrel{\hat{=}} v : c \\ var_a^{c_t}(R_v^{c \to c_t}[\alpha_c]) \ , & \begin{array}{l} \text{if } \phi \mathrel{\hat{=}} \alpha_c.a \\ \text{with } \alpha_c \in \mathsf{Expr}_{t_c},\ a : t_{c'} \to t \in \textsc{Att}_c^* \end{array} \\ var_a^{\hat{c}_t}(R_v^{c \to c_t}[\alpha_{\hat{c}}]) \ , & \begin{array}{l} \text{if } \phi \mathrel{\hat{=}} \alpha_{\hat{c}}.a \text{ and } c \neq \hat{c}, \\ \text{with } R_v^{c \to c_t}[\alpha_{\hat{c}}] \in \mathsf{Expr}_{t_{\hat{c}_t}},\ \alpha_{\hat{c}} \in \mathsf{Expr}_{t_{\hat{c}}} \\ \hat{c}, \hat{c}_t \in \textsc{Class},\ \hat{c} \sim_{\bowtie} \hat{c}_t, a \in \textsc{Att}_{\hat{c}}^* \end{array} \\ var_r^{c_t}(R_v^{c \to c_t}[\alpha_c]) \ , & \begin{array}{l} \text{if } \phi \mathrel{\hat{=}} \alpha_c.r, \\ \text{with } \alpha_c \in \mathsf{Expr}_{t_c},\ r \in \textsc{Ref} \end{array} \\ var_r^{\hat{c}_t}(R_v^{c \to c_t}[\alpha_r]) \ , & \begin{array}{l} \text{if } \phi \mathrel{\hat{=}} \alpha_{\hat{c}}.r \text{ and } c \neq \hat{c}, \\ \text{with } R_v^{c \to c_t}[\alpha_{\hat{c}}] \in \mathsf{Expr}_{t_{\hat{c}_t}},\ \alpha_{\hat{c}} \in \mathsf{Expr}_{t_{\hat{c}}} \\ \hat{c}, \hat{c}_t \in \textsc{Class},\ \hat{c} \sim_{\bowtie} \hat{c}_t, r \in \textsc{Ref} \end{array} \\ \begin{array}{l} \mathbf{let}\ v_1 = R_v^{c \to c_t}[\alpha_1], \ldots \\ \mathbf{in}\ R_{v_1}^{c \to c_t}\left[\ldots R_{v_n}^{c \to c_t}[\alpha] \ldots\right] \end{array} , & \begin{array}{l} \text{if } \phi \mathrel{\hat{=}} \mathbf{let}\ v_1{=}\alpha_1, \ldots\ \mathbf{in}\ \alpha \\ \alpha_1 \in \mathsf{Expr}_{t_c},\ \alpha \in \mathsf{Expr} \end{array} \\ \begin{array}{l} R_V^{c \to c_t}\left[\alpha_{\mathsf{Set}(\hat{c})}\right] \text{->} iter(v_1, \ldots v_n \mid \\ \quad R_{v_1}^{c \to c_t}\left[\ldots R_{v_n}^{c \to c_t}\left[R_v^{\hat{c} \to \hat{c}_t}[\alpha]\right] \ldots\right] \ , \\ ) \end{array} & \begin{array}{l} \text{if } \phi \mathrel{\hat{=}} \alpha_{\mathsf{Set}(\hat{c})}\text{->}iter(v_1, \ldots v_n \mid \alpha) \\ \text{with } \alpha_{\mathsf{Set}(\hat{c})} \in \mathsf{Expr}_{\mathsf{Set}(t_{\hat{c}})},\ \alpha \in \mathsf{Expr} \\ \hat{c}, \hat{c}_t \in \textsc{Class},\ c \sim_{\bowtie} c_t,\ r = \langle c, \hat{c} \rangle \in \textsc{Ref} \\ v_1, \ldots, v_n \in \mathsf{Var}_{t_{\hat{c}}} \end{array} \\ \gamma(R_v^{c \to c_t}[\alpha_1], \ldots, R_v^{c \to c_t}[\alpha_n]), & \begin{array}{l} \text{if } \phi \mathrel{\hat{=}} \gamma(\alpha_1, \ldots, \alpha_n), \\ \text{with } \gamma \text{ an OCL expression with} \\ \text{subexpressions } \alpha_1, \ldots, \alpha_n \end{array} \end{cases}$$

The result of applying the k target rewrite function $R_{v_1}^{c_1 \to c_{t,1}}, \ldots, R_{v_k}^{c_k \to c_{t,k}}$ to ϕ should be called $var_{\phi, v_1, \ldots, v_k}^{c_1 \to c_{t,1}, \ldots, c_k \to c_{t,k}}$:

$$var_{\phi, v_1, \ldots, v_k}^{c_1 \to c_{t,1}, \ldots, c_k \to c_{t,k}} = R_{v_1}^{c_1 \to c_{t,1}}\left[\ldots R_{v_k}^{c_k \to c_{t,k1}}[\phi] \ldots\right]$$

Further let $\{\tilde{v}_{c_1}, \ldots \tilde{v}_{c_n}\} = \mathrm{free}(var_{\phi, v_1, \ldots, v_k}^{c_1 \to c_{t,1}, \ldots, c_k \to c_{t,k}})$ the set of free variables in $var_{\phi, v_1, \ldots, v_k}^{c_1 \to c_{t,1}, \ldots, c_k \to c_{t,k}}$ with types $t_{\tilde{c}_i}$ for $i = 1, \ldots n$. The interpretation function $\underline{var}_{\phi, v_1, \ldots, v_k}^{c_1 \to c_{t,1}, \ldots, c_k \to c_{t,k}, \sigma} : I(\hat{c}_1) \times \ldots \times I(\hat{c}_n) \to I(t)$ is defined as:

$$\underline{var}_{\phi, v_1, \ldots, v_k}^{c_1 \to c_{t,1}, \ldots, c_k \to c_{t,k}, \sigma}(\underline{\hat{c}}_1, \ldots, \underline{\hat{c}}_n) = I\left[\left[var_{\phi, v_1, \ldots, v_k}^{c_1 \to c_{t,1}, \ldots, c_k \to c_{t,k}}\right]\right](\langle \sigma, \{\hat{v}_{c_1} \to \underline{\hat{c}}_0, \ldots, \hat{v}_{c_n} \to \underline{\hat{c}}_n\} \rangle).$$

To show an application of the target rewrite function, we rewrite the expression $\phi = v$.members->select(m | m.name = “onEvent”)->size(). To save space we abbreviate jointarget with jt, classifier with cl and Interface with If:

$$
\begin{aligned}
R_v^{\text{cl.If}\to\text{jt.If}}[\phi] &= R_v^{\text{cl.If}\to\text{jt.If}}[v\text{.members->select(m | m.name = “onEvent”)->size()}]\\
&= R_v^{\text{cl.If}\to\text{jt.If}}[v\text{.members->select(m | m.name = “onEvent”)}]\text{ ->size()}\\
&= \begin{array}{l} R_v^{\text{cl.If}\to\text{jt.If}}[v\text{.members}]\text{ ->select(m |}\\ \quad R_m^{\text{m.Member}\to\text{jt.Method}}\left[R_v^{\text{cl.If}\to\text{jt.If}}[\text{m.name = “onEvent”}]\right]\\ \text{)->size()}\end{array}\\
&= \begin{array}{l} var^{\text{jt.If}}_{\text{members}}(R_v^{\text{cl.If}\to\text{jt.If}}[c])\text{->select(m |}\\ \quad R_m^{\text{m.Member}\to\text{jt.Method}}\left[R_v^{\text{cl.If}\to\text{jt.If}}[\text{m.name = “onEvent”}]\right]\\ \text{)->size()}\end{array}\\
&= \begin{array}{l} var^{\text{jt.If}}_{\text{members}}(v)\text{->select(m |}\\ \quad R_m^{\text{m.Member}\to\text{jt.Method}}\left[R_v^{\text{cl.If}\to\text{jt.If}}[\text{m.name = “onEvent”}]\right]\\ \text{)->size()}\end{array}\\
&= \begin{array}{l} var^{\text{jt.If}}_{\text{members}}(v)\text{->select(m |}\\ \quad R_m^{\text{m.Member}\to\text{jt.Method}}\left[R_v^{\text{cl.If}\to\text{jt.If}}[\text{m.name}] = R_v^{\text{cl.If}\to\text{jt.If}}[\text{“onEvent”}]\right]\\ \text{)->size()}\end{array}\\
&= \begin{array}{l} var^{\text{jt.If}}_{\text{members}}(v)\text{->select(m |}\\ \quad R_m^{\text{m.Member}\to\text{jt.Method}}\left[R_v^{\text{cl.If}\to\text{jt.If}}[\text{m}]\text{.name = “onEvent”}\right]\\ \text{)->size()}\end{array}\\
&= \begin{array}{l} var^{\text{jt.If}}_{\text{members}}(v)\text{->select(m |}\\ \quad R_m^{\text{m.Member}\to\text{jt.Method}}[\text{m.name = “onEvent”}]\\ \text{)->size()}\end{array}\\
&= \begin{array}{l} var^{\text{jt.If}}_{\text{members}}(v)\text{->select(m |}\\ \quad R_m^{\text{m.Member}\to\text{jt.Method}}[\text{m.name}] = \text{“onEvent”}\\ \text{)->size()}\end{array}\\
&= \begin{array}{l} var^{\text{jt.If}}_{\text{members}}(v)\text{->select(m |}\\ \quad var^{\text{jt.Method}}_{\text{name}}(\text{m}) = \text{“onEvent”}\\ \text{)->size()}\end{array}
\end{aligned}
$$

However in general a rewrite $R_v^{c\to c_t}[\phi]$ for an expression ϕ can be undefined or not well-typed. The rewrite is undefined, if one of the meta variables in the resulting expression is undefined. It can be not well-typed, if the resulting types do not confirm. An example for a non well-typed rewrite is the result of the following rewrite of the expression $\phi = \text{v1} = \text{v2}$ with the two free variables v1, v2 of type t_c:

$$\phi' := R_{\text{v2}}^{c\to c_t}[\phi] = R_{\text{v2}}^{c\to c_t}[\text{v1} = \text{v2}] = R_{\text{v2}}^{c\to c_t}[\text{v1}] = R_{\text{v2}}^{c\to c_t}[\text{v2}] = \text{v1} = \text{v2}$$

The result ϕ' is not well-typed, because v1 has type t_{c_t}, however v2 still has type t_c (after the rewrite) and equality is not defined for different class types without a subclass relationship.

4.3.3. Class Instance Meta Functions

For some constraints we need to get the set of all source class instance of a particular class $c \in \textsc{Class}$. For this purpose we define the *class instances meta functions*:

Definition 18 (class instances meta function)**.** Let $c \in \textsc{Class}$ be a source class. The initial value of the *class instances meta function* $\mathit{Instances}_c \in \mathsf{Expr}_{\mathsf{Set}(t_c)}$ is:

$$\mathit{Instances}_c := c.\mathsf{allInstances()}$$

So initially the set contains all instances of the corresponding type. However if instances shall get deleted on update translation, we exclude them from the set. We define two essential properties for the class instance meta functions:

Theorem 10 (correctness of class instance meta functions)**.** *A class instances meta function $\mathit{Instances}_c \in \mathit{Expr}_{\mathsf{Set}(t_c)}$ is correct, if*

(i) *For a unmodified target model obtained from a query, $\mathit{Instances}_c$ contains all existing instances:*

$$\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target}) \qquad \text{(Get-Equality)}$$
$$(m_t = q(m_s) \rightarrow \sigma^{m'_s}_{\textsc{Class}}(c) = \underline{\mathit{Instances}_c})$$

(ii) *After the translation of a target model back to a source model, $\mathit{Instances}_c$ contains all source class instances of c, which existed before the translation and exist after the translation:*

$$\forall m_s \in I(M_{source}) \forall m_t \in I(M_{target}) \qquad \text{(Put-Equality)}$$
$$(m'_s = q^{-1}(m_t, m_s) \rightarrow \sigma^{m_s}_{\textsc{Class}}(c) \cap \sigma^{m'_s}_{\textsc{Class}}(c) = \underline{\mathit{Instances}_c})$$

4.4. The Trace Model

The mapping between source class instances and target class instances is originally only implicitly given by the ModelJoin view definition Q. Like proposed in Definition 10 of chapter 3.4 an explicit trace model should be introduced.

The trace model will contain trace classes, which represent the mapping between source and target class instances. These trace classes allow the prohibition of update operations, that would change the mapping of a target class instance to a different source class instance. We want to ensure, that a target class instance $\underline{c_t} \in I(c_t)$ with a mapping $\underline{c} \sim_{\bowtie}$ to a source class instance $\underline{c} \in I(c)$ does not map to a different source class instance $\underline{c'} \in I(c)$ with $\underline{c} \neq \underline{c'}$ after the update. Because the target class instances represent their corresponding source class instances, this is a desired property. The user does not have to worry about changing the identity of the target class instance $\underline{c}_t$ behind the scene, by accident, through applying an update operation.

For example the intention of changing the value of the name attribute of a target class instance $\underline{c}_t \in I(\text{jointarget.Interface})$ is probably to change the name of the corresponding source class instance $\underline{c} \in I(\text{classifier.Interface})$ with $\underline{c} \sim_{\bowtie} \underline{c}_t$ and not to change the mapping of the target class instance $\underline{c}_t$ to a different source class instance. However there may be cases, where adding a new mapping is desired. We further discuss this in chapter 5.1.

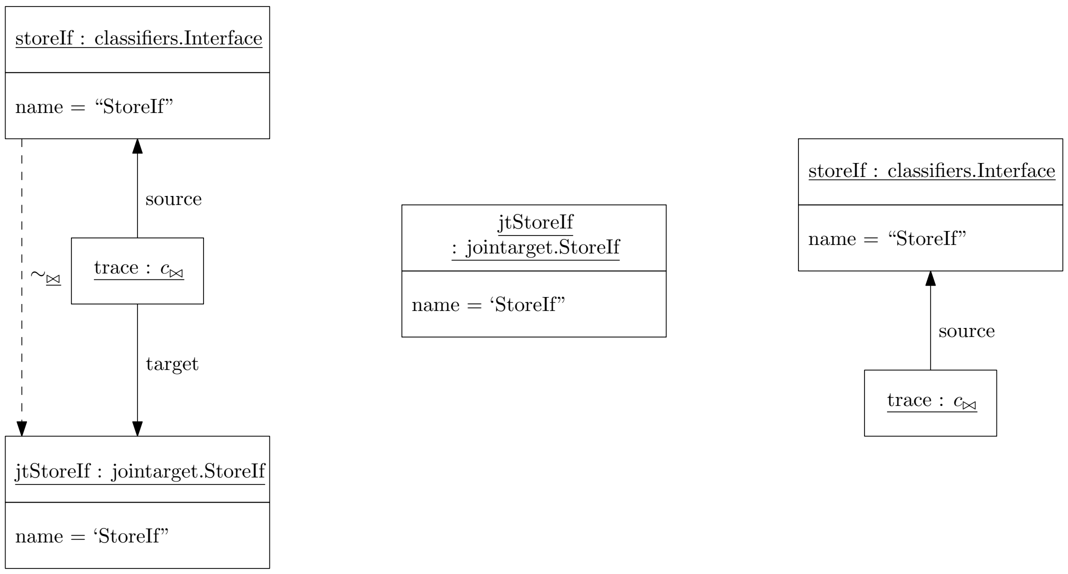


Figure 4.4.: Left: Trace model with unmodified target class instance. Center: Untranslated newly created target class instance with missing trace class instance. Right: Untranslated deleted target class instance leading to a trace class instance with dangling target reference.

The trace model does not only make it easy to maintain the mapping it also allows the easy detection of new and deleted target class instances and links. A trace class instance $\underline{c}_{\bowtie} \in I(c_{\bowtie})$ has links to its source class instances $\underline{c} \in I(c)$ and to its target class instance $\underline{c}_t \in I(c_t)$. We make the trace model non-editable for the user, because it only serves book keeping purpose and is maintained by the query and translation functions. The query function creates the explicit trace class instances with their links. For changed target class instances, these traces will be used to propagate the changes to the corresponding source class instances in the translation function. New target class instances created by update operation can easily be detected, because these have no trace class instance with mapping information. Deleted target class instances can also be easily detected, because the trace class still exists with a dangling reference to the deleted target class instance (see Figure 4.4). The anatomy of the trace classes will be defined in more detail in the following section.

While in practice the trace model is created as a separate model in addition to the target model when running the query, in our theoretical setting we will consider the trace metamodel as part of the target metamodel M_t and the trace model embedded in the target model $m_t \in I(M_t)$. However we disallow all changes to the trace model in update operations without explicitly specified constraints in the next section. The trace model can be seen as an explicit form of the $\sim_{\underline{\bowtie}}$-Relation, that can be used in OCL expressions.

4.5. Constraints Creation

In this section the definition of the ModelJoin operations from [17] should be extended in two ways: (1) In addition to the target model an explicit trace model, describing the mappings between the source and the target model, should be created. (2) Further for each ModelJoin operation a set of OCL expressions should be derived. The OCL expressions should be chosen in such way, that all target models that satisfy the OCL-constraints can be translated back to the source models.

Definition 19 (Translatable view). Let M_{source} be a source metamodel, M_{target} be a target metamodel with trace model and $q : I(M_{source}) \to I(M_{target})$ be a ModelJoin expression. For a pair of source and target model instances $\langle m_s, m_t \rangle \in I(M_{source}) \times I(M_{target})$ the target model instance m_t is called *translatable*, if there exists a translation q^{-1} that satisfies the GETPUT- and the PUTGET -Property.

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

For the induction we make the following assumption:

Theorem 11 (Induction statement). *Let M_{source} be a source metamodel, M_{target} be a target model and $q : I(M_{source}) \to I(M_{target})$ be a ModelJoin expression with a set of OCL-Constraints C derived according to the following definitions, then the following three statements hold:*

(i) *For all source model instances $m_s \in I(M_{source})$, 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.*

(iii) *The definitions var_a and var_r are correct for all $a \in \mathrm{ATT}^*_c$ and $r \in \mathrm{REF}$.*

First we show that the Induction statement holds for an empty ModelJoin expression. Then we show by induction, that the induction statement holds for a ModelJoin expression extended by an additional ModelJoin operator.

Theorem 12 (Induction base case). *Let M_{source} be a source metamodel, M_{target} be an empty target metamodel and $q : I(M_{source}) \to I(M_{target})$ be a ModelJoin expression, then the Induction statement holds for the empty set $C = \emptyset$ of OCL-constraints. More precisely:*

(i) *For all pairs $\langle m_s, m_t \rangle \in I(M_{source}) \times I(M_{target})$ the target model instance m_t is translatable.*

(ii) *The definitions var_a and var_r are correct for all $a \in \textsc{Att}^*_c$ and $r \in \textsc{Ref}$.*

Proof. Because the target metamodel is empty, in particular does not contain any classes and references, the target model m_t cannot contain any elements. Therefore no class instances can be created or updated and the trivial translation $q^{-1}(m_t, m_s) := m_s$ satisfies the GetPut- and the PutGet-Property. Further, the Get-Equality for var_a and var_r immediately follows from their definitions, because the meta variables map to the source model elements directly. The Put-Equality holds, because no source model elements are changed by q^{-1}. □

4.5.1. Constraints for Join Expressions

The natural join operation is one of the basic join operations of ModelJoin. It takes two classes and creates a new joined class with the join confirming attributes of both classes. To define the corresponding OCL expressions and trace elements, we extend the set of join-confirming attribute pairs $A^{\bowtie}_{c_1,c_2}$ from [17] to include the corresponding target class attributes:

Definition 20 (Join-confirming attributes with mapping). Let $c_1, c_2, c_t \in \textsc{Class}$ be classes, then the set of join confirming attributes with mapping $A^{\bowtie}_{c_1,c_2,c_t}$ is:

$$A^{\bowtie}_{c_1,c_2,c_t} = \{\langle a_1, a_2, a_t \rangle \in \textsc{Att}^*_{c_1} \times \textsc{Att}^*_{c_2} \times \textsc{Att}^*_{c_t} \mid a_1 \tilde{=}_{\textsc{Att}} a_2 \wedge a_1 \sim_{\bowtie} a_t \wedge a_2 \sim_{\bowtie} a_t\}$$

Definition 21 (Extensions for natural join). Let $c_1, c_2 \in \textsc{Class}$ be two source classes, $c_t \in \textsc{Class}$ be a target class and $\bowtie = \langle c_1, c_2, c_t \rangle$ be a natural join operator.

(i) Trace model The trace metamodel is extended by

$$\begin{aligned} \textsc{Class}' &= \textsc{Class} \cup \{c_{\bowtie}\} \\ \textsc{Ref}' &= \textsc{Ref} \cup \{\mathsf{left}, \mathsf{right}, \mathsf{target}\} \end{aligned}$$

with a new class $c_{\bowtie}$ and

$$\begin{aligned} associates(\mathsf{left}) &= \langle c_{\bowtie}, c_1 \rangle, & multiplicity(\mathsf{left}) &= \{1\}, \\ associates(\mathsf{right}) &= \langle c_{\bowtie}, c_2 \rangle, & multiplicity(\mathsf{right}) &= \{1\}, \\ associates(\mathsf{target}) &= \langle c_{\bowtie}, c_t \rangle, & multiplicity(\mathsf{target}) &= \{0, 1\}. \end{aligned}$$

The trace model is extended by additional instances of $c_{\bowtie}$, which represent the mapping:

$$\begin{aligned} &\forall \underline{c}_t \in \sigma_{\textsc{Class}}(c_t) \forall \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1) \forall \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2) \\ &(\underline{c}_1 \sim_{\underline{\bowtie}} \underline{c}_t \wedge \underline{c}_2 \sim_{\underline{\bowtie}} \underline{c}_t \Rightarrow \\ &\exists \underline{c}_{\bowtie} \in \sigma_{\textsc{Class}}(c_{\bowtie})(\underline{c}_1 \in L(\mathsf{left})(\underline{c}_{\bowtie})) \wedge (\underline{c}_2 \in L(\mathsf{right})(\underline{c}_{\bowtie})) \wedge (\underline{c}_t \in L(\mathsf{target})(\underline{c}_{\bowtie}))) \end{aligned}$$

(ii) Constraints The target metamodel should satisfy the following OCL-constraints:

```
context $c_{\bowtie}$
-- If the created attributes of the target class are updated, the new values will be used for the left and right source classes, therefore they must be confirm to all other constraints involving the source attribute values:
inv attributeMappingLeft:
$\mathbf{AND}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$(self.target.$a_t$ = $var_{a_1}$(self.left))
inv attributeMappingRight:
$\mathbf{AND}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$(self.target.$a_t$ = $var_{a_2}$(self.right))
-- The mapping should not change after translation. Exactly the instances which had join confirming attribute values before the update, should have join confirming attributes after the update:
inv keepMappingPairs:
$Instances_{c_1}$->forAll(left |
  $Instances_{c_2}$->forAll(right |
    ($\mathbf{AND}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$(left.$a_1$ = right.$a_2$)) =
    ($\mathbf{AND}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$($var_{a_1}$(left) = $var_{a_2}$(right)))
  )
)
```

The following constraints for the target class c_t determine how new instances are handled. For simplicity the creation of new target class instances, mapping to existing source class instances after the update translation, will be forbidden. However other strategies are possible and lead to different constraints. We will discuss them further in chapter 5.1.

```
context $c_t$
-- New created target class instances should not map to old source class instances:
inv newTargetInstancesLeft:
isNew(self) implies $Instances_{c_1}$->forAll( left | $\mathbf{OR}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$(self.$a_t$ <> $var_{a_1}$(left)))
inv newTargetInstancesRight:
isNew(self) implies $Instances_{c_2}$->forAll( right | $\mathbf{OR}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$(self.$a_t$ <> $var_{a_2}$(right)))
-- New created target class instances should have no join confirming attributes with other new created target class instances
inv newTargetInstances:
isNew(self) implies $c_t$.allInstances->select(other |
  isNew(other) and other <> self
)->forAll(other |
  $\mathbf{OR}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$(self.$a_t$ <> other.$a_t$)
)
-- The target meta variables must confirm to the corresponding attributes:
inv newAttributesLeft:
isNew(self) implies $\mathbf{AND}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$(self.$a_t$ = $var^{c_t}_{a_1}(self)$)
inv newAttributesRight:
isNew(self) implies $\mathbf{AND}_{\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1, c_2, c_t}}$(self.$a_t$ = $var^{c_t}_{a_2}(self)$)
```

```
-- New instances may only be created if the source class is not mapped anywhere else:
inv noConflictsWithOtherMappings:
$\exists c'_t \in$ CLASS$(c'_t \neq c_t \wedge (c_1 \sim_{\bowtie} c'_t \vee c_1 \sim_{\bowtie} c'_t) \Rightarrow c_{\bowtie}$.allInstances()->select(nj| nj.target = self
)->notEmpty())
```

with the substitution

```
isNew($v$) := $c_{\bowtie}$.allInstances()->select(nj | nj.target = $v$)->isEmpty()
```

Finally we must ensure that if an instance of c_t gets deleted we can delete both corresponding source class instances of c_1 respectively c_2. Other deletion strategies are possible and will be discussed later in chapter 5.2.

```
context $c_{\bowtie}$
-- If a target class instance got deleted, the source class instances get delete. So if a source class
instance got deleted, ensure that all corresponding target class instances got deleted
inv consistentDeletionLeft:
not $Instances_{c_1}$->includes(self.left) implies self.target.oclIsUndefined()
inv consistentDeletionRight:
not $Instances_{c_2}$->includes(self.right) implies self.target.oclIsUndefined()
```

(iii) Meta variables The attributes of the target class can be used as canonical attribute for the meta variables var_{a_1} and var_{a_2} (Recall Section 4.2.1 Figure 4.2). We therefore extend the definition of the meta variables var_{a_1} and var_{a_2} to use the corresponding target class attributes a_t. If an instance of the source class c_1 respectively c_2 is mapped by an instance of the trace class $c_{\bowtie}$ to an instance of the target class c_t, then the value of the attribute a_t of the target class is returned. Otherwise the original definition is used:

```
$\forall \langle a_1, a_2, a_t \rangle \in A^{\bowtie}_{c_1,c_2,c_t}$
$var'_{a_1}(v_1)$ := (let j = $c_{\bowtie}$.allInstances()->select(j| j.left = $v_1$)->asOrderedSet()
  in if j->notEmpty() and not j->first().target.oclIsUndefined() then
    j->first().target.$a_t$
  else
    $var_{a_1}(v_1)$
  endif
)
$var'_{a_2}(v_2)$ := (let j = $c_{\bowtie}$.allInstances()->select(j| j.right = $v_2$)->asOrderedSet()
  in if j->notEmpty() and not j->first().target.oclIsUndefined() then
    j->first().target.$a_t$
  else
    $var_{a_2}(v_2)$
  endif
)
```

If an instance of the target class gets deleted and leads to the deletion of the source class instance at update translation, it can be the case that dangling references to the deleted source classes must be deleted as well. This must be respected in the meta variable definition of all incoming references. The references to source class instances, which get deleted by the update translation, are excluded:

$\forall c \in \textsc{Class}\ \forall r \in \{r \in \textsc{Ref} \mid \mathit{associates}(r) = \langle c, c_1 \rangle\}$
$\mathit{var}'_r(v) := \mathit{var}_r(v)$->exclude(c | *toDeleteLeft*(c))
$\forall c \in \textsc{Class}\ \forall r \in \{r \in \textsc{Ref} \mid \mathit{associates}(r) = \langle c, c_2 \rangle\}$
$\mathit{var}'_r(v) := \mathit{var}_r(v)$->exclude(c | *toDeleteRight*(c))

with the substitutions

toDeleteLeft(v_1) := (**let** j = $c_{\bowtie}$.allInstances()->select(j| j.left = v_1)->asOrderedSet()
  in j->notEmpty() **and** j->first().target.oclIsUndefined()
)
toDeleteRight(v_2) := (**let** j = $c_{\bowtie}$.allInstances()->select(j| j.right = v_2)->asOrderedSet()
  in j->notEmpty() **and** j->first().target.oclIsUndefined()
)

(iv) Instance meta functions To exclude the deleted class instances in constraints for other ModelJoin expressions, we extend the definition of the class instance meta functions:

$\mathit{Instances}'_{c_1} := \mathit{Instances}_{c_1}$->exclude(left | *toDeleteLeft*(left))
$\mathit{Instances}'_{c_2} := \mathit{Instances}_{c_2}$->exclude(left | *toDeleteRight*(right))

Definition 22 (Translation for natural join)**.** Let $\bowtie = \langle c_1, c_2, c_t \rangle$ be a natural join operator and $\underline{c}_t \in \sigma_{\textsc{Class}}(c_t)$ be a target class instance. The instance $\underline{c}_t$ should be translated according the following rule:

(i) If there exists a trace class instance $\underline{c}_{\bowtie} \in \sigma_{\textsc{Class}}(c_{\bowtie})$ with $L(\texttt{target})(\underline{c}_{\bowtie}) = \underline{c}_t$, then for all $\langle a_1, a_2, a_t \rangle \in A^{\bowtie}_{c_1,c_2,c_t}$ emit the update operations $\textsc{update}_{\textsc{Att}(a_1)}(\underline{c}_1, \sigma_{\textsc{Att}}(a_t)(\underline{c}_t))$ and $\textsc{update}_{\textsc{Att}(a_2)}(\underline{c}_2, \sigma_{\textsc{Att}}(a_t)(\underline{c}_t))$ where $\underline{c}_1 \in L(\texttt{left})(\underline{c}_{\bowtie})$ and $\underline{c}_2 \in L(\texttt{right})(\underline{c}_{\bowtie})$.

(ii) Else the two create class instance updates $\textsc{create}_{\textsc{Class}(c_1)}(V_1)$, $\textsc{create}_{\textsc{Class}(c_2)}(V_2)$ with the attribute value sets $V_1 = \{v_{a_1} = \sigma_{\textsc{Att}}(a_t)(\underline{c}_t) \mid \langle a_1, a_2, a_t \rangle \in A^{\bowtie}_{c_1,c_2,c_t}\}$ and $V_2 = \{v_{a_2} = \sigma_{\textsc{Att}}(a_t)(\underline{c}_t) \mid \langle a_1, a_2, a_t \rangle \in A^{\bowtie}_{c_1,c_2,c_t}\}$ should be emitted.

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

Theorem 13 (Induction step for natural join)**.** *Let $q : I(M_{source}) \rightarrow I(M_{target})$ be a ModelJoin expression with a set of OCL-Constraints C for which the Induction statement holds. If q is extended by a arbitrary natural join expression to $q' : I(M_{source}) \rightarrow I(M'_{target})$ 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*

(iii) *The new definitions for var'_{a_1}, var'_{a_2} for all $\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}$ are correct.*

(iv) *The new definitions for the class instance meta functions Instances$'_{c_1}$ and Instances$'_{c_2}$ are correct.*

Proof. Let $c_1, c_2 \in$ Class be the source classes and $c_t \in$ Class the target class of the natural join expression: $\bowtie = \langle c_1, c_2, c_t\rangle$.

First (i) is proven. Therefore let $m_s \in I(M_{source})$ be a source model instance.

We first want to show, that $m_t = q'(m_s)$ satisfies all constraints in C. All constraints in C do not depend directly on instances of c_t, because c_t does not exist in the ModelJoin view definition without the join expression. Constraints in C can only depend on instances of c_t by the usage of meta variables. However if all the new definition var'_{a_1}, var'_{a_2} are correct, which will be shown in (iii), then the Get-Equality assures that the new definitions have the same values as the original ones. So 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 invariant attributeMappingLeft is satisfied because for all $\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}$ it is var_{a_1}(self.left) = self.left.a_1 according to the Get-Equality of var_{a_1} and self.left.a_1 = self.target.a_t according to the natural join definition. All links of `self` $\in \sigma_{\text{Class}}(c_{\bowtie})$ are not empty according to the the natural join definition. The analog argument can be used to show that attributeMappingRight is satisfied.

The invariant keepMappingPairs is satisfied immediately because for all $\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}$ it is var_{a_1}(left) = left.a_1 and var_{a_2}(right) = right.a_2 according to the Get-Equality of var_{a_1} and var_{a_2}.

All invariants in the context of c_t are immediately true, because the set $c_{\bowtie}$.allInstances()->select(nj | nj.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 $\underline{c}_{\bowtie} \in \sigma_{\text{Class}}(c_{\bowtie})$ according to the natural join definition.

Next (ii) should be shown. We show that for a given pair $\langle m_s, m'_t\rangle \in I(M_{source}) \times I(M'_{target})$ the target model instance m'_t can be translated, by constructing a translation q'^{-1}. Because m'_t without the instance of the class c_t is translatable according to the premise, it will just be shown that creation, update and deletion operations for c_t instances are translatable.

