Electronic Communications of the EASST
Volume 42 (2011)

Proceedings of the
5th International Workshop on
Multi-Paradigm Modeling
(MPM 2011)

Reusing Model Transformations across Heterogeneous Metamodels

Manuel Wimmer, Angelika Kusel, Werner Retschitzegger,
Johannes Schönböck, Wieland Schwinger,
Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara

12 pages

Guest Editors: Vasco Amaral, Hans Vangheluwe, Cécile Hardebolle, Lazlo Lengyel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Reusing Model Transformations across Heterogeneous Metamodels

Manuel Wimmer[1], **Angelika Kusel**[2], **Werner Retschitzegger**[2],
Johannes Schönböck[1], **Wieland Schwinger**[2],
Jesús Sánchez Cuadrado[3], **Esther Guerra**[4], **and Juan de Lara**[4]

[1] Vienna University of Technology, Austria
[2] Johannes Kepler University Linz, Austria
[3] Universidad de Murcia, Spain
[4] Universidad Autónoma de Madrid, Spain

Abstract: Model transformations are key enablers for multi-paradigm modeling. However, currently there is little support for reusing transformations in different contexts since they are tightly coupled to the metamodels they are defined upon, and hence reusing them for other metamodels becomes challenging. Inspired from generic programming, we proposed *generic model transformations*, which are defined over so-called metamodel concepts, which are later bound to specific metamodels. Nevertheless, the current binding mechanism lacks automated resolution support for recurring structural heterogeneities between metamodels. Therefore, based on a systematic classification of heterogeneities, we propose a flexible binding mechanism being able to *automatically resolve recurring structural heterogeneities* between metamodels. For this, the binding model is analyzed and required adaptors are automatically added to the transformation.

Keywords: Transformation Reuse, Metamodel Heterogeneities

1 Introduction

Multi-paradigm modeling [MV02] fosters the description of systems using the most appropriate formalism and level of abstraction. Hence, *transformations* become key enablers in this approach to transform models between different languages and abstraction levels. So far, transformation development is a type-centric activity, because transformations are defined over and thus, tightly coupled to the types of specific metamodels (MMs). However, the defined transformation may apply to other MMs as well, which share similar characteristics. The main obstacle in reusing model-to-model transformations are structural heterogeneities between MMs, a well-known problem in database integration [LN07], which occur if semantically similar concepts are realized in different ways in different MMs. In particular, such structural heterogeneities may break existing transformations if applied to other MMs. Thus, an automatic resolution of these heterogeneities is desirable to increase reuse of transformations and thus, to avoid a tedious and error-prone manual adaptation.

In this respect, this paper extends ideas from our previous work (cf. [CGL11, LG10]), where – inspired from generic programming – we proposed *generic model transformations*. Generic model transformations are not defined over specific MMs, but over so-called *concept MMs*. This offers an extra level of indirection, because concept MMs can be *bound* to *specific MMs*, making

the generic transformation reusable for specific MMs. If structural heterogeneities occur between the concept MMs and the specific MMs, adapters have to be manually defined for their resolution. To ease the burden of resolving recurring structural heterogeneities, we propose an approach to automatically derive adapters for generic transformations. These adapters realize a virtual view on the specific MM providing required features of the concept MM which are expressed structurally different in the specific MM. These adapters effectively establish a *subtype* relationship [SJ07] between the concept MM and the specific MM. Please note that concept and the specific MM are assumed to correspond to a single meta-metamodel, e.g., Ecore. Based on our previous work for establishing a classification of MM heterogeneities [WKK+10b], we provide a comprehensive set of adapters for resolving structural heterogeneities between Ecore[1]-based MMs. As a proof of concept we discuss an implementation on top of ATL [JABK08] where the reuse of generic transformations is achieved by means of a Higher Order Transformation (HOT) [TCJ10] by adding helper functions realizing the virtual view. However, our approach is applicable to other transformation languages supporting helper functions (e.g., QVT).

Outline. Section 2 shortly reviews generic model transformations and highlights points for improvements. Whereas Section 3 presents obstacles for transformation reuse on the basis of a running example, the subsequent Section 4 introduces the approach for automatically deriving adapters. Finally, Section 5 elaborates on related work, and Section 6 concludes the paper.

