A Framework for Building Mapping Operators Resolving Structural Heterogeneities*

Gerti Kappel[1], Horst Kargl[1], Thomas Reiter[2], Werner Retschitzegger[2], Wieland Schwinger[3], Michael Strommer[1], and Manuel Wimmer[1]

[1] Institute of Software Technology and Interactive Systems
Vienna University of Technology, Austria
{kappel|wimmer|kargl|strommer}@big.tuwien.ac.at
[2] Information Systems Group
Johannes Kepler University Linz, Austria
{reiter|retschitzegger}@ifs.uni-linz.ac.at
[3] Department of Telecooperation
Johannes Kepler University Linz, Austria
wieland.schwinger@jku.ac.at

Abstract. Seamless exchange of models among different modeling tools increasingly becomes a crucial prerequisite for the success of model-driven engineering. Current best practices use model transformation languages to realize necessary mappings between concepts of the metamodels defining the modeling languages supported by different tools. Existing model transformation languages, however, lack appropriate abstraction mechanisms for resolving recurring kinds of structural heterogeneities one has to primarily cope with when creating such mappings.
We propose a framework for building reusable mapping operators which allow the automatic transformation of models. For each mapping operator, the operational semantics is specified on basis of Colored Petri Nets, providing a uniform formalism not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations, thus facilitating understanding and debugging. To demonstrate the applicability of our approach, we apply the proposed framework for defining a set of mapping operators which are intended to resolve typical structural heterogeneities occurring between the core concepts usually used to define metamodels.

1 Introduction

Interoperability between modeling tools. With the rise of Model-Driven Engineering (MDE) [20] models become the main artifacts of the software development process. Hence, a multitude of modeling tools is available supporting different tasks, such as model creation, model simulation, model checking, model

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.0

transformation, and code generation. Seamless exchange of models among different modeling tools increasingly becomes a crucial prerequisite for effective model-driven engineering. Due to lack of interoperability, however, it is often difficult to use tools in combination, thus the potential of MDE cannot be fully utilized. For achieving interoperability in terms of transparent model exchange, current best practices (cf., e.g. [21]) comprise creating model transformations based on mappings between concepts of different tool metamodels, i.e., the metamodels describing the modeling languages supported by the tools.

Problem Statement. We have followed the aforementioned approach in various projects such as the ModelCVS project [12] focusing on the interoperability between legacy case tools (in particular CA's AllFusion Gen) with UML tools and the MDWEnet project [1] trying to achieve interoperability between different tools and languages for web application modeling. The prevalent form of heterogeneity one has to cope with when creating such mappings between different metamodels is *structural heterogeneity*, a form of heterogeneity well-known in the area of database systems [3, 14]. In the realm of metamodeling structural heterogeneity means that semantically similar modeling concepts are defined with different metamodeling concepts leading to differently structured metamodels. Current model transformation languages, e.g., the OMG standard QVT [17], provide no appropriate abstraction mechanisms or libraries for resolving recurring kinds of structural heterogeneities. Thus, resolving structural heterogeneities requires to manually specify partly tricky model transformations again and again which simply will not scale up having also negative influence on understanding the transformation's execution and on debugging.

Contribution. The contribution of this paper is twofold. First, a framework is proposed for building reusable mapping operators which are used to define so-called metamodel bridges. Such a metamodel bridge allows the automatic transformation of models since for each mapping operator the operational semantics is specified on basis of Colored Petri Nets. Colored Petri Nets provide a uniform formalism not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations, thus facilitating understanding and debugging. Second, to demonstrate the applicability of our approach we apply the proposed framework for defining a set of mapping operators subsumed in our mapping language called CAR. This mapping language is intended to resolve typical structural heterogeneities occurring between the core concepts usually used to define metamodels, i.e., class, attribute, and reference, as provided by the OMG standard MOF [16].

Structure. The rest of the paper is structured as follows. In Section 2 we introduce our framework for defining mapping operators in order to establish metamodel bridges. In Section 3 the mapping language CAR is presented. Section 4 discusses related work, and finally, in Section 5 a conclusion and a discussion of future work is given.

2 Metamodel Bridging at a glance

In this section, we describe the conceptual architecture of the proposed *Metamodel Bridging Framework* in a by-example manner. The proposed framework provides two views on the metamodel bridge, namely a *mapping view* and a *transformation view* as illustrated in Figure 1.