Let $\underline{c}_t$ be an instance of c_t in m'_t. We consider the case that the instance $\underline{c}_t$ has a corresponding mapping instance $\underline{c}_{\bowtie}$, that means $L(\texttt{target})(\underline{c}_{\bowtie}) = \underline{c}_t$. For each attribute $a_t \in \text{Att}^*_{c_t}$ of $\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}$ the corresponding source attributes a_1 and a_2 should be updated by $\text{update}_{\text{Att}(a_1)}(\underline{c}_1, \sigma_{\text{Att}}(a_t)(\underline{c}_t))$ and $\text{update}_{\text{Att}(a_2)}(\underline{c}_2), \sigma_{\text{Att}}(a_t)(\underline{c}_t))$ where $\underline{c}_1 \in L(\texttt{left})(\underline{c}_{\bowtie})$ and $\underline{c}_2 \in L(\texttt{right})(\underline{c}_{\bowtie})$ according to Definition 22.

Assuming the Put-Equality holds for all join confirming attributes var_{a_1} and var_{a_2}, then these update do not break the Put-Equality, because the invariants attributeMappingLeft and attributeMappingRight ensure that $var_{a_1}(\underline{c}_1) = \sigma_{\text{Att}}(a_t)(\underline{c}_t)$ and $var_{a_2}(\underline{c}_2) = \sigma_{\text{Att}}(a_t)(\underline{c}_t)$, which are the update values.

The invariant keepMappingPairs ensure that the mapping between the existing source class instances and the target class instance does not change by the attribute updates:

Assume that two instances $\underline{c}_1, \underline{c}_2$ have join confirming attribute values, then it is $\forall\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}(\sigma_{\text{Att}}(a_1)(\underline{c}_1) = \sigma_{\text{Att}}(a_2)(\underline{c}_2))$. The invariant keepMappingPairs states then that $\forall\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}(var_{a_1}(\underline{c}_1) = var_{a_2}(\underline{c}_2))$. According to the PUT-EQUALITY of var_{a_1} and var_{a_2} the instances have join confirming attribute values after the update. With the same argumentation it can be shown that if $\underline{c}_1, \underline{c}_2$ do not have join confirming attributes, they also do not have them after the update translation.

Consider now the case that an instance $\underline{c}_t$ has no corresponding mapping instance $\underline{c}_{\bowtie}$, that means it was newly created. For new instances the following update operations should be created $\text{CREATE}_{\text{Class}(c_1)}(V_1)$, $\text{CREATE}_{\text{Class}(c_2)}(V_2)$ with $V_1 = \{v_{a_1} = \sigma_{\text{Att}}(a_t)(\underline{c}_t) \mid \langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}\}$ and $V_2 = \{v_{a_2} = \sigma_{\text{Att}}(a_t)(\underline{c}_t) \mid \langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}\}$ according to Definition 22. Let $\underline{c}_1$ and $\underline{c}_2$ be the class instances created by these update operations. We show that these instances only influence the instance $\underline{c}_t$, so that it cannot have side effects to mapping expressions other than this natural join. The instances do not influence other ModelJoin class mapping expressions because the noConflictsWithOtherMappings invariant forbids the creation of new $\underline{c}_t$, if the class c_1 or c_2 is used in another mapping. From the natural join definition it follows that $\underline{c}_1 \sim_{\bowtie} \underline{c}_t$ and $\underline{c}_2 \sim_{\bowtie} \underline{c}_t$, because $v_{a_1} = v_{a_t}$ for all $\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}$. $\underline{c}_1$ has no join confirming attribute values with another existing source instance $\underline{c}'_2$ because of newTargetInstancesRight. Further $\underline{c}_1$ also has no join confirming attribute values with new source instance $\underline{c}'_2$ that will be created by other update translations: Instances $\underline{c}'_2$ can only be created by the create class instance operation for this natural join expression, because the noConflictsWithOtherMappings and newTargetInstances forbid that $\underline{c}_t$ has join confirming attributes with another newly created target class instance $\underline{c}'_t$. Therefore $\underline{c}_1$ has no join confirming attributes with instance $\underline{c}'_1$ created by translating $\underline{c}'_t$. The same argument can be given for $\underline{c}_2$.

Last consider the case that there's a mapping $\underline{c}_{\bowtie}$ with no target instance $\underline{c}_t$, that means $L(\texttt{target})(\underline{c}_{\bowtie}) = \emptyset$. This can only happen, if the target instance got deleted. In this case the following delete operation for the translation should be created: $\text{DELETE}_{\text{Class}(c)}(\underline{c}_1)$ and $\text{DELETE}_{\text{Class}(c)}(\underline{c}_2)$ where $\underline{c}_1 \in L(\texttt{left})(\underline{c}_{\bowtie})$ and $\underline{c}_2 \in L(\texttt{right})(\underline{c}_{\bowtie})$ according to Definition 22. That the translation only deletes exactly the source instances that map to already deleted target class instances follows from the consistentDeletionLeft and consistentDeletionRight invariant.

Next (iii) will be shown. Let $\underline{c}_1 \in \sigma_{\text{Class}}(c_1)$ be a class instance. In the case there exists no class instance $\underline{c}_{\bowtie}$ with $L(\texttt{left})(\underline{c}_{\bowtie}) = \underline{c}_1$ then $var'_{a_1}(\underline{c}_1) = var_{a_1}(\underline{c}_1)$ for $\langle a_1, a_2, a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}$ and the correctness follows from the correctness of var'_{a_1}. Consider now the case that there exists a $\underline{c}_{\bowtie}$ with $L(\texttt{left})(\underline{c}_{\bowtie}) = \underline{c}_1$, then $var'_{a_1} = \sigma_{\text{Att}}(a_t)(\underline{c}_t)$. The GET-EQUALITY follows from the natural join definition for a_t. The PUT-EQUALITY from the chosen translations in (ii).

Let now $r \in \text{Ref}$ be a reference with $associates(r) = \langle c, c_1\rangle$. If a target class instance $\underline{c}_t$ with $\underline{c}_1 \sim \underline{c}_t$ got deleted, then for the corresponding trace class instance $\underline{c}_{\bowtie}$ the *target* reference is undefined, therefore $\underline{c}_1$ is excluded from the links in the definition of $var'_r(\underline{c})$. The delete reference operation chosen in (ii) deletes the link on update translation. If the target class instance did not got deleted, the link from the original meta variable is included in the result. Hence the PUT-EQUALITY is satisfied. The GET-EQUALITY is satisfied because all trace class instances have a target in m_t.

Last (iv) will be shown. Let $\underline{c}_1 \in \sigma_{\text{CLASS}}(c_1)$ be a class instance. If $\underline{c}_1$ does not have a join partner, then it has no corresponding trace class instance $\underline{c}_{\bowtie} \in \sigma_{\text{CLASS}}(c_{\bowtie})$. So it is not excluded in the definition of $\mathit{Instances}'_{c_1}$. If $\underline{c}_1$ does not have a join partner, then it has a corresponding trace class instance $\underline{c}_{\bowtie} \in \sigma_{\text{CLASS}}(c_{\bowtie})$ and the `target` link is set. So $\underline{c}_1$ is not excluded as well. So $\mathit{Instances}'_{c_1} = \mathit{Instances}_{c_1}$ and the GET-EQUALITY of $\mathit{Instances}'_{c_1}$ follows from the GET-EQUALITY of $\mathit{Instances}_{c_1}$.

The PUT-EQUALITY follows from the translation rule in Definition 22 and that $\underline{c}_1$ is excluded if the trace class has an empty `target` link. The same argument can be given for $\mathit{Instances}_{c_2}$.

□

The outer join works very similar to the natural join. However target class instances may map to one source class only.

Definition 23 (Extensions for outer join)**.** Let $c_1, c_2 \in \text{CLASS}$ be two source classes, $c_t \in \text{CLASS}$ be a target class and $\bowtie = \langle c_1, c_2, c_t\rangle$ a outer join operator.

(i) Trace model The trace metamodel is extended similarly to the natural join by

$$\begin{aligned}\text{CLASS}' &= \text{CLASS} \cup \{c_{\bowtie}\}\\ \text{REF}' &= \text{REF} \cup \{\mathsf{left}, \mathsf{right}, \mathsf{target}\}\end{aligned}$$

with a new class $c_{\bowtie}$ and

$$\begin{aligned}\mathit{associates}(\mathsf{left}) &= \langle c_{\bowtie}, c_1\rangle, & \mathit{multiplicity}(\mathsf{left}) &= \{0,1\},\\ \mathit{associates}(\mathsf{right}) &= \langle c_{\bowtie}, c_2\rangle, & \mathit{multiplicity}(\mathsf{right}) &= \{0,1\},\\ \mathit{associates}(\mathsf{target}) &= \langle c_{\bowtie}, c_t\rangle, & \mathit{multiplicity}(\mathsf{target}) &= \{0,1\}.\end{aligned}$$

Note that the lower bound of the multiplicities for the references left and right changed to zero, because the target class instance may map only to one of the source classes.

The trace model is extended by additional instances of $c_{\bowtie}$ that represent the mapping:

$$\begin{aligned}&\forall \underline{c}_t \in \sigma_{\text{CLASS}}(c_t) \forall \underline{c}_1 \in \sigma_{\text{CLASS}}(c_1)\\ &\quad(\underline{c}_1 \sim_{\underline{\bowtie}} \underline{c}_t \Rightarrow \exists \underline{c}_{\bowtie} \in \sigma_{\text{CLASS}}(c_{\bowtie})(\underline{c}_1 \in L(\mathsf{left})(\underline{c}_{\bowtie})) \wedge (\underline{c}_t \in L(\mathsf{target})(\underline{c}_{\bowtie})))\\ &\wedge \forall \underline{c}_t \in \sigma_{\text{CLASS}}(c_t) \forall \underline{c}_2 \in \sigma_{\text{CLASS}}(c_2)\\ &\quad(\underline{c}_2 \sim_{\underline{\bowtie}} \underline{c}_t \Rightarrow \exists \underline{c}_{\bowtie} \in \sigma_{\text{CLASS}}(c_{\bowtie})(\underline{c}_2 \in L(\mathsf{right})(\underline{c}_{\bowtie})) \wedge (\underline{c}_t \in L(\mathsf{target})(\underline{c}_{\bowtie})))\end{aligned}$$

(ii) Constraints The OCL-constraints are similar to the OCL-constraints for the natural join, however they handle the case, that one of the source classes is undefined:

context $c_{\bowtie}$
−− *If the created attributes of the target class are updated, the new values will be used for the left and right source classes, therefore they must be confirm to all other constraints involving the source attribute values:*
inv attributeMappingLeft:

```
not self.left.oclIsUndefined() implies AND$_{\langle a_1,a_2,a_t\rangle \in A^{⟗}_{c_1,c_2,c_t}}$(self.target.$a_t$ = $var_{a_1}$(self.left))
inv attributeMappingRight:
not self.right.oclIsUndefined() implies AND$_{\langle a_1,a_2,a_t\rangle \in A^{⟗}_{c_1,c_2,c_t}}$(self.target.$a_t$ = $var_{a_2}$(self.right))
-- The mapping should not change when the attributes are update, therefore, exactly the instances
that had join confirming attribute values before the update, should have join confirming attributes
after the update:
inv keepMappingPairs:
$Instances_{c_1}$->forAll(left |
  $Instances_{c_2}$->forAll(right |
    (AND$_{\langle a_1,a_2,a_t\rangle \in A^{⟗}_{c_1,c_2,c_t}}$(left.$a_1$ = right.$a_2$)) =
    (AND$_{\langle a_1,a_2,a_t\rangle \in A^{⟗}_{c_1,c_2,c_t}}$($var_{a_1}$(left) = $var_{a_2}$(right)))
  )
)
```

The constraints for new target class instances, namely newTargetInstancesLeft, newTargetInstancesRight, newTargetInstances, newAttributesLeft, newAttributesRight and the constraints consistentDeletionLeft, consistentDeletionRight are equal to the one in the natural join definition.

(iii) Meta variables and Instance meta functions The definition of the meta variables var_{a_1} and var_{a_2} is expanded like in the natural join definition. The same applies to the meta variables of the incoming references and the class instances meta functions.

Definition 24 (Translation for outer join). Let $⋈= \langle c_1, c_2, c_t\rangle$ be an outer 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 to the the translation rule for the natural join in Definition 22. Instances $\underline{c}_1 \in I(c_1)$ or respectably $\underline{c}_2 \in I(c_2)$ are only updated, created or deleted, if $\underline{c}_1 \neq \bot$ or respectably $\underline{c}_2 \neq \bot$.

Theorem 14 (Induction step for outer join). *Let $q : I(M_{source}) \rightarrow I(M_{target})$ be a ModelJoin expression with a set of OCL-Constraints C for which the Induction statement holds. If q is extended by a arbitrary outer join expression to $q' : I(M_{source}) \rightarrow I(M'_{target})$ 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.*

(iii) *The new definitions for var'_{a_1}, var'_{a_2} for all $\langle a_1, a_2, a_t\rangle \in A^{⋈}_{c_1,c_2,c_t}$ are correct.*

(iv) *The new definitions for the class instance meta functions $Instances'_{c_1}$ and $Instances'_{c_2}$ are correct.*

Proof. Let $c_1, c_2 \in \textsc{Class}$ be the source classes and $c_t \in \textsc{Class}$ the target class of the outer join expression: $\bowtie = \langle c_1, c_2, c_t \rangle$.

For (i) the proof is the same as for the induction step the natural join.

We show (ii) by constructing a translation q'^{-1} that shows that c_t is translatable like in the proof for the correctness of the natural join. Let $\underline{c}_t$ be an instance of c_t in m'_t.

We consider the case that the instance $\underline{c}_t$ has a corresponding mapping instance $\underline{c}_{\bowtie}$, that means $L(\texttt{target})(\underline{c}_{\bowtie}) = \underline{c}_t$. For each attribute $a_t \in \textsc{Att}^*_{c_t}$ of $\langle a_1, a_2, a_t \rangle \in A^{\bowtie}_{c_1,c_2,c_t}$ the corresponding source attributes a_1 and a_2 should be updated according to Definition 24. The updates are the same such as the ones used in the proof for the correctness of the natural join, thus they preserve the Put-Equality.

The invariant keepMappingPairs ensure that the mapping between the existing source class instances and the target class instance does not change by the attribute updates: If $\underline{c}_1 \neq \perp$ and $\underline{c}_2 \neq \perp$ the argumentation from the natural join can be used, so we just consider without loss of generality the case that $\underline{c}_1 \neq \perp$ and $\underline{c}_2 = \perp$.

Assume that there is no $\underline{c}_2$ with join confirming attributes with $\underline{c}_1$. Then $\forall \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2)$ $\neg(\forall \langle a_1, a_2, a_t \rangle \in A^{\bowtie}_{c_1,c_2,c_t} (\sigma_{\textsc{Att}}(a_1)(\underline{c}_1) = \sigma_{\textsc{Att}}(a_2)(\underline{c}_2)))$. The invariant keepMappingPairs then states that $\forall \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2) \neg(\forall \langle a_1, a_2, a_t \rangle \in A^{\bowtie}_{c_1,c_2,c_t} (var_{a_1}(\underline{c}_1) = var_{a_2}(\underline{c}_2)))$. According to the Put-Equality of var_{a_1} and var_{a_2} all instances $\underline{c}_2$ have no join confirming attribute values after the update.

Consider now the case that an instance $\underline{c}_t$ has no corresponding mapping instance $\underline{c}_{\bowtie}$, that means it was newly created. For new instances at least one of the following two update operations should be created $\textsc{create}_{\textsc{Class}(c_1)}(V_1)$, $\textsc{create}_{\textsc{Class}(c_2)}(V_2)$ according to Definition 24. That these updates only influence the instance $\underline{c}_t$ is shown in the correctness proof of the natural join.

Last consider the case that there is a mapping $\underline{c}_{\bowtie}$ with no target instance $\underline{c}_t$, that means $L(\texttt{target})(\underline{c}_{\bowtie}) = \emptyset$. This can only happen, if the target instance got deleted. In this case the following update translations should be created $\textsc{delete}_{\textsc{Class}(c)}(\underline{c}_1)$ and $\textsc{delete}_{\textsc{Class}(c)}(\underline{c}_2)$ for $\underline{c}_1 \in L(\texttt{left})(\underline{c}_{\bowtie})$ and $\underline{c}_2 \in L(\texttt{right})(\underline{c}_{\bowtie})$, if $\underline{c}_1$ respectively $\underline{c}_2$ is defined according to Definition 24. That the translation only deletes exactly the target model class instances that are already deleted follows from the consistentDeletionLeft and consistentDeletionRight invariant.

For (iii) and (iv) the proof is the same as for the induction step for the natural join. □

Definition 25 (Extensions for theta join). Let $c_1, c_2 \in \textsc{Class}$ be two source classes, $c_t \in \textsc{Class}$ be a target class, $\theta \in \mathsf{Expr}_{t_{c_1}} \times \mathsf{Expr}_{t_{c_2}} \rightarrow \mathsf{Expr}_{\mathsf{Boolean}}$ be a logical expression and $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$ be theta join operator.

(i) Trace model The trace metamodel is extended by

$$\begin{aligned} \textsc{Class}' &= \textsc{Class} \cup \{c_{\bowtie}\} \\ \textsc{Ref}' &= \textsc{Ref} \cup \{\mathsf{left}, \mathsf{right}, \mathsf{target}\} \end{aligned}$$

with a new class $c_\bowtie$ and

$$\begin{aligned} associates(\mathsf{left}) &= \langle c_\bowtie, c_1\rangle, & multiplicity(\mathsf{left}) &= \{1\},\\ associates(\mathsf{right}) &= \langle c_\bowtie, c_2\rangle, & multiplicity(\mathsf{right}) &= \{1\},\\ associates(\mathsf{target}) &= \langle c_\bowtie, c_t\rangle, & multiplicity(\mathsf{target}) &= \{0,1\}.\end{aligned}$$

The trace model is extended by additional instances of $c_\bowtie$ that represent the mapping:

$$\begin{aligned}&\forall \underline{c}_t \in \sigma_{\text{CLASS}}(c_t)\forall \underline{c}_1 \in \sigma_{\text{CLASS}}(c_1)\forall \underline{c}_2 \in \sigma_{\text{CLASS}}(c_2)\\ &(\underline{c}_1 \sim_\bowtie \underline{c}_t \wedge \underline{c}_2 \sim_\bowtie \underline{c}_t \Rightarrow\\ &\exists \underline{c}_\bowtie \in \sigma_{\text{CLASS}}(c_\bowtie)(\underline{c}_1 \in L(\mathsf{left})(\underline{c}_\bowtie)) \wedge (\underline{c}_2 \in L(\mathsf{right})(\underline{c}_\bowtie)) \wedge (\underline{c}_t \in L(\mathsf{target})(\underline{c}_\bowtie)))\end{aligned}$$

(ii) Constraints The target metamodel should satisfy the OCL-constraints:

context $c_\bowtie$
−− *The mapping should not change when the target model is updated, therefore, exactly the instances for that θ hold before the update, should θ hold after the update. However deleted instances must be excluded:*
inv keepMappingPairs:
Instances$_{c_1}$->forAll(left |
  Instances$_{c_2}$->forAll(right |
    θ(left, right) = $var_{\theta(\mathsf{left},\mathsf{right})}$
)

We further consider two cases here. The first case is that either $var^{c_1\to c_t}_{\theta(\mathsf{v1},\mathsf{v2}),\mathsf{v1}}$, $var^{c_2\to c_t}_{\theta(\mathsf{v1},\mathsf{v2}),\mathsf{v2}}$ or $var^{c_1\to c_t,c_2\to c_t}_{\theta(\mathsf{v1},\mathsf{v2}),\mathsf{v1},\mathsf{v2}}$ is not defined or well typed. In this case we cannot reason about new target instances and therefore no new target class instances are allowed to be created.

context c_t
−− *It is not allowed to create new target instances:*
inv noNewTargetInstances:
$c_\bowtie$.allInstances()->select(tj | tj.target = self)->notEmpty()

In the case that $var^{c_1\to c_t}_{\theta(\mathsf{v1},\mathsf{v2}),\mathsf{v1}}$, $var^{c_2\to c_t}_{\theta(\mathsf{v1},\mathsf{v2}),\mathsf{v2}}$ and $var^{c_1\to c_t,c_2\to c_t}_{\theta(\mathsf{v1},\mathsf{v2}),\mathsf{v1},\mathsf{v2}}$ are defined and well-typed, new target classes must satisfy the following constraints:

context c_t
inv newTargetInstancesLeft:
isNew(self) **implies** *Instances*$_{c_1}$->forAll(left |
  not $var^{c_2\to c_t}_{\theta(\mathsf{left},\mathsf{self}),\mathsf{self}}$
)
inv newTargetInstancesRight:
isNew(self) **implies** *Instances*$_{c_2}$->forAll(right |
  not $var^{c_1\to c_t}_{\theta(\mathsf{self},\mathsf{right}),\mathsf{self}}$
)
inv newTargetInstances:

isNew(self) **implies** c_t.allInstances->select(other |
  $c_{\bowtie}$.allInstances()->select(tj | tj.target = other)->isEmpty() **and** other <> self
)->forAll(other |
  not $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self}, \text{other}), \text{self}, \text{other}}$
)
inv isJoinConfirming:
isNew(self) **implies let** self2 = self **in** $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self}, \text{self2}), \text{self}, \text{self2}}$
–– *New instances may only be created if the source class is not mapped anywhere else:*
inv noConflictsWithOtherMappings:
$\exists c'_t \in \textsc{Class}(c'_t \neq c_t \wedge (c_1 \sim_{\bowtie} c'_t \vee c_1 \sim_{\bowtie} c'_t) \Rightarrow c_{\bowtie}$.allInstances()->select(oj| oj.target = self
)->notEmpty())

(iii) Meta variables and Instance meta functions The OCL-constraint consistentDeletionLeft and consistentDeletionRight defined in the natural join extensions should also be created to ensure consistent deletion for theta joins, too. Further the meta variables of the incoming references and the class instances meta functions should be expanded like in the natural join extension.

Definition 26 (Translation for theta join). Let $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$ be a theta join operator and $\underline{c}_t \in \sigma_{\textsc{Class}}(c_t)$ 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 $\underline{c}_{\bowtie} \in \sigma_{\textsc{Class}}(c_{\bowtie})$ with $L(\texttt{target})(\underline{c}_{\bowtie}) = \underline{c}_t$ then the create class instance operations $\textsc{create}_{\textsc{Class}(c_1)}(V_1)$, $\textsc{create}_{\textsc{Class}(c_2)}(V_2)$ with $V_1 = \emptyset$ and $V_2 = \emptyset$ should be emitted.

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

Theorem 15 (Induction step 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 a 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.*

(iii) *The new definitions for the class instance meta functions $Instances'_{c_1}$ and $Instances'_{c_2}$ are correct.*

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

First (i) is proven. Therefore 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 $var_{\theta(\text{left},\text{right})}$. 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 $\underline{c}_t \in \sigma_{\textsc{Class}}(c_t)$ are a target of a join trace instance $\underline{c}_{\bowtie} \in \sigma_{\textsc{Class}}(c_{\bowtie})$ according to the theta join definition.

We show (ii) by constructing a translation q'^{-1} and show that c_t is translatable like in the proof for the correctness of the natural join. 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 $var_{\theta(\text{left},\text{right})}$.

Consider now the case that an instance $\underline{c}_t$ has no corresponding mapping instance $\underline{c}_{\bowtie}$, that means it was newly created. For new instances at least one of the following two update operations should be created $\text{CREATE}_{\textsc{Class}(c_1)}(V_1)$, $\text{CREATE}_{\textsc{Class}(c_2)}(V_2)$ according to Definition 26. Because of the isJoinConfirming invariant and the PUT-EQUALITY of $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self},\text{self2}),\text{self},\text{self2}}$, we have $\underline{\phi(\text{left}, \text{right})}^{\sigma}_{\text{left},\text{right}}(\underline{c_1}, \underline{c_2}) = \mathit{true}$ 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 $var^{c_1 \to c_t}_{\theta(\text{self},\text{right}),\text{self}}$, there exists no class instance $\underline{c}'_1$ with $\underline{\phi(\text{left}, \text{right})}^{\sigma}_{\text{left},\text{right}}(\underline{c'_1}, \underline{c_2}) = \mathit{true}$ and no new other joining pairs are created. The same argument can be given for other class instances $\underline{c}'_2$.

For (iii) the proof is the same as for the induction step for the natural join. □

4.5.2. Constraints for Keep Expressions

Definition 27 (Extensions for keep reference). Let $c_1, \hat{c} \in \textsc{Class}$ be two source classes, $r \in \textsc{Ref}$ be a source reference with $\mathit{associates}(r) = \langle c_1, \hat{c} \rangle$, $c_t, \hat{c}_t \in \textsc{Class}$ be two target classes, $r_t \in \textsc{Ref}$ be a target metamodel reference with $\mathit{associates}(r_t) = \langle c_t, \hat{c}_t \rangle$ and $\kappa_{\textsc{Ref}} = \langle r, r_t \rangle$ a keep reference expression of a join $\bowtie = \langle c_1, c_2, c_t \rangle$ where c_1 is the left joined class.

(i) Trace model The metamodel is extended by

$$\textsc{Class}' = \textsc{Class} \cup \{c_{\kappa_{\textsc{Ref}}}\}$$
$$\textsc{Ref}' = \textsc{Ref} \cup \{\text{source}, \text{target}\}$$

with a new class $c_{\kappa_{\textsc{Ref}}}$ and

$$\mathit{associates}(\text{source}) = \langle c_{\kappa_{\textsc{Ref}}}, \hat{c} \rangle, \qquad \mathit{multiplicity}(\text{source}) = \{1\},$$
$$\mathit{associates}(\text{target}) = \langle c_{\kappa_{\textsc{Ref}}}, \hat{c}_t \rangle, \qquad \mathit{multiplicity}(\text{target}) = \{0, 1\}.$$

The trace model is extended by additional instances of $c_{\kappa_{\textsc{Ref}}}$ that represent the mapping:

$$\forall \underline{\hat{c}} \in \sigma_{\textsc{Class}}(\hat{c}) \forall \underline{\hat{c}}_t \in \sigma_{\textsc{Class}}(\hat{c}_t)$$
$$(\underline{\hat{c}} \sim_{\underline{\bowtie}} \underline{\hat{c}}_t \Rightarrow \exists \underline{c}_{\kappa_{\textsc{Ref}}} \in \sigma_{\textsc{Class}}(c_{\kappa_{\textsc{Ref}}})(\underline{\hat{c}} \in L(\mathsf{source})(\underline{c}_{\kappa_{\textsc{Ref}}}) \wedge \underline{\hat{c}_t} \in L(\mathsf{target})(\underline{c}_{\kappa_{\textsc{Ref}}})))$$

(ii) Constraints The following constraints should be created:

context $\hat{c}_t$
−− *Each target class instance must be either a join target class instance or referenced from least one target class instance, that contains the target reference:*
inv isLinked:
$\mathbf{OR}_{\{\bowtie = \langle c'_1, c'_2, \hat{c}_t \rangle\}}$($c_{\bowtie}$.allInstances()->select(j|j.target = self)->notEmpty()) **or**
$\mathbf{OR}_{\{\kappa_{\textsc{Ref}} = \langle r, r_t \rangle \mid associates(r_t) = \langle c_t, \hat{c}_t \rangle\}}$($c_{\kappa_{\textsc{Ref}}}$.allInstances()->select(k|k.target = self)->notEmpty())
−− *New instances may only be created if the source class is not mapped anywhere else:*
inv noConflictsWithOtherMappings:
$\exists c'_t \in \textsc{Class}(c'_t \neq \hat{c}_t \wedge \hat{c} \sim_{\bowtie} c'_t \Rightarrow c_{\kappa_{\textsc{Ref}}}$.allInstances()->select(k| k.target = self)->notEmpty())

Further the references must be equal to the the corresponding meta variable and target meta variable:

context $c_{\kappa_{\textsc{Ref}}}$
−− *The value of the target attribute must be the same as the updated value of the source attribute:*
inv referenceMapping:
self.target.r->collect(t |
  $c_{\kappa_{\textsc{Ref}}}$.allInstances()->select(k | k.target = t)->asOrderedSet()->first().source
) = var_r(self.left)

context c_t
inv referenceTargetMapping:
self.r = $var_r^{c_t}$(self)

(iii) Meta variables The references in the target model can be used as canonical references for the reference r, thus the definition of var_r is extended:

```
$var'_r(v_1)$ := (let j = $c_{\bowtie}$.allInstances()->select(j| j.left = $v_1$)->asOrderedSet()
  in if j->notEmpty() and not j->first().target.oclIsUndefined() then
    $Instances_{\hat{c}}$->select(c |
      j->first().target.$r_t$->includes(
        $c_{\kappa_{\text{REF}}}$.allInstances()->select(k| k.source = c)->asOrderedSet()->first().target
      )
    )
  else
    $var_r(v_1)$
  endif
)
```

In case c_1 is not the source class of a join, but of a keep reference expression $\tilde{\kappa}_{\text{REF}} = \langle \tilde{r}, \tilde{r}_t \rangle$ with *associates*$(\tilde{r}) = \langle \tilde{c}, c_1 \rangle$, then the definition var_r is extended by:

```
$var'_r(v_1)$ := (let k = $c_{\tilde{\kappa}_{\text{REF}}}$.allInstances()->select(k| k.source = $v_1$)->asOrderedSet()
  in if k->notEmpty() and not k->first().target.oclIsUndefined() then
    $Instances_{\hat{c}}$->select(c |
      k->first().target.$r_t$->includes(
        $c_{\kappa_{\text{REF}}}$.allInstances()->select(k| k.source = c)->asOrderedSet()->first().target
      )
    )
  else
    $var_r(v_1)$
  endif
)
```

Definition 28 (Translation for keep reference)**.** Let $\kappa_{\text{REF}} = \langle r, r_t \rangle$ be a keep reference expression and $\underline{c}_t \in \sigma_{\text{CLASS}}(\underline{c}_t)$ and $\underline{\hat{c}}_t \in \sigma_{\text{CLASS}}(\underline{\hat{c}}_t)$ be target class instances. We only consider the case, where κ_{REF} is a keep reference expression of a join $\bowtie = \langle c_1, c_2, c_t \rangle$ where c_1 is the left joined class. The case for a keep reference expression is defined analogically. The pair $\underline{c}_t, \underline{\hat{c}}_t$ should be translated according the following rules:

(i) Let $\underline{c}_{\kappa_{\text{REF}}} \in \sigma_{\text{CLASS}}(c_{\kappa_{\text{REF}}})$ be a trace class instance of $\underline{\hat{c}}_t$ with $L(\texttt{target})(\underline{c}_{\kappa_{\text{REF}}}) = \underline{c}_t$, $\underline{\hat{c}} \in L(\texttt{source})(\underline{c}_{\kappa_{\text{REF}}})$, $\underline{c}_{\bowtie}$ be a trace class instance with $\underline{c}_t \in L(\texttt{target})(\underline{c}_{\bowtie})$ and $\underline{c}_1 \in L(\texttt{source})(\underline{c}_{\bowtie})$.

If $\underline{\hat{c}}_t \in L(r_t)(c_t)$ and $\underline{\hat{c}} \notin L(r)(c_1)$ then the create link operation $\text{CREATE}_{\text{REF}(r)}(\underline{c}_1, \underline{\hat{c}})$ should be emitted. If $\underline{\hat{c}}_t \notin L(r_t)(c_t)$ and $\underline{\hat{c}} \in L(r)(c_1)$, then the delete link operation $\text{DELETE}_{\text{REF}(r)}(\underline{c}_1, \underline{\hat{c}})$ should be emitted.

(ii) If there exists no trace class instance $\underline{c}_{\kappa_{\text{REF}}} \in \sigma_{\text{CLASS}}(c_{\kappa_{\text{REF}}})$ with $\underline{c}_t \in L(\texttt{target})(\underline{c}_{\kappa_{\text{REF}}})$, then the create class instance operation $\text{CREATE}_{\text{CLASS}(\hat{c}_1)}(V_1)$ with $V_1 = \emptyset$ should be emitted, if no other translation of $\underline{\hat{c}}_t$ creates an instance already. Let $\underline{\hat{c}}$ the new created instance. Further the create reference operation $\text{CREATE}_{\text{REF}(r)}(\underline{c}_1, \underline{\hat{c}})$ should be emitted.

(iii) Further for each $\underline{c}_{\kappa_{\text{Ref}}} \in \sigma_{\text{Class}}(c_{\kappa_{\text{Ref}}})$ with $L(\texttt{target})(\underline{c}_{\kappa_{\text{Ref}}}) = \emptyset$, the delete reference operation $\text{DELETE}_{\text{Ref}(r)}(\underline{c}_1, \underline{\hat{c}})$ with $\underline{c}_1 = L(\texttt{source})(\underline{c}_{\bowtie})$ and $\underline{\hat{c}} \in L(\texttt{source})(\underline{c}_{\kappa_{\text{Ref}}})$ should be emitted.