2 Generic Model Transformations in a Nutshell

In our previous work [CGL11], the notion of *generic model transformations* (i.e., decoupling of transformation logic and MMs) as a reuse mechanism has been presented which are defined between so-called *concept MMs*. Concept MMs specify the requirements (e.g., needed attributes and references etc.), which a specific MM must fulfill to qualify for the source domain or target domain of a transformation (cf. upper part of Fig. 1 for a simple example). The elements (classes, attributes, and references) of the concept MMs can be seen on a conceptual level as variables that have to be bound to the elements of specific MMs (cf. lower part of Fig. 1) by means of a binding model (cf. middle part of Fig. 1). The established bindings between concept MMs and specific MMs are used as input for a HOT, which rewrites the generic model transformation, i.e., the concept types are replaced by specific types, to obtain a specific model transformation.

As can be seen in Fig. 1, in the course of the binding not only naming differences but also more challenging structural heterogeneities have to be resolved between the source concept MM and the specific source MM. In our example, `UMLClass` exhibits a reference to an optional superclass whereas `Component` exhibits the *inverse* reference to its subcomponents. Thus, a crucial issue to increase reuse of generic transformations is a flexible binding mechanism which is able to overcome structural heterogeneities automatically. Although the binding language we proposed in [CGL11] allows specifying OCL-based adapters to resolve heterogeneities, there exist three points for improvements. First, although heterogeneities are recurring, *no automatic resolution* thereof is supported by the binding language forcing the transformation reuser to specify complex OCL code manually (cf. ① in Fig. 1). Second, the adaptations for resolving heterogeneities between concept MMs and specific MMs are *scattered across* the specific model

[1] http://www.eclipse.org/modeling/emf

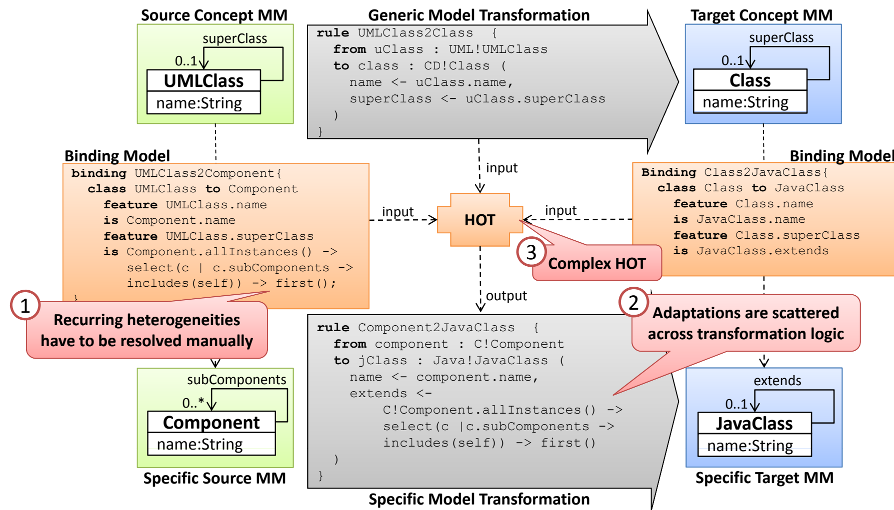


Figure 1: Generic Model Transformations at a Glance

transformation leading to hardly maintainable transformations (cf. ② in Fig. 1). Finally, since the adaptation of the transformation code comprises the challenging rewriting of OCL expressions, a *complex HOT* has to be developed (cf. ③ in Fig. 1), which is still challenging [TCJ10].

In order to alleviate these problems, in this paper we propose an automated mechanism to resolve commonly occurring heterogeneities, based on a library of *generic* and *composable* adapters. We also simplify the HOT proposed in [CGL11], leading to a simpler adaptation mechanism and more comprehensible specific transformations. We restrict to the case of generic transformations defined over a concept MM on the source domain, and a specific MM on the target. Supporting genericity on the target is left to future work.

3 Metamodel Heterogeneities: Obstacles for Transformation Reuse

In this section, first, a motivating example for transformation reuse is presented, and subsequently, an overview on potentially arising heterogeneities between Ecore-based MMs is given.

3.1 Motivating Example

In order to motivate our approach, Fig. 2 depicts an example of a generic model transformation between `ClassDiagrams` and `ERDiagrams` for demonstration purposes. Since our approach is independent of the concrete implementation of the generic transformation, we omit implementation details thereof. Instead, the focus is on how to reuse the existing transformation such that it works for a specific source MM (cf. bottom of Fig. 2). In this respect, one can see that a major obstacle to reuse the existing transformation are the heterogeneities that exist between the source concept MM and the specific source MM (cf. ①-④ in Fig. 2), as detailed in the following.

Concept Package. Although the concept `Package` serves in both MMs as a container for model elements, the containment hierarchies differ. Whereas in the source concept MM, the class `Package` nests `UMLClasses` via the reference `ownedClasses`, the specific class `Package` offers the reference `ownedElements` to nest any kind of named elements (cf. ① in Fig. 2).