At the mapping view level, the user defines mappings between elements of two metamodels (M2). Thereby a mapping expresses also a relationship between model elements, i.e., the instances of the metamodels [4]. In our approach, we define these mappings between metamodel elements with mapping operators standing for a processing entity encapsulating a certain kind of transformation logic. A mapping operator takes as input elements of the source model and produces as output semantically equivalent elements of the target model. Thus, it declaratively describes the semantic correspondences on a high-level of abstraction. A set of applied mapping operators defines the mapping from a left hand side (LHS) metamodel to a right hand side (RHS) metamodel further on subsumed as *mapping model*.

For actually exchanging models between different tools, the mapping models have to be executed. Therefore, we propose, in addition to the mapping view, a transformation view which is capable of transforming models (M1) from the LHS to the RHS on basis of Colored Petri Nets [6].

2.1 The Mapping View

For defining mapping operators and consequently also for building mapping models, we are using a subset of the UML 2 component diagram concepts. With this formalism, each mapping operator can be defined as a dedicated component, representing a modular part of the mapping model which encapsulates an arbitrary complex structure and behavior, providing well-defined interfaces to the environment. The resulting components are collected in a mapping operator library which can be seen as a domain-specific language for bridging metamodels. The user can apply the mapping operators expressed as components in a plug&play manner, i.e., only the connections to the provided and required interfaces have to be established manually.

Our motivation for using UML 2 component diagrams for the mapping view is the following. First, many software engineers are likely to be familiar with the UML component diagram notation. Second, the provided and required interfaces which can be typed, enable the composition of mapping operators to resolve more complex structural heterogeneities. Third, the clear separation between *black-box* view and *white-box* view of components allows switching between a high-level mapping view and a detailed transformation view, covering the operational semantics, i.e., the transformation logic, of an operator.

Anatomy of a mapping operator. Each mapping operator (as for example shown in the mapping model of Figure 1) has *input ports* with required interfaces (left side of the component) as well as *output ports* with provided interfaces (right side of the component). Because each mapping operator has its own *trace model*,