Theorem 16 (Induction step for keep reference). *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 a arbitrary keep reference 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.*

(iii) *The new definition for var'_r is correct.*

Proof. Let $c_1, c_2 \in \text{Class}$ be the source classes and $c_t \in \text{Class}$ the target class of the join $\bowtie = \langle c_1, c_2, c_t \rangle$ where c_1 is the left joined class of the keep reference expression $\kappa_{\text{Ref}} = \langle r, r_t \rangle$. We only consider the case, where κ_{Ref} is a keep reference expression of the join $\bowtie$. The case for a keep reference expression is defined analogically.

First (i) is proven. Therefore 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 $\hat{c}$ and all instances of other classes then $\hat{c}_t$ satisfy C according to the premises. So it just has to be shown that all new constraints in $C' \setminus C$ are satisfied.

The isLinked invariant is true, because of the definition of the keep reference expression. The noConflictsWithOtherMappings invariant is true because for every instance $\hat{c}_t$ there exists a keep reference class instance according to the definition.

We show (ii) by constructing a translation q'^{-1}. Let $\underline{\hat{c}}_t$ be an instance of $\hat{c}_t$ in m'_t. Since $\underline{\hat{c}}_t$ has no attribute values by default, no attributes can be changed. We consider the case that an instance $\underline{\hat{c}}_t$ has a corresponding mapping instance $\underline{c}_{\kappa_{\text{Ref}}}$, that means $L(\texttt{target})(\underline{c}_{\kappa_{\text{Ref}}}) = \underline{\hat{c}}_t$. For each deleted link with $(\underline{c}_t, \underline{\hat{c}}_t) \in \sigma_{\text{Ref}}(r_t)$ a delete reference operation $\text{DELETE}_{\text{Ref}(r)}(\underline{c}_1, \underline{\hat{c}})$ should be created and for each new link a create reference operation $\text{CREATE}_{\text{Ref}(r)}(\underline{c}_1, \underline{\hat{c}})$ should be created according to Definition 28.

Consider now the case where an instance $\underline{\hat{c}}_t$ has no corresponding mapping instance $\underline{c}_{\kappa_{\text{Ref}}}$, that means it was newly created. The update operation $\text{CREATE}_{\text{Class}(\hat{c}_1)}(V_1)$ with $V_1 = \emptyset$ should be created according to Definition 28. For each Link $(\underline{c}_t, \underline{\hat{c}}_t) \in \sigma_{\text{Ref}}(r_t)$ a new link with $\text{CREATE}_{\text{Ref}(r)}(\underline{c}, \underline{\hat{c}})$ should be created. Because of the isLinked invariant, there exists at least one link. Because of the Put-Equality of var_r all $\text{DELETE}_{\text{Ref}(r)}$ and $\text{DELETE}_{\text{Ref}(r)}$ do not conflict with other updates.

Finally we show (iii). Let $\underline{c}_1$ be a class instance of c_1. If there exists no $\underline{c}_{\bowtie}$ with the link $L(\texttt{target})(\underline{c}_{\bowtie}) = \hat{\underline{c}}_1$ then $var'_r(\underline{c}_1) = var_r(\underline{c}_1)$ and the GET-EQUALITY and PUT-EQUALITY follows from $var_r(\underline{c}_1)$. If there exist such an instance $\underline{c}_{\bowtie}$ then it follows from the keep reference definitions that there is for each link in r a link in r_t with corresponding mapping instances $\underline{c}_{\kappa_{\text{REF}}}$ and its easy to verify that the GET-EQUALITY holds. The PUT-EQUALITY holds, because of the chosen translations above.

□

Definition 29 (Extensions for keep attribute)**.** Let $a : t_c \to t \in \text{ATT}^*_c$ be a source attribute of a class $c_1 \in \text{CLASS}$, $a_t : t_{\tilde{c}} \to t \in \text{ATT}^*_{c_t}$ be a target attribute of a target class $c_t \in \text{CLASS}$ and a $\kappa_{\text{ATT}} = \langle a_1, a_t \rangle$ be a keep attribute expression of a join $\bowtie = \langle c_1, c_2, c_t \rangle$ where c_1 is the left joined class.

(i) Constraints The target model should satisfy the OCL-constraints:

context $c_{\bowtie}$
−− *The value of the target attribute must be the same as the updated value of the source attribute:*
inv attributeMapping:
self.target.a_t = var_{a_1}(self.left)

context c_t
inv attributeTargetMapping:
self.a_t = $var^{c_t}_{a_1}$(self)

If the keep attribute expression κ_{ATT} references a attribute of the right class c_2 the same OCL constraints are created, but using c_2 instead of c_1 and right instead of left.

If the keep attribute expression $\kappa_{\text{ATT}} = \langle a_1, a_t \rangle$ is the parent of a keep reference expression $\kappa_{\text{REF}} = \langle r, r_t \rangle$ instead of a join, the following OCL-constraints are created instead:

context $c_{\kappa_{\text{REF}}}$
−− *The value of the target attribute must be the same as the updated value of the source attribute:*
inv attributeMapping:
self.target.a_t = var_{a_1}(self.source)

context c_t
inv attributeTargetMapping:
self.a_t = $var^{c_t}_{a_1}$(self)

(ii) meta variables The new attributes of the target class can be used as canonical attribute for var_{a_1}. The definition of var_{a_1} is extended in the join case by

```
$var'_{a_1}(v_1)$ := (let j = $c_{\bowtie}$.allInstances()->select(j| j.left = $v_1$)->asOrderedSet()
  in if j->notEmpty() and not j->first().target.oclIsUndefined() then
    j->first().target.$a_t$
  else
    $var_{a_1}(v_1)$
  endif
)
```

and in the keep reference case by

```
$var'_{a_1}(v_1)$ := (let k = $c_{\kappa_{\text{Ref}}}$.allInstances()->select(k| k.source = $v_1$)->asOrderedSet()
  in if k->notEmpty() and not k->first().target.oclIsUndefined() then
    k->first().target.$a_t$
  else
    $var_{a_1}(v_1)$
  endif
)
```

Definition 30 (Translation for keep attribute)**.** Let $\kappa_{\text{Att}} = \langle a_1, a_t \rangle$ be a keep reference expression 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 $L(\texttt{target})(\underline{c}_{\bowtie}) = \underline{c}_t$ then the attribute value set V_1 in the create class instance operation $\text{CREATE}_{\text{Class}(c_1)}(V_1)$ from the context class, should be extended by the value $v_{a_1} = \sigma_{\text{Att}}(a_t)(\underline{c}_t)$.

(ii) If there exists a trace class instance $\underline{c}_{\bowtie} \in \sigma_{\text{Class}}(c_{\bowtie})$, then the attribute a_1 should be updated by the update operation $\text{UPDATE}_{\text{Att}(a_1)}(\underline{c}_1, \sigma_{\text{Att}}(a_t)(\underline{c}_t))$ with $\underline{c}_t \in L(\texttt{left})(\underline{c}_{\bowtie})$.

Theorem 17 (Induction step for keep attribute)**.** *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 a arbitrary keep attribute 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.*

(iii) *The new definition for var'_{a_1} is correct.*

Proof. Let $c_1, c_2 \in \textsc{Class}$ be the source classes and $c_t \in \textsc{Class}$ the target class of the join $\bowtie = \langle c_1, c_2, c_t \rangle$ where c_1 is the left joined class and $\kappa_{\textsc{Att}} = \langle a_1, a_t \rangle$ a keep attribute expression.

First (i) is proven. Therefore 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 a_t and are satisfied according to the premises. So it just has to be shown that all new constraints in $C' \setminus C$ are satisfied.

The attributeMapping invariant is true, because of the definition of the keep attribute expression. The attributeTargetMapping invariant is true because of the Get-Equality of $var_{a_1}^{c_t}$.

We show (ii) by constructing a translation q'^{-1}. Let $\underline{c}_t$ be an instance of c_t in m_t'.

We consider the case that the instance $\underline{c}_t$ has a corresponding mapping instance $\underline{c}_{\bowtie}$, that means $L(\texttt{target})(\underline{c}_{\bowtie}) = \underline{c}_t$. The attribute a_1 should be updated by the update operation $\textsc{update}_{\textsc{Att}(a_1)}(\underline{c}_1, \sigma_{\textsc{Att}}(a_t)(\underline{c}_t))$ according to Definition 30.

Assuming that the Put-Equality holds for var_{a_1}, then the update dos not break the Put-Equality, because the invariants attributeMapping ensure that $var_{a_1}(\underline{c}_1) = \sigma_{\textsc{Att}}(a_t)(\underline{c}_t)$, which are the update values.

Consider now the case that an instance $\underline{c}_t$ has no corresponding mapping instance $\underline{c}_{\bowtie}$, that means it was newly created. For these instances the attribute value set V_1 should be extended according to Definition 30. This does not conflict with other Updates because of the Put-Equality of $var_{a_1}^{c_t}$.

(iii) can be shown like in the proof of the induction step for the natural join. □

Definition 31 (Extensions for calculate attribute)**.** Let $a_t : t_c \to t \in \textsc{Att}_{c_t}^*$ be a target attribute of a target class $c_t \in \textsc{Class}$, $\phi : \mathsf{Expr}_{t_{c_1}} \times \mathsf{Expr}_{t_{c_2}} \to \mathsf{Expr}_{\mathsf{t}}$ be a function and $\delta_\phi = \langle c_1, c_2, a_t \rangle$ be a calculate attribute expression of a join $\bowtie = \langle c_1, c_2, c_t \rangle$ where c_1 is the left and c_2 the right joined class.

(i) Constraints The target model should satisfy the OCL-constraints:

context $c_{\bowtie}$
-- The value of the target attribute must be calculated from the source attributes
inv attributeMapping:
self.target.a_t = $var_{\phi(\text{self.left, self.right})}$

If $var_{\phi(\text{v1,v2}),\text{v1,v2}}^{c_1 \to c_t, c_2 \to c_t}$ is not defined then no new instances can be created

context c_t
inv attributeTargetMapping:
$c_{\bowtie}$.allInstances()->select(j| j.target = self)->notEmpty()

If $var_{\phi(\text{v1,v2}),\text{v1,v2}}^{c_1 \to c_t, c_2 \to c_t}$ is defined, new instances are allowed if the following OCL-constraints are satisfied:

context c_t
inv newAttributes:
$c_{\bowtie}$.allInstances()->select(j| j.target = self)->isEmpty() **implies**
self.a_t = **let** self2 = self **in** $var^{c_1 \to c_t, c_2 \to c_t}_{\phi(\mathsf{self}, \mathsf{self2}), \mathsf{self}, \mathsf{self2}}$
inv attributeTargetMapping:
$\exists c'_t \in \textsc{Class}(c'_t \neq c_t \wedge (c_1 \sim_{\bowtie} c'_t \vee c_2 \sim_{\bowtie} c'_t) \Rightarrow$
$c_{\bowtie}$.allInstances()->select(j| j.target = self)->notEmpty())

If the keep attribute expression κ_{ATT} references a attribute of the right class c_2 the same OCL constraints are created, but using c_2 instead of c_1 and right instead of left.

If the keep attribute expression $\kappa_{\text{ATT}} = \langle a_1, a_t \rangle$ is the parent of a keep reference expression $\kappa_{\text{REF}} = \langle r, r_t \rangle$ instead of a join, ϕ only depends on one source class and the following OCL-constraints are created instead:

context $c_{\kappa_{\text{REF}}}$
−− *The value of the target attribute must be calculated from the source attributes*
inv attributeMapping:
self.target.a_t = $var_{\phi(\mathsf{self.source})}$

If $var^{c_1 \to c_t}_{\phi(\mathsf{v}), \mathsf{v}}$ is not defined then no new instances can be created

context c_t
inv attributeTargetMapping:
$c_{\bowtie}$.allInstances()->select(j| j.target = self)->notEmpty()

If $var^{c_1 \to c_t}_{\phi(\mathsf{v}), \mathsf{v}}$ is defined, new instances are allowed if the following OCL-constraints are satisfied:

context c_t
inv newAttributes:
$c_{\kappa_{\text{REF}}}$.allInstances()->select(k| k.target = self)->isEmpty() **implies** self.a_t = $var^{c_1 \to c_t}_{\phi(\mathsf{self}), \mathsf{self}}$
inv attributeTargetMapping:
$\exists c'_t \in \textsc{Class}(c'_t \neq c_t \wedge (c_1 \sim_{\bowtie} c'_t \vee c_2 \sim_{\bowtie} c'_t) \Rightarrow$
$c_{\kappa_{\text{REF}}}$.allInstances()->select(k| k.target = self)->notEmpty())

Definition 32 (Translation for calculate attribute). Let $\delta_\phi = \langle c_1, c_2, a_t \rangle$ be a calculate attribute expression. No update operations are emitted at update translation.

Theorem 18 (Induction step for calculate attribute). *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 a arbitrary calculate attribute 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 $c_1, c_2 \in \textsc{Class}$ be the source classes and $c_t \in \textsc{Class}$ the target class of the join $\bowtie = \langle c_1, c_2, c_t \rangle$ where c_1 is the left joined class and $\kappa_{\textsc{Att}} = \langle a_1, a_t \rangle$ a calculate attribute expression.

First (i) is proven. Therefore 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 a_t and are satisfied according to the premises. So it just has to be shown that all new constraints in $C' \setminus C$ are satisfied.

The attributeMapping invariant is true, because of the definition of the calculate attribute expression. All other constraints are immediate satisfied because all target class instances have a mapping and so the sets $c_{\bowtie}$.allInstances()->select(j| j.target = self) are not empty.

We show (ii) by constructing a translation q'^{-1}. Let $\underline{c}_t$ be an instance of c_t in m'_t.

For all instances $\underline{c}_t$ no additional update operations are emitted and so the Put-Equality is not influenced. □

Aggregation expressions in ModelJoin can be handled similar to calculate attributes, because the aggregation function can be expressed as OCL expression and therefore no extra definition extension is given here.

4.5.3. Handling of Additional Supertypes and Subtypes

Because keep supertype and keep subtype operations do not influence the system state of the target model, in particular no additional instances of the super-/subtype are created and only the metamodel hierarchy is extended, no additional trace classes are needed. The constraints for keep operation nested in a keep supertype and keep subtype operators should be created such as they would be part of the outer join or keep reference operator.

4.5.4. Regarding Multiplicity Restrictions of the Source Metamodels

We silently ignored the multiplicity restrictions of references and attributes in the source model. The multiplicity restrictions can be violated by the translation of a create class instance operation. This is for example the case, if a source attribute does not get mapped to a target class attribute and so its value is missing for new class instances. This problem could be handled in three ways:

(1) For each primitive type there is an default value, that is used for missing values.

(2) The syntax of ModelJoin is extended, so that a value or function to generate the missing attributes can be given at view definition time.

(3) The missing attribute values are requested from the user at translation time or updated by the user manually after the translation.

The same problem arises with outgoing references for newly created source classes and could be handled in the same way as for attributes. We propose a solution for (1) and (2) by introducing source attribute update expressions in chapter 5.3.

4.6. Deciding Translatability

We have shown that the induction statement holds for all ModelJoin operators and an empty ModelJoin view definition. In conclusion, we can formulate the induction statement for an arbitrary ModelJoin view definition:

Theorem 19 (Translatability)**.** *Let $q : I(M_s) \to I(M_t)$ be a ModelJoin expression with a 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 $\langle m_s, m_t' \rangle \in I(M_s) \times I(M_t')$ a target model instance m_t', that satisfies C, is translatable.*

Proof. The properties can be shown for every ModelJoin expressions by induction over the used ModelJoin operators using the corresponding induction step theorems. □

To check the OCL constraints, we need to generate the OCL constraints for the ModelJon view definition and check them against the source and target model.

Definition 33. Let $q : I(M_s) \to I(M_t)$ be a ModelJoin view definition. We define $\mathtt{OCL}_q : I(M_s) \times I(M_t) \to \{true, false\}$ as the function, that decides, if the OCL expression for q do hold for a given instance pair $\langle m_s, m_t' \rangle \in I(M_s) \times I(M_t)$.

Using the OCL constraints and the inductive definition of q^{-1}, we have found a solution for the Restricted View-Update-Problem:

Theorem 20 (Solution for the Restricted View-Update-Problem)**.** *Let $Q \in M_s \times M_t$ be a ModelJoin view definition, then the inductively defined translation q^{-1} together with the set $V_r(m_s) = \{m_t \in I(M_t) \mid \mathit{OCL}_q(m_s, m_t)\}$ solve the Restricted View-Update-Problem.*

Proof. Let $Q \in M_s \times M_t$ be a ModelJoin view definition. A translation q^{-1} exists by construction. Further Theorem 19 show the totality of $V_r(m_s)$ and for all pairs $\langle m_s, m_t' \rangle \in I(M_s) \times I(M_t')$ with $m_t' \in V_r(m_s)$, that the model m_t' is translatable. So the PutGet- and GetPut-Property holds for q^{-1}. □

4.7. Inferring About the Translatability of Updates

It might be desired to show the translatability of a given update sequence in general. Currently we need a concrete models to check the OCL constraints and decide the translatability. To decide the translatability in general, we need to check the OCL constraints for generality. This can be done manually or using an automatic solver. Clavel et al. [19] showed for a subset of OCL that OCL Constraints can be checked for unsatisfiability by translating them into first order logic. Then automated theorem prover (e.g., Prover9 [47]) and SMT solver (e.g., Yices [5]) can automatically check the unsatisfiability. Alternatively interactive proof tools could may be used. For example the *KeY tool* [10] can translate OCL constraints into first order predicate logic. The user then can interact wit the KeY theorem prover to logically reason about a update operation. The proposed tools and approaches may not support all needed OCL constructs. We will demonstrate only the basic approach by manually proofing an easy example.

We know, that the constraints are satisfied before the update. Therefore we can use the OCL constraints generated by the ModelJoin expression as premise:

Theorem 21. *Let $q : I(M_s) \to I(M_t)$ be a ModelJoin view definition and $u : I(M_t) \to I(M_t)$ be a update operation. If*

$$\forall m_s \in I(M_s)(OCL_q(m_s, q(m_s)) \Rightarrow OCL_q(m_s, u(q(m_s))))$$

then u is translatable for all target models.

Showing the generality of OCL constraints is very hard. However for some cases it is possible and Theorem 21 can be applied. For example we can show the translatability in the following case:

Consider the ModelJoin view definition in Listing 4.4 and the update operation in Listing 4.5.

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

Listing 4.4: Example ModelJoin view definition.

```
update jointarget.Interface {
    name: "foo",
    alias: "foo"
} where "jointarget.Interface.name = 'bar'"
```

Listing 4.5: Example ModelJoin update operation.

The generated OCL expressions are:

```
context $c_{\bowtie}$
inv keepMappingPairs:
classifier.Interface.allInstances->forAll(left |
  uml.Interface.allInstances->forAll(right |
    (left.name = right.name) = ($var_{\text{name}}$(left) = $var_{\text{name}}$(right))
)
inv attributeMappingName:
self.target.name = $var_{\text{name}}$(self.left)
inv attributeMappingAlias:
self.target.alias = $var_{\text{name}}$(self.right)
context jointarget.Interface
inv noNewTargetInstances:
$c_{\bowtie}$.allInstances()->select(tj | tj.target = self)->notEmpty()
inv attributeTargetMappingName:
self.name = $var_{\text{name}}^{\text{jointarget.Interface}}$(self)
inv attributeTargetMappingAlias:
self.alias = $var_{\text{name}}^{\text{jointarget.Interface}}$(self)
```

The update semantic of the update operation with m_s, m_t, m_t' like in Theorem 21 is

$$\begin{aligned}
\sigma_{\text{ATT}}^{m_t'}(\mathsf{name})(\underline{c}) &= \begin{cases} \text{“foo”}, & \sigma_{\text{ATT}}^{m_t}(\mathsf{name})(\underline{c}) = \text{“bar”} \\ \sigma_{\text{ATT}}^{m_t}(\mathsf{name})(\underline{c}), & \text{else} \end{cases} \\
\sigma_{\text{ATT}}^{m_t'}(\mathsf{alias})(\underline{c}) &= \begin{cases} \text{“foo”}, & \sigma_{\text{ATT}}^{m_t}(\mathsf{alias})(\underline{c}) = \text{“bar”} \\ \sigma_{\text{ATT}}^{m_t}(\mathsf{alias})(\underline{c}), & \text{else} \end{cases} \\
\sigma_{\text{REF}}^{m_t'}(r) &= \sigma_{\text{REF}}^{m_t}(r) \\
\sigma_{\text{CLASS}}^{m_t'}(c) &= \sigma_{\text{CLASS}}^{m_t}(c)
\end{aligned}$$

We want to show the translatability of the update operation in general. As additional premise we suppose, that there exists no source class instance of the classes classifier.Interface and uml.Interface named “foo”. Combining this premise and Theorem 21, the following statement must be shown.

$$\begin{aligned}
&\forall \underline{c}_1 \in \sigma_{\text{CLASS}}^{\sigma^{m_s}}(\mathsf{classifier.Interface})\colon \sigma_{\text{ATT}}^{m_s}(\mathsf{name})(\underline{c}_1) \neq \text{“foo”} \\
&\wedge \forall \underline{c}_2 \in \sigma_{\text{CLASS}}^{\sigma^{m_s}}(\mathsf{uml.Interface})\colon \sigma_{\text{ATT}}^{m_s}(\mathsf{name})(\underline{c}_2) \neq \text{“foo”} \\
&\wedge \mathsf{OCL}_q(m_s, m_t) \\
\Longrightarrow\ &\mathsf{OCL}_q(m_s, m_t')
\end{aligned}$$

We first show the invariant keepMappingPairs: Let $\underline{c}_1 \in \sigma_{\text{CLASS}}^{\sigma^{m_s}}(\mathsf{classifier.Interface})$ and $\underline{c}_2 \in \sigma_{\text{CLASS}}^{\sigma^{m_s}}(\mathsf{uml.Interface})$ be two source instances.

According to the keep attribute definition it is

$$\sigma^{m_t}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_1) \text{ and } \sigma^{m_t}_{\textsc{Att}}(\mathsf{alias})(\underline{c}_t) = \sigma^{m_s}_{\textsc{Att}}(\mathsf{alias})(\underline{c}_1)$$

Assume that $\sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_1) = \sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_2)$. Let $\underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})$ be a target class instance with $\underline{c}_1 \sim_{\bowtie} \underline{c}_t$ and $\underline{c}_2 \sim_{\bowtie} \underline{c}_t$. We either have

$$\begin{aligned}
&\sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_1) \neq \text{“}\mathit{bar}\text{''}\\
\Longrightarrow\ &\sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \sigma^{m_t}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) \text{ and } \sigma^{m_t'}_{\textsc{Att}}(\mathsf{alias})(\underline{c}_t) = \sigma^{m_t}_{\textsc{Att}}(\mathsf{alias})(\underline{c}_t)\\
\Longrightarrow\ &\sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \sigma^{m_t'}_{\textsc{Att}}(\mathsf{alias})(\underline{c}_t)
\end{aligned}$$

or

$$\begin{aligned}
&\sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_1) = \text{“bar”}\\
\Longrightarrow\ &\sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \text{“foo” and } \sigma^{m_t'}_{\textsc{Att}}(\mathsf{alias})(\underline{c}_t) = \text{“foo”}\\
\Longrightarrow\ &\sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \sigma^{m_t'}_{\textsc{Att}}(\mathsf{alias})(\underline{c}_t)
\end{aligned}$$

According to the Put-Equality we have in both cases

$$\begin{aligned}
&\underline{\mathit{var}}_{\text{name}}(\underline{c}_1) = \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) \text{ and}\\
&\underline{\mathit{var}}_{\text{name}}(\underline{c}_2) = \sigma^{m_t'}_{\textsc{Att}}(\mathsf{alias})(\underline{c}_t)\\
\Longrightarrow\ &\underline{\mathit{var}}_{\text{name}}(\underline{c}_1) = \underline{\mathit{var}}_{\text{name}}(\underline{c}_2)
\end{aligned}$$

Assume now that $\sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_1) \neq \sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_2)$. For $\underline{c}_1$ we have either

$$\begin{aligned}
&\sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_1) \neq \text{“bar”}\\
\Longrightarrow\ &\forall \underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})\colon \underline{c}_1 \sim_{\bowtie} \underline{c}_t \to \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \sigma^{m_t}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t)\\
&\wedge \forall \underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})\colon \underline{c}_2 \sim_{\bowtie} \underline{c}_t \to\\
&(\sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \sigma^{m_t}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) \vee \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \text{“foo”})\\
\Longrightarrow\ &\forall \underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})\colon \underline{c}_1 \sim_{\bowtie} \underline{c}_t \to \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \sigma^{m_t}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t)\\
&\wedge \forall \underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})\colon \underline{c}_2 \sim_{\bowtie} \underline{c}_t \to \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) \neq \sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_1)
\end{aligned}$$

or

$$\begin{aligned}
&\sigma^{m_s}_{\textsc{Att}}(\mathsf{name})(\underline{c}_1) = \text{“bar”}\\
\Longrightarrow\ &\forall \underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})\colon \underline{c}_1 \sim_{\bowtie} \underline{c}_t \to \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \text{“foo”}\\
&\wedge \forall \underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})\colon \underline{c}_2 \sim_{\bowtie} \underline{c}_t \to \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \sigma^{m_t}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t))\\
\Longrightarrow\ &\forall \underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})\colon \underline{c}_1 \sim_{\bowtie} \underline{c}_t \to \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) = \text{“foo”}\\
&\wedge \forall \underline{c}_t \in \sigma^{\sigma^{m_t}}_{\textsc{Class}}(\mathsf{jointarget.Interface})\colon \underline{c}_2 \sim_{\bowtie} \underline{c}_t \to \sigma^{m_t'}_{\textsc{Att}}(\mathsf{name})(\underline{c}_t) \neq \text{“foo”}
\end{aligned}$$

According to the Put-Equality we have in both cases

$$\begin{aligned}
&\underline{var}_{\text{name}}(\underline{c}_1) = \sigma_{\text{Att}}^{m'_t}(\mathsf{name})(\underline{c}_t) \text{ and}\\
&\underline{var}_{\text{name}}(\underline{c}_2) = \sigma_{\text{Att}}^{m'_t}(\mathsf{alias})(\underline{c}_t)\\
\Longrightarrow\ &\underline{var}_{\text{name}}(\underline{c}_1) \neq \underline{var}_{\text{name}}(\underline{c}_2)
\end{aligned}$$

In conclusion, the invariant keepMappingPairs holds.

The invariants attributeMappingName, attributeMappingAlias, attributeTargetMapping-Name and attributeTargetMappingAlias hold because the used meta variables do map to the instance itself. The invariant noNewTargetInstances holds, because no new instances were created by the update operation.

In summary, we have shown that, the update operation is translatable in all cases, if there exists no source class instance of the classes classifier.Interface and uml.Interface named “foo”.

5. Additional Translation Strategies for Updated Views

In the previous chapter we have chosen one fixed translation strategy for each ModelJoin operator. However for some operations, various translation strategies are possible. In this chapter we propose additional translation strategies. For each translation strategy we define the translation algorithm and the set of constraints, which should be generated, if the translation strategy is chosen.

5.1. Translation of New Target Class Instances

5.1.1. Mapping Meta Functions

If target class instances are created by an update operations, different source class instances could by created at update translation. Since there is no mapping information present for the new instances, the mapping must be either guessed from the join definition or explicitly provided by the user.

To handle both cases in an unified way, we define the following *mapping meta functions*:

Definition 34 (mapping meta function)**.** Let $c, c_t \in \textsc{Class}$ be classes with $c \sim_{\bowtie} c_t$. A function $\mathit{mapsTo}_{\langle c,c_t\rangle} : \mathsf{Expr}_{t_c} \times \mathsf{Expr}_{t_{c_t}} \rightarrow \mathsf{Expr}_{\mathsf{Boolean}}$ is called a *mapping meta function*, iff $\underline{\mathit{mapsTo}}^{\sigma}_{\langle c,c_t\rangle}(\underline{c}, \underline{c}_t) = \mathit{true}$ implies $\underline{c} \mathrel{\underline{\sim_{\bowtie}}} \underline{c}_t$. The interpretation function is defined as

$$\underline{\mathit{mapsTo}}^{\sigma}_{\langle c,c_t\rangle}(\underline{c}, \underline{c}_t) = I\left[\left[\mathit{mapsTo}_{\langle c,c_t\rangle}(\mathsf{c}, \mathsf{ct})\right]\right](\langle \sigma, \{\mathsf{c} \rightarrow \underline{c}, \mathsf{ct} \rightarrow \underline{c}_t\}\rangle)$$

For a natural join, the join confirming attributes define the mapping relation in the get direction. Similarly for theta joins, the theta expression defines the mapping relation. Therefore, we can argue, that the join confirming attributes, respectively the attributes and references used in the theta join expression are a good criteria to decide whether a source class should map to a new target class. More precisely, if the attribute values and links of a source class instance $\underline{c}$ are equal to the corresponding attributes and links of the target class $\underline{c}_t$, then $\underline{c}$ should map to $\underline{c}_t$.

A natural join expression $\bowtie = \langle c_1, c_2, c_t\rangle$ can be interpreted as a theta join $\bowtie_\theta = \langle c_1, c_2, c_t\rangle$ with the join expression $\theta(v_1, v_2) = \mathbf{AND}_{\langle a_1,a_2,a_t\rangle \in A^{\bowtie}_{c_1,c_2,c_t}}(v_1.a_1 = v_2.a_2)$. Therefore, we will just consider the case for the theta join here.