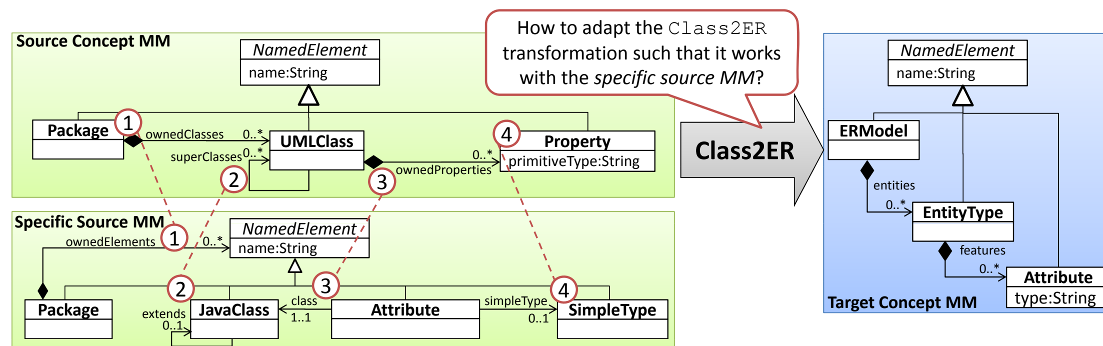


Figure 2: Motivating Example with Exemplary Heterogeneities

Concept UMLClass. In the context of the concept `UMLClass` several heterogeneities arise. Whereas the source concept MM supports multiple inheritance, the specific source MM supports single inheritance only (cf. ② in Fig. 2). Second, a difference in linking can be detected since in the source concept MM the class `UMLClass` owns its properties by means of the reference `UMLClass.ownedProperties` whereas in the specific source MM, attributes refer to their owning class via the reference `Attribute.class` (cf. ③ in Fig. 2).

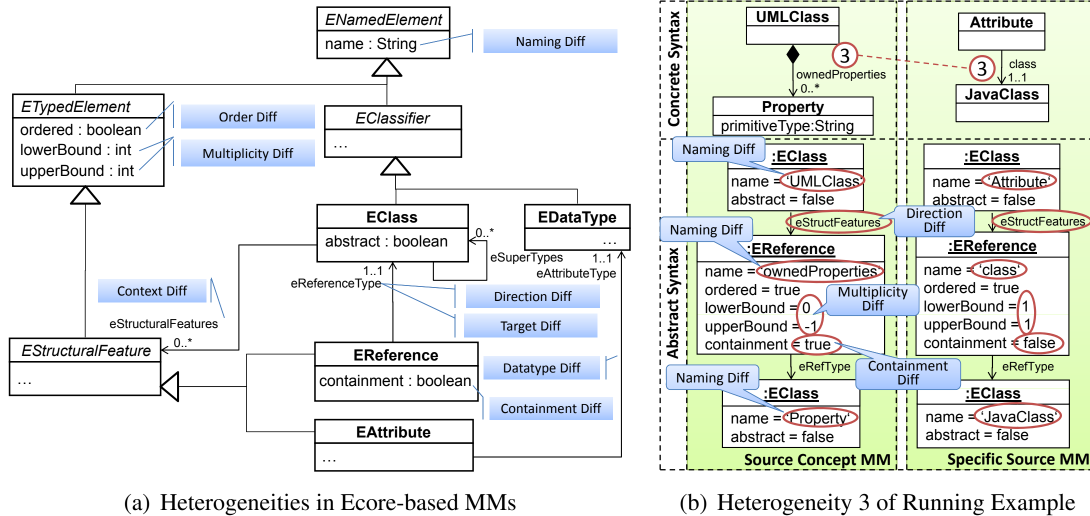
Concept Property. Finally, there is a heterogeneity in the context of the concept `Property`. Whereas the source concept MM allows to describe the primitive type of a property by means of a simple attribute value (cf. attribute `Property.primitiveType`), the specific source MM explicates this fact by means of an object of type `SimpleType` (cf. ④ in Fig. 2).

Before delving into details about how to resolve these heterogeneities, a major question is, which heterogeneities between Ecore-based MMs might occur at all. For this, a summary on potential heterogeneities based on our previous work [WKK+10b] is provided in the following, allowing to clearly factor out the applicability but also the boundaries of our approach.