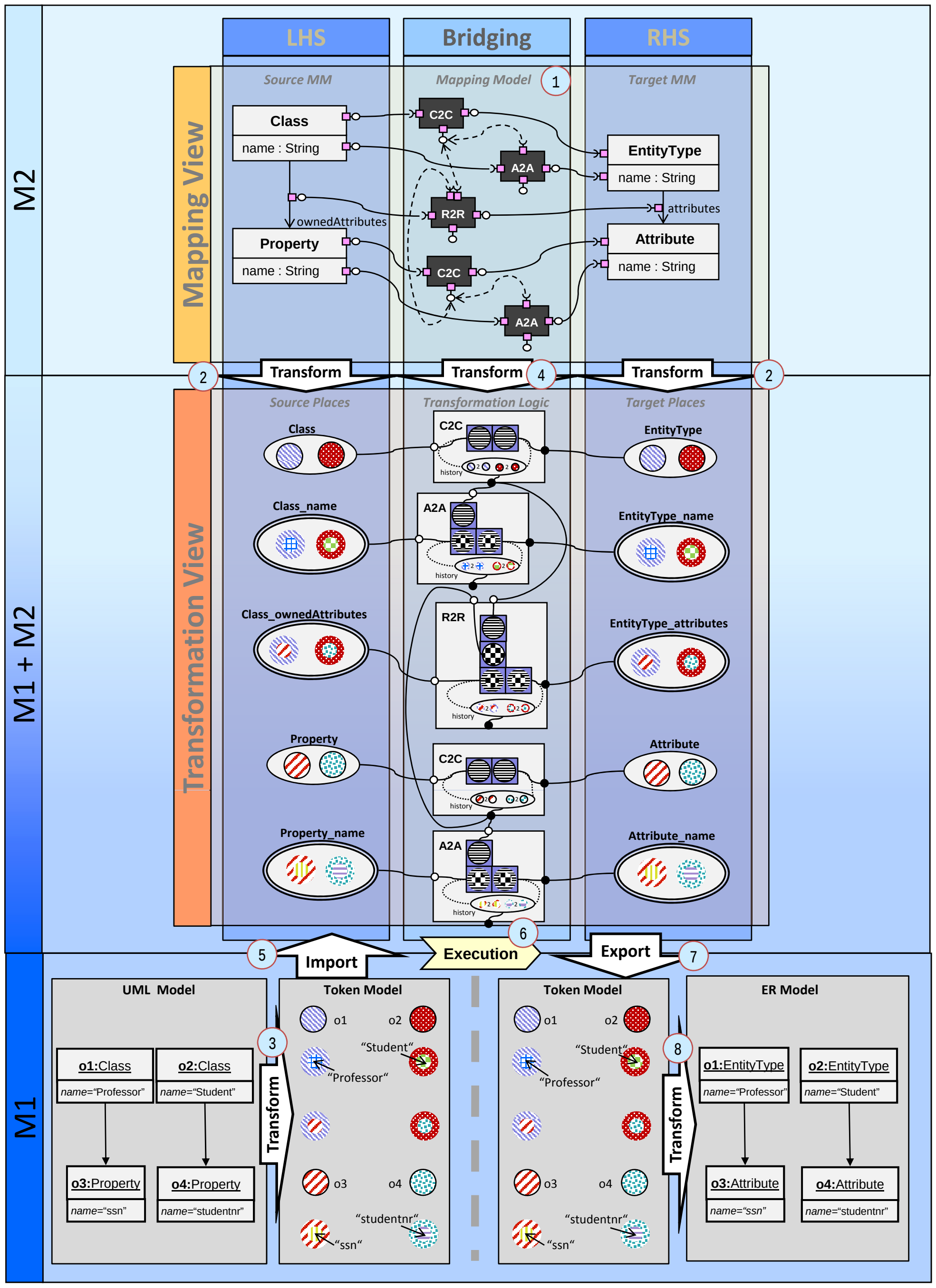


Fig. 1. Metamodel Bridging Framework by-example

i.e., providing a log about which output elements have been produced from which input elements, an additional *providedContext port* with a corresponding interface is available on the bottom of each mapping operator. This port can be used by other operators to access the trace information for a specific element via *requiredContext ports* with corresponding interfaces on top of the operator.

In the mapping view of Figure 1 (cf. step 1), an example is illustrated where a small part of the metamodel of the UML class diagram (cf. source metamodel) is mapped to a part of the metamodel of the Entity Relationship diagram (cf. target metamodel). In the mapping view, source metamodel elements have provided interfaces and target metamodel elements have required interfaces. This is due to the fact that in our scenario, models of the LHS are already available whereas models of the RHS must be created by the transformation, i.e., the elements of the LHS must be streamed to the RHS according to the mapping operators. Consequently, *Class* and *Property* of the source metamodel are mapped to *EntityType* and *Attribute* of the target metamodel with *Class2Class* (*C2C*) operators, respectively. In addition, the C2C operator owns a providedContext port on the bottom of the component which shall be used by the requiredContext ports of the appropriate *Attribute2Attribute* (*A2A*) and *Reference2Reference* (*R2R*) operators to preserve validity of target models. In particular, with this mechanism it can be ensured that values of attributes are not transformed before their owning objects has been transformed and links as instances of references are not transformed before the corresponding source and target objects have been transformed.

2.2 The Transformation View

The transformation view is capable of executing the defined mapping models. For this, so called *transformation nets* [18] are used which are a special kind of Colored Petri Nets consisting of source places at the LHS and target places at the RHS. Transitions between the source and target places describe the transformation logic located in the bridging part of the *transformation net* as shown in Figure 1.

Transformation nets provide a suitable formalism to represent the operational semantics of the mapping operators, i.e., the transformation logic defined in the white-box view of the component due to several reasons. First, they enable the execution of the transformation thereby generating the target model out of the source model, which favors also debugging of a mapping model. Second, it allows a homogeneous representation of all artefacts involved in a model transformation (i.e., models, metamodels, and transformation logic) by means of a simple formalism, thus being especially suited for gaining an understanding of the intricacies of a specific metamodel bridge.

In the next paragraphs, we discuss rules for assembling metamodels, models, and mapping models into a single transformation net and how the transformation can actually be executed.