Now we want to derive the *mapsTo* meta function from a join expression θ. Therefore we have to extract the attributes and references used in θ. For this purpose, we define the *extraction functions* $E_{\textsc{Att}}$ and $E_{\textsc{Ref}}$:

Definition 35 (attribute extraction function). Let $\phi \in \mathsf{Expr}_t$ be an OCL expression and $v \in \text{free}(\phi)$ be a free variable in ϕ of type t_c, then the *attribute extraction function* $E_v^{\textsc{Att}} : \mathsf{Expr}_t \to \mathcal{P}(\textsc{Att})$ is defined as

$$E_v^{\textsc{Att}}[\phi] = \begin{cases} \{a\}, & \text{if } \phi = v.a, \\ & \text{with } a \in \textsc{Att}_c^* \\[2ex] \textbf{let } \ldots, v_i = v, \ldots & \text{if } \phi = \textbf{let } \ldots, v_i = v, \ldots \textbf{ in } \alpha \\ \textbf{in}\,{}_v^{\textsc{Att}}\left[E_{v_i}^{\textsc{Att}}[\alpha]\right], & \alpha \in \mathsf{Expr} \\[2ex] & \text{if } \phi = \gamma(\alpha_1, \ldots, \alpha_n), \\ E_v^{\textsc{Att}}[\alpha_1] \cup \ldots \cup E_v^{\textsc{Att}}[\alpha_n], & \text{with } \gamma \text{ an OCL expression with} \\ & \text{subexpressions } \alpha_1, \ldots, \alpha_n \end{cases}$$

Definition 36 (reference extraction function). Let $\phi \in \mathsf{Expr}_t$ be an OCL expressions and $v \in \text{free}(\phi)$ a free variable in ϕ of type t_c, then the *reference extraction function* $E_v^{\textsc{Ref}} : \mathsf{Expr}_t \to \mathcal{P}(\textsc{Ref})$ is defined as

$$E_v^{\textsc{Ref}}[\phi] = \begin{cases} \{r\}, & \text{if } \phi = v.r, \\ & \text{with } r \in \textsc{Ref} \\[2ex] \textbf{let } \ldots, v_i = v, \ldots & \text{if } \phi = \textbf{let } \ldots, v_i = v, \ldots \textbf{ in } \alpha \\ \textbf{in}\,{}_v^{\textsc{Ref}}\left[E_{v_i}^{\textsc{Ref}}[\alpha]\right], & \alpha \in \mathsf{Expr} \\[2ex] & \text{if } \phi = \gamma(\alpha_1, \ldots, \alpha_n), \\ E_v^{\textsc{Ref}}[\alpha_1] \cup \ldots \cup E_v^{\textsc{Ref}}[\alpha_n], & \text{with } \gamma \text{ an OCL expression with} \\ & \text{subexpressions } \alpha_1, \ldots, \alpha_n \end{cases}$$

With the extraction function we can define the mapping function for a join expression θ:

Definition 37 (mapping meta functions for join conditions). Let $c, c_t \in \textsc{Class}$ be classes with $c \sim_{\bowtie} c_t$ and $\theta : \mathsf{Expr}_{t_{c_1}} \times \mathsf{Expr}_{t_{c_2}} \to \mathsf{Expr}_{\mathsf{Boolean}}$ be an OCL expression, then $mapsTo_{\langle c, c_t \rangle}^{\theta} : \mathsf{Expr}_{t_c} \times \mathsf{Expr}_{t_{c_t}} \to \mathsf{Expr}_{\mathsf{Boolean}}$ is the *mapping meta* function for c_t with

$$\begin{aligned} mapsTo_{\langle c_1, c_t \rangle}^{\phi(c_1,c_2)}(\underline{c}_1, \underline{c}_t) = {} & \textbf{AND}_{a \in E_{c_1}^{\textsc{Att}}[\phi(c_1,c_2)]}(var_a^{c_t}(\underline{c}_t) = var_a(\underline{c}_1)) \textbf{ and} \\ & \textbf{AND}_{r \in E_{c_1}^{\textsc{Ref}}[\phi(c_1,c_2)]}(var_r^{c_t}(\underline{c}_t)\text{->forAll(t} \mid var_a(\underline{c}_1)\text{->exists(s} \mid mapsTo(s, t)))) \textbf{ and} \\ & \textbf{AND}_{r \in E_{c_1}^{\textsc{Ref}}[\phi(c_1,c_2)]}(var_a(\underline{c}_1)\text{->forAll(s} \mid var_r^{c_t}(\underline{c}_t)\text{->exists(t} \mid mapsTo(s, t)))) \end{aligned}$$

In the previous section we disallowed to map existing source class instances to new target class instances. We want to relax this restriction and allow two or more new target class instances to map to the source class instance created by an update operation. However since these newly created source class instances do not exist at the constraint checking time, we cannot use the previous *mapsTo* functions. Instead, we need to represent the source class instance by the corresponding new target class instance:

Definition 38 (target mapping meta function)**.** Let $c, c_t \in \textsc{Class}$ be classes with $c \sim_{\bowtie} c_t$. A function $\mathit{mapsTo}^{c_t}_{\langle c,c_t\rangle} : \mathsf{Expr}_{t_{c_t}} \times \mathsf{Expr}_{t_{c_t}} \rightarrow \mathsf{Expr}_{\mathsf{Boolean}}$ is called a *target mapping meta function*, iff $\underline{\mathit{mapsTo}^{\sigma}_{\langle c,c_t\rangle}}(\underline{c}'_t, \underline{c}_t) = \mathit{true}$ implies there should exists an instance $\underline{c}$, so that $\underline{c} \mathrel{\underline{\sim_{\bowtie}}} \underline{c}_t$ and $\underline{c} \mathrel{\underline{\sim_{\bowtie}}} \underline{c}'_t$ after the translation. The interpretation function is defined as

$$\underline{\mathit{mapsTo}^{c_t,\sigma}_{\langle c,c_t\rangle}}(\underline{c}, \underline{c}_t) = I\left[\left[\mathit{mapsTo}^{c_t}_{\langle c,c_t\rangle}(\mathsf{c}, \mathsf{ct})\right]\right](\langle\sigma, \{\mathsf{c} \rightarrow \underline{c}, \mathsf{ct} \rightarrow \underline{c}_t\}\rangle)$$

We can derive the target mapping function from a join expression θ:

Definition 39 (target mapping meta functions for join conditions)**.** Let $c, c_t \in \textsc{Class}$ be classes with $c \sim_{\bowtie} c_t$ and $\theta : \mathsf{Expr}_{t_{c_1}} \times \mathsf{Expr}_{t_{c_2}} \rightarrow \mathsf{Expr}_{\mathsf{Boolean}}$ be a OCL expression, then $\mathit{mapsTo}^{c_t,\theta}_{\langle c,c_t\rangle} : \mathsf{Expr}_{t_{c_t}} \times \mathsf{Expr}_{t_{c_t}} \rightarrow \mathsf{Expr}_{\mathsf{Boolean}}$ is the *target mapping meta function* for c_t with

$$\begin{aligned}\mathit{mapsTo}^{\phi(c_1,c_2)}_{\langle c_1,c_t\rangle}(\underline{c}'_t, \underline{c}_t) = {} & \mathbf{AND}_{a \in E^{\textsc{Ref}}_{c_1}[\phi(c_1,c_2)]}(\mathit{var}^{c_t}_a(\underline{c}_t) = \mathit{var}^{c_t}_a(\underline{c}'_t)) \ \mathbf{and} \\ & \mathbf{AND}_{r \in E^{\textsc{Ref}}_{c_1}[\phi(c_1,c_2)]}(\mathit{var}^{c_t}_r(\underline{c}_t) = \mathit{var}^{c_t}_a(\underline{c}'_t))\end{aligned}$$

5.1.2. Mapping Strategies

If we guess the mapping, we may argue that it is not always desired to map new target class instances to existing source class instances, when the mapping meta function returns true, because there is a one-to-one or one-to-many relationship between the source class and target class and the mapping of new target instances to already mapped source class instances may disregards this relationship and maps the identity of the new target class instance to a already mapped source class behind the scene.

Therefore we propose three possible strategies for a update translation:

Map to left (right) Use all possible existing left (right) classes as instance mapping source for the new target class instance and create the missing right (left) source class instance at update translation.

Map to either left or right Use either all left or all right class instances as instance mapping source for the new target class instance. Create the missing right or left source class instance at update translation

Do not map Do not use any existing left or right class instance as mapping source for the new target class instance.

In all cases: If the new target class cannot be mapped to an existing instance, then new source class instances should be created for both sides at update translation. Map to both is not possible, because then there is already a target class instance that maps to these source classes and an update translation would not be possible, because for one matching source pair, only one instance is created in the query function.

As an example, consider the ModelJoin view definition in Listing 5.1 with the two source class instances and two new target class instances given in Figure 5.1. The result for each mapping strategy for the translation of the new target class instances are given in the following table:

```
theta join classifiers.Interface with uml.Interface
where "classifiers.Interface.name = uml.Interface.name" as jointarget.Interface {
    keep attributes commons.NamedElement.name as nameInJava,
                    uml.NamedElement.name as nameInUml
}
```

Listing 5.1: Example ModelJoin view definition joining java interfaces with um interface by name and keeping the name attributes of each source model class.

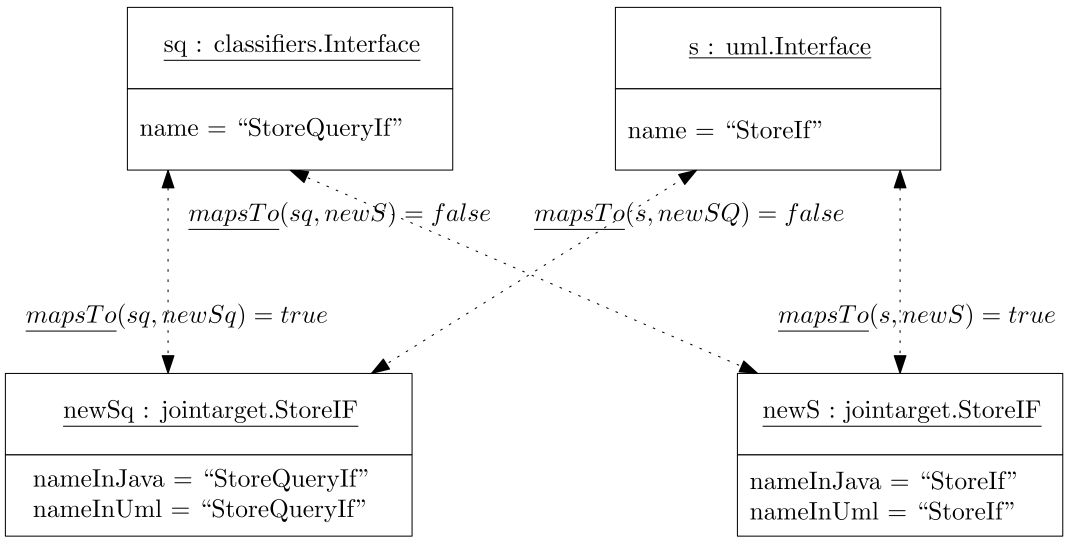

sq : classifiers.Interface
name = “StoreQueryIf”
s : uml.Interface
name = “StoreIf”
$mapsTo(sq, newS) = false$
$mapsTo(s, newSQ) = false$
$mapsTo(sq, newSq) = true$
$mapsTo(s, newS) = true$
newSq : jointarget.StoreIF
nameInJava = “StoreQueryIf”
nameInUml = “StoreQueryIf”
newS : jointarget.StoreIF
nameInJava = “StoreIf”
nameInUml = “StoreIf”

Figure 5.1.: A source model and target model instance with a source class instance for each source model and two new target class instances that map to one source class each.

Translation Strategy	Translation of new Class Instance	Translation Operations	New Mappings
Map to left	*newSq*	$\text{CREATE}_{\text{CLASS}(\text{uml.Interface})}($ $\{a_{\text{name}} = \text{“StoreQueryIf”}\}$ $)$	$sq \sim_{\underline{\bowtie}} newSq$
	newS	Translation not allowed	
Map to right	*newSq*	Translation not allowed	
	newS	$\text{CREATE}_{\text{CLASS}(\text{classifier.Interface})}($ $\{a_{\text{name}} = \text{“StoreIf”}\}$ $)$	$s \sim_{\underline{\bowtie}} newS$
Map to either left or right	*newSq*	$\text{CREATE}_{\text{CLASS}(\text{uml.Interface})}($ $\{a_{\text{name}} = \text{“StoreQueryIf”}\}$ $)$	$sq \sim_{\underline{\bowtie}} newSq$
	newS	$\text{CREATE}_{\text{CLASS}(\text{classifier.Interface})}($ $\{a_{\text{name}} = \text{“StoreIf”}\}$ $)$	$s \sim_{\underline{\bowtie}} newS$
Do not map	*newSq*	Translation not allowed	
	newS	Translation not allowed	

5.1.2.1. Notation

To derive the OCL constraints for the different strategies, we define some shorthand notations:

$isNew(v_t)$ = $c_{\bowtie}$.allInstances()->select(j | j.target = v_t)->isEmpty()

$toDeleteLeft(v_1)$ = $c_{\bowtie}$.allInstances()->select(j | j.left = v_1).target.oclIsUndefined()
$toDeleteRight(v_2)$ = $c_{\bowtie}$.allInstances()->select(j | j.right = v_2).target.oclIsUndefined()

$matchingLeft(v_t)$ = $Instances_{c_1}$->select(left | $var^{c_1 \rightarrow c_t}_{\theta(\text{left}, v_t), v_t}$)
$matchingRight(v_t)$ = $Instance_{c_2}$->select(right | $var^{c_2 \rightarrow c_t}_{\theta(v_t, \text{right}), v_t}$)

5.1.2.2. Do Not Map Strategy

We have already used the do not map strategy in the join definition extensions. For completeness we repeat it here for a general mapping expression θ:

context c_t
−− *The new target class instances should not map to an existing left source class instance*
inv newTargetInstancesLeft:
isNew(self) **implies** *matchingLeft*(self)->isEmpty()
−− *The new target class instances should not map to an existing right source class instance*

```
inv newTargetInstancesRight:
isNew(self) implies matchingRight(self)->isEmpty()
-- No two new target class instances should map to the same left class or map to the same right
class
inv newTargetInstances:
isNew(self) implies
c_t.allInstances->select(other | isNew(other) and other <> self)->forAll(other |
  not $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self}, \text{other}), \text{self}, \text{other}}$ and not $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{other}, \text{self}), \text{self}, \text{other}}$
)
-- The new created source class instances map to this target instance
inv newTargetInstancesMap:
isNew(self) implies let self2 = self in $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self}, \text{self2}), \text{self}, \text{self2}}$
```

Let $\underline{c}_t \in \sigma_{\text{CLASS}}(c_t)$ be a new target class instance. The translation creates a new instance of both source classes c_1 and c_2.

The newTargetInstancesLeft and newTargetInstancesRight invariant ensures that the target class instances $\underline{c}_t$ does not map to an existing source class instance $\underline{c}_1 \in \sigma_{\text{CLASS}}(c_1)$ or $\underline{c}_2 \in \sigma_{\text{CLASS}}(c_2)$.

Two different target instances can map to the same left or right source class instance. However if two target class instances map to the same left source class instance, then they can not map to the same right source class instance and vice versa. For each source class instance pair for which the join condition holds, only one target class instance can exist. The newTargetInstances invariant ensures, that this is the case for all source class pairs.

Lastly the newTargetInstancesMap invariant ensures, that the join condition holds for the pair of source class instance of each target class instance.

5.1.2.3. Map to Left (Right) Strategy

In the map to left (respectively right) strategy, new target instances may only map to existing left (respectively right) source class instances. New target instances mapping to existing right (respectively left) source class instances will be forbidden.

```
context c_t
-- The new target class instances should not map to an existing right source class instance
inv newTargetInstancesRight:
isNew(self) implies matchingRight(self)->isEmpty()
-- For each existing matching left source class instance there must be exactly one new target class
instance that only maps to the matching left source class instance.
inv newTargetInstancesLeft:
isNew(self) implies matchingLeft(self)->forAll(left |
  let mappingTarget = $c_t$.allInstances->select(target | isNew(target))->select(target |
    $mapsTo^{\theta}_{\langle c_1, c_t \rangle}$(left, target) and $mapsTo^{c_t,\theta}_{\langle c_2, c_t \rangle}$(self, target)
  )
  in mappingTarget->size() = 1 and matchingLeft(self)->forAll(otherLeft |
    $mapsTo^{\theta}_{\langle c_1, c_t \rangle}$(otherLeft, mappingTarget) implies otherLeft = left
  )
)
-- If there are no matching left source class instances, then a new left source class instance should
be created at update translation. In this case, for all matching new created right class instances,
there must be exactly one target class instance that maps only to the new left and new right class
instances.
inv newTargetInstancesLeftNew:
isNew(self) and matchingLeft(self)->isEmpty() implies
$c_t$.allInstances->select(right | isNew(right) and $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self, right}), \text{self, right}}$)->forAll(right |
  let mappingTarget = $c_t$.allInstances->select(target | isNew(target))->select(target |
    $mapsTo^{c_t,\theta}_{\langle c_1, c_t \rangle}$(self, target) and $mapsTo^{c_t,\theta}_{\langle c_2, c_t \rangle}$(right, target)
  )
  in mappingTarget->size() = 1 and $c_t$.allInstances->select(otherRight |
    isNew(otherRight) and $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self, right}), \text{self, right}}$
  )->forAll(otherRight |
    $mapsTo^{c_t,\theta}_{\langle c_2, c_t \rangle}$(otherRight, mappingTarget) implies otherRight = right
  )
)
-- No two new target class instances should map to the same left class and right class instance.
inv newTargetInstances:
isNew(self) implies
$c_t$.allInstances->select(other | isNew(other) and other <> self)->forAll(other |
  not ($mapsTo^{c_t,\theta}_{\langle c_1, c_t \rangle}$(other, self) and $mapsTo^{c_t,\theta}_{\langle c_2, c_t \rangle}$(other, self))
)
-- The new created source class instances map to this target instance
inv newTargetInstancesMap:
isNew(self) implies let self2 = self in $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self, self2}), \text{self, self2}}$
```

Let $\underline{c}_t \in \sigma_{\text{CLASS}}(c_t)$ be a new target class instance.

The translation strategy for match only left is: If there is an existing matching left class instance $\underline{c}_1 \in \sigma_{\text{CLASS}}(c_1)$, then no left source class instance is created and the existing source class is used as left mapping source: $\underline{c}_1 \sim_{\bowtie} \underline{c}_t$. If no matching left class instance $\underline{c}_1$ exists, then a new class instance $\underline{c}_1^{new} \in I(c_1)$ is created and used as mapping source: $\underline{c}_1^{new} \sim_{\bowtie} \underline{c}_t$. A right source class instance $\underline{c}_2 \in I(c_2)$ with $\underline{c}_2 \sim_{\bowtie} \underline{c}_t$ is always be created. The case for match only right is symmetric.

The newTargetInstancesRight invariant ensures, that the new target class instance $\underline{c}_t$ does not map to existing right source class instances $\underline{c}_2 \in \sigma_{\text{CLASS}}(c_2)$.

The newTargetInstancesLeft invariant ensures that, for each pair of an existing left source class $\underline{c}_1 \in \sigma_{\text{CLASS}}(c_1)$ and a right source class instance to create $\underline{c}_2^{new} \in I(c_2)$ with

$\underline{\theta(c_1, c_2)}_{\underline{c}_1, \underline{c}_2^{new}} = \mathit{true}$, there exists exactly one target class instance $\underline{c}_t$ with $\underline{c}_1 \sim_{\underline{\bowtie}} \underline{c}_t$ and $\underline{c}_2^{new} \sim_{\underline{\bowtie}} \underline{c}_t$. Further it checks, that the target class instance $\underline{c}_t$ does not map to another left source class $\underline{c}_1' \in \sigma_{\text{Class}}(c_1)$, because a target class can only map to one left source class.

The newTargetInstancesLeft invariant ensures that, if there is no pair of an existing left source class $\underline{c}_1 \in \sigma_{\text{Class}}(c_1)$ and a right source class instance to create $\underline{c}_2^{new} \in I(c_2)$ with $\underline{\theta(c_1, c_2)}_{\underline{c}_1, \underline{c}_2^{new}} = \mathit{true}$, then for all pairs of the new left source class instance to create $\underline{c}_1^{new} \in I(c_1)$ and an existing right source class instance or right source class to create $\underline{c}_2 \in I(c_2)$, there exists exactly one target class instance $\underline{c}_t \in \sigma_{\text{Class}}(c_t)$ with $\underline{c}_1^{new} \sim_{\underline{\bowtie}} \underline{c}_t$ and $\underline{c}_2 \sim_{\underline{\bowtie}} \underline{c}_t$. Further it check, that $\underline{c}_t$ only maps to the right source class instance $\underline{c}_2$, because a target class can only map to one right source class.

The newTargetInstances and newTargetInstancesMap invariants are the same as in the do not map strategy.

5.1.2.4. Match Left or Right

```
context c_t
-- If there are no matching left source class instances, then for each existing matching right source
class instance there must be exactly one new target class instance that only maps to the matching
right source class instance, else the new target class instance should not map to an existing right
source class instance.
inv newTargetInstancesRight:
isNew(self) implies if matchingLeft(self)->isEmpty() then
  matchingRight(self)->forAll(right |
    let mappingTarget = c_t.allInstances->select(target | isNew(target))->select(target |
      $mapsTo^{\theta}_{\langle c_2, c_t \rangle}$(right, target) and $mapsTo^{c_t,\theta}_{\langle c_1, c_t \rangle}$(self, target)
    )
    in mappingTarget->size() = 1 and matchingRight(self)->forAll(otherRight |
      $mapsTo^{\theta}_{\langle c_2, c_t \rangle}$(otherRight, mappingTarget) implies otherRight = right
    )
  )
else
  matchingRight(self)->isEmpty()
endif
-- If there are no matching right source class instances, then for each existing matching left
source class instance there must be exactly one new target class instance that only maps to the
matching left source class instance, else the new target class instance should not map to an exist-
ing left source class instance.
inv newTargetInstancesLeft:
isNew(self) implies if matchingRight(self)->isEmpty() then
```

```
  matchingLeft(self)->forAll(left |
    let mappingTarget = $c_t$.allInstances->select(target | isNew(target))->select(target |
      $mapsTo^{\theta}_{\langle c_1, c_t\rangle}$(left, target) and $mapsTo^{c_t,\theta}_{\langle c_2, c_t\rangle}$(self, target)
    )
    in mappingTarget->size() = 1 and matchingLeft(self)->forAll(otherLeft |
      $mapsTo^{\theta}_{\langle c_1, c_t\rangle}$(otherLeft, mappingTarget) implies otherLeft = left
    )
  )
else
  matchingLeft(self)->isEmpty()
endif
-- If there are no matching left source class instances, then a new left source class instance should
be created at update translation. In this case, for all matching new created right class instances,
there must be exactly one target class instance that maps only to the new left and new right class
instances.
inv newTargetInstancesLeftNew:
isNew(self) and matchingLeft(self)->isEmpty() implies
$c_t$.allInstances->select(right | isNew(right) and $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self, right}), \text{self, right}}$)->forAll(right |
  let mappingTarget = $c_t$.allInstances->select(target | isNew(target))->select(target |
    $mapsTo^{c_t,\theta}_{\langle c_1, c_t\rangle}$(self, target) and $mapsTo^{c_t,\theta}_{\langle c_2, c_t\rangle}$(right, target)
  )
  in mappingTarget->size() = 1 and $c_t$.allInstances->select(otherRight |
    isNew(otherRight) and $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self, otherRight}), \text{self, otherRight}}$
  )->forAll(otherRight |
    $mapsTo^{c_t,\theta}_{\langle c_2, c_t\rangle}$(otherRight, mappingTarget) implies otherRight = right
  )
)
-- If there are no matching right source class instances, then a new right source class instance
should be created at update translation. In this case, for all matching new created left class in-
stances, there must be exactly one target class instance that maps only to the new left and new
right class instances.
inv newTargetInstancesRightNew:
isNew(self) and matchingRight(self)->isEmpty() implies
$c_t$.allInstances->select(left | isNew(left) and $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self, left}), \text{self, left}}$)->forAll(left |
  let mappingTarget = $c_t$.allInstances->select(target | isNew(target))->select(target |
    $mapsTo^{c_t,\theta}_{\langle c_2, c_t\rangle}$(self, target) and $mapsTo^{c_t,\theta}_{\langle c_1, c_t\rangle}$(left, target)
  )
  in mappingTarget->size() = 1 and $c_t$.allInstances->select(otherLeft |
    isNew(otherLeft) and $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self, otherLeft}), \text{self, otherLeft}}$
  )->forAll(otherLeft |
    $mapsTo^{c_t,\theta}_{\langle c_1, c_t\rangle}$(otherLeft, mappingTarget) implies otherLeft = left
  )
```

)
−− *No two new target class instances should map to the same left class and right class*
inv newTargetInstances:
isNew(self) **implies**
c_t.allInstances->select(other | *isNew*(other) **and** other <> self)->forAll(other |
  not ($mapsTo^{c_t,\theta}_{\langle c_1,c_t\rangle}$(other, self) **and** $mapsTo^{c_t,\theta}_{\langle c_2,c_t\rangle}$(other, self))
)
−− *The new created source class instances map to this target instance*
inv newTargetInstancesMap:
isNew(self) **implies let** self2 = self **in** $var^{c_1\to c_t,c_2\to c_t}_{\theta(\text{self},\text{self2}),\text{self},\text{self2}}$

Let $\underline{c}_t \in \sigma_{\text{CLASS}}(c_t)$ be a new target class instance.

The translation strategy for match left or right is: If there is an existing matching left class instance $\underline{c}_1 \in \sigma_{\text{CLASS}}(c_1)$, then no left source class instance is created and the existing source class is used as left mapping source: $\underline{c}_1 \sim_{\underline{\bowtie}} \underline{c}_t$. If no matching left class instance $\underline{c}_1$ does exist, then a new class instance $\underline{c}_1^{new} \in \overline{I(c_1)}$ is created and used as mapping source: $\underline{c}_1^{new} \sim_{\underline{\bowtie}} \underline{c}_t$. The same applies to the right source class.

The invariants are similar to the ones in Map to left (right) strategy. However depend now on the existence of a new left or new right source class instance.

5.1.3. Handling of Keep Attribute, Calculate Attribute and Keep References

If we allow to map to existing source classes, we have to make sure that the GET-EQUALITY and PUT-EQUALITY holds for these mappings, too. Therefore we have to extend the constraints for keep attribute, calculate attribute and keep reference expressions. We use the *mapsTo* meta functions to introduce the additional constraints.

For keep attribute expressions, the following additional constraints should be created:

context c_t
inv attributeNewMapping:
isNew(self) **implies**
$Instances_{c_1}$->forAll(source | $mapsTo^{\theta}_{\langle c_1,c_t\rangle}$(source, self) **implies** self.a_t = var_{a_1}(source))
inv attributeNewTargetMapping:
isNew(self) **implies**
c_t.allInstances()->forAll(other | $mapsTo^{c_t,\theta}_{\langle c_1,c_t\rangle}$(other, self) **implies** self.a_t = $var^{c_t}_{a_1}$(other))

The constraints for a source class attribute of the right join source are symmetric.

For calculate attribute, the following additional constraints should be created:

context c_t
inv attributeNewMapping:
isNew(self) **implies**
$Instances_{c_1}$->forAll(left |

$\mathit{Instance}_{c_2}$->forAll(right |
  $\mathit{mapsTo}^{\theta}_{\langle c_1, c_t\rangle}$(left, self) **and** $\mathit{mapsTo}^{\theta}_{\langle c_2, c_t\rangle}$(right, self) **implies** self.a_t = ϕ(left, right)
)
)
inv attributeNewTargetMappingLeft:
isNew(self) **implies**
c_t.allInstances()->forAll(other |
  $\mathit{mapsTo}^{c_t,\theta}_{\langle c_1, c_t\rangle}$(other, self) **implies** self.a_t = $\mathit{var}^{c_1\to c_t, c_2\to c_t}_{\phi(\text{other}, \text{self}), \text{other}, \text{self}}$
)
inv attributeNewTargetMappingRight:
isNew(self) **implies**
c_t.allInstances()->forAll(other |
  $\mathit{mapsTo}^{c_t,\theta}_{\langle c_2, c_t\rangle}$(other, self) **implies** self.a_t = $\mathit{var}^{c_1\to c_t, c_2\to c_t}_{\phi(\text{self}, \text{other}), \text{self}, \text{other}}$
)

For keep reference, the following additional constraints should be created:

context c_t
inv referenceNewMapping:
isNew(self) **implies**
$\mathit{Instances}_{c_1}$->forAll(source | $\mathit{mapsTo}^{\theta}_{\langle c_1, c_t\rangle}$(source, self) **implies** self.r_t = var_r(source))

5.1.4. Translation Algorithm for Target Class Instances

Because the do not map and map to left (right) strategies are more restricted versions of the map to left or right strategy the same algorithm can be used for all three strategies. Algorithm 1 shows the translation algorithm for all three strategies.

Algorithm 1 Translation algorithm for new target class instances of a theta join $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$

```
1:  procedure TRANSLATEJOINTARGET$_{c_t}$(target)
2:      if isNew(target) then
3:          left ← matchingLeft(target)->find(left |
4:              $mapsTo^{\theta}_{\langle c_1, c_t \rangle}$(left, target)
5:            )
6:          if left.oclIsUndefined() then
7:              $V_1$ ← attribute values for the left source class instance
8:              left ← CREATE$_{\text{CLASS}(c_1)}$($V_1$).
9:          else
10:             update attributes and references of left
11:         right ← matchingRight(target)->find(right |
12:             $mapsTo^{\theta}_{\langle c_2, c_t \rangle}$(right, target)
13:           )
14:         if right.oclIsUndefined() then
15:             $V_2$ ← attribute values for the right source class instance
16:             right ← CREATE$_{\text{CLASS}(c_2)}$($V_2$).
17:         else
18:             update attributes and references of right
19:     else
20:         join = ← $c_\bowtie$.allInstances->find(j | j.target = target)
21:         Update attribute values and links of join.left and join.right
```

5.1.5. Limitations of Guessed Mappings

Guessing the mapping works well, if the set $E_{left} = E^{\text{ATT}}_{c_1}[\phi(c_1, c_2)] \cup E^{\text{REF}}_{c_1}[\phi(c_1, c_2)]$ forms a unique key for all left source class instances of c_1 and the set $E_{right} = E^{\text{ATT}}_{c_2}[\phi(c_1, c_2)] \cup E^{\text{REF}}_{c_2}[\phi(c_1, c_2)]$ forms a unique key for all right source class instances of c_2. For each new target class the mapping left source class and right source class instance is then uniquely determined. More precisely, for each new target class instance $\underline{c}_t \in I(c_t)$, there exists at most one left source class instance $\underline{c}_1 \in I(c_1)$ with $\underline{mapsTo}^{\theta}_{\langle c_1, c_t \rangle}(c_1, c_t) = true$ and there exists at most one right source class instance $\underline{c}_2 \in I(c_2)$ with $\underline{mapsTo}^{\theta}_{\langle c_2, c_t \rangle}(c_2, c_t) = true$. This can be easily verified by making the assumption, that for a new target class instance $\underline{c}_t \in \sigma_{\text{CLASS}}(c_t)$, there would be two left source class instances $\underline{c}_1, \underline{c}'_1 \in \sigma_{\text{CLASS}}(c_1)$ with $\underline{c}_1 \neq \underline{c}'_1$ which map to $\underline{c}_t$. Then from the definition of the mapping meta function, it follows, that for the two source class instances $\underline{c}_1$ and $\underline{c}'_1$ the values of E_{left} are equal. A contradiction to the unique key property of E_{left}.

If E_{left} or E_{right} does not form a unique key, then the mapping for a new target class instance may be impossible to be guessed uniquely. In this case the translation for the new target class would be disallowed by the OCL constraints, because the mapping is ambiguous. To handle such cases, the mapping must be manually specified to resolve the ambiguities.

Consider for example the ModelJoin expression in Listing 5.2 with the simplified metamodel and model given in Figure 5.2. In this example the sets E_{left} and E_{right} are $E_{left} = \{\text{name}\}$, $E_{right} = \{\text{namespace}\}$, where E_{right} does not form an unique key. For example the instances cu1, cu2 $\in I$(java.CompilationUnit) have the same value for the attribute namespace. Let new $\in I$(jt.ComponentFile) be a new target model class instance. The class instance new could map to either cu1 or cu2 (see Figure 5.3). The mapping is ambiguous. A unique mapping could be determined by the filename attribute. However this is not part of the join condition and therefore not used by the mapping meta function.

```
theta join uml.Component with java.CompilationUnit
where "containers.CompilationUnit.namespace.includes(uml.Component.name)" as jointarget.
    ComponentFile {
    keep attributes uml.Component.name as componentName,
                    commons.NamedElement.name as filename,
                    commons.CompilationUnit.namespaces
}
```

Listing 5.2: Example ModelJoin view definition for ComponentFiles joining each uml component with its compilation units.

5.1.6. Manual Mapping

The mapping of new target class instances to existing source class instances could be either manually specified at view definition time or at the update translation.

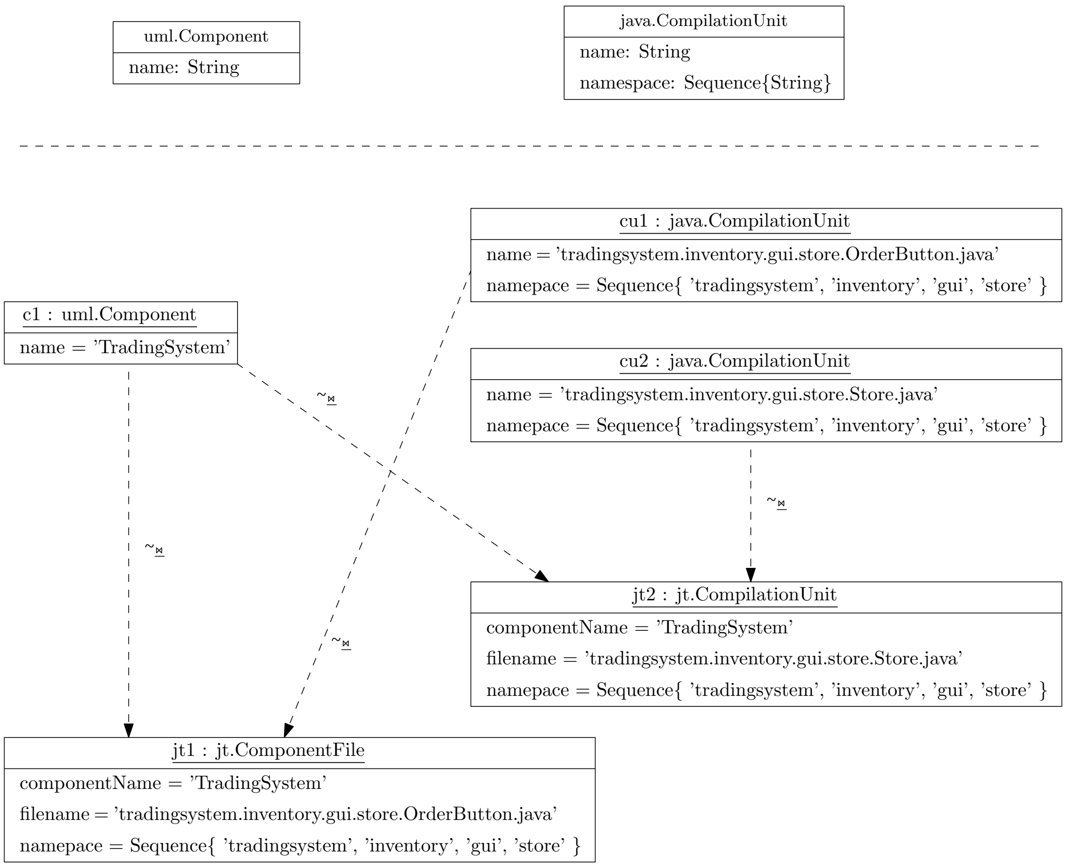

uml.Component
name: String
java.CompilationUnit
name: String
namespace: Sequence{String}
cu1 : java.CompilationUnit
name = 'tradingsystem.inventory.gui.store.OrderButton.java'
namepace = Sequence{ 'tradingsystem', 'inventory', 'gui', 'store' }
c1 : uml.Component
name = 'TradingSystem'
cu2 : java.CompilationUnit
name = 'tradingsystem.inventory.gui.store.Store.java'
namepace = Sequence{ 'tradingsystem', 'inventory', 'gui', 'store' }
jt2 : jt.CompilationUnit
componentName = 'TradingSystem'
filename = 'tradingsystem.inventory.gui.store.Store.java'
namepace = Sequence{ 'tradingsystem', 'inventory', 'gui', 'store' }
jt1 : jt.ComponentFile
componentName = 'TradingSystem'
filename = 'tradingsystem.inventory.gui.store.OrderButton.java'
namepace = Sequence{ 'tradingsystem', 'inventory', 'gui', 'store' }

Figure 5.2.: Simplified metamodels and models for the ModelJoin view definition in Listing 5.2

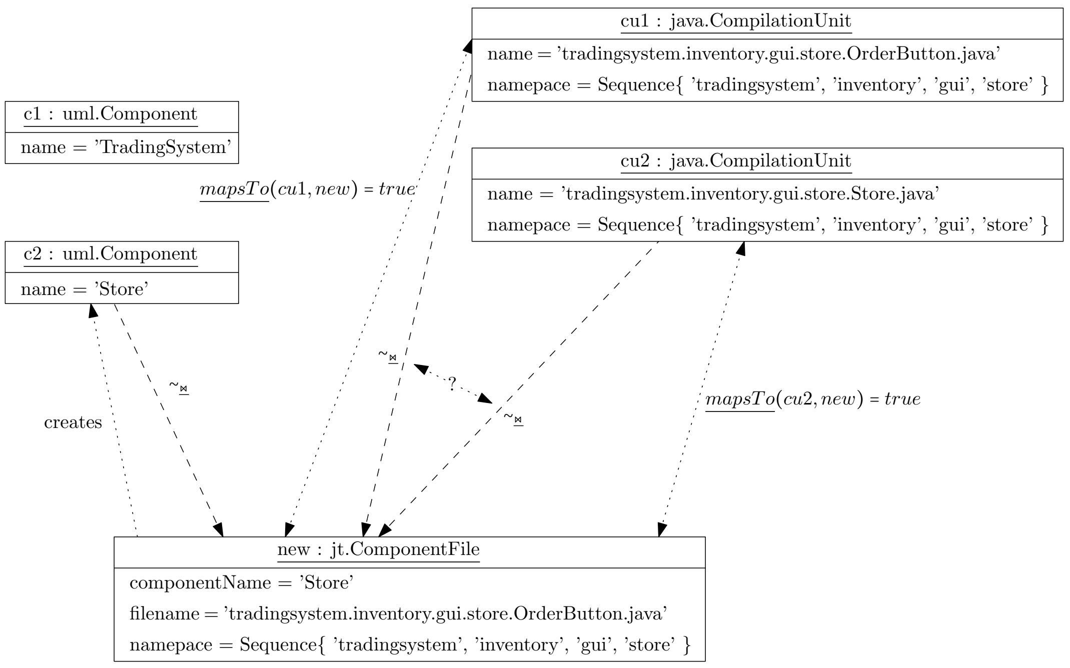
At view definition time the mapping could be specified by supplying either the meta mapping function explicitly or by giving a unique key, that is used instead of the attributes and references in the mapping expression.