3.2 Metamodel Heterogeneities at a Glance

Heterogeneities occur if semantically similar concepts are realized in different ways leading to differently structured MMs [WKK+10a]. Since we focus on Ecore-based MMs, a systematic classification of different kinds of heterogeneities is obtained by investigating potential variation points [WKK+10b]. To shortly recall this classification for the purposes needed in this paper, Fig. 3(a) depicts the relevant extract of the Ecore meta-metamodel which is used for defining MMs. Assuming that both, concept MM and specific MM use the *same* Ecore concept, differences in all the owned *meta-features* may arise (cf. Fig. 3(a)). The term meta-feature denotes features of Ecore elements, e.g., the feature `name` of `ENamedElement`. In this paper, we focus on heterogeneities of meta-features of `EAttributes` and `EReferences`, and leave the heterogeneities of `EClasses` as future work except naming differences between classes.

Fig. 3(b) shows the application of the classification to heterogeneity ③ of the example. When comparing the values of the meta-features in the abstract syntax, one can see that six differences occur, comprising three *naming differences*, a *multiplicity difference*, a *containment difference*, and a *direction difference* (since the reference `UMLClass.ownedProperties` in the concept MM, exhibits the *inverse* direction in the specific MM (cf. reference `Attribute.class`)).



(a) Heterogeneities in Ecore-based MMs (b) Heterogeneity 3 of Running Example

Figure 3: Heterogeneity Classification and Exemplary Application

4 Automatic Generation of Adapters

To resolve the afore presented heterogeneities, two approaches might be taken to allow the generic model transformation to work with the specific MM. First, the specific source MM might enforce a change of the generic transformation, whereby this idea is closely related to *program transformation* [Vis01]. Second, the specific source models might be transformed such that they conform to the source concept MM and thus, might act as input of the generic transformation definition. Thereby, the transformation definition remains unchanged, but the models get transformed by a preceding transformation, being closely related to *data transformation* [Len02].

We follow a hybrid approach by combining the advantages of both ideas, namely (i) *direct trace links* between specific source and target models, since the instantiation of a generic transformation leads to a single transformation (advantage of program transformation approach), and (ii) an *lightweight adaptation process*, since the generic transformation code does not have to be rewritten – except of renaming classes which is a trivial rewriting rule (advantage of data transformation approach). The core idea is to extend the generic model transformation by adapters which provide a *virtual view* on the specific models such that they correspond to the interpretation of the generic MM (cf. Fig. 4). By this, a *subtype* relationship between the concept MM and the specific MM is established, whereby a specific MM is a subtype of a concept MM if the specific MM provides at least the classes with features that the concept MM provides [SJ07]. By this, transformations depending on the concept MM may also work with any sub-MM.

Considering the example (cf. Fig. 4), obviously, there is no subtype relationship between the concept MM and the specific MM so far, since the specific MM misses four features, which the concept MM defines, namely `Package.ownedClasses`, `JavaClass.superClasses`, `JavaClass.ownedProperties`, `Attribute.primitiveType` (highlighted in dashed lines in Fig. 4). To establish the required subtype relationship, the binding model allows to describe “fuzzy” correspondences between semantically related features in a simple one-to-one

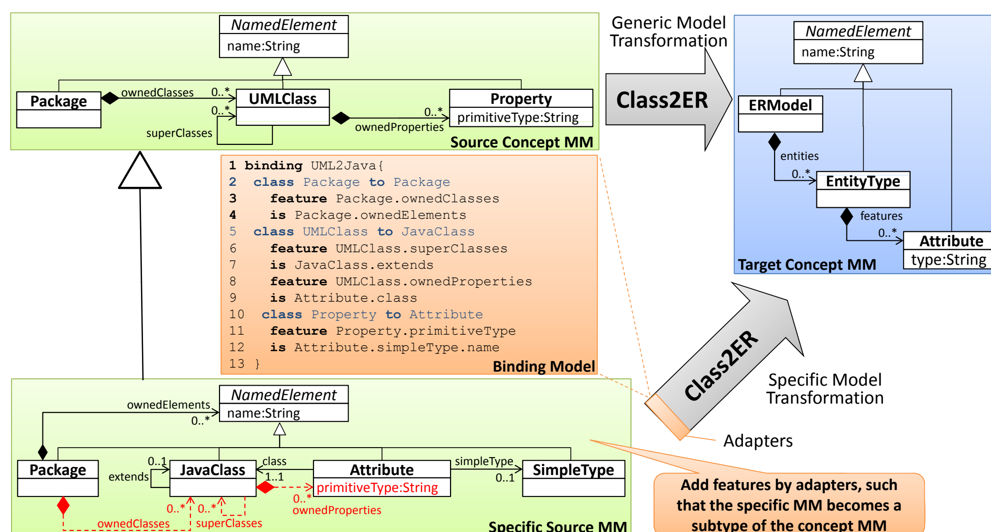


Figure 4: Exemplary Bindings between Heterogenous MMs

fashion. These are used to automatically derive adapters to actually resolve the heterogeneities as described in the following subsection. This corresponds to the idea of keeping the bindings as simple as possible and derive information required to actually resolve the heterogeneities automatically [ABM05]. Since the adapters are merely added to a generic transformation, but do not change it, imperative and declarative code is supported.