Places represent Metamodels. First of all, places of a transformation net are used to represent the elements of the source and target metamodels (cf. step 2 in Figure 1). In this respect, we currently focus on the three major building blocks of metamodels (provided, e.g. by meta-metamodels such as MOF), namely *class*, *attribute*, and *reference*. In particular, classes are mapped onto one-colored places whereby the name of the class becomes the name of the place. The notation used to visually represent one-colored places is a circle or oval as traditionally used in Petri Nets. Attributes and references are represented by two-colored places, whereby the name of the containing class plus the name of the reference or of the attribute separated by an underline becomes the name of the place (cf. e.g. *Class_name* and *Class_ownedAttributes* in Figure 1). To indicate that these places contain two-colored tokens, the border of two-colored places is double-lined.

Tokens represent Models. The tokens of the transformation net are used to represent the source model which should be transformed according to the mapping model. Each element of the source model is expressed by a certain token, using its color as a means to represent the model element's identity in terms of a String (cf. step 3 in Figure 1). In particular, for every object, a one-colored token is produced, whereby for every link as an instance of a reference, as well as for every value of an attribute, a two-colored token is produced. The *fromColor* for both tokens refers to the color of the token that corresponds with the containing object. The *toColor* is given by the color of the token that corresponds with the referenced target object or the primitive value, respectively. Notationally, a two-colored token consist of a ring (carrying the fromColor) surrounding an inner circle (depicting the toColor).

Considering our example shown in Figure 1, the objects *o1* to *o4* of the UML model shown in the M1-layer are transformed into one-colored tokens. Each one-colored token represents an object identity, pointed out by the object name beneath the token. E.g., the tokens with the inner-color *"Student"* and *"Professor"* have the same outer-color as their containing objects and the token which represents the link between object *o1* and *o3* has the same outer-color as the token representing object *o1* and the inner-color corresponds to the one-colored token representing object *o3*.

Transitions represent Mapping Models. The mapping model is expressed by the transformation logic of the transformation net connecting the source and the target places (cf. Step 4 in Figure 1). In particular, the operational semantics of the mapping operators are described with transitions, whereby the behavior of a transition is described with the help of preconditions called *query-tokens* (LHS of a transition) and postconditions called *generator-tokens* (RHS of a transition). Query-tokens and generator-tokens can be seen as templates, simply visualized as color patterns, describing a certain configuration of tokens. The pre-condition is fulfilled and the transitions fires, if the specified color pattern described by the query-tokens matches a configuration of available input tokens. In this case, the postcondition in terms of the generator-tokens produces the required output tokens representing in fact the necessary target model concepts.

In the following, the most simple mapping operators used in our example are described, namely C2C, A2A, and R2R.

C2C. The white-box view of the C2C operators as shown in the transformation view of Figure 1 ensures that each object instantiated from the class connected to the input port is streamed into the mapping operator, the transition matches a single token from the input port, and streams the exact token to the output port. This is expressed in the transition by using the most basic query-token and generator-token combination, both having the same color pattern. In addition, every input and output token combination is saved in a history place representing the trace model which is connected to the *providedContext* port and can be used as trace information by other operators.

A2A. The white-box view of the A2A operator is also illustrated in the bridging part of the transformation view in Figure 1. Two-colored tokens representing attribute values are streamed via the input port into the mapping operator. However, a two-colored token is only streamed to the output port if the owning object of the value has been already transformed by a C2C operator. This is ensured in that the transition uses the same color pattern for the one-colored query-token representing the owning object streamed from the *requiredContext* port and for the outer color of the two-valued query-token representing the containing object of the attribute value. Only, if a token configuration matches this pre-condition, the two-colored token is streamed via the generator-token to the output port. Again, the input tokens and the corresponding output tokens are stored in a history place which is connected to the *providedContext* port.

R2R. The white-box view of the R2R operator shown in the transformation view of Figure 1 consists of three query-tokens, one two-colored query-token representing the link and two one-colored query-tokens for accessing trace information from C2C operators. The two-colored query-token must have the same inner and outer colors as provided by the C2C trace information, i.e., the source and target objects must be already transformed. When this precondition is satisfied by a token configuration, the two-colored token representing the link is streamed via the generator-token to the output port.

Execution of the transformation logic. As soon as the metamodels are represented as places, which are furthermore marked with the respective colored tokens depicting the concepts of the source model (cf. step 5 in Figure 1), the transformation net can be started. Now, tokens are streamed from the source places over the transitions into the target places (cf. step 6 in Figure 1).