In the example of Figure 5.3, the attribute filename could be used as unique key for the mapping function. To specify the key at view creation time, the ModelJoin syntax must be extended. We propose the syntax extension with mapping keys *listOfAttributes* for join expressions. An example for an extended ModelJoin view definition is given in Listing 5.3.

The mapping meta functions then would use the provided key attributes to find the mapping source classes. Therefore let $A^{key}_{c_1} \subseteq \textsc{Att}^*_{c_1}$ the given key attributes of the left source class and $A^{key}_{c_2} \subseteq \textsc{Att}^*_{c_2}$ the given key attributes of the right source class. The used meta mapping function for the left source class are then defined as:

$$mapsTo_{\langle c_1, c_t \rangle}(\underline{c}_1, \underline{c}_t) = \textbf{AND}_{a \in A^{key}_{c_1}}(var^{c_t}_a(\underline{c}_t) = var_a(\underline{c}_1))$$
$$mapsTo^{c_t}{}_{\langle c_1, c_t \rangle}(\underline{c}'_t, \underline{c}_t) = \textbf{AND}_{a \in A^{key}_{c_1}}(var^{c_t}_a(\underline{c}_t) = var^{c_t}_a(\underline{c}'_t))$$

The meta mapping functions for the right source class are defined symmetrically.



cu1 : java.CompilationUnit
name = 'tradingsystem.inventory.gui.store.OrderButton.java'
namepace = Sequence{ 'tradingsystem', 'inventory', 'gui', 'store' }
c1 : uml.Component
name = 'TradingSystem'
cu2 : java.CompilationUnit
name = 'tradingsystem.inventory.gui.store.Store.java'
namepace = Sequence{ 'tradingsystem', 'inventory', 'gui', 'store' }
mapsTo(cu1, new) = true
c2 : uml.Component
name = 'Store'
creates
?
mapsTo(cu2, new) = true
new : jt.ComponentFile
componentName = 'Store'
filename = 'tradingsystem.inventory.gui.store.OrderButton.java'
namepace = Sequence{ 'tradingsystem', 'inventory', 'gui', 'store' }

Figure 5.3.: The mapping for the new instance of jointarget.ComponentFile cannot be guessed from the join condition in Listing 5.2, because the attribute namespace of the right source class is not unique.

5.1.7. Mappings Between Different ModelJoin Expressions

We have disallowed the creation of new target class instances, if one of its source classes was involved in another ModelJoin join expression. We now want to relax this constraint. Let $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$ and $\bowtie_\theta = \langle c_1, c_2', c_t' \rangle$ be two join expressions with a common source class c_1 (see Figure 5.4). If we allow the creation of new instances $\underline{c}_t \in I(c_t)$, new instances of c_1 may be created during translation. This instances $\underline{c}_1 \in I(c_1)$ could form new join pairs with existing instances $\underline{c}_2' \in \sigma_{\textsc{Class}}(c_2')$. This may lead to new instances $\underline{c}_t' \in I(c_t')$, that must exist in the view before translation (see Figure 5.5). Further $\underline{c}_1$ could not only form mapping pairs with existing instance $\underline{c}_2'$, it could form mapping pairs with instanced $\underline{c}_2' \in I(c_2')$ that are created by the update translation of new instance $\underline{c}_t' \in I(c_t')$ (see Figure 5.6). To check, that these target class instances exist, additional constraints are necessary. Let $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$ be a theta join expression. The noConflictsWithOtherMappings invariant from the join extensions should be dropped and new constraints should be created. To formulate these constraints we must extend the target mapping meta function from Definition 39 to allow the mapping between instances of two different target class instances:

Definition 40 (extended target mapping meta function)**.** Let $c, c_t, c_t' \in \textsc{Class}$ be classes with $c \sim_\bowtie c_t$, $c \sim_\bowtie c_t'$ and $\theta : \mathsf{Expr}_{t_{c_1}} \times \mathsf{Expr}_{t_{c_2}} \rightarrow \mathsf{Expr}_{\mathsf{Boolean}}$ be an OCL expression, then let $mapsTo^{c_t', c_t, \theta}_{\langle c, c_t \rangle} : \mathsf{Expr}_{t_{c_t'}} \times \mathsf{Expr}_{t_{c_t}} \rightarrow \mathsf{Expr}_{\mathsf{Boolean}}$ be the target mapping meta function for c_t with

```
theta join uml.Component with java.CompilationUnit
where "containers.CompilationUnit.namespace.includes(uml.Component.name)"
as jointarget.ComponentFile
with mapping keys java.CompilationUnit, uml.Component.name {
    keep attributes uml.Component.name as componentName,
                    commons.NamedElement.name as filename,
                    commons.CompilationUnit.namespaces
}
```

Listing 5.3: Extended version of the ModelJoin view definition in Listing 5.2 to specify the key for the meta mapping functions.

$$\begin{tikzcd}
c'_2 \arrow[dr, dashed, "\sim\bowtie"] & & c_1 \arrow[dl, dashed, "\sim\bowtie"] \arrow[dr, dashed, "\sim\bowtie"] & & c_2 \arrow[dl, dashed, "\sim\bowtie"] \\
& c'_t & & c_t &
\end{tikzcd}$$

Figure 5.4.: Source class c_1 that is used in two join expressions $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$ and $\bowtie_{\theta'} = \langle c_1, c'_2, c'_t \rangle$.

$$mapsTo^{\phi(c_1,c_2)}_{\langle c_1,c_t \rangle}(\underline{c}'_t, \underline{c}_t) = \textbf{AND}_{a \in E^{\text{Ref}}_{c_1}[\phi(c_1,c_2)]}(var^{c_t}_a(\underline{c}_t) = var^{c'_t}_a(\underline{c}'_t)) \textbf{ and}$$
$$\textbf{AND}_{r \in E^{\text{Ref}}_{c_1}[\phi(c_1,c_2)]}(var^{c_t}_r(\underline{c}_t) = var^{c'_t}_a(\underline{c}'_t))$$

The extended target mapping function is only defined, if the used target meta variables are defined. It the extended target mapping meta function is undefined, the content of this subsection can not be applied. The expression for the equality of the target meta variables for reference can only be evaluated, if the target class of the target reference is the same in both target classes. If this is not the case, the content of this subsection can not be applied too. Now we want to formulate new constraints for the map left or right strategy, that respect new target class instance in other join expressions:

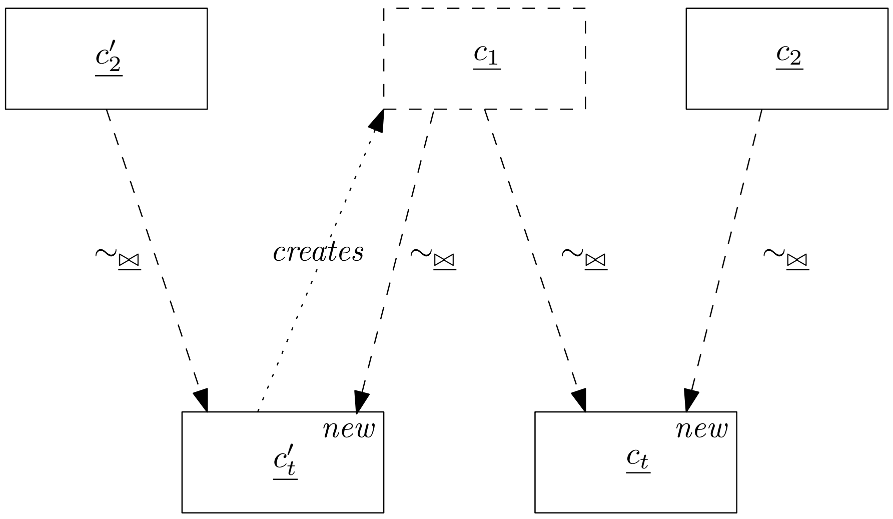

$\underline{c'_2}$
$\underline{c_1}$
$\underline{c_2}$
$\sim_{\underline{\bowtie}}$
creates
$\sim_{\underline{\bowtie}}$
$\sim_{\underline{\bowtie}}$
$\sim_{\underline{\bowtie}}$
new
$\underline{c'_t}$
new
$\underline{c_t}$

Figure 5.5.: If the source class c_1 is involved in two join expressions $\bowtie_\theta = \langle c_1, c_2, c_t\rangle$ and $\bowtie_\theta = \langle c_1, c'_2, c'_t\rangle$, the creation of a new target instance $\underline{c'_t}$ could lead to the creation of a new source class instance $\underline{c_1}$ that forms a mapping pair with an existing source class instance $\underline{c_2}$.

context c_t
−− *If there are no matching left source class instances then a new left source class instance is created at translation. For all other join expressions, that share the left source class as one of its join sources, for each matching*
inv newTargetInstancesRight:
isNew(self) **implies if** $matching_{c_1}$(self)->isEmpty() **then**
  AND$_{\{\langle c', c'_t, \theta'\rangle \mid \bowtie_{\theta'}=\langle c_1, c', c'_t\rangle \vee \bowtie_{\theta'}=\langle c', c_1, c'_t\rangle\}}$($matching_{c'}$(self)->forAll(other |
    let mappingTarget = c'_t.allInstances->select(target | *isNew*(target))->select(target |
      $mapsTo^{\theta'}_{\langle c_1, c'_t\rangle}$(other, target) **and** $mapsTo^{c_t, c'_t, \theta'}_{\langle c_1, c'_t\rangle}$(self, target)
)
    in mappingTarget->size() = 1 **and** $matching_{c'}$(self)->forAll(otherOther |
      $mapsTo^{\theta'}_{\langle c', c_t\rangle}$(otherOther, mappingTarget) **implies** otherOther = other
)
))
else
  matchingRight(self)->isEmpty()
endif
−− *If there are no matching left source class instances then a new left source class instance should be created at update translation. In this case, for all matching new created right class instances, there must be exactly one target class instance that maps only to the new left and new right class instances.*
inv newTargetInstancesLeftNew:
isNew(self) **and** *matchingLeft*(self)->isEmpty() **implies**

```
$\textbf{AND}_{\{\langle c', c'_t, \theta'\rangle \mid \bowtie_{\theta'} = \langle c_1, c', c'_t\rangle \vee \bowtie_{\theta'} = \langle c', c_1, c'_t\rangle\}}$( $c'_t$.allInstances->select(other |
  $\mathit{isNew}$(other) and $\mathit{var}^{c_1 \to c_t, c' \to c'_t}_{\theta'(\text{self, other}), \text{self, other}}$
)->forAll(other |
  let mappingTarget = $c'_t$.allInstances->select(target | $\mathit{isNew}$(target))->select(target |
    $\mathit{mapsTo}^{c_t, \theta'}_{\langle c_1, c'_t\rangle}$(self, target) and $\mathit{mapsTo}^{c'_t, \theta'}_{\langle c', c'_t\rangle}$(other, target)
  )
  in mappingTarget->size() = 1 and $c'_t$.allInstances->select(otherOther |
    $\mathit{isNew}$(otherOther) and $\mathit{var}^{c_1 \to c_t, c' \to c'_t}_{\theta(\text{self, otherOther}), \text{self, otherOther}}$
  )->forAll(otherOther |
    $\mathit{mapsTo}^{c'_t, \theta'}_{\langle c', c'_t\rangle}$(otherOther, mappingTarget) implies otherOther = other
  )
))
```

where

```
$\mathit{matching}_c(v_t)$ = $\mathit{Instances}_c$->select(left | $\mathit{var}^{c \to c_t}_{\theta(\text{left}, v_t), v_t}$)
```

if v_t is of type c_t and c is the left source class of a join with condition θ. If c is the right source class of a join instead, then the symmetric expression is used. The invariants newTargetInstancesLeft and newTargetInstancesRightNew are defined analogously.

If c_t is not the target class of a keep reference expression $\kappa_{\text{REF}} = \langle r, r_t\rangle$ with $associates(r) = \langle c, \hat{c}\rangle$ and $associates(r_t) = \langle c_t, \hat{c}_t\rangle$ instead of a join, then the following constraints should be created instead:

```
context $\hat{c}_t$
-- If the new created source instance forms a join pair in a join expression with an existing join
source class, then the new target instance must exist.
inv newTargetInstances:
$\textbf{AND}_{\{\langle c', c'_t, \theta'\rangle \mid \bowtie_{\theta'} = \langle \hat{c}, c', c'_t\rangle \vee \bowtie_{\theta'} = \langle c', \hat{c}, c'_t\rangle\}}$( $\mathit{matching}_{c'}$(self)->forAll(other |
  let mappingTarget = $c'_t$.allInstances->select(target | $\mathit{isNew}$(target))->select(target |
    $\mathit{mapsTo}^{\theta'}_{\langle \hat{c}, c'_t\rangle}$(other, target) and $\mathit{mapsTo}^{c_t, \theta'}_{\langle \hat{c}, c'_t\rangle}$(self, target)
  )
  in mappingTarget->size() = 1 and $\mathit{matching}_{c'}$(self)->forAll(otherOther |
    $\mathit{mapsTo}^{\theta'}_{\langle c', c_t\rangle}$(otherOther, mappingTarget) implies otherOther = other
  )
) )
-- If the new created source instance forms a join pair in a join expression with a new to create
join source class, then the new target instance must exist.
inv newTargetInstancesNew:
$\textbf{AND}_{\{\langle c', c'_t, \theta'\rangle \mid \bowtie_{\theta'} = \langle \hat{c}, c', c'_t\rangle \vee \bowtie_{\theta'} = \langle c', \hat{c}, c'_t\rangle\}}$( $c'_t$.allInstances->select(other |
  $\mathit{isNew}$(other) and $\mathit{var}^{\hat{c} \to c_t, c' \to c'_t}_{\theta'(\text{self, other}), \text{self, other}}$
)->forAll(other |
  **let** mappingTarget = $c'_t$.allInstances->select(target | *isNew*(target))->select(target |
    $mapsTo^{c_t,\theta'}_{\langle\hat{c},c'_t\rangle}$(self, target) **and** $mapsTo^{c'_t,\theta'}_{\langle c',c'_t\rangle}$(other, target)
  )
  **in** mappingTarget->size() = 1 **and** $c'_t$.allInstances->select(otherOther |
    *isNew*(otherOther) **and** $var^{\hat{c}\to c_t,c'\to c'_t}_{\theta(\text{self, otherOther}),\text{self},\text{otherOther}}$
  )->forAll(otherOther |
    $mapsTo^{c'_t,\theta'}_{\langle c',c'_t\rangle}$(otherOther, mappingTarget) **implies** otherOther = other
  )
))
```

Because target class instances can now map to source class instances, that are not their left or right join source class, we need additional constraints to prevent conflicting attribute values:

```
**context** $c_t$
**inv** attributeNewMapping:
*isNew*(self) **implies**
**AND**$_{\{\langle c',c'_t,\theta'\rangle \mid \bowtie_{\theta'}=\langle c_1,c',c'_t\rangle\vee\bowtie_{\theta'}=\langle c',c_1,c'_t\rangle\}}$(
  $Instances_{c_1}$->forAll(source | $mapsTo^{\theta}_{\langle c_1,c_t\rangle}$(source, self) **implies**
  **AND**$_{\langle a_t,a\rangle\in\{\langle a_t,a\rangle\in\textsc{Att}^*_{c_t}\times\textsc{Att}^*_{c_1} \mid a\sim_{\bowtie}a_t\wedge a\sim_{\bowtie}a'_t,a'_t\in\textsc{Att}^*_{c'_t}\}}$(self.$a_t$ = $var_a$(source))
)
**inv** attributeNewMappingNew:
*isNew*(self) **implies**
**AND**$_{\{\langle c_t,c'_t,\theta'\rangle \mid \bowtie_{\theta'}=\langle c_1,c',c'_t\rangle\vee\bowtie_{\theta'}=\langle c',c_1,c'_t\rangle\}}$(
  $c'_t$.allInstances()->select(other |
    *isNew*(other) **and** $matching_{c'}$(other)->isEmpty()
  )->forAll(other |
    $mapsTo^{\theta}_{\langle c'_t,c_t\rangle}$(other, self) **implies**
    **AND**$_{\langle a_t,a'_t\rangle\in\{\langle a_t,a'_t\rangle\in\textsc{Att}^*_{c_t}\times\textsc{Att}^*_{c'_t} \mid a\sim_{\bowtie}a_t\wedge a\sim_{\bowtie}a'_t,a\in\textsc{Att}^*_{c_1}\}}$(self.$a_t$ =other.$a'_t$)
  )
)
```

Links must be checked in a similar way:

```
**context** $c_t$
**inv** linksNewMapping:
*isNew*(self) **implies**
**AND**$_{\{\langle c',c'_t,\theta'\rangle \mid \bowtie_{\theta'}=\langle c_1,c',c'_t\rangle\vee\bowtie_{\theta'}=\langle c',c_1,c'_t\rangle\}}$(
  $Instances_{c_1}$->forAll(source | $mapsTo^{\theta}_{\langle c_1,c_t\rangle}$(source, self) **implies**
  **AND**$_{\langle r_t,r\rangle\in\{\langle r_t,r\rangle\in\textsc{Ref}^*_{c_t}\times\textsc{Ref}^*_{c_1} \mid r\sim_{\bowtie}r_t\wedge r\sim_{\bowtie}r'_t,r'_t\in\textsc{Ref}^*_{c'_t}\}}$(self.$r_t$ = $var_r$(source))
)
**inv** attributeNewMappingNew:
*isNew*(self) **implies**
**AND**$_{\{\langle c_t,c'_t,\theta'\rangle \mid \bowtie_{\theta'}=\langle c_1,c',c'_t\rangle\vee\bowtie_{\theta'}=\langle c',c_1,c'_t\rangle\}}$(
  $c'_t$.allInstances()->select(other |
    $isNew$(other) and $matching_{c'}$(other)->isEmpty()
  )->forAll(other |
    $mapsTo^{\theta}_{\langle c'_t, c_t \rangle}$(other, self) implies
    $\mathbf{AND}_{\langle r_t, r'_t \rangle \in \{\langle r_t, r'_t \rangle \in \textsc{Ref}^*_{c_t} \times \textsc{Ref}^*_{c'_t} \mid a \sim_{\bowtie} r_t \wedge a \sim_{\bowtie} r'_t, a \in \textsc{Att}^*_{c_1}\}}$(self.$r_t$ =other.$r'_t$)
  )
)
```

5.2. Translation of Deleted Target Class Instances

There are different ways to translate a deleted target class instance, that was created by a join expression. In this section we want to evaluate the possible translation strategies for deleted target class instances and check how the constraints must be adapted for each strategy.

5.2.1. Deletion Strategies

A join target class instance can have a left and a right source class instance. If one of these source class instances would not exist, then the target class instance would not exist, if the join is not an outer join. So there are three possible delete strategies to satisfy the PutGet-Property:

Delete left and right Delete both, the left and right source class instance.

Delete only left (right) Delete only the left (right) source class instance.

We enforce a consistent deletion by updating the class instance meta functions in the following way:

```
$Instances'_{c_1}$ := $Instances_{c_1}$->exclude(left | $toDeleteLeft$(left))
$Instances'_{c_2}$ := $Instances_{c_2}$->exclude(left | $toDeleteRight$(right))
```

where *toDeleteLeft* respectively *toDeleteRight* evaluated to *true*, if the target class was deleted in the target model. To adopt the behavior for a different translation strategy, just these meta functions must be adopted.

5.2.1.1. Delete Left and Right

For the delete left and right strategy the meta function evaluates to *true* for both source class instances, if the target class instance was deleted.

```
$toDeleteLeft(v_1)$ := (let n = $c_{\bowtie}$.allInstances()->select(n| j.left = $v_1$)->asOrderedSet()
  in j->notEmpty() and j->first().target.oclIsUndefined()
)
$toDeleteRight(v_2)$ := (let n = $c_{\bowtie}$.allInstances()->select(n| j.right = $v_2$)->asOrderedSet()
  in j->notEmpty() and j->first().target.oclIsUndefined()
)
```

5.2.1.2. Delete Left (Right)

For the delete left strategy the meta function evaluates only to *true* for the left source class instance, if a target class instance was deleted.

```
toDeleteLeft($v_1$) := (let n = $c_{\bowtie}$.allInstances()->select(n| j.left = $v_1$)->asOrderedSet()
  in j->notEmpty() and j->first().target.oclIsUndefined()
)
toDeleteRight($v_2$) = false
```

5.3. ModelJoin Expressions for Source Attribute Updates

For the view definition keep calculated attribute and keep aggregate expressions can be used to define the query function for attribute values. If these attribute values are changed, the changes can not easily be propagated back to the source models, because the query function can be an arbitrary OCL expression and so can not be inverted in general. The generated constraints for these ModelJoin constructs can currently only ensure, that the source model elements or respectively the corresponding target model elements already have the correct values at translation time to produce the updated calculated attribute value. If the source model elements, the expression depends on, do not have corresponding target model elements the view can not be translated without updating the source model elements manually.

As an example consider the ModelJoin View Definition in Listing 5.4. The calculated attribute implementationName is derived from the source attribute name by striping the “If” prefix from the interface name. The ModelJoin view definition is based on the convention that the interface of a type is named as the name of the type with a prefixed “If”. For example the interface of the class Store is named StoreIf. In the current form changes to the implementationName attribute value can not be translated. If the attribute value would be changed, the OCL-constraints that ensure the GetPut-Property would be violated and a translation would not be possible without changing the value of the source attribute name of the corresponding source instance manually.

Because the function defined by the OCL expression used for the calculated attribute is not invertible in general and even if it would be, automatically computing the inverse is a hard problem on its own, we do not want not to try to invert the function automatically. Instead we want to allow the author of the ModelJoin view definition to supply the OCL expressions, that should be used at translation to update the source attributes. We introduce a new *source attribute update* ModelJoin operator:

Definition 41 (Source attribute update). Let $c \in \textsc{Class}$ be a source, $c_t \in \textsc{Class}$ be a target class with $c \sim_{\bowtie} c_t$, $a_t : t_c \to t \in \textsc{Att}^*_c$ be an attribute of c and $\phi : \mathsf{Expr}_{t_{c_t}} \to \mathsf{Expr}_\mathsf{t}$ a function. The source attribute update operator is defined as:

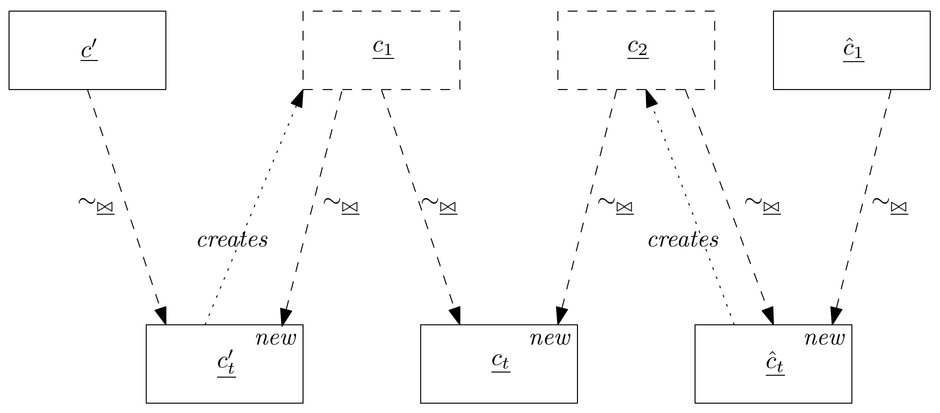


Figure 5.6.: If the source class c_1 is involved in two join expressions $\bowtie_\theta = \langle c_1, c_2, c_t\rangle$ and $\bowtie_\theta = \langle c_1, c_2', c_t'\rangle$, the creation of a new target instance $\underline{c}_t'$ could lead to the creation of a new source class instance $\underline{c_1}$ that forms a mapping pair with class instance $\underline{c}_2$ that are created by the translation of the target class instance $\underline{\hat{c}}_t$ of c_t .

```
theta join classifiers.Interface with uml.Interface
where "classifiers.Interface.name = uml.Interface.name" as jointarget.Interface {
    keep calculated attribute commons.NamedElement.name.substring(
        1, commons.NamedElement.name.size() - 2
    ) as implementationName : String
}
```

Listing 5.4: ModelJoin view definition with calculated attribute expression.

$$\delta^{-1}{}_{\phi} = \langle c_t, a\rangle \in \textsc{Class} \times \textsc{Att}_c^*$$

where the new attribute value of a for after the translation is defined by the function ϕ:

$$\forall \underline{c} \in \sigma_{\textsc{Class}}(c) \forall \underline{c}_t \in \sigma_{\textsc{Class}}(c_t)(\underline{c} \sim_{\underline{\bowtie}} \underline{c}_t \Rightarrow \sigma_{\textsc{Att}}(a)(\underline{c}) = \underline{\phi}(\underline{c}_t))$$

A proposal for a concrete syntax can be found in the example of Listing 5.5.

It is to note, that a source attribute update expression is may not well-defined. A source class instance $\underline{c}$ can map to two different target class instances $\underline{c}_t, \underline{c}_t'$ and the update expression may evaluate two different values depended on the given target instance: $\phi(\underline{c}_t) \neq \phi(\underline{c}_t')$. For a given model it can be easily checked if the source attribute updates expression is well defined, by using the mapping relation and evaluating ϕ for all target class instances that map to the same source class instance and comparing the result. So an error can be thrown, if there are conflicts for a given model. In general we leave it up to the ModelHoin view author to supply a well defined update expression.

Next we want to integrate the update expression into our OCL constraints by extending the definition of the corresponding meta variables;

```
theta join classifiers.Interface with uml.Interface
where "classifiers.Interface.name = uml.Interface.name" as jointarget.Interface {
    keep calculated attribute commons.NamedElement.name.substring(
        1, commons.NamedElement.name.size() - 2
    ) as implementationName : String
    source attribute commons.NamedElement.name updates to
        jointarget.Interface.implementationName.concat('If')
}
```

Listing 5.5: ModelJoin view definition from Listing 5.4 extended by a source attribute update expression

Definition 42 (Extension for source attribute update)**.** Let $c \in \textsc{Class}$ be a source, $c_t \in \textsc{Class}$ be a target class with $c \sim_{\bowtie} c_t$, $a_t : t_c \to t \in \textsc{Att}^*_c$ be an attribute of c and $\phi : \mathsf{Expr}_{t_{c_t}} \to \mathsf{Expr}_{\mathsf{t}}$ be an OCL expression and $\delta^{-1}{}_{\phi} = \langle c_t, a \rangle$ be a source attribute update expression.

If c_t is created by a join expression and c is the left join source class, then the following meta variable extension is used:

```
$var'_a(v)$ := (**let** j = $c_{\bowtie}$.allInstances()->select(j| j.left = $v$)->asOrderedSet()
  **in** **if** j->notEmpty() **and not** j->first().target.oclIsUndefined() **then**
    $\phi$(j->first().target)
  **else**
    $var_a(v)$
  **endif**
)
```

If c_t is created by a keep reference expression, then the following meta variable extension is used:

```
$var'_a(v)$ := (**let** k = $c_{\kappa_{\textsc{Ref}}}$.allInstances()->select(k| k.source = $v$)->asOrderedSet()
  **in** **if** k->notEmpty() **and not** k->first().target.oclIsUndefined() **then**
    $\phi$(k->first().target)
  **else**
    $var_a(v)$
  **endif**
)
```

In addition the following target meta variable can be defined:

$$var^{c_t}_a(v) := \phi(v)$$

A source attribute update expression solves one of our previous problems: In section 4.3.2 we have seen that not all OCL expressions can be rewritten for new target class instances, because some target meta variables may not be defined. This limits the creation of new target class instances, because some attributes or references are missing and cannot be derived automatically. The extended ModelJoin definition for Listing 4.3 is given in Listing 5.7.

```
theta join classifiers.Interface with uml.Interface
where "classifiers.Interface.name = uml.Interface.name" as jointarget.Interface {
    keep attributes commons.NamedElement.name as interfaceName
    keep calculated attribute commons.NamedElement.name.substring(
        1, commons.NamedElement.name.size() - 2
    ) as implementationName
    source attribute uml.NamedElement.name updates to jointarget.Interface.interfaceName
}
```

Listing 5.6: Extended version of the ModelJoin definition in Listing 4.3 with a update to expression.

Further we want to introduce a variant of the source attribute update expression for default attribute value for new created source class instances at update translation. We have discussed in chapter 4.5.4, that some source attributes may be undefined for new source class instances created by update translations. To be able to supply default values for these new instances, we introduce the *source attribute default* ModelJoin operator. In this variation the given formula ϕ is only used for the attribute value of new source class instances. If the source class instance does already exist, the value of the source attribute is left untouched. We propose a concrete syntax for this operator in Listing 5.7.

```
source attribute uml.NamedElement.name defaults to jointarget.Interface.interfaceName
```

Listing 5.7: Default source attribute value expression.

6. Automatic Fixes for Untranslatable Views

Executing an update operation on the target model can lead to an untranslatable target model, because certain constraints are violated. Further updates are required to restore the translatability of the updated target model. Consider the following example: An update operation creates a new target class instance. The translation of the new instance would lead to further new target classes, which are not present in the view. The corresponding constraints are violated. These missing target class instances, could be created automatically by detecting the problem and creating the missing target class instance before translation. Such an automatic 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. The proposed automatic fix should not undo the update operation, instead it should apply a minimal set of necessary changes to reflect the update in a translatable target model. In the example above, the automatic fix should not delete the new target class instance, it should create the missing target class instance.