4.1 Automatic Resolution of Heterogeneities by Reasoning

In order to obtain a heterogeneity model including the heterogeneities as classified in Fig. 3(a) from the binding model, the values of the meta-features are compared (cf. Section 3.2). With the help of this heterogeneity model, in the subsequent transformation adaptation, first, corresponding different class names are resolved by rewriting using a HOT (cf. Fig. 5). To exemplify this, lines 1-5 of List. 1 depicts an exemplary rule of the generic model transformation of our running example as well as the rewritten rule in lines 1-5 of List. 2. Second, required *adapter templates* are instantiated in the HOT by calling predefined rules, which allow to resolve heterogeneities as described in Section 3.2. Adapter templates define how to resolve recurring heterogeneities in a generic way by depending only on the input provided by the heterogeneity model. Thus, they can be instantiated to establish a subtype relationship between the source concept MM and the specific MM. Adapters (representing the virtual view) are realized by helper functions in ATL (so-called attribute helpers) which allow to extend the specific MM by missing features required by the source concept MM. Whenever a feature is invoked by the transformation, which is only available in the source concept MM, this invocation is intercepted by a helper function which provides the adapter to the specific MM. To exemplify this, List. 2 (cf. lines 7-9) includes the helper function generated for the reference `Package.ownedClasses` of the MM concept. Thereby, a correspondingly named feature in terms of the specific MM concepts is introduced, thereby resolving naming differences. Without this adapter the transformation would break in line 4, since the class `Package` in the specific MM does not contain a reference named `ownedClasses`.

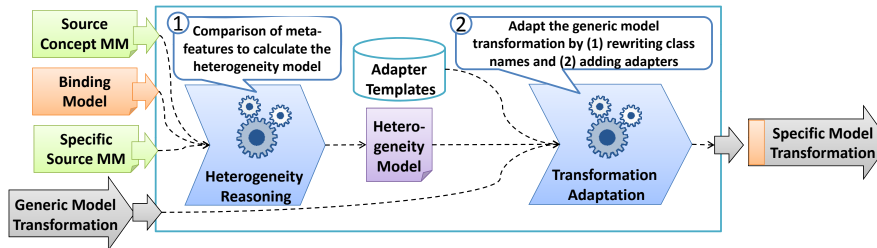


Figure 5: Overview of Reasoning Process

Listing 1: Exemplary ATL Rule of Generic Model Transformation

```
1 -- Extract of generic transformation
2 rule Package2ERModel {
3     from s: UML!Package
4     to t: ER!ERModel ( entities <- s.ownedClasses )
5 }
```

Listing 2: Exemplary ATL Rule of Specific Model Transformation

```
1 -- Extract of specific transformation -> only class names have been changed
2 rule Package2ERModel {
3     from s: Java!Package
4     to t: ER!ERModel ( entities <- s.ownedClasses )
5 }
6
7 -- Added helper function to resolve the heterogeneity, realizing the adapter
8 helper context Java!Package def : ownedClasses : Sequence(Java!JavaClass) =
9   self.ownedElements -> select(x| x.oclIsKindOf(Java!JavaClass));
```

4.2 Heterogeneity Reasoning and Adapter Generation in Action

This section applies the proposed approach to the example by presenting the automatically generated adapters for each binding of Fig. 4. Due to space limitations, only for the first binding the *adapter template* is shown, while for the rest only the resulting *concrete adapters* are given.

4.2.1 Binding `Package.ownedClasses` $\rightarrow$ `Package.ownedElements`

Discovered Differences: NamingDiff, TargetDiff

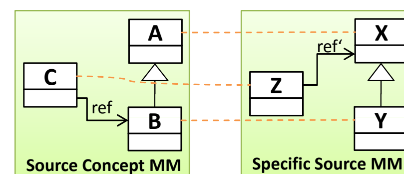


Figure 6: Target Difference

Target Difference. A target difference arises if two corresponding references as specified in the binding model point to different target classes. Such a binding is only valid iff the corresponding target class (cf. `Y` in Fig. 6) of the concept reference (cf. `ref` in Fig. 6) in the specific MM is a subclass of the target class (cf. `X` in Fig. 6) of the specific reference (cf. `ref′` in Fig. 6), since the set of elements referred by `ref` is then a subset of the elements referred by `ref′`. Considering our running example, a target difference exists between the references `Package.ownedClasses` and `Package.ownedElements`. Since this binding satisfies the above condition, it is valid. Therefore, the adaptation can be fulfilled by a helper function computing the corresponding subset (cf. List. 3).