Considering our running example, in a first step only the transitions of the C2C operators are able to fire due to the dependencies of the A2A and R2R operators. Hence, tokens from the places *Class* and *Property* are streamed to the appropriate places of the RHS and all combinations of the queried input and generated output tokens are stored in the trace model of the C2C operator. As soon as all necessary tokens are available in the trace model, depending operators, i.e., the A2A and R2R operators, are also able to fire.

Generation of the target model. After finishing the transformation, the tokens from the target places can be exported (cf. step 7 in Figure 1) and transformed back into instances of the RHS metamodel (cf. step 8 in Figure 1).

In our example, the one-colored tokens *o1* to *o4* contained in the target places are transformed back into objects of type *EntityType* and *Attribute*. The two-colored tokens which represent attribute values, e.g., *"Professor"* and *"Student"*, are assigned to their containing objects, e.g., *o1* and *o2* whereas *"ssn"* and *"studentnr"* are assigned to *o3* and *o4*. Finally, the two-colored tokens which represent links between objects are transformed back into links between *o1* and *o3*, as well as between *o2* and *o4*.

3 Mapping Operators of the CAR Mapping Language

3.1 Motivating Example

Based on experiences gained in various interoperability projects [9, 10, 23, 24] it has been shown that although most meta-metamodels such as MOF offer only a core set of language concepts for defining metamodels, numerous structural heterogeneities occur when defining modeling languages.

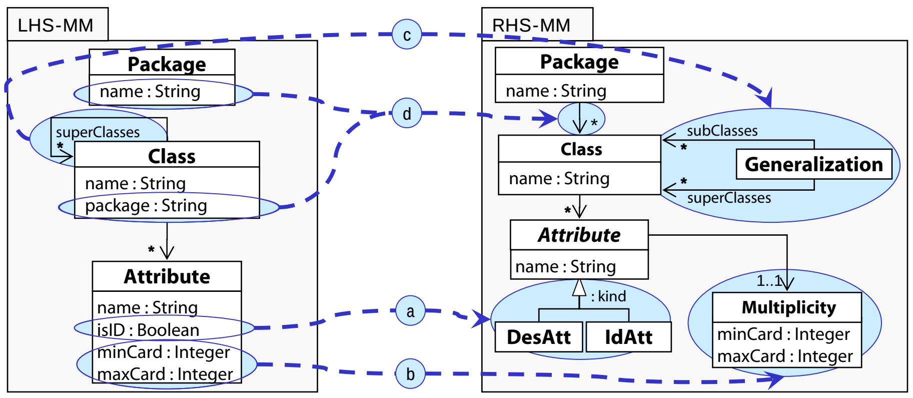
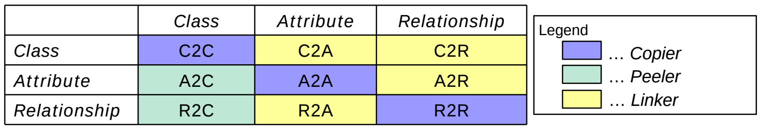


Fig. 2. Structural Heterogeneities Between Metamodels - Example

As an example for structural metamodel heterogeneity consider the example shown in Figure 2. Two MOF-based metamodels represent semantically equivalent core concepts of the UML class diagram in different ways. Whereas the LHS metamodel uses only a small set of classes, the RHS metamodel employs a much larger set of classes thereby representing most of the UML concepts which are in the LHS metamodel implicitly defined as attributes or references explicitly as first class citizens. More specifically, four structural metamodel heterogeneities can be found which require mapping operators going beyond the simple *one-to-one* mappings provided by the mapping operators in Section 2.

3.2 CAR Mapping Language at a glance

At this time, we provide nine different core mapping operators for resolving structural metamodel heterogeneities as depicted in Figure 2. These nine mapping operators result from the possible combinations between the core concepts of meta-metamodels, namely *class*, *attribute*, and *reference*, which also led to the name of the CAR mapping language. These mapping operators are designed to be declarative and bi-directional and it is possible to derive executable transformations based on transformation nets. One important requirement for the CAR mapping language is that it should be possible to reconstruct the source models from the generated target models, i.e., any loss of information during transformation should be prevented. In Figure 3, the mapping operators are divided according to their functionality into the categories *Copier*, *Peeler*, and *Linker* which are explained in the following.