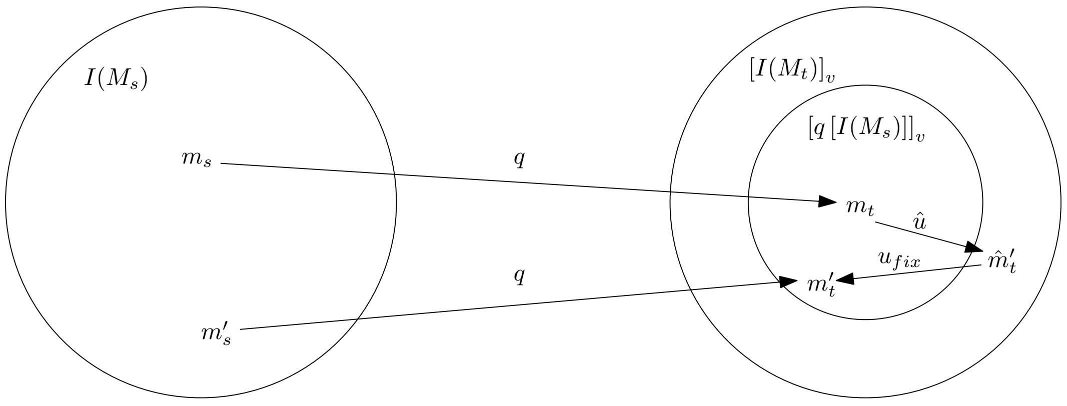


Figure 6.1.: The update operation $\hat{u}$ leads to an untranslatable target model $\hat{m}_t'$. But there may be an update operation u_{fix} that can be automatically derived and that leads to a translatable target model m_t' again.

More precisely we want to claim the following properties for an automatic fix operation:

Definition 43 (Valid fix)**.** A update operation $u_{fix} : I(M_t) \to I(M_t)$ is a valid fix for a update operation $u : I(M_t) \to I(M_t)$, if the following two properties do hold:

1. The fixed target model is translatable: $\forall m_t \in I(M_t)(u_{fix}(u(m_t))$ *is translatable*)

2. The fix reflects the update operation, this means that the attributes and links changes by the update operation are preserved by the fix operation:

$$\begin{aligned}
&\forall c \in \textsc{Class}\ \forall \underline{c} \in \sigma_{\textsc{Class}}^{u(m_t)}(c)(\\
&\quad \forall a \in \textsc{Att}_c^*(\sigma_{\textsc{Att}}^{u(m_t)}(a)(\underline{c}) \neq \sigma_{\textsc{Att}}^{m_t}(a)(\underline{c}) \Rightarrow \sigma_{\textsc{Att}}^{u_{fix}(u(m_t))}(a)(\underline{c}) = \sigma_{\textsc{Att}}^{u(m_t)}(a)(\underline{c})) \\
&\wedge \forall r \in \textsc{Ref}_c^*(L^{u(m_t)}(r)(\underline{c}) \neq L^{m_t}(r)(\underline{c}) \Rightarrow L^{u_{fix}(u(m_t))}(r)(\underline{c}) = L^{u(m_t)}(r)(\underline{c})))
\end{aligned}$$

An automatic fix can be interpreted as a relaxation of the OCL-constraints. We replace an OCL constraint with a weaker one. The weaker oen is a precondition for the algorithm executing the fix. A target model, which satisfies only the weaker constraint, can then be fixed by the algrthm to satisfies the original stronger constraint.

Not for all update operation such a fix may exist. Gruschko et al. [12] introduce a classification for changes to Ecore-based metamodels into three classes, Burger and Gruschko [15] adapted these for MOF-based metamodels. We adapt them further for updates to the target model:

- A *non-breaking* update leads to a translatable target model and does not need any fixes.
- For *breaking and resolvable* updates, an algorithm can be defined, that fixes the target model, so that it is translatable after the fix.
- A *breaking and non resolvable* update can not be fixed automatically and needs manual interaction.

6.1. Automatic Creation of Missing 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. We enforce this by the OCL constraints allowing only the translation of a target model, if the required target class instances exist. However, the missing target class instances could be created automatically at update translation, because these can be derived from the corresponding source class instances.

For a theta join we propose Algorithm 2 for this purpose. In the procedure createMissingJoinTargets we check for each new target class instance, if the translation of the instance would lead to new left or right source class instances. This is the case if there are no left or right source class instances, which map to the new instance (the set *matchingLeft*(target) respectively *matchingRight*(target) is empty). If this is the case, we check in the procedure createMissingJoinTargetsForNewLeft respectively createMissingJoinTargetsForNewRight, if all target class instances for the new source class exist.

In CREATEMISSINGJOINTARGETSFORNEWLEFT we first check for each existing right source class instance, for which the join condition with the new left source class instance holds, if there is a new target class 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 class instances could form new join pairs with the new left source class, also new created right source class instances could be join partners. Therefore we collect all new target class instances, for which the join condition holds for its right source class in matchingRightTarget.

For each instance in matchingRightTarget we check if a target class 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 for the case, if a new right source class is created by the translation.

The algorithm is directly derived from the OCL constraints given in Section 5.1.2.4. The newTargetInstancesLeft, newTargetInstancesLeftNew and newTargetInstancesRight, newTargetInstancesRightNew invariants from the Match left or right strategy in Section 5.1.2.4 can be relaxed to:

```
context $c_t$
inv newTargetInstancesLeft:
$isNew$(self) implies if $matchingRight$(self)->isEmpty() then
  $matchingLeft$(self)->forAll(left |
    let mappingTarget = $c_t$.allInstances->select(target | $isNew$(target))->select(target |
      $mapsTo^{\theta}_{\langle c_1, c_t \rangle}$(left, target) and $mapsTo^{c_t,\theta}_{\langle c_2, c_t \rangle}$(self, target)
    )
    in (mappingTarget->size() = 1 and $matchingLeft$(self)->forAll(otherLeft |
      $mapsTo^{\theta}_{\langle c_1, c_t \rangle}$(otherLeft, mappingTarget) implies otherLeft = left
    )) or mappingTarget->size() = 0
  )
else
  $matchingLeft$(self)->isEmpty()
endif
inv newTargetInstancesLeftNew:
$isNew$(self) and $matchingLeft$(self)->isEmpty() implies
$c_t$.allInstances->select(right | $isNew$(right) and $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self}, \text{right}), \text{self}, \text{right}}$)->forAll(right |
  let mappingTarget = $c_t$.allInstances->select(target | $isNew$(target))->select(target |
    $mapsTo^{c_t,\theta}_{\langle c_1, c_t \rangle}$(self, target) and $mapsTo^{c_t,\theta}_{\langle c_2, c_t \rangle}$(right, target)
  )
  in (mappingTarget->size() = 1 and $c_t$.allInstances->select(otherRight |
    $isNew$(otherRight) and $var^{c_1 \to c_t, c_2 \to c_t}_{\theta(\text{self}, \text{otherRight}), \text{self}, \text{otherRight}}$
  )->forAll(otherRight |
    $mapsTo^{c_t,\theta}_{\langle c_2, c_t \rangle}$(otherRight, mappingTarget) implies otherRight = right
  )) or mappingTarget->size() = 0
)
```

The invariants for the right source class, namely newTargetInstancesRight and newTargetInstancesRightNew can be relaxed analogically.

We show that the original constraints hold after the execution of createMissingJoinTargets, if the relaxed constraints did hold and the mapping of the missing target class instances is uniquely determined. Suppose that the relaxed constraints from above hold. We first check the not relaxed version of newTargetInstancesLeft after the execution of the algorithm. Let $\underline{c}_t$ be a arbitrary target class instance for which both expressions: *isNew*(self) and *matchingRight*(self)->isEmpty() evaluate to true. For all left source class instances $\underline{c}_1$ in the set *matchingLeft*(self) either mappingTarget->size() = 1 or mappingTarget->size() = 0 holds, before the execution. At execution of Algorithm 2 the procedure createMissingJoinTargetsForNewRight is called for the instance $\underline{c}_t$. In this procedure for all left in the set *matchingLeft*(self) it is checked if mappingTarget->isEmpty() and if this is the case, a new target class instance is created, else no actions is performed. The definition of mappingTarget is the same as in the relaxed and not relaxed constraint newTargetInstancesLeft. Therefore in the case of mappingTarget->size() = 1 no action is performed. In the case of mappingTarget->size() = 0 the new target class instance is created with the source attribute values and links of the left source class $\underline{c}_1$ and the corresponding right source attribute values and links of $\underline{c}_t$, resulting in a target class instance $\underline{c}'_t$ for which $\underline{\mathit{mapsTo}}^{\sigma}_{\langle c_1,c_t\rangle}(\underline{c}_1,\underline{c}'_t) = \mathit{true}$ and $\underline{\mathit{mapsTo}}^{c_t,\sigma}_{\langle c_2,c_t\rangle}(\underline{c}_t,\underline{c}'_t) = \mathit{true}$. After the creation the set mappingTarget for $\underline{c}_1$ contains exactly the instance $\underline{c}'_t$, so mappingTarget->size() = 1. According to the premise the left mapping of $\underline{c}'_t$ is unique, this means there exists no other left source class instances that map to $\underline{c}'_t$. So the not relaxed version of newTargetInstancesLeft is satisfied.

For the relaxed version of newTargetInstancesLeftNew a similar argument can be given: If mappingTarget->isEmpty(), then the missing target class instance is created, that satisfies the mapping relation. After the execution of Algorithm 2 the set mappingTarget exactly contains the new created instance and because the mapping is uniquely determined, the not relaxed version of newTargetInstancesLeftNew is satisfied.

For newTargetInstancesRight and newTargetInstancesRightNew the argumentation can be given symmetrically.

6.2. Automatic Calculated Values of Derived Model Elements

Defining a view has the advantage, that certain attributes and references are automatically derived from the source model. However the PutGet-Property requires that all model elements, including calculated attributes or aggregations, are equal to the ones calculated by the query function after the translation. So the derived elements have to be adopted manually before the translation.

Instead we could make the derived model elements non-editable for the user. The values for the derived elements, could be calculated automatically on the fly at the update of the target model. This can be done, because the derived elements depend on certain elements in the source model. These source model elements can have in turn corresponding target model elements, which can be used as data source.

Whenever an target model element is updated, the depended calculated attributes can be updated. Let $\delta_\phi = \langle c_1, c_2, a_t\rangle$ be a calculate attribute expression with the OCL expression

Algorithm 2 Automatically create missing target class instances $\bowtie_\theta = \langle c_1, c_2, c_t \rangle$

```
1: procedure CREATEMISSINGJOINTARGETS
2:     for target ∈ $c_t$.allInstances() do
3:         if isNew(target) then
4:             if matchingLeft(target)->isEmpty() then
5:                 ▷ On Update translation a new left source class would be created,
                     so create the missing target class instances for this new left source
                     class instance
6:                 left ← target
7:                 CREATEMISSINGJOINTARGETSFORNEWLEFT(left)
8:             if matchingRight(target)->isEmpty() then
9:                 ▷ On Update translation a new right source class would be created,
                     so create the missing target class instances for this new right source
                     class instance
10:                right ← target
11:                CREATEMISSINGJOINTARGETSFORNEWRIGHT(right)
```

$\phi : \mathsf{Expr}_{t_{c_1}} \times \mathsf{Expr}_{t_{c_2}} \rightarrow \mathsf{Expr}_{t}$. The target attribute a_t can be automatically updated by Algorithm 5. The algorithm uses the normal rewrite and the rewrite for new target classes for ϕ to update the value of a_t.

6.3. Automatic Propagation of Updated Attribute Values

If an attribute value in the view is changed, there can be other target model elements that depend on the same source attribute and must therefore be updated as well. These additional updates could be proposed to the user, so that these must not be done manually, before the translation. The constraints, that enforce that attributes are changed in a consistent way have the form $v_1.a_t = var_a(v_2)$. If such a constraint is violated, either the attribute value of a was changed and the meta variable has the old value or the attribute referenced from the meta variable was changed and the value of the attribute must be updated. If both changed, it is unclear which change is the intended. The set of changed (updated) attributes $A_{changed}$ can be determined by comparing the source attribute with the target attributes or by the knowledge about the update operation.

Algorithm 6 can be used to propagate the changes. The algorithm runs as long as attribute gets changes or it detects conflicting changes. For each OCL constraint it extracts all comparisons, that have the form $v_1.a_t = var_a(v_2)$ or $v_1.a_t = var_a^{c_t}(v_2)$ where v_1, v_2 are variables and a_t, a are attributes. If the value of the attribute and the meta variable are different, then it is checked, which one was changed, updates the unchanged attribute and adds the attribute to the set of changed attributes.

Algorithm 3 Automatically create missing target class instances for a new left source class instance

```
 1: procedure createMissingJoinTargetsForNewLeft(left : $c_t$)
 2:     ▷ For each existing right class instance for which the join expression holds
          a new target class instance must exist.
 3:     for right ∈ matchingRight(target) do
 4:         mappingTargets ← $c_t$.allInstances->select(target | isNew(target))
 5:            ->select(target |
 6:               $mapsTo^{c_t,\theta}_{\langle c_1,c_t\rangle}$(left, target) and $mapsTo^{\theta}_{\langle c_2,c_t\rangle}$(right, target)
 7:            )
 8:         if mappingTargets->isEmpty() then
 9:             $V_1 \leftarrow \{v_a \mid a \in \textsc{Att}^*_{c_1},\ v_a = var^{c_t}_a(\text{left})\}$
10:             $V_2 \leftarrow \{v_a \mid a \in \textsc{Att}^*_{c_1},\ v_a = var_a(\text{right})\}$
11:             $L_1 \leftarrow \{l_r \mid r = \langle c_1, \hat{c}\rangle \in \textsc{Ref},\ l_r = var^{c_t}_r(\text{left})\}$
12:             $L_2 \leftarrow \{l_r \mid r = \langle c_2, \hat{c}\rangle \in \textsc{Ref},\ l_r = var_r(\text{right})\}$
13:             create new target class instance with source attributes $V_1$, $V_2$
                and links $L_1$, $L_2$ according to the definition of $c_t$.
14:     ▷ For each new right class instance that would be created by update translation
          and for which the join expression expression holds a new target class
          instance must exist.
15:     matchingRightTarget ← $c_t$.allInstances->select(right |
16:          isNew(right) and $var^{c_1\to c_t, c_2\to c_t}_{\theta(\text{left},\text{right}),\text{left},\text{right}}$
17:     )
18:     for right ∈ matchingRightTarget do
19:         mappingTargets ← $c_t$.allInstances->select(target | isNew(target))
20:            ->select(target |
21:               $mapsTo^{c_t,\theta}_{\langle c_1,c_t\rangle}$(left, target) and $mapsTo^{c_t,\theta}_{\langle c_2,c_t\rangle}$(right, target)
22:            )
23:         if mappingTargets->isEmpty() then
24:             $V_1 \leftarrow \{v_a \mid a \in \textsc{Att}^*_{c_1},\ v_a = var^{c_t}_a(\text{left})\}$
25:             $V_2 \leftarrow \{v_a \mid a \in \textsc{Att}^*_{c_1},\ v_a = var^{c_t}_a(\text{right})\}$
26:             $L_1 \leftarrow \{l_r \mid r = \langle c_1, \hat{c}\rangle \in \textsc{Ref},\ l_r = var^{c_t}_r(\text{left})\}$
27:             $L_2 \leftarrow \{l_r \mid r = \langle c_2, \hat{c}\rangle \in \textsc{Ref},\ l_r = var^{c_t}_r(\text{right})\}$
28:             create new target class instance with source attributes $V_1$, $V_2$
                and links $L_1$, $L_2$ according to the definition of $c_t$.
```

Algorithm 4 Automatically create missing target class instances for a new right source class instances

```
 1: procedure createMissingJoinTargetsForNewRight(right : $c_t$)
 2:     Analogous to Algorithm 3, by exchanging “left” with “right” and vise versa.
```

Algorithm 5 Automatically updates the calculated attribute $\delta_\phi = \langle c_1, c_2, a_t \rangle$

```
1: procedure UPDATECALCULATEDATTRIBUTE$_{a_t}$
2:     for target ∈ $c_t$.allInstances()->select(target| not isNew(target)) do
3:         join ← $c_\bowtie$.allInstances()->find(j | j.target = target)
4:         target.$a_t$ ← $var_{\phi(\text{join.left,join.right})}$
5:     for target ∈ $c_t$.allInstances()->select(target| isNew(target)) do
6:         target.$a_t$ ← let target2 = target in $var^{c_1 \to c_t, c_2 \to c_t}_{\phi(\text{target,target2}),\text{target,target2}}$
```

Algorithm 6 Automatically propagation updated attribute values

```
1: procedure PROPAGATEATTRIBUTEVALUES($A_{changed}$)
2:     repeat
3:         attributeChanged ← false
4:         for constraint ∈ OCL-Constraints do
5:             for $v_1.a_t = var_a(v_2)$ ∈ constraint or $v_1.a_t = var^{c_t}_a(v_2)$ ∈ constraint do
6:                 if $v_1.a_t$ <> $var_a(v_2)$ or $v_1.a_t$ <> $var^{c_t}_a(v_2)$ then
7:                     $v'_2.a'_t$ ← target attribute usage in $var_a(v_2)$ or $var^{c_t}_a(v_2)$
8:                     if $\langle v_1, a_t \rangle \in A_{changed}$ and $\langle v'_2, a'_t \rangle \notin A_{changed}$ then
9:                         $v_1.a_t \leftarrow var_a(v_2)$
10:                        $A_{changed} \leftarrow A_{changed} \cup \{\langle v_1, a_t \rangle\}$
11:                    if $\langle v_1, a_t \rangle \notin A_{changed}$ and $\langle v'_2, a'_t \rangle \in A_{changed}$ then
12:                        $v'_2.a'_t \leftarrow v_1.a_t$
13:                        $A_{changed} \leftarrow A_{changed} \cup \{\langle v'_2, a'_t \rangle\}$
14:                    if $\langle v_1, a_t \rangle \in A_{changed}$ and $\langle v'_2, a'_t \rangle \in A_{changed}$ then
15:                        return Error(“Conflicting attribute changes”)
16:                    attributeChanged ← true
17:    until not attributeChanged
```

7. Evaluation

In this section the findings from the previous chapters will be evaluated. We first summarize under which conditions atomic update operations are translatable in general. Then we evaluate the translatability of updates in two case study examples.

7.1. Translatability of Updated Model Elements in General

It would be desired, that all atomic update operation on the target model can be translated. However this cannot be fully archived, because the PutGet-Property implies certain constraints for the target model $m_t \in I(M_t)$, so that $m_t \in q\,[I(M_s)]$. Therefore we evaluate for each atomic update operation in which cases a translation is not possible and how the proposed approach limits the translatability of updates. Further we discuss, if these limitations are useful or could be avoided if another approach would have been chosen. We use atomic update operation for this part of the evaluation, because common sequences of update operation can be hardly found without a concrete metamodel example. We would have to analyze a large set of concrete metamodels to find those common update patterns, which is beyond the scope of this thesis.

7.1.1. Translatability of a Updated Attribute Value

First we consider an attribute update operation $\textsc{update}_{\textsc{Att}(a_t)}(\underline{c}_t, v)$ for a target attribute $a_t : t_{c_t} \rightarrow t \in \textsc{Att}^*_{c_t}$ of the target model class instance $\underline{c}_t \in \sigma_{\textsc{Class}}(c_t)$ with a value v of type t. In our approach the changed attribute value can be translated, if all of the following requirements hold for the target model:

C1.1 If a source attribute $a \in \textsc{Att}^*_c$ with $a \sim_{\bowtie} a_t$ is involved in a join condition θ of a join $\bowtie_\theta = \langle c_1, c_2, c'_t \rangle$, for all other elements $e \in \textsc{Att} \cup \textsc{Ref}$ used in the join condition θ, the value of the corresponding target model elements $e_t \in \textsc{Att} \cup \textsc{Ref}$ with $e \sim_{\bowtie} e_t$ must be updated, so that the value of the join condition θ does not change:

$$\forall \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1) \forall \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2)$$
$$\left(\underline{\theta(\text{left, right})}_{\text{left,right}}(\underline{c}_1, \underline{c}_2) = \underline{var}_{\theta(\text{left,right}),\text{left,right}}(\underline{c}_1, \underline{c}_2)\right)$$

For example consider the update operation

$$\textsc{update}_{\textsc{Att}(\text{jointarget.Interface.javaName})}(\underline{c}_t, \text{“StoreIf”})$$

for a target class instance $\underline{c}_t \in \sigma_{\textsc{Class}}(\text{jointarget.Interface})$ of the target model defined by the ModelJoin view definition in Listing 7.1. Let $\underline{c}_1 \in \sigma_{\textsc{Class}}(\text{classifiers.Interface})$

and $\underline{c}_2 \in \sigma_{\textsc{Class}}(\text{uml.Interface})$ be the source classes of $\underline{c}_t$. Then the join condition $\theta(v1, v2) = v1.\text{name} = v2.name$ holds for $\underline{c}_1$ and $\underline{c}_2$ before the translation. However it does not hold after translation, because

$$\underline{var}_{\text{commons.NamedElement.name}}(\underline{c}_1) = \text{“StoreIf”}$$

and $\underline{var}_{\text{uml.NamedElement.name}}(\underline{c}_2)$ has the old value $\sigma_{\textsc{Att}}(\text{uml.NamedElement.name})(\underline{c}_2)$. So the further update operation

$$\textsc{update}_{\textsc{Att}(\text{jointarget.Interface.umlName})}(\underline{c}_t, \text{“StoreIf”})$$

is necessary to update $\underline{var}_{\text{uml.NamedElement.name}}(\underline{c}_2)$.

C1.2 If a source attribute $a \in \textsc{Att}_c^*$ with $a \sim_{\bowtie} a_t$ is used in a calculated attribute expression $\delta_\phi = \langle c_1, c_2, a'_t \rangle$, the value of the calculated attribute must be updated as well, so that:

$$\forall \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1) \forall \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2) \forall \underline{c}'_t \in \sigma_{\textsc{Class}}(c'_t)$$
$$(\underline{c}_1 \underline{\sim_{\bowtie}} \underline{c}'_t \wedge \underline{c}_2 \underline{\sim_{\bowtie}} \underline{c}'_t \to \sigma_{\textsc{Att}}(a'_t)(\underline{c}'_t) = \underline{var}_{\phi(\text{left,right})}(\underline{c}_1, \underline{c}_2))$$

This can be done manually or automatically using the algorithm given in Section 6.2. For example consider the update operation

$$\textsc{update}_{\textsc{Att}(\text{jointarget.Interface.javaName})}(\underline{c}_t, \text{“StoreIf”})$$

for a target class instance $\underline{c}_t \in \sigma_{\textsc{Class}}(\text{jointarget.Interface})$ of the target model defined by the ModelJoin view definition in Listing 7.1. The corresponding source attribute commons.NamedElement.name is used in the keep calculated attribute expression for implementationName, so the further update operation

$$\textsc{update}_{\textsc{Att}(\text{jointarget.Interface.implementationName})}(\underline{c}_t, \text{“Store”})$$

is necessary.

C1.3 If a source attribute $a \in \textsc{Att}_c^*$ with $a \sim_{\bowtie} a_t$ is mapped to multiple target attributes $A = \{a'_t \in \textsc{Att} \mid a \sim_{\bowtie} a'_t\}$, the value of all these target attribute values must be updated as well, so that:

$$\forall a'_t : t_{c'_t} \to t \in A \forall \underline{c}'_t \in \sigma_{\textsc{Class}}(c'_t) \forall \underline{c}_t \in \sigma_{\textsc{Class}}(c_t) \forall c \in \textsc{Class}$$
$$(\forall \underline{c} \in \sigma_{\textsc{Class}}(c)(\underline{c} \underline{\sim_{\bowtie}} \underline{c}'_t \wedge \underline{c} \underline{\sim_{\bowtie}} \underline{c}_t) \to \sigma_{\textsc{Att}}(a'_t)(\underline{c}'_t) = \sigma_{\textsc{Att}}(a_t)(\underline{c}_t))$$

This can be done manually or using the algorithm given in Section 6.3. For example consider the update operation

$$\textsc{update}_{\textsc{Att}(\text{jointarget.Interface.javaName})}(\underline{c}_t, \text{“StoreIf”})$$

for a target class instance $\underline{c}_t \in \sigma_{\textsc{Class}}(\text{jointarget.Interface})$ of the target model defined by the ModelJoin view definition in Listing 7.1. The corresponding source attribute commons.NamedElement.name does also map to jointarget.Interface.javaNameAlis, so the further update operation

$$\textsc{update}_{\textsc{Att}(\text{jointarget.Interface.javaNameAlis})}(\underline{c}_t, \text{“StoreIf”})$$

is necessary.

```
theta join classifiers.Interface with uml.Interface
where "classifiers.Interface.name = uml.Interface.name" as jointarget.Interface {
    keep calculated attribute commons.NamedElement.name.substring(
        1, commons.NamedElement.name.size() - 2
    ) as implementationName : String
    keep attributes commons.NamedElement.name as javaName
    keep attributes commons.NamedElement.name as javaNameAlias
    keep attributes uml.NamedElement.name as umlName
    keep outgoing members.MemberContainer.members as type jointarget.Operation {
        keep attributes commons.NamedElement.name
    }
}
```

Listing 7.1: ModelJoin view definition with different attribute expression.

So without further actions not all attribute value changes can be translated directly.

Condition C1.1 would not be strictly necessary, if it was allowed to change the mapping of a join target class instance $\underline{c}_t$ with the attribute update, so that $\underline{c}_t$ has different source class instances after the translation than before. Like discussed in Section 4.4 this would change the identity, in terms of the mapping relation, of the target class instance $\underline{c}_t$ behind the scene. We argued, that this is not a desired behavior, because the target class instances in the target model represent their mapping sources and an update to a target class instance should therefore update the source instances and not change the mapping. Dropping this condition would not be possible with the proposed approach. The unchangeable trace model implies a fixed source class instance to target class instance mapping for existing source class instances. So the limitation is useful, if the mapping for existing instances should not be changed.

Condition C1.2, C1.3 would not be strictly necessary, if we did not want not consider the PutGet-Property. In an approach that does not fulfill the PutGet-Property, these condition could be dropped. While a single attribute update cannot conflict with other update, an update sequence with multiple attribute updates can have conflicts. Such a conflict occurs, if two target attributes $a_t, a_t' \in \textsc{Att}$, which map to the same source attribute $a \in \textsc{Att}$ get updated with different values. For such a conflict it is unclear how to translate the updated attribute values, because a could get either the value of a_t or a_t'. The PutGet-Property ensures, that a translation can be determined and so is very useful in the multiple update case. However in the case of the single attribute update the PutGet-Property could be relaxed to only the updated elements. Unchanged target model elements then need to be omitted at update translation. In the current approach an extra step is necessary: The user has to fix the target model before the translation. This can be done either manually or automatically using one of the proposed algorithms. That is a disadvantage, because this extra step is more work for the user. An advantage however is, that the user gets an explicit feedback over which attributes need to be updated as well. He or she then can check, if these changes are desired, before performing the translation. If the update algorithm can be applied, the additional work is minimal. In the case of single attribute updates, the algorithm can always be applied, because no conflict can occur.

7.1.2. Translatability of a Updated Link

Next we want to consider a create link operation $\textsc{create}_{\textsc{Ref}(r_t)}(\underline{c}_t, \underline{\hat{c}}_t)$ and a delete link operation $\textsc{delete}_{\textsc{Ref}(r_t)}(\underline{c}_t, \underline{\hat{c}}_t)$ for a reference $r_t \in \textsc{Ref}$ between two target classes $c_t, \hat{c}_t \in \textsc{Class}$. In our approach a removed or added link can be translated, if all of the following requirements do hold for the target model:

C2.1 If a source reference $r \in \textsc{Ref}$ with $r \sim_{\bowtie} r_t$ is involved in a join condition θ of a join $\bowtie_\theta = \langle c_1, c_2, c'_t \rangle$, for all other elements $e \in \textsc{Att} \cup \textsc{Ref}$ used in the join condition θ, the value of the corresponding target model elements $e_t \in \textsc{Att} \cup \textsc{Ref}$ with $e \sim_{\bowtie} e_t$ must be updated, so that the value of the join condition θ does not change:

$$\begin{aligned}&\forall \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1) \forall \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2)\\ &\quad (\underline{\theta(\text{left}, \text{right})}_{\text{left},\text{right}}(\underline{c}_1, \underline{c}_2) = \underline{var}_{\theta(\text{left},\text{right}),\text{left},\text{right}}(\underline{c}_1, \underline{c}_2))\end{aligned}$$

C2.2 If a source reference $r \in \textsc{Ref}$ with $r \sim_{\bowtie} r_t$ is used in a calculated attribute expression $\delta_\phi = \langle c_1, c_2, a'_t \rangle$, the value of the calculated attribute must be updated as well, so that:

$$\begin{aligned}&\forall \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1) \forall \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2) \forall \underline{c}'_t \in \sigma_{\textsc{Class}}(c'_t)\\ &\quad (\underline{c}_1 \sim_{\underline{\bowtie}} \underline{c}'_t \wedge \underline{c}_2 \sim_{\underline{\bowtie}} \underline{c}'_t \rightarrow \sigma_{\textsc{Att}}(a'_t)(\underline{c}'_t) = \underline{var}_{\phi(\text{left},\text{right})}(\underline{c}_1, \underline{c}_2))\end{aligned}$$

This can be done manually or using the given algorithm in Section 6.2.

C2.3 If a source reference $r \in \textsc{Ref}$ with $r \sim_{\bowtie} r_t$ is mapped to multiple target references $R = \{r'_t \in \textsc{Ref} \mid r \sim_{\bowtie} r'_t\}$, the links of all these target references must be updated as well, so that:

$$\begin{aligned}&\forall r'_t \in \{r'_t \in R \mid \mathit{associates}(r'_t) = \langle c'_t, \hat{c}_t \rangle\} \forall \underline{c}'_t \in \sigma_{\textsc{Class}}(c'_t) \forall \underline{c}_t \in \sigma_{\textsc{Class}}(c_t) \forall c \in \textsc{Class}\\ &\quad (\forall \underline{c} \in \sigma_{\textsc{Class}}(c)(\underline{c} \sim_{\underline{\bowtie}} \underline{c}'_t \wedge \underline{c} \sim_{\underline{\bowtie}} \underline{c}_t) \rightarrow (L(r'_t)(\underline{c}'_t) = L(r_t)(\underline{c}_t)))\end{aligned}$$

This can only be done manually.

C2.4 If a link is removed and the linked target class instance $\underline{\hat{c}}_t$ is not linked by another target reference or is a join target, then the $\underline{\hat{c}}_t$ must be deleted from the target model, so that:

$$\forall r_t \in \{r_t \in \textsc{Ref} \mid \mathit{associates}(r_t) = \langle c'_t, \hat{c}_t \rangle\} \forall \underline{c}'_t \in \sigma_{\textsc{Class}}(c'_t)(\underline{\hat{c}}_t \notin L(r_t) \rightarrow (\underline{\hat{c}}_t \notin \sigma_{\textsc{Class}}(\hat{c}))$$

For example consider the delete link operation

$$\textsc{delete}_{\textsc{Ref}(\text{members.MemberContainer.members})}(\underline{c}_t, \underline{\hat{c}}_t)$$

for the both target class instances $\underline{c}_t \in \sigma_{\textsc{Class}}(\text{jointarget.Interface})$ and $\underline{\hat{c}}_t \in \sigma_{\textsc{Class}}(\text{jointarget.Operation})$ of the target model defined by the ModelJoin view definition in Listing 7.1. The class instance $\underline{\hat{c}}_t$ would not exist without being linked in the reference members. So it must be delete with a further update operation:

$$\textsc{delete}_{\textsc{Class}(\text{jointarget.Operation})}(\underline{\hat{c}}_t)$$

C2.5 If a link is added, the linked target class instance $\underline{\hat{c}}_t$ must have a corresponding source class instance:

$$\forall \underline{\hat{c}}_t \in L(r_t) \exists \underline{\hat{c}} \in \sigma_{\textsc{Class}}(\hat{c})(\underline{\hat{c}} \sim_{\bowtie} \underline{\hat{c}}_t)$$

For example, the create reference operation

$$\textsc{create}_{\textsc{Ref}(\mathsf{members.MemberContainer.members})}(\underline{c}_t, \underline{\hat{c}}_t)$$

is not possible, if $\underline{c}_t \in \sigma_{\textsc{Class}}(\mathsf{jointarget.Interface})$ is an existing target class instance and $\underline{\hat{c}}_t \in \sigma_{\textsc{Class}}(\mathsf{jointarget.Operation})$ was newly created and so has not source class instance.

For Conditions C2.1 and C2.2 the same applies like in the cases C1.1 and C1.2.

For Condition C2.3 we have not defined an algorithm for an automatic fix. However such an algorithm may would be possible, because the case is very similar to the one for attributes.

Condition C2.4 would not be strictly necessary, if the PutGet-Property was not be considered. Let $\underline{\hat{c}}_t$ be a target class instance, that is only created by one or multiple keep reference expressions. In ModelJoin the instance $\underline{\hat{c}}_t$ only exists in the target model, if it is linked by a target reference r_t with $associates(r_t) = \langle c'_t, \hat{c}_t \rangle$. This property and the PutGet-Property forces the user to delete $\underline{\hat{c}}_t$, if it is not linked by any target reference r_t anymore. This has the disadvantage, that for the user it may not be clear, that the deletion of the target class instance $\underline{\hat{c}}_t$ does not lead to a deletion of the corresponding source class instance $\underline{\hat{c}}$ with $\underline{\hat{c}} \sim_{\bowtie} \underline{\hat{c}}_t$. Not forcing the user to delete the target class would lead to a violation of the PutGet-Property. However it would be more intuitive, if the target class instance $\underline{\hat{c}}_t$ was not needed to be deleted explicitly. A further disadvantage is, that the user can only remove the link, but cannot delete the source class instance $\underline{\hat{c}}$, because these cases cannot be distinguished in our approach.