Listing 3: Adapter for Resolving the Naming Difference & Target Difference

```
helper context Java!Package def : ownedClasses : Sequence(Java!JavaClass) =
  self.ownedElements -> select(x| x.oclIsKindOf(Java!JavaClass));
```

List. 4 shows the template for producing such adapters. Thereby the OCL context is set to the specific source class of the reference (`<specificRef.owner>`). The helper has to be named equally to the feature in the concept model (`conceptRef.name>`) and the return type has to correspond to the type in the specific MM. In order to get this return type we need to first get the target class of the reference in the concept model. This target class is then used to resolve the specific target class by means of the binding model (denoted in the template by `<conceptRef.type.resolve>`). In order to select only elements of the corresponding subclass, we select those elements that are direct and indirect instances of the equivalent type in the specific MM.

Listing 4: Adapter Template for Target Difference

```
helper context <specificRef.owner> def : <conceptRef.name> : <conceptRef.type.resolve> =
    self.<specificRef.name> -> select(x| x.oclIsKindOf(<conceptRef.type.resolve>));
```

4.2.2 Binding `UMLClass.superClasses` → `JavaClass.extends`

Discovered Differences: NamingDiff, MultiplicityDiff

Multiplicity Difference. Multiplicity differences arise in case of different lower or upper bounds of features. To ensure a correct resolution of a multiplicity difference, several preconditions must hold. In particular, the lower bound of the specific feature must be greater or equal than the lower bound of the feature in the concept MM. This is since the generic transformation expects at least a certain number of elements (i.e., the lower bound), which could lead to an error if the lower bound is underrun. In the context of the running example, the lower bounds of the references `UMLClass.superClasses` and `JavaClass.extends` are both 0, i.e., the condition is fulfilled and the binding is valid. To resolve a multiplicity difference, it is furthermore of special interest if the upper bound is set to exactly one, i.e., only a single element might be returned, or to a value greater than one, i.e., a collection of elements might be returned. If a feature returning a single element in the specific model should be bound to a feature returning a collection, the element needs to be wrapped into a collection, excluding `OclUndefined` in case the feature is unset. This resolution strategy is needed in the context of our running example to resolve the heterogeneity between the references `UMLClass.superClasses` and `JavaClass.extends` as can be seen in List. 5. If a feature returning a collection in the specific MM should be bound to a feature returning a single element in the concept MM, it is possible to return, e.g., the first element of the collection to ensure syntactic correctness, although this does not reflect the full semantics of the specific MM's feature, since elements are discarded.

Listing 5: Adapter for Multiplicity Difference

```
helper context Java!JavaClass def : superClasses : Sequence(Java!JavaClass) =
    Sequence{self.extends} -> excluding(OclUndefined);
```

4.2.3 Binding `UMLClass.ownedProperties` → `Attribute.class`

Discovered Differences: NamingDiff, DirectionDiff, MultiplicityDiff, ContainmentDiff

Direction Difference. A direction difference occurs, if a reference has the inverse direction of another one. In such situations, logic is needed to compute from one reference the corresponding inverse reference. In the context of our example the reference `UMLClass.ownedProperties` of the concept MM has to be expressed by means of the reference `Attribute.class` of the specific MM. Since `UMLClass.ownedProperties` is assumed to return a set of referred `Properties`, the corresponding set of `Attributes` has to be computed from the information provided by the reference `Attribute.class`. For this purpose, first `allInstances()` of `Attributes` are queried, and then filtered by an appropriate OCL condition (`select(a| a.class = self)`) as shown in List. 6. Please note, that in case that the inverse reference has an upper bound greater than 1, the object (i.e., `self`) has to be contained in the set of objects returned by the reference (e.g., `select(a| a.classes->including(self))`. Furthermore, since the multiplicity of the inverse end of the reference `Attribute.class` is unknown, the multiplicity difference that occurred at first sight no longer has any effect.