	Class	*Attribute*	*Relationship*
Class	🟦 C2C	🟨 C2A	🟨 C2R
Attribute	🟩 A2C	🟦 A2A	🟨 A2R
Relationship	🟩 R2C	🟨 R2A	🟦 R2R

Legend
🟦 ... *Copier*
🟩 ... *Peeler*
🟨 ... *Linker*

Fig. 3. CAR Mapping Operators

Copier. The diagonal of the matrix in Figure 3 depicts the symmetric mapping operators of the CAR mapping language which have been already discussed in Section 2. The term symmetric means that the input and outport ports of the left side and the right side of the mapping operators are of the same type. This category is called *Copier*, because these mapping operators copy one element of the LHS model into the RHS model without any further manipulations.

Peeler. This category consists of mapping operators which create new objects by "peeling"[1] them out of values or links. The A2C operator bridges heterogeneities which are resulting from the fact that a concept is expressed as an attribute in one metamodel and as a class in another metamodel. Analogously, a concept can be expressed on the LHS as a reference and on the RHS as a class which can be bridged by a R2C operator.

Linker. The last category consists of mapping operators which either link two objects to each other out of value-based relationships (cf. A2R and R2A operator) or assign values or links to objects for providing the inverse variants of the A2C and R2C operators (cf. C2A and C2R operator).

To resolve the structural heterogeneities depicted in Figure 2, in the following subsections the necessary mapping operators are discussed in detail, comprising

[1] Note that the term "peeling" is used since when looking at the white-box view the transformation of an attribute value into an object requires in fact to generate a one-colored token out of a two-colored token.

besides a variation of the C2C operator mainly mapping operators falling into the above mentioned peeler and linker category.

3.3 Conditional C2C Mapping Operator

Problem. In MOF-based metamodels, a property of a modeling concept can be expressed via a discriminator of an inheritance branch or with an additional attribute. An example for this kind of heterogeneity can be found in Figure 2(a), namely between *Attribute.isID* on the LHS and the subclasses of the class *Attribute* on the RHS. This heterogeneity is not resolvable with a primitive C2C operator per se, because one class on the LHS corresponds to several classes on the RHS whereby each mapping is only valid under a certain condition. On the model level, this means that a set of objects has to be splitted into several subsets based on the object's attribute values.

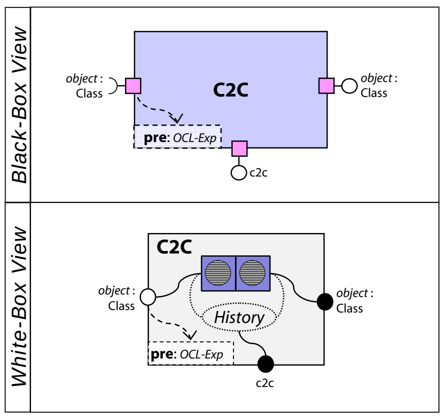
Solution. To cope with this kind of heterogeneity, the C2C operator has to be extended with the capability of splitting a set of objects into several subsets. For this we are annotating the C2C operator with OCL-based preconditions assigned to ports as depicted in Figure 4(a). These preconditions supplement the query-tokens of the transitions by additionally allowing to specify constraints on the source model elements. The reason for introducing this additional mechanism is that the user should be able to configure the C2C operator without having to looking into the white-box view of the operator, realizing its basic functionality.

Example Application. In the example shown in Figure 5, we can apply two C2C mapping operators with OCL conditions, one for mapping *Attribute* to *DesAtt* with the precondition *Attribute.isID = false*, and one for mapping *Attribute* to *IdAtt* with the precondition *Attribute.isID = true*. In addition, this example shows a way how mappings can be reused within a mapping model by allowing inheritance between mappings. This mechanism allows to define certain mappings directly between *superClasses* and not for each *subClass* combination again and again (cf., e.g., the A2A mapping between the *Attribute.name* attributes). For more details about the inheritance mechanism see Subsection 3.7.