Condition C2.5 limits the update operation significantly. New target class instances can not be added to existing references. This is the case, because of the Put-Equality of the corresponding meta variable of the reference. To satisfy the Put-Equality the meta variable var_r would need to contain the source instance $\underline{\hat{c}}$ of $\underline{\hat{c}}_t$. However this is not possible, if $\underline{\hat{c}}$ does not exist, yet. The condition would not be necessary to satisfy the PutGet-Property.

7.1.3. Translatability of a Class Instance Creation

In this section a create class instance operation $\textsc{create}_{\textsc{Class}(c_t)}(V)$, which created a new instance $\underline{c}_t$ of the target model class $c_t \in \textsc{Class}$ with the attribute values $V = \{v_a \in I(t) \mid a \in \textsc{Att}^*_{c_t}\}$ will be considered. In our approach a created target class instance can be translated, if all of the following requirements do hold for the target model:

C3.1 All model elements $E = \{e^1, \dots, e^n\}$ of the source classes $c \in \textsc{Class}$ with $c \sim_{\bowtie} c_t$ used in either a join condition θ or a keep calculated attribute expression ϕ must have corresponding target model elements:

$$\forall e^i \in E\, (\exists e_t \in \textsc{Att} \cup \textsc{Ref}(e^i \sim_{\bowtie} e_t)$$

Alternatively a source attribute update expression can be used instead of attributes in the target model (see chapter 5.2). Further for all join conditions and the formulas of all calculated attribute expression a rewrite that only depends on target model elements must be possible:

$$\begin{aligned}
&\forall \theta_{\bowtie} = \langle c_1, c_2, c_t\rangle\\
&\quad var^{c_1\to c_t}_{\theta(\text{left},\text{right}),\text{left}}, var^{c_2\to c_t}_{\theta(\text{left},\text{right}),\text{right}}, var^{c_1\to c_t,c_2\to c_t}_{\theta(\text{left},\text{right}),\text{left},\text{right}} \text{ are defined and well-typed}\\
&\forall \delta_{\phi} = \langle c_1, c_2, a_t\rangle\\
&\quad var^{c_1\to c_t,c_2\to c_t}_{\phi(\text{left},\text{right}),\text{left},\text{right}} \text{ is defined and well-typed}
\end{aligned}$$

For example in the ModelJoin view definition in Listing 7.1 all necessary target meta variables are defined. The set of model elements is

$$E = \{\text{commons.NamedElement.name}, \text{uml.NamedElement.name}\}$$

The attributes map to their corresponding target class elements: jointarget.Insterface.javaName and jointarget.Insterface.umlName.

C3.2 If one of the source classes $c \in \textsc{Class}$ with $c \sim_{\bowtie} c_t$ is referenced by a reference $r \in \textsc{Ref}$ with $associates(r) = \langle c', c\rangle$, all corresponding target references $R = \{r_t \in \textsc{Ref} \mid r \sim_{\bowtie} r_t\}$ must have the same target class:

$$\exists \hat{c}'_t \in \textsc{Class}\, \forall r_t \in \{r_t \in \textsc{Ref} \mid associates(r_t) = \langle c''_t, \hat{c}_t\rangle\}(associates(r) = \langle c''_t, \hat{c}'_t\rangle)$$

C3.3 If the translation of the new created target class instance $\underline{c}_t \in \sigma_{\textsc{Class}}(c_t)$ leads to new join pairs, the new target class instances must be created:

$$\begin{aligned}
\forall \theta_{\bowtie} =&\langle c_1, c_2, c_t\rangle\\
&(\nexists \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1)(\underline{mapsTo}^{\sigma}_{\langle c_1,c_t\rangle}(\underline{c}_1,\underline{c}_t) = \text{true}) \to\\
&\quad(\forall \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2)(\underline{var}^{c_1\to c_t}_{\theta(\text{left},\text{right}),\text{left}}(\underline{c}_t,\underline{c}_2) = \text{true} \to\\
&\quad\quad \exists! \underline{c}'_t \in \sigma_{\textsc{Class}}(c_t)(\underline{mapsTo}^{\sigma,c_t}_{\langle c_1,c_t\rangle}(\underline{c}_t,\underline{c}'_t) \wedge \underline{mapsTo}^{\sigma}_{\langle c_2,c_t\rangle}(\underline{c}_2,\underline{c}'_t))\\
&\quad\wedge(\forall \underline{c}''_t \in \sigma_{\textsc{Class}}(c_t)(\underline{var}^{c_1\to c_t,c_2\to c_t}_{\theta(\text{left},\text{right}),\text{left},\text{right}}(\underline{c}_t,\underline{c}''_t) = \text{true} \to\\
&\quad\quad \exists! \underline{c}'_t \in \sigma_{\textsc{Class}}(c_t)(\underline{mapsTo}^{\sigma,c_t}_{\langle c_1,c_t\rangle}(\underline{c}_t,\underline{c}'_t) \wedge \underline{mapsTo}^{\sigma,c_t}_{\langle c_2,c_t\rangle}(\underline{c}''_t,\underline{c}'_t)))\\
&\wedge(\nexists \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2)(\underline{mapsTo}^{\sigma}_{\langle c_2,c_t\rangle}(\underline{c}_2,\underline{c}_t) = \text{true}) \to\\
&\quad(\forall \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1)(\underline{var}^{c_2\to c_t}_{\theta(\text{left},\text{right}),\text{right}}(\underline{c}_1,\underline{c}_t) = \text{true} \to\\
&\quad\quad \exists! \underline{c}'_t \in \sigma_{\textsc{Class}}(c_t)(\underline{mapsTo}^{\sigma,c_t}_{\langle c_2,c_t\rangle}(\underline{c}_t,\underline{c}'_t) \wedge \underline{mapsTo}^{\sigma}_{\langle c_1,c_t\rangle}(\underline{c}_1,\underline{c}'_t))\\
&\quad\wedge(\forall \underline{c}''_t \in \sigma_{\textsc{Class}}(c_t)(\underline{var}^{c_1\to c_t,c_2\to c_t}_{\theta(\text{left},\text{right}),\text{left},\text{right}}(\underline{c}_t,\underline{c}''_t) = \text{true} \to\\
&\quad\quad \exists! \underline{c}'_t \in \sigma_{\textsc{Class}}(c_t)(\underline{mapsTo}^{\sigma,c_t}_{\langle c_2,c_t\rangle}(\underline{c}_t,\underline{c}'_t) \wedge \underline{mapsTo}^{\sigma,c_t}_{\langle c_1,c_t\rangle}(\underline{c}''_t,\underline{c}'_t)))
\end{aligned}$$

This can be done manually or using the algorithm given in Section 6.1. For example, recall the example in chapter 5.1.4.

C3.4 The new created target class instance $\underline{c}_t \in \sigma_{\textsc{Class}}(c_t)$ must map uniquely to either only one left, only one right or no source class instance:

$$\begin{aligned}
&\forall \theta_{\bowtie} = \langle c_1, c_2, c_t \rangle \\
&\quad (\exists! \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1)(\underline{mapsTo^{\sigma}_{\langle c_1, c_t \rangle}}(\underline{c}_1, \underline{c}_t)) \land \nexists \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2)(\underline{mapsTo^{\sigma}_{\langle c_2, c_t \rangle}}(\underline{c}_2, \underline{c}_t))) \\
&\quad \lor (\exists! \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2)(\underline{mapsTo^{\sigma}_{\langle c_2, c_t \rangle}}(\underline{c}_2, \underline{c}_t)) \land \nexists \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1)(\underline{mapsTo^{\sigma}_{\langle c_1, c_t \rangle}}(\underline{c}_1, \underline{c}_t))) \\
&\quad \lor (\nexists \underline{c}_1 \in \sigma_{\textsc{Class}}(c_1)(\underline{mapsTo^{\sigma}_{\langle c_1, c_t \rangle}}(\underline{c}_1, \underline{c}_t)) \land \nexists \underline{c}_2 \in \sigma_{\textsc{Class}}(c_2)(\underline{mapsTo^{\sigma}_{\langle c_2, c_t \rangle}}(\underline{c}_2, \underline{c}_t)))
\end{aligned}$$

For example, where this is not the case, recall the example in chapter 5.1.5.

Condition C3.1 could be relaxed using another approach: For example the missing values could be requested at translation time. Because we want to check the PutGet-Property before translation and the OCL constraints are created at view creation time the constraint is required in our approach. To infer the value of the join conditions θ and formula ϕ of keep calculated attribute expressions for the new source class instances $\underline{c} \in \sigma_{\textsc{Class}}(c)$, that may get created at the translation of the create class instance operation, the values must be present in the target model. Without these values, the constraints necessary to check the PutGet-Property could not be evaluated. The condition only depends on the view definition and not on the target model. Therefore the condition must be considered at view definition time. If the view was designed with C3.1 in mind, all needed values for the translation are present in the target model and so the PutGet-Property can be checked without getting further values from the user.

Condition C3.2 is not strictly necessary. Let $r \in \textsc{Ref}$ be a reference in the source model with $associates(r) = \langle c, \hat{c}_t \rangle$. In another approach it could be allowed to use the source class $\hat{c} \in \textsc{Class}$ in two different keep reference expressions $\kappa_{\textsc{Ref}} = \langle r, r_t \rangle, \kappa'_{\textsc{Ref}} = \langle r, r'_t \rangle$ with $associates(r_t) = \langle c_t, \hat{c}_t \rangle$, $associates(r'_t) = \langle c'_t, \hat{c}'_t \rangle$, which do not have the same target class: $\hat{c}_t \neq \hat{c}'_t$. In this case the source class $\hat{c}$ in would map to two different target classes $\hat{c}_t$ and $\hat{c}'_t$. This can lead to multiple target class instances $\underline{\hat{c}}_t \in \sigma_{\textsc{Class}}(\hat{c}_t), \underline{\hat{c}}_t \in \sigma_{\textsc{Class}}(\hat{c}'_t)$ of different classes that map to the same source class instance $\underline{\hat{c}} \in \sigma_{\textsc{Class}}(\hat{c})$. A user may not expect this behavior and the dependencies between the target references r_t and r'_t of different types may not be obvious from the target metamodel. However the condition limits the set of possible view definitions. To check if the two or more target references r_t, r'_t of different target type, can be translated back to the one source reference r, so that the PutGet-Property is satisfied, further constraints would be necessary. We would need to check for each linked target class instance $\underline{\hat{c}}_t \in L(r_t)(\underline{c}_t)$, if there is a target class instance $\underline{\hat{c}}'_t \in L(r'_t)(\underline{c}_t)$ linked in the other target reference that shares the same source class: $\underline{\hat{c}} \sim_{\underline{\bowtie}} \hat{c}_t$ and $\underline{\hat{c}} \sim_{\underline{\bowtie}} \hat{c}'_t$. For new created target class instances, this is difficult, because their source classes do not exist before translation. It may be possible to express these checks with OCL and extend our approach, but this needs to be evaluated further.

Condition C3.3 would not be strictly necessary, if the PutGet-Property is considered. The missing target class instances could be fully created at translation time or at the next query after translation. In our approach the additional step of creating the missing target class instances manually or by using the given algorithm is more work for the user. However the user has the ability to check the result, before doing the actual translation and may adapt the new created instance for further requirements.

Condition C3.4 is not strictly necessary. However, without this restriction there could be situations in which the new target class instance $\underline{c}_t$ of a join $\theta_{\bowtie} = \langle c_1, c_2, c_t \rangle$ would map to two different left source class instances $\underline{c}_1, \underline{c}'_1$. If the translation of the new created target class instance $\underline{c}_t$, creates a new instance of the right join source class $\underline{c}_2 \in I(c_2)$, it is unclear if $\underline{c}_t$ is the target class of the source instances $\underline{c}_1$ and $\underline{c}_2$ or the source instances $\underline{c}'_1$ and $\underline{c}_2$. There may are situations where the exact mapping does not matter. However in our approach a unique mapping is needed. This could be provided manually (see chapter 5.1.6).

7.1.4. Translatability of a Class Instance Deletion

Finally a delete class instance operation $\textsc{delete}_{\textsc{Class}(c_t)}(\underline{c}_t)$, which deletes the instance $\underline{c}_t$ of the target class $c_t \in \textsc{Class}$ will be considered. In our approach the operation can be translated, if all of the following requirements hold for the target model:

C4.1 If the translation of the deleted class instance leads to a deletion of a source class instance $\underline{c_t} \in \sigma_{\textsc{Class}}(c_t)$ with $\underline{c} \sim_{\bowtie} \underline{c}_t$, then all other target class instances $\underline{c}'_t \in \sigma_{\textsc{Class}}(c_t)$, which would not exist without the instances $\underline{c}$ must be deleted as well:

$$\forall c'_t \in \textsc{Class}(c \sim_{\bowtie} c'_t \rightarrow (\nexists \underline{c}'_t \in \sigma_{\textsc{Class}}(c'_t)(\underline{c} \sim_{\underline{\bowtie}} \underline{c}'_t))$$

With the exception that $\underline{c}$ is a source class of a outer join with the target class c'_t, where $\underline{c}'_t$ exists without $\underline{c}$.

Condition C4.1 would not be strictly necessary, if the PutGet-Property was not considered. The target class instances $\underline{c}'_t$, which would not exist in the target model after translation, could be deleted directly as part of the translation. This would have the advantage, that the user does not have to perform this additional step. However the user would not see the full effect before the translation was performed. In our approach an algorithm, that could delete the corresponding target class instances, could be defined, similar to the algorithm for missing target class instances.

7.2. Application in Case Study Examples

In this section we will evaluate, if the findings are applicable in concrete example cases. Since ModelJoin is not used in real practical applications yet, no real application can be used and general modeling examples are used instead. For the case study we have therefore chosen the Common Component Modelling Example (CoCoME) [34] and the Media Store example from the upcoming Palladio Book [58].

7.2.1. Common Component Modelling Example (CoCoME)

The CoCoME describes a trading system for a supermarket that handles sales. This includes the Cash Desk activities, like using a Bar Code Scanner or paying by credit card or cash as well as administrative tasks like ordering products or generating reports. The description of the trading system includes requirements, use case analysis, architectural component model with several views, system tests and test scenarios. In addition there is a Java implementation.

Using the Java Model Parser and Printer (JaMoPP) [33], the source code of the Java implementation can be converted from the text representation to a model representation and vice versa.

Hence we have two models describing the trading system: A UML model of the architecture and a Java metamodel instance of the implementation. With ModelJoin a UML view of the trading system can therefore be extended with implementation details using the model representation of the source code. Further more, translated updates cannot only update the UML models, but also the Java source code by using JaMoPP to keep the model and text representation in sync.

The findings of the thesis will be evaluated by analyzing a concrete ModelJoin view definition. It will be checked, if desired updates are possible and lead to the expected changes to the source models.

7.2.1.1. metamodels

EMFText Java Using JaMoPP the java source code can be converted to a EMFText Java model [24]. The model contains the abstract syntax tree of the java source with source code positions and comments. For each java source code file, a Compilation Unit class instance is created which contains in particular the classes and interface including their members defined in the java source file.

UML component diagram The UML component diagram [56] shows the components of the software system and which service these provide and require, including the interface used for communication.

7.2.1.2. Component Diagram and Java Source Code Example

For the evaluation we use the UML Inventory component diagram from [34]. The used EMF UML2 model is given in Figure 7.1. It shows the components of the Inventory forming the different layers of the trading system. The components are connected through provided and required interface connectors. A component is implemented as a Java package containing multiple classes, implementing the component behavior. While in the UML diagram only the interface names of the components services are given, the interface definitions in the java source codes do contain additional information, such as the method signatures. We assume the software engineer wants to create a view containing the interfaces with their method signatures and the components providing these interfaces and therefore creates the ModelJoin view definition in Listing 7.2.

7.2.1.3. Changing the Name of an Interface

First the effect of changing an attribute in the target model will be evaluated.

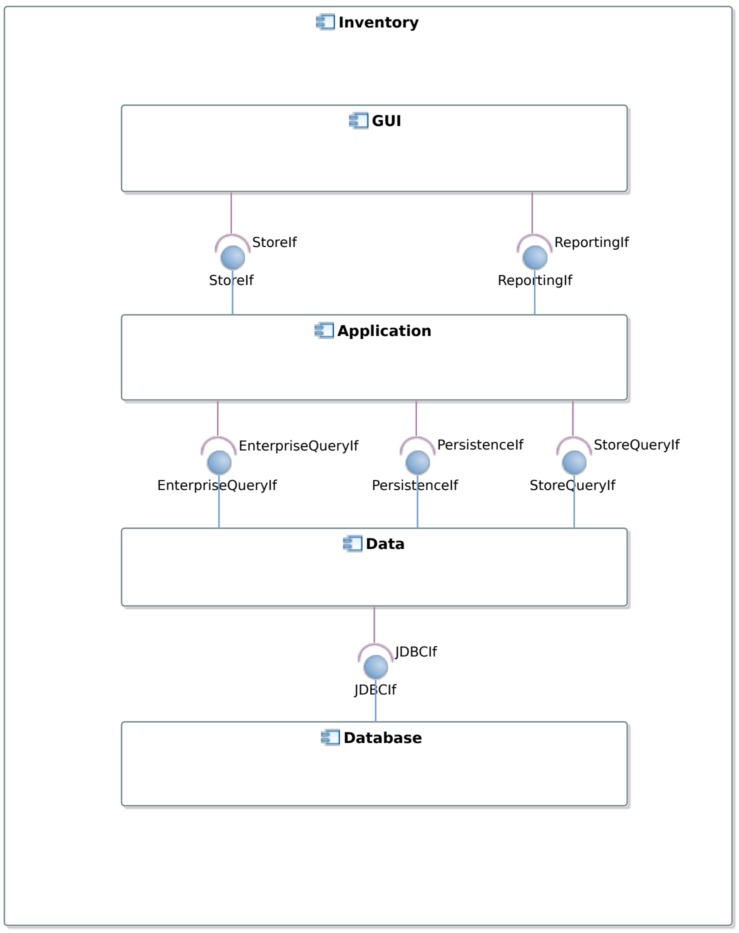


Figure 7.1.: The Inventory component of the CoCoME trading system with subcomponents and provided and required interfaces.

Task We assume the user wants to update the name of the jointarget.Interface class instance with the name “StoreIf” to “StoreInfoIf”. The user performs the update operations in Listing 7.3.

Expected result The source attributes uml.Interface.name, classifiers.Interface.name and classifier.Class.name of the corresponding source class instances gets updated.

Actual result All OCL constraints are fulfilled and on translation the source attributes get updated. The result meets the expectation.

Notes It is to note that if an update operation for one of the attributes would be dropped, the translation would not be allowed anymore. For example dropping the update for the implName attribute of the jointarget.Interface class instance, would lead to the following errors:

```
theta join classifiers.Interface with uml.Interface
where "classifiers.Interface.name = uml.Interface.name" as jointarget.Interface {
    keep attributes commons.NamedElement.name
    keep calculated attribute "classifiers.Interface.name.substring(1, classifiers.Interface
    .name.size() - 2).concat('Impl')" as jointarget.Interface.implName : String
    keep outgoing members.MemberContainer.members of type members.InterfaceMethod as type
    jointarget.Operation {
        keep attributes commons.NamedElement.name
        keep outgoing parameters.Parametrizable.parameters of type parameters.
    OrdinaryParameter as type jointarget.Parameter {
            keep attributes commons.NamedElement.name
        }
    }
    source attribute uml.NamedElement.name updates to "jointarget.Interface.name"
}

theta join classifiers.Class with uml.InterfaceRealization
where "uml.InterfaceRealization.contract.name.oclAsType(String).substring(1, uml.
    InterfaceRealization.contract.name.oclAsType(String).size() - 2).concat('Impl') =
    classifiers.Class.name"
as jointarget.InterfaceRealization {
    keep attributes commons.NamedElement.name
    keep outgoing uml.Dependency.client of type uml.Component as type jointarget.Component {
        keep attributes uml.NamedElement.name
    }
    keep outgoing uml.InterfaceRealization.contract of type uml.Interface as type jointarget
    .Interface
}
```

Listing 7.2: ModelJoin view definition for the UML and Java example. It includes the UML interfaces joined with the java interface definitions, the UML interface realization elements joined with the implementing classes, the corresponding components and interface operations including parameters.

```
inv mjtrace::join_Interface_Interface_Interface::attributeMapping_Interface_implName:
    The value of the calculated attribute "Interface.implName" is not up to date.
```

This could be solved by either updating the calculated attribute values manually or using the proposed update fix in Section 6.2.

7.2.1.4. Adding an Interface

Next the translatability of created instances will be evaluated.

Task We assume the user wants to create a new jointarget.Interface instance with the name "OrderIf", a operation with name "getOrderDetails" and a parameter named "orderNo". The user performs the update operation in Listing 7.4.

```
update jointarget.Interface {
    name: "StoreInfoIf"
    implName: "StoreImpl"
} where "jointarget.Interface.name = 'StoreIf'"

update jointarget.InterfaceRealization {
    name: "StoreInfoImpl"
} where "jointarget.InterfaceRealization.name = 'StoreImpl'"
```

Listing 7.3: Update operation to change the name of the interface and the name of the corresponding interface realization.

```
create jointarget.Parameter {
    name: "orderNo"
} as parameter

create jointarget.Operation {
    name: "getOrderDetails"
    parameters: OrderedSet{parameter}
} as operation

create jointarget.Interface {
    name: "OrderIf"
    implName: "OrderImpl"
    members: OrderedSet{operation}
}
```

Listing 7.4: Update operation to create a new interface with a new operation with one parameter.

Expected result A new instance of the source classes classifiers.Interface, uml.Interface, members.InterfaceMethod and parameters.OrdinaryParameter are created and the attribute values and links are set according to the corresponding target class instances.

Actual result All OCL constraints are fulfilled and on translation the source class instances get created and the links and attribute values are set. The result meets the expectation. However the new created class instance does not form a complete compilation unit in the EMFText Java model. The missing elements, like for example imports and package declarations, must be created manually. There is currently no mechanism for creating additional model elements.

7.2.1.5. Deleting an Interface

Next the translatability of created instances will be evaluated.

Task We assume the user wants to delete the jointarget.Interface instance with the name “StoreIf” and the corresponding instance of jointarget.InterfaceRealization. The user performs the update operation in Listing 7.5.

```
cascade delete jointarget.Interface
where "jointarget.Interface.name = 'StoreIf'"

delete jointarget.InterfaceRealization
where "jointarget.InterfaceRealization.name = 'StoreImpl'"
```

Listing 7.5: Update operation to delete an interface and the corresponding interface realization.

The *cascade* keyword indicates that all linked instances in the target model shall be deleted as well.

Expected result The corresponding source class instances of uml.Interface, classifier.Interface, classifiers.Class, uml.InterfaceRealization, members.InterfaceMethod and parameters.OrdinaryParameter get deleted.

Actual result All OCL constraints are fulfilled. The corresponding source classes instances of uml.Interface, classifier.Interface, classifiers.Class and uml.InterfaceRealization get deleted. However the instances of type members.InterfaceMethod and parameters.Ordinary-Parameter do not get deleted, because removing the link and instance from the target model only removes the corresponding link in the source model and does not delete the instance. The two intends: Delete a source class instance and remove the link to the source class instance are not distinguishable for a class created by keep reference expressions, because in both cases the link and the instances in the target model are removed. This is necessary to fulfill the PUTGET-Property, however seems to be a problematic case.

7.2.2. Palladio Media Store Example

As further example the media store example from the upcoming Palladio Book will be used. The Media Store example describes a file hosting system for audio files. Users can primarily download audio files, but also shall be allowed to upload own audio files. There exists an example project for Palladio with System, Execution Environment, Component Allocation and Usage Models.

7.2.2.1. metamodels

Service Effect Specification (SEFF) For performance measurements a service of a component can be described by a Service Effect Specification (SEFF) [9]. It describes called internal and external actions, resource acquire and release and the control flow including loops, branches and forks. An action must have one start and one stop action, which mark the beginning and ending of activities. An internal action models a component internal activity like doing a computation. An external call action describes a call to an external component. Conditional branches can be handled by Probabilistic Branches which contain the branch probabilities and actions to execute in each branch.

UML activity diagram The source of an SEFF can be a UML activity diagram [56]. Activity diagrams can be used to model the stepwise activities or actions of computations or organizational workflows. The modeled workflow or computation can be composed among other nodes from Actions, Start State, Stop State and Decision Nodes.

7.2.2.2. DB-Cache SEFF and UML Activity Diagram Example

For the evaluation we use the DB-Cache example from the Palladio Media Store Example Project. We suppose that the caching behavior was modeled by a UML activity diagram and then the corresponding SEFF was derived from it. The graphical representation of the two models can be seen in Figure 7.2. 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.

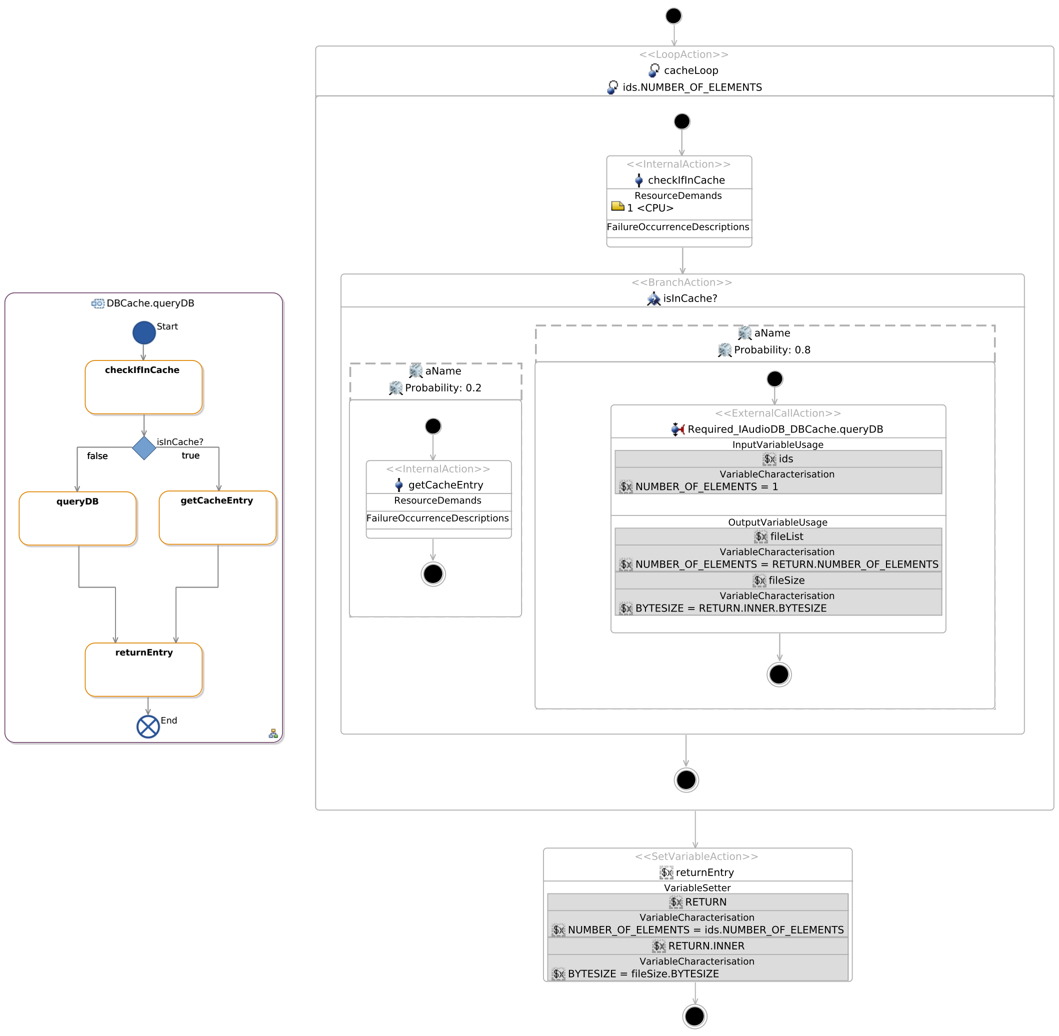


Figure 7.2.: The UML activity diagram (left) and SEFF diagram (right) for the queryDB operation of the DBCache component.

To combine the information from the two models and keep the models in sync, a view with the ModelJoin view definition in Listing 7.6 is created. In the view the Internal Actions and External Action of the SEFF are joined with the OpaqueAction. The name of the UML OpaqueAction respectively the entityName of the SEFF action is used to identify the correspondence between the source model classes. Similarly the UML DecisionNode and SEFF BranchAction are joined. As common subclass the ActivityNode from the UML model and the AbstractAction from PCM is kept. The control flow in the UML Model is modeled by instances of the ControlFlow class with a source and target reference. In SEFF the control flow is not a first class entity. It is defined through predecessor and successor references of the actions. Therefore the UML ControlFlow actions are joined with the Actions directly, where the name of the predecessor or successor matches the name of the target or source references.

7.2.2.3. Renaming an Action

First the effect of changing an attribute in the target model will be evaluated:

Task We assume the user wants to update the name of the jointarget.ExternalCallAction by changing the name from “queryDB” to “queryDatabase”. The user performs the update operations in Listing 7.7:

Expected result The source attribute uml.OpaqueAction.name and the source attribute pcm.seff.ExternalCallAction.entityName are updated for the corresponding source class instances to“queryDatabase”.

Actual result All constraints are fulfilled and on translation the source attributes uml.OpaqueAction.name and pcm.seff.ExternalCallAction.entityName of the corresponding source class instances are updated. The actual result therefore meets the expectation.

Notes If the update operations for the Edges would be dropped, the checking of the OCL constraints would lead to a violation of the following invariants with corresponding error messages:

```
inv mjtrace::join_ControlFlow_AbstractAction_BackEdge::attributeMapping_BackEdge_name:
    The value of the calculated attribute "BackEdge.name" is not up to date.
inv mjtrace::join_ControlFlow_AbstractAction_ForwardEdge::attributeMapping_ForwardEdge_name:
    The value of the calculated attribute "ForwardEdge.name" is not up to date.
```

The translation would get rejected, because the OCL checking detected two problems with the updated target model:

P1 The value of the calculated attributes jointarget.ForwardEdge.name and jointarget.BackEdge.name does not confirm to the updated attributes values. The PutGet-Property is violated because after the translation the value of these attributes would change, so the Problem was correctly detected.

Problem P1 can be solved by either updating the calculated attribute values manually or by using the proposed update fix in Section 6.2.

7.2.2.4. Deleting an Edge

Task We assume the user wants to delete an edge. The user performs the update operations in Listing 7.8:

Expected result The corresponding source class instance uml.ControlFlow get deleted and the links pcm.seff.AbstractActtion.successor_AbstractAction and pcm.seff.AbstractActtion.predecessor_AbstractAction get unset.

Actual result All constraints are fulfilled and on update translation the source classes uml.ControlFlow, pcm.seff.StartAction and pcm.seff.LoopAction get deleted. This behavior is unexpected by the user, because he or she may expect that only the references get unset in the seff model. However the defined join condition indicates the actual behavior and our approach in its current form does not allow to model this behavior.

Notes Dropping the delete operation for the jointarget.BackEdge instance, the checking of the OCL constraints would lead to a violation of the following OCL invariants with corresponding error messages:

```
mjtrace::join_ControlFlow_AbstractAction_ForwardEdge::consistentDeletion_BackEdge:
    The deletion of a target class instance of type "ForwardEdge" leads to the deletion of
    other target class instances of type "BackEdge".
```

The translation was rejected, because the OCL checking detected one problem with the updated target model:

P1 The deletion of the jointarget.ForwardEdge instance would lead to the deletion of a jointarget.BackEdge as well. However the jointarget.BackEdge was not deleted in the target model. Therefore the PutGet-Property would be violated.

Problem P1 could be solved by deleting the corresponding jointarget.BackEdge instance as well.

7.2.2.5. Creating a New Action

Task We assume the user wants to create a new jointarget.InternalAction class instance. The user performs the update operation in Listing 7.9:

Expected result An instance of the corresponding source class instances uml.OpaqueAction and pcm.seff.InternalAction are created.

Actual result The checking of the OCL constraints leads to a violation of the following invariants with corresponding error messages:

```
jointarget::ExternalCallAction::noNewTargetInstances:
    It is not allowed to create new instaces of type "ExternalCallAction" because these
    cannot be checked for side effects.
```

The translation was rejected, because the OCL checking detected one problem with the updated target model:

P1 The source class pcm.seff.AbstractAction (or a concrete subclass) is used in all four theta join expressions. However not all used source mode elements in the join conditions for the target classes jointarget.ForwardEdge and jointarget.BackEdge have a corresponding target model element. The not mapped source model elements are the references pcm.seff.AbstractActtion.successor_AbstractAction and pcm.seff.AbstractAction.predecessor_AbstractAction.