Listing 6: Adapter for Direction Difference

```
helper context Java!JavaClass def :  ownedProperties : Sequence(Java!Attribute) =
  Java!Attribute.allInstances() -> select(a| a.class = self);
```

Containment Difference. Ecore-based MMs allow to define a containment hierarchy of MM elements by corresponding references which may not only be accessed in the forward direction (i.e., parent → children), but also in the backward direction (i.e., children → parent). For this purpose, ATL provides access to the parent of an element by calling `refImmediateComposite`. Thus, if there are differences in the containment hierarchies of the concept MM and the specific MM, a corresponding attribute helper named `refImmediateComposite` has to be defined. In the context of the running example an attribute helper has to be defined for the class `Attribute`. To realize the parent access in the specific MM a corresponding path must exist in the binding model, enabling the derivation of the adapter as shown in List. 7.

Listing 7: Adapter for Containment Difference

```
helper context Java!Attribute def :  refImmediateComposite() : Java!JavaClass =
  self.class;
```

4.2.4 Binding `Property.primitiveType` → `Attribute.simpleType.name`

Discovered Differences: NamingDiff, ContextDiff

Context Difference. A context difference arises if corresponding features are owned by different classes, e.g., the attribute `Property.primitiveType` in the concept MM corresponds to the attribute `SimpleType.name`. Since the context of these attributes differs, the binding model has to specify how to find the attribute in the specific MM that corresponds to the attribute of the concept MM, i.e., a corresponding path specification is required (cf. `simpleType.name` in the binding model). The resulting adapter code is shown in List. 8. The example assumes only single-valued references, but in general also multi-valued references might occur on the specified path. In this case, suitable `collect` operations are needed.

Listing 8: Adapter for Context Difference

```
helper context Java!Attribute def : primitiveType : String =
    if self.simpleType <> OclUndefined then self.simpleType.name else OclUndefined endif;
```

4.2.5 Remaining Classification Differences

In the following, we shortly elaborate on the resolution of the not yet discussed differences.

Order Difference. Order Differences occur if one feature is defined to be ordered whereas the other one is not. Thus, the underlying implementation has to either maintain ordering information (e.g., OCL type `Sequence`) or omit it (OCL type `Set`), whereby ATL uses sequences per default. Nevertheless, in general, to resolve order differences, corresponding collection casts (`asSequence()`, `asSet()`) might be introduced.

Datatype Difference. Finally, datatype differences occur if the datatypes of specified attributes differ. To resolve this kind of heterogeneity, appropriate datatype casts might be used, if the involved datatypes are compatible.

4.2.6 Composition of Adapters

After discussing the adapters in isolation, the resolution of more complex scenarios comprising several differences at once demands for a composition of adapters. This composition has to follow certain rules. For *attributes*, composition of adapters is only allowed in the following order (evaluation starts from right to left as detailed below): $NamingDiff \circ OrderDiff \circ MultiplicityDiff \circ DatatypeDiff \circ ContextDiff$. To exemplify this, Fig. 7 shows a simple example requiring the composition of three adapters to resolve a naming, datatype and multiplicity difference. Following the order defined above, first the datatype difference (i.e., cast from `Int` to `Real`), followed by the multiplicity difference (i.e., wrapping a single element into a set) and finally, the naming difference is resolved.

Concerning *references*, the situation is slightly different: $NamingDiff \circ ((OrderDiff \circ MultiplicityDiff \circ TargetDiff \circ ContextDiff) \parallel DirectionDiff) \parallel ContainmentDiff$. Although the corresponding differences (i.e., NamingDiff – ContextDiff) are resolved analogously, the resolution of direction differences and containment differences is independent of any other difference and thus, require isolated adapters (cf. parallel composition operator).

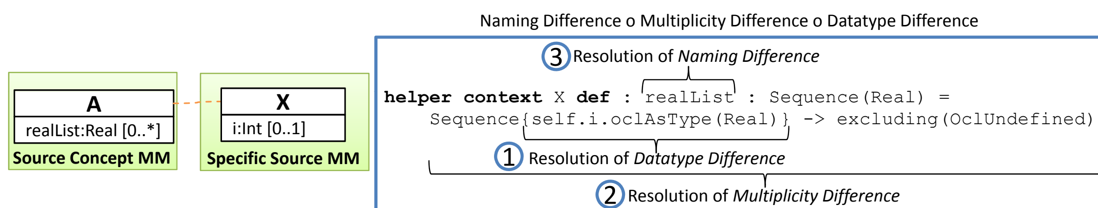


Figure 7: Composition of Adapters

5 Related Work

In this section, we relate our approach to existing transformation reutilization approaches, that either adapt the transformations, or adapt the MMs.

Adaptation of Transformations. The adaptation of transformations to evolved MMs has been presented in [LBNK09, MEM10, GD10]. While Méndez et al. [MEM10] provide a classification of typical evolution scenarios and their impact on transformations, the authors of [LBNK09]

and [GD10] propose solutions for typical evolution scenarios (e.g., extract superclass). Both approaches assume the availability of a difference model, comprising the changes that occurred between two versions of a MM, which is used to semi-automatically derive potential adaptation rules for the model transformation. In contrast to our approach, their focus is on evolution-specific scenarios, e.g., renaming, add/delete of attributes or references, but they do not consider reusing transformations for independent MMs exhibiting arbitrary heterogeneities.

Adaptation of MMs. Instead of adapting the transformations, another possibility is to adapt the MMs, as done in [SMM+10], for reusing transformations. Although following the same idea of adapting the specific MMs making them subtypes of the concept MMs, the adaptation has to be done manually by providing aspects defined in Kermeta[2]. Therefore, the burden of resolving heterogeneities is entirely left to the transformation reuser. Furthermore, the approach of [SMM+10] fully depends on a transformation language's capability to change the MM by aspect-orientation, which is specific to Kermeta.

6 Critical Discussion and Future Work

In this paper we have presented an automatic approach to resolve common heterogeneities when binding a concept MM to a specific MM for the purpose of reusing a generic transformation. On critically reflecting the presented approach, three main points remain for future work.

Handling Heterogeneities between Classes. The classification of Fig. 3(a) presents heterogeneities that might occur between features, for which we presented adapters. With the exception of naming differences, we did not deal with heterogeneities between classes. We omitted these heterogeneities since they currently result in complex rewriting rules for the transformation logic. For tackling these kinds of heterogeneities in the same way as heterogeneities between features, a way has to be found to express a virtual view on classes in ATL. Finally, resolution of semantic heterogeneities is totally left to the transformation designer.

Reusing Transformations for Specific Target MMs. Our approach focuses on adaptations concerning MMs in the source domain. The application to target MMs is not supported, since it is not possible to query the target model to provide virtual features. Thus, further mechanisms have to be explored for adapting the transformations regarding the target domain.

Specialization of Constraints. The focus of this paper was to adapt transformations specified for concept MMs such that they are also applicable to specific MMs. Nevertheless, not only the transformations have to be adapted, but also constraints on the concept MMs have to be translated for specific MMs. Imagine the case that there are stronger constraints on the source concept MM than on the specific MM. Thus, only a subset of the instances of the specific MM may be a valid input for the transformation. However, to exactly identify this subset, the constraints of the concept MMs have to be translated to constraints operating on the specific MMs.

Bibliography

[ABM05] Y. An, A. Borgida, J. Mylopoulos. Inferring complex semantic mappings between relational tables and ontologies from simple correspondences. *Proc. of OTM'05*,

[2] http://www.kermeta.org

pp. 1152–1169, 2005.

[CGL11] J. S. Cuadrado, E. Guerra, J. de Lara. Generic Model Transformations: Write Once, Reuse Everywhere. In *Proc. of ICMT'11*. 2011.

[GD10] J. Garcia, O. Díaz. Adaptation of transformations to metamodel changes. In *Desarrollo de Software Dirigido por Modelos*. Pp. 1–9. 2010.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming* 72(1-2):31–39, 2008.

[LBNK09] T. Levendovszky, D. Balasubramanian, A. Narayanan, G. Karsai. A Novel Approach to Semi-automated Evolution of DSML Model Transformation. In *Proc. of SLE'09*. 2009.

[Len02] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. of PODS'02*. 2002.

[LG10] J. de Lara, E. Guerra. Generic Meta-modelling with Concepts, Templates and Mixin Layers. In *Proc. of MoDELS'10*. 2010.

[LN07] F. Legler, F. Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. In *Proc. of BTW'07*. 2007.

[MEM10] D. Méndez, A. Etien, R. Muller, Alexis nad Casallas. Towards Transformation Migration After Metmodel Evolution. In *Proc. of ME @ MoDELS'10*. 2010.

[MV02] P. J. Mosterman, H. Vangheluwe. Guest editorial: Special issue on computer automated multi-paradigm modeling. *TOMACS Journal* 12(4):249–255, 2002.

[SJ07] J. Steel, J.-M. Jézéquel. On model typing. *SoSyM Journal* 6(4):401–413, 2007.

[SMM+10] S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, J.-M. Jézéquel. Reusable model transformations. *SoSyM Journal*, pp. 1–15, 2010.

[TCJ10] M. Tisi, J. Cabot, F. Jouault. Improving Higher-Order Transformations Support in ATL. In *Proc. of ICMT'10*. 2010.

[Vis01] E. Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science* 57:109–143, 2001.

[WKK+10a] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In *Proc. of ICMT'10*. 2010.

[WKK+10b] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger. Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities. In *Proc. of MDI @ MoDELS'10*. 2010.