The actual result does not match the users expectations. The translation cannot be performed in this case because of Problem P1, which cannot be solved by the user. A source attribute update expression cannot be used in this case, because the not mapped source model elements are references and not attributes. A similar construct for references would be needed here.

7.3. Conclusion of the Evaluation

For atomic update operation on the target model most of the conditions for translatability are useful considering the PutGet-Property. The conditions ensure that the translation is uniquely determined.

If one of the conditions for the translatability of updated attribute values or links is violated, it requires minimal manual work to restore the condition. In some cases even an algorithm to restore the conditions can be used. However Condition C2.4 is problematic: A deletion of a linked target class instance does not lead to a deletion of the corresponding source class instance may be confusing for the user. Further Condition C2.5 is a strong restriction. It disallows the linking of new target class instances from existing target class instances. A new target class instance must be translated first, before it can be linked from an existing target class instance. However this is not possible in all cases.

The translatability of created class instances depends mostly on the view definition. Because the constraints get created at view definition time, all elements necessary to check the translatability of the new created class instance must be present in the target model. If the view definition is not designed with updatability in mind, a violation of these conditions leads to untranslatable new target class instances.

The translation of a deleted class instance is always possible under Condition C4.1. It requires some work to restore the condition manually, however an algorithm can likely be created.

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. Further we have seen that the constraint checking prevents the translation of inconsistent or ambiguous updates. In most cases the actual translation result meets our intuitive expectation. An exception is the deletion of a class instance created by a keep reference expression in 7.2.1.5. Another limitation is, that new created target class instances only lead to the creation of the corresponding source class instances. It is not possible to create a complete compilation unit in 7.2.1.4, because the classes of the missing source class instances do not have corresponding target classes.

The Palladio Media Store Example case shows, 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 are one way to supply the value for missing target attributes, however there is no construct for references yet. However such a construct is needed in the case in 7.2.2.5.

7.4. Limitations and Validity of the Case Study

Since ModelJoin is a new 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 not from real world examples and are created specifically for this case study. We tried to model practical scenarios and therefore used common model examples. However it is unclear, if the results of this evaluation fulfills 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 author of this thesis. To determine if the expected results are empirically valid, a user study is necessary. However this is beyond the scope of this work.

```
theta join uml.OpaqueAction with pcm.seff.InternalAction
where "uml.OpaqueAction.name = pcm.seff.InternalAction.entityName"
as jointarget.InternalAction {
    keep attributes uml.OpaqueAction.body
    keep supertype uml.ActivityNode as type jointarget.Action {
        keep attributes uml.NamedElement.name
    }
    keep supertype pcm.seff.AbstractAction as type jointarget.AbstractAction {}
    source attribute pcm.core.entity.NamedElement.entityName updates to "jointarget.InternalAction.name"
}

theta join uml.OpaqueAction with pcm.seff.ExternalCallAction
where "uml.OpaqueAction.name = pcm.seff.ExternalCallAction.entityName"
as jointarget.ExternalCallAction {
    keep supertype uml.ActivityNode as type jointarget.Action
    keep supertype pcm.seff.AbstractAction as type jointarget.AbstractAction
    keep outgoing pcm.seff.ExternalCallAction.calledService_ExternalService as type jointarget.OperationSignature {
        keep attributes pcm.core.entity.NamedElement.entityName
    }
    source attribute pcm.core.entity.NamedElement.entityName updates to "jointarget.ExternalCallAction.name"
}

theta join uml.DecisionNode with pcm.seff.BranchAction
where "uml.DecisionNode.name = pcm.seff.BranchAction.entityName"
as jointarget.DecisionNode {
    keep supertype uml.ActivityNode as type jointarget.Action
    keep supertype pcm.seff.AbstractAction as type jointarget.AbstractAction
    keep outgoing pcm.seff.BranchAction.branches_Branch of type pcm.seff.ProbabilisticBranchTransition as type jointarget.Tansition {
        keep attributes pcm.seff.ProbabilisticBranchTransition.branchProbability
        keep outgoing pcm.seff.AbstractBranchTransition.branchBehaviour_BranchTransition of type pcm.seff.ResourceDemandingBehaviour as type jointarget.Behaviur {
            keep outgoing pcm.seff.ResourceDemandingBehaviour.steps_Behaviour as type jointarget.AbstractAction
        }
    }
    source attribute pcm.core.entity.NamedElement.entityName updates to "jointarget.DecisionNode.name"
}
```

```
theta join uml.ControlFlow with pcm.seff.AbstractAction
where "uml.ControlFlow.target.name = (if pcm.seff.AbstractAction.successor_AbstractAction.
    oclIsUndefined() then '' else pcm.seff.AbstractAction.successor_AbstractAction.
    entityName endif)"
as jointarget.ForwardEdge {
    keep supertype uml.ActivityEdge as type jointarget.Edge {
        keep calculated attribute "'Edge from '.concat(uml.ActivityEdge.source.name.toString
    ()).concat(' to ').concat(uml.ActivityEdge.target.name.toString())" as jointarget.Edge.
    name : String
        keep outgoing uml.ActivityEdge.source as type jointarget.Action
        keep outgoing uml.ActivityEdge.target as type jointarget.Action
        keep outgoing uml.ActivityEdge.guard of type uml.LiteralBoolean as type jointarget.
    BooleanLiteral {
            keep attributes uml.LiteralBoolean.value
        }
    }
}

theta join uml.ControlFlow with pcm.seff.AbstractAction
where "uml.ControlFlow.source.name = (if pcm.seff.AbstractAction.predecessor_AbstractAction.
    oclIsUndefined() then '' else pcm.seff.AbstractAction.predecessor_AbstractAction.
    entityName endif)"
as jointarget.BackEdge {
    keep supertype uml.ActivityEdge as type jointarget.Edge
}
```

Listing 7.6: ModelJoin view definition for the PCM example.

```
update jointarget.ExternalCallAction {
    name: "queryDatabase"
} where "jointarget.OpaqueAction.name = 'queryDB'"

update jointarget.ForwardEdge {
    name: "Edge from isInCache? to queryDatabase"
} where "jointarget.FowardEdge.name = 'Edge from isInCache? to queryDB'"

update jointarget.ForwardEdge {
    name: "Edge from queryDatabase to returnEntry"
} where "jointarget.ForwardEdge.name = 'Edge from queryDB to returnEntry'"

update jointarget.BackEdge {
    name: "Edge from queryDatabase to returnEntry"
} where "jointarget.BackEdge.name = 'Edge from queryDB to returnEntry'"

update jointarget.BackEdge {
    name: "Edge from isInCache? to queryDatabase"
} where "jointarget.BackEdge.name = 'Edge from isInCache? to queryDB'"
```

Listing 7.7: Update operations to rename an action and the corresponding edges.

```
delete jointarget.ForwardEdge
where "jointarget.ForwardEdge.name = 'Edge from getCacheEntry to returnEntry'"

delete jointarget.BackEdge
where "jointarget.BackEdge.name = 'Edge from getCacheEntry to returnEntry'"
```

Listing 7.8: Update operations to delete pair of ForwardEdge and BackEdge.

```
create jointarget.InternalAction {
    name: 'putInCache',
    body: Set{'Puts the entry from the query into the cache'}
}
```

Listing 7.9: Update operation to create a new action.

8. Related work

8.1. Update Translation for Relational Views

For relational views the research of the View-Update-Problem mainly focuses on the concept of "translation under constant complement" introduced by Bancilhon and Spyratos [7]. In this case the problem of translating a view update is solved by finding a suitable "complementary" view that does not change under update translations and the information of the database can be reconstructed from the view and its complement view. Cosmadakis and Papadimitrious [21] showed that finding a minimum complement of a given view is NP-complete. They use the concept of functional dependencies [4] in the relational model.

While the complement view approach gives us the theoretical background and shows that view update are not always possible, not always unique and need additional information to solve ambiguities (i.e. complement view) it does not provide a useful mechanism for translating view updates.

Masunaga[45] described an algorithm for propagating updates against views defined in relational algebra to their base relations. He proposes that whenever semantic ambiguities arise these must be resolved by either the user or knowledge about the database scheme.

Keller [38] has formulated five criteria for acceptable view update translations. He relaxes the no side effects constraints of the the complementary view concept by arguing that it is to restrictive and does not allow certain useful translators. Further he enumerates a complete list of translator that satisfy these criteria for select, project and join views on relations in Boycee-Codd Normal Form [20].

This approach was extended by Barsalou, Siambela, Keller and Wiederhold [8] for object-based views. They describe an algorithm that enumerates all valid translations of the various update operations on view objects. A view object is a hierarchical subset of a normalized database scheme allowing object-oriented access to a relational database. The translator can then be chosen at view object generation time and used to translate all view updates.

Medeiros and Tompa [48] have developed a tool that predicts the side effects of an arbitrary mapping policies. In addition the algorithm shows if a desired update is reflected back the view, if the translated updates is applied to the database.

8.2. Update Translation for Tree Views

Of particularly interest in relational databases is the creation of XML views that provide an XML-document like view to a legacy relation database. For this purposes views are defined by a nested relational algebra [26]. A nested relational algebra is an extension of the classical relational algebra operations by a nest and unnest operation. Braganholo, Davidson and Heuser [13] studied first the updated of those nested views. They identified classes of XML views for which update can automatically translated, because the nest operation is invertible. In their case, there exists an isomorphic mapping between the view and the database relations, so that the translation of the view updates is unambiguous.

Tatarinov, Ives, Halevy and Weld [60] proposed a more fine grained set of update operations for XML views and integrates these in an XQuery language extensions. They compared numerous approaches for implementing a core set (inset, delete) of this operations. Since the mapping is also unambiguous in their cases the algorithms only differ in performance aspects.

In contrast to XML views of relational database the View-Update-Problem was also studied directly in the context of object-oriented database systems. School, Laasch and Tresch [59] prosed a query language that produces updateable views. The defined query language satisfies the concept of "object preservation". This means that a query never generates new object, so that a query result is always a set of existing objects. An update on the view is then directly an update on the database, which greatly simplifies the problem.

8.3. Linguistic Approaches to the View-Update-Problem

Another 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 programming have been developed in different use cases and communities including programming languages, databases, program transformations, constrained-based user interfaces and quantum computing.

Foster et al. [27] identify three major classes for theses languages:

Bi-directional languages form pairs of a query and translation function to lenses. Like in our approach the query function creates a view from the source data. The translation function takes both, the source data and the view, and returns the updated source data. The get function can project away some information and the translation function can restore them.

Bijective languages form a bijection between the source data and the view. The translation function only takes the view as an argument and returns a (perhaps partial) updated source data.

Reversible language consist of functions that can be applied in reverse, so that from the output the original source data can be computed. The inverse function can only be applied to a unmodified output, so no update translation is possible.

8.3.1. Bi-Directional Transformation Languages

Bi-Directional Transformations are studied intensively in the context of constraint-based user interfaces.

Foster et al. [27] propose a domain specific programming language for the View-Update-Problem in the case of tree-structured data. All expressions in this language denote bi-directional transformation between a “concrete” tree and an “abstract view”. They call these transformations dubbed lenses and define two essential laws that guarantee the “well behaved” lenses of these lenses. These laws are similar to the correct update translations proposed by Dayal and Bernstein [22] for relational databases. The GetPut law states that, if an abstract view is generated from a concrete tree and is immediately putback without modification the concrete tree does not change. The PutGet law on the other hand archives that the putback operation must capture all the information contained in the abstract view. If an updated abstract view is put back, getting the abstract view again without changing the concrete tree yields the same abstract view.

Their lenses feature familiar constructs from functional programming (composition, mapping, projection, conditionals, recursion) together with some special operations for tree manipulation.

Another approach taken by Meertens [49] is even more general: He defines get and putback as a relation between the view and model and not just as functions. He introduced the concept of constraint maintainers which are a pair of functions, that computes the necessary updates so that a constraint of the relation is maintained. One function for each directions of update translation from one side of the relation to the other. His constraint maintainers operate on list like data structures.

Mu, Hu and Takeichi [51] recently proposed a functional injective language to define transformations, so that synchronization behavior can be automatically derived by algebraic reasoning. In another paper [35] they developed an editor for structured documents, where the user can perform a sequence of editing operations on the document view and the editor automatically derives an document source and a transformation that produces the document view. The used bidirectional transformation language not only describes the relationship between the source document and view, it also describes the data dependencies in the view.

8.3.2. Bijective Transformation Languages

This field of languages includes program inversion and inverse computing. Dijkstra introduced program inversion as deriving the inverse program from a program [23]. Inverse computing computes a possible input of a program for a particular output [46]. Abramov and Glück [3] survey fundamental concepts in inverse programming and present an algorithm for inverse computation in a first-order functional language.

Abiteboul et al. [2] proposed a declarative language for the correspondence and translation between data stored in different formats. The defined rules can be used to translate the data from one format to another. The rules are defined by predicates which express the relation between two data sources. In their approach the two data models in the correspondence contain the same information in a different tree structure. Later [1] they proposed a query language for structured data stored in files. They use a grammar annotated with program information, so that a file can be viewed as database structure, such that it can be queried and updated. Querying involves parsing the file and updating serializing in the grammar.

Ohori and Tajima [53] developed a typed polymorphic calculus for views and object sharing. Their calculus allows the definition of views which includes objects from different classes in an object-orientated database. Similar to ModelJoin the classes in the view (the view type) and the transformation is defined by the calculus. As operations union of different classes, projection and calculated properties can be used.

8.3.3. Reversible Transformation Languages

The languages in this class can only partially be used for update translation, because only the unmodified output of the original function is accepted by the reverse function.

Landauer [40] argued that functions with a not single-valued inverse are logically irreversible and this is associated with physical irreversibility. Bennet [11] and [41] showed independently that a Turing machine can be made logically reversible. This is done by saving the intermediate results of each step. Bekkers [6] proposed an abstract computer model and programming language with just injective primitive operations. This has the advantage that no check pointing or saving of intermediate results has to be done. More languages with just injective primitives have been developed (e.g. [52], [44] or [30]).

9. Conclusion

We have formulated the View-Update-Problem for ModelJoin views, and adapted two fundamental well known properties, the GetPut- and PutGet-Property, for models defined in MOF or Ecore metamodels. To check if an update target model satisfies these properties, we have chosen OCL as validation language. We have extended ModelJoin with a trace model and defined a scheme to derive OCL constraints for a ModelJoin view definition. We have shown that the target model is translatable if all constraints are fulfilled and that an unmodified target model satisfies all constraints.

Furthermore, we have discussed alternative translation strategies for target model updates and described how the OCL constraints need to be adapted to reflect these strategies. We have introduced multiple strategies for translating newly created target class instances and deleted target class instances. We have extended ModelJoin by source attribute update expressions to allow the update of unmapped attributes in the source model at update translation. These can be used to fully define the translation behavior at view definition time.

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. These include the automatic creation of missing target class instance, the automatic calculation of derived model elements, and the propagation of updated attribute values.

Finally, we have evaluated our approach for atomic update operations on the target model and in two case study examples. The evaluation has shown that almost all of the conditions for translatability are useful, the PutGet-Property ensures consistent target models and that the translation can be uniquely determined. Further, all but one chosen update operation were translatable in the case study example cases.

Our approach introduces a translation semantic to allow the editability of ModelJoin views. By choosing one of the proposed translation strategies, the semantic of a target model update can be fixed. Furthermore, the derived OCL constraints can be used to check the translatability of an updated view. Using standard OCL constraints allows the easy integration of the translatability check into existing tools. With the introduction of the trace model, target model updates can be translated in a state based approach, without losing the concrete mapping of the target model elements to the source model elements.

We have seen that not all updated target models, which fulfill the GetPut-Property, can be translated. The generated constraints are too restrictive in some cases. Especially join conditions involving unmapped references cannot be checked for new instances easily. Further, if source classes are involved in multiple joins, newly created target model instances cannot be translated in all 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.

Currently the checking of the OCL constraints is unnecessarily slow, because OCL constraints for Ecore models do not support bidirectional navigation for references, and finding a referencing class instance has linear time cost. In the future, a formulation of the constraints without using OCL or by extending OCL for Ecore with support for bidirectional navigation [18] may speed up the translatability check significantly for large models.

We have always used the OCL constraints to decide the translatability for a concrete target model. It would be desirable to decide the translatability for a sequence of update operation on the target model in general. 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.

The validity of the case study for the chosen examples is questionable, because we were not able to use real world cases. When there are applications of ModelJoin, these could be used in a second case study to find more problematic cases.

Bibliography

[1] Serge Abiteboul, Sophie Cluet, and Tova Milo. "A logical view of structure files". In: *VLDB Journal* (1998).

[2] Serge Abiteboul, Sophie Cluet, and Tova Milo. "Correspondence and translation for heterogeneous data". In: *Database Theory—ICDT'97*. Springer, 1997, pp. 351–363.

[3] Sergei M Abramov and Robert Glück. "Principles of inverse computation and the universal resolving algorithm". In: *The essence of computation* 2566 (2002), pp. 269–295.

[4] W.W. Armstrong. "Dependency Structures of Database Relationship". In: *Proceedings of IFIP 14*. North-Holland, Amsterdam, 1974, pp. 580–583.

[5] L. Moura B. Dutertre. "Yices: An SMT Solver". 2008. URL: `http://yices.csl.sri.com/`.

[6] HenryG. Baker. "NREVERSAL of fortune — The thermodynamics of garbage collection". English. In: *Memory Management*. Ed. by Yves Bekkers and Jacques Cohen. Vol. 637. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 507–524. ISBN: 9783540559405. DOI: `10.1007/BFb0017210`. URL: `http://dx.doi.org/10.1007/BFb0017210`.

[7] F. Bancilhon and N. Spyratos. "Update Semantics of Relational Views". In: *ACM Trans. Database Syst.* 6.4 (Dec. 1981), pp. 557–575. ISSN: 0362-5915. DOI: `10.1145/319628.319634`. URL: `http://doi.acm.org/10.1145/319628.319634`.

[8] Thierry Barsalou et al. "Updating Relational Databases Through Object-based Views". In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. SIGMOD '91. Denver, Colorado, USA: ACM, 1991, pp. 248–257. ISBN: 0897914252. DOI: `10.1145/115790.115831`. URL: `http://doi.acm.org/10.1145/115790.115831`.

[9] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22. ISSN: 0164-1212.

[10] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 3-540-68977-X, 978-3-540-68977-5.

[11] C. H. Bennett. "Logical Reversibility of Computation". In: *IBM J. Res. Dev.* 17.6 (Nov. 1973), pp. 525–532. ISSN: 0018-8646. DOI: `10.1147/rd.176.0525`. URL: `http://dx.doi.org/10.1147/rd.176.0525`.

[12] Richard F. Paige Boris Gruschko, Dimitrios S. Kolovos. "Towards synchronizing models with evolving metamodels". In: *In Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR.* 2007.

[13] Vanessa P. Braganholo, Susan B. Davidson, and Carlos A. Heuser. "On the updatability of XML views over relational databases". In: *International Workshop on Web and Databases, San Diego, California, June 12-13, 2003.* 2003, pp. 31–36. URL: `http://www.cse.ogi.edu/webdb03/papers/06.pdf`.

[14] Erik Burger. "Flexible Views for View-Based Model-Driven Development". In: *Proceedings of the 18th international doctoral symposium on Components and architecture.* WCOP '13. Vancouver, British Columbia, Canada: ACM, 2013, pp. 25–30. ISBN: 9781450321259. DOI: `10.1145/2465498.2465501`. URL: `http://doi.acm.org/10.1145/2465498.2465501`.

[15] 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. URL: `http://sdqweb.ipd.kit.edu/publications/pdfs/burger2010a.pdf`.

[16] 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`.

[17] Erik Burger et al. "View-based model-driven software development with ModelJoin". English. In: *Software & Systems Modeling* (2014), pp. 1–24. ISSN: 1619-1366. DOI: `10.1007/s10270-014-0413-5`. URL: `http://dx.doi.org/10.1007/s10270-014-0413-5`.

[18] Jordi Cabot et al. "Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010): Navigating Across Non-Navigable Ecore References via OCL". In: *Electronic Communications of the EASST* 36 (2011).

[19] Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. "Checking unsatisfiability for OCL constraints". In: *Electronic Communications of the EASST* 24 (2010).

[20] Edgar F Codd. *Recent Investigations in Relational Data Base Systems.* IBM Thomas J. Watson Research Division, 1974.

[21] Stavros S. Cosmadakis and Christos H. Papadimitriou. "Updates of Relational Views". In: *J. ACM* 31.4 (Sept. 1984), pp. 742–760. ISSN: 0004-5411. DOI: `10.1145/1634.1887`. URL: `http://doi.acm.org/10.1145/1634.1887`.

[22] Umeshwar Dayal and Philip A. Bernstein. "On the Correct Translation of Update Operations on Relational Views". In: *ACM Trans. Database Syst.* 7.3 (Sept. 1982), pp. 381–416. ISSN: 0362-5915. DOI: `10.1145/319732.319740`. URL: `http://doi.acm.org/10.1145/319732.319740`.

[23] Edsger W Dijkstra. “Program inversion”. In: *Program Construction*. Springer, 1979, pp. 54–57.

[24] *EMFText Concrete Syntax Zoo Java 5*. `http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Java_5`.

[25] Anthony Finkelstein et al. “Viewpoints: A framework for integrating multiple perspectives in system development”. In: *International Journal of Software Engineering and Knowledge Engineering* 2.01 (1992), pp. 31–57.

[26] S. J. Fischer and P. C. Fischer. “Nested relational structures”. In: *Advances in Computing Research* (1986), 3:269–307.

[27] 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. ISSN: 0362-1340. DOI: `10.1145/1047659.1040325`. URL: `http://doi.acm.org/10.1145/1047659.1040325`.

[28] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. `http://www.eclipse.org/modeling/emf/`. retrieved 2 June 2015.

[29] Eclipse Foundation. *EMF Documentation*. `http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html`. retrieved 2 June 2015.

[30] Robert Glück and Masahiko Kawabe. “A Program Inverter for a Functional Language with Equality and Constructors”. English. In: *Programming Languages and Systems*. Ed. by Atsushi Ohori. Vol. 2895. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 246–264. ISBN: 9783540205364. DOI: `10.1007/978-3-540-40018-9_17`. URL: `http://dx.doi.org/10.1007/978-3-540-40018-9_17`.

[31] Thomas Goldschmidt. “View-based textual modelling”. PhD thesis. Karlsruhe, 2011. ISBN: 9783866446427. URL: `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022234`.

[32] Thomas Goldschmidt, Steffen Becker, and Erik Burger. “View-based Modelling-A Tool Oriented Analysis”. In: *Proceedings of the Modellierung*. 2012.

[33] Florian Heidenreich et al. *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik, 2009.

[34] Sebastian Herold et al. “CoCoME-the common component modeling example”. In: *The Common Component Modeling Example*. Springer, 2008, pp. 16–53.

[35] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. “A programmable editor for developing structured documents based on bidirectional transformations”. In: *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM. 2004, pp. 178–189.

[36] IBM. *INSTEAD OF Triggers - All Views are Updatable!* 2002. URL: `http://www.ibm.com/developerworks/data/library/techarticle/0210rielau/0210rielau.html` (visited on 11/18/2015).

[37] ANSI/ISO/IEC International Standard (IS). *Database Language SQL – Part 2: Foundation (SQL/Foundation).* ANSI/ISO/IEC 9075:2011. Dec. 2011.

[38] Arthur M. Keller. “Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins”. In: *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems.* PODS ’85. Portland, Oregon, USA: ACM, 1985, pp. 154–163. ISBN: 0897911539. DOI: `10.1145/325405.325423`. URL: `http://doi.acm.org/10.1145/325405.325423`.

[39] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. “Model Comparison: A Foundation for Model Composition and Model Transformation Testing”. In: *Proceedings of the 2006 International Workshop on Global Integrated Model Management.* GaMMa ’06. Shanghai, China: ACM, 2006, pp. 13–20. ISBN: 1595934103. DOI: `10.1145/1138304.1138308`. URL: `http://doi.acm.org/10.1145/1138304.1138308`.

[40] R. Landauer. “Irreversibility and Heat Generation in the Computing Process”. In: *IBM Journal of Research and Development* 5.3 (July 1961), pp. 183–191. ISSN: 0018-8646. DOI: `10.1147/rd.53.0183`.

[41] Yves Lecerf. “Machines de Turing réversibles. Récursive insolubilité en $n \in N$ de l’équation $u = \theta^n$, où θ est un “isomorphisme de codes””. In: *Comptes Rendus* 257.18 (1963), p. 2597.

[42] 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. San Diego, California: ACM, 2003, pp. 49–55. ISBN: 1581136706. DOI: `10.1145/773153.773159`. URL: `http://doi.acm.org/10.1145/773153.773159`.

[43] Yuehua Lin, Jing Zhang, and Jeff Gray. “Model comparison: A key challenge for transformation testing and version control in model driven software development”. In: *Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development.* Springer, 2004, pp. 219–236.

[44] Christopher Lutz and Howard Derby. “Janus: a time-reversible language”. In: *Caltech class project* (1982).

[45] Yoshifumi Masunaga. “A Relational Database View Update Translation Mechanism”. In: *Proceedings of the 10th International Conference on Very Large Data Bases.* VLDB ’84. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1984, pp. 309–320. ISBN: 0934613168. URL: `http://dl.acm.org/citation.cfm?id=645912.671143`.

[46] John McCarthy. “The inversion of functions defined by Turing machines”. In: *Automata studies* (1956), pp. 177–181.

[47] W. McCune. “Prover9 and Mace4”. 2005–2010. URL: `http://www.cs.unm.edu/~mccune/prover9/`.

[48] ClaudiaBauzer Medeiros and FrankWm. Tompa. “Understanding the implications of view update policies”. English. In: *Algorithmica* 1.1-4 (1986), pp. 337–360. ISSN: 0178-4617. DOI: `10.1007/BF01840451`. URL: `http://dx.doi.org/10.1007/BF01840451`.

[49] Lambert Meertens. *Designing Constraint Maintainers for User Interaction.* Tech. rep. 1998.

[50] Microsoft. *Transact-SQL-Reference CREATE VIEW (Transact-SQL).* 2014. URL: `https://msdn.microsoft.com/de-de/library/ms187956(v=sql.120).aspx` (visited on 11/18/2015).

[51] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. "An Algebraic Approach to Bi-Directional Updating". In: *In ASIAN Symposium on Programming Languages and Systems (APLAS.* Springer, 2004, pp. 2–18.

[52] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. "An Injective Language for Reversible Computation". English. In: *Mathematics of Program Construction.* Ed. by Dexter Kozen. Vol. 3125. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 289–313. ISBN: 9783540223801. DOI: `10.1007/978-3-540-27764-4_16`. URL: `http://dx.doi.org/10.1007/978-3-540-27764-4_16`.

[53] Atsushi Ohori and Keishi Tajima. "A Polymorphic Calculus for Views and Object Sharing (Extended Abstract)". In: *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* PODS '94. Minneapolis, Minnesota, USA: ACM, 1994, pp. 255–266. ISBN: 0897916425. DOI: `10.1145/182591.182623`. URL: `http://doi.acm.org/10.1145/182591.182623`.

[54] Object Management Group (OMG). *Meta Object Facility (MOF) Core.* `http://www.omg.org/spec/MOF/2.4.1/`. Version 2.4.1. 2013.

[55] Object Management Group (OMG). *Object Constraint Language (OCL).* `http://www.omg.org/spec/OCL/2.4/`. Version 2.4. 2014.

[56] Object Management Group (OMG). *UML Superstructure.* `http://www.omg.org/spec/UML/2.2/`. Version 2.2. 2009.

[57] Oracle. *Database Administrator's Guide Managing Views.* 2008. URL: `https://docs.oracle.com/cd/B28359_01/server.111/b28310/views001.htm` (visited on 11/18/2015).

[58] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures - The Palladio Approach.* To appear. MIT Press, 2015.

[59] MarcH. Scholl, Christian Laasch, and Markus Tresch. "Updatable views in object-oriented databases". English. In: *Deductive and Object-Oriented Databases.* Ed. by C. Delobel, M. Kifer, and Y. Masunaga. Vol. 566. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pp. 189–207. ISBN: 9783540550150. DOI: `10.1007/3-540-55015-1_10`. URL: `http://dx.doi.org/10.1007/3-540-55015-1_10`.

[60] Igor Tatarinov et al. "Updating XML". In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data.* SIGMOD '01. Santa Barbara, California, USA: ACM, 2001, pp. 413–424. ISBN: 1581133324. DOI: `10.1145/375663.375720`. URL: `http://doi.acm.org/10.1145/375663.375720`.

A. Appendix

A.1. Technical Changes to the ModelJoin-Implementation

As part of the thesis the existing experimental ModelJoin implementation [16] was extended by a trace model and constraint generator. The extended ModelJoin Workflow can be found in Figure A.1.

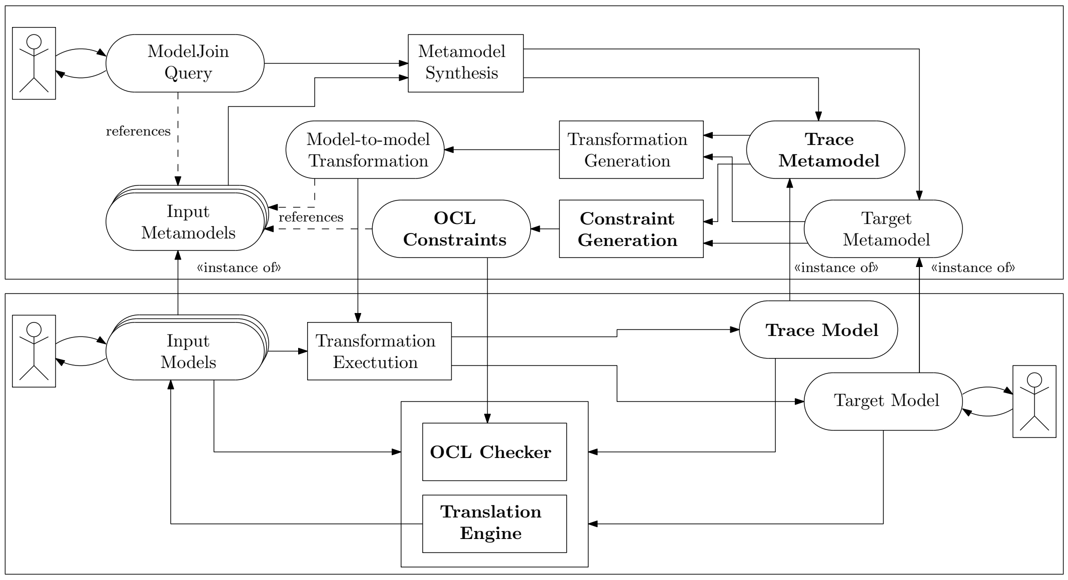


Figure A.1.: Extended ModelJoin Workflow. Based on Figure 6.7 in [14]. New elements are highlighted in bold.

A.1.1. Trace Model Generation

The Metamodel synthesis was extended, so that it generates a trace metamodel in addition to the target metamodel. The trace metamodel contains trace classes like described in chapter 4.4 and following. These classes reference classes in the input metamodel and target metamodel. Therefore the trace metamodel depends on the input models and is not input model independent like the target metamodel. The trace metamodel is currently only needed for the translatability check.

Further the transformation generator was extended. The generated transformation now outputs and trace model in addition to the target model.

A.1.2. OCL Constraint Generation and Checking

To generate the OCL-constraints, a constraint generator was created. It takes the trace and target metamodels as input and outputs an OCL-File (.ocl) with OCL constraints. These OCL constraints can be used to check if an updated target model can be translated back to the input models. The OCL generator uses the same target metamodel annotations as the metamodel generator. The generated OCL constraints can be loaded into the Ecore Model Editor of the Eclipse IDE. The editor then shows the violated constraints and marks the corresponding model elements with an error.

Additional an OCL checker was developed that takes the input models, the target and trace model and the OCL-Constraints as input and checks if the updated target model fulfills all constraints.

A.1.3. SVN Locations

The source for the implementation can be found under the following SVN locations:

https://svnserver.informatik.kit.edu/i43/svn/code/MDSD/ModelJoin/Core/

OCL Constraint Generation:	trunk/edu.kit.ipd.sdq.mdsd.mj.constraints.generator
OCL Constraint Checker:	trunk/edu.kit.ipd.sdq.mdsd.mj.constraints.checker
Trace Model Generation:	trunk/edu.kit.ipd.sdq.mdsd.mj.metamodel.generator

A.1.4. Future Enhancements

Translation Engine The actual model transformation, which applies the translation to the input models, was not yet developed. It could be generated by an additional transformation generator or use the target and trace model directly to derive the source model updates. The source model updates, that result from an updated target and trace model are given in chapter 4.5 and for different translation strategies in chapter 5.

Target Model Annotations With the new trace model some of the annotations in the target model are redundant. In future versions of ModelJoin, the annotations could be removed and the information could be obtained from the trace model instead.