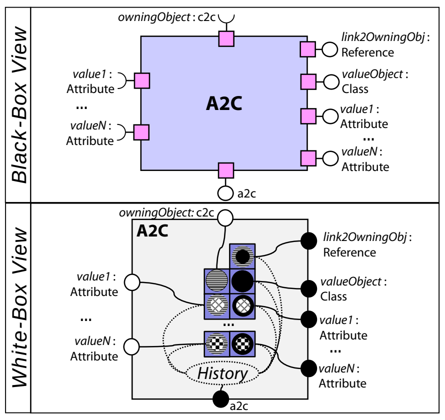
3.4 A2C Mapping Operator

Problem. In Figure 5(b), the attributes *minCard* and *maxCard*, which are part of the class *Attribute* at the LHS, are at the RHS part of a dedicated class *Multiplicity*. Therefore, on the instance level, a mechanism is needed to "peel" objects out of attribute values and to additionally take into account the structure of the LHS model in terms of the attribute's owning class when building the RHS model, i.e., instances of the class *Multiplicity* must be connected the corresponding instances of class *Attribute*.

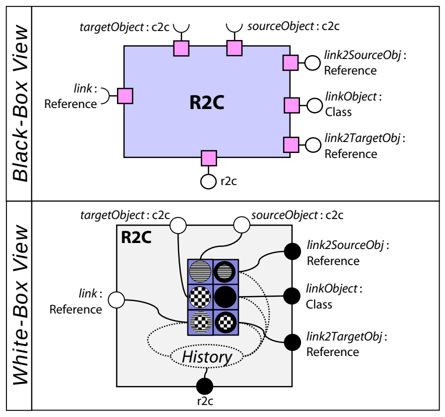
Solution. The black-box view of the A2C mapping operator as illustrated in Figure 4(b) consists of one or more required interfaces for attributes on the LHS depending on how many attributes are contained by the additional class, and has in minimum three provided interfaces on the RHS. The first of these interfaces is used to mark the reference which is responsible to link the two target



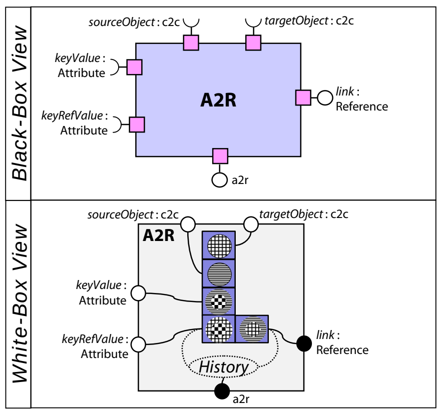
(a) Conditional C2C operator



(b) A2C operator



(c) R2C operator



(d) A2R operator

Fig. 4. Black-box and white-box views of CAR mapping operators

classes, the second is used to mark the class that should be instantiated, and the rest is used to link the attributes of the LHS to the RHS. Additionally, an A2C operator has a required interface to a C2C, because the source object is splitted into two target objects, thereby only one object is created by the A2C, the other has to be generated by an C2C operator which maps the LHS class to its corresponding target RHS class.

The white-box view of the A2C operator shown in Figure 4(b) comprises a transition consisting of at least two query-tokens. The first query-token guarantees that the *owningObject* has been already transformed by a C2C operator. The other query-tokens are two-colored tokens representing the attribute values which have as fromColor the same color as the first query-token. The post-condition of the transition consists of at least three generator-tokens. The second generator-token introduces a new color, i.e., this color is not used in the pre-condition part of the transition, and therefore, the generator-token produces a new object with an unique identity. The first generator-token is used for linking the newly created object appropriately into the target model and the other two-colored generator tokens are used to stream the values into the newly generated object by changing the fromColor of the input values.

Example Application. In Figure 5, the attributes *minCard* and *maxCard* are mapped to attributes of the class *Multiplicity*. Furthermore, the reference between the classes *Attribute* and *Multiplicity* is marked by the A2C mapping as well as the class *Multiplicity*. To assure that the generated *Multiplicity* objects can be properly linked to *Attribute* objects, the A2C mapping is in the context of the C2C mapping between the *Attribute* classes.

3.5 R2C Mapping Operator

Problem. In Figure 5(c), the reference *superClasses* of the LHS metamodel corresponds to the class *Generalization* of the RHS metamodel. This kind of heterogeneity requires an operator which is capable of "peeling" an object out of a link and to additionally preserve the structure of the LHS in terms of the classes connected by the relationships at the RHS.

Solution. The black-box view of the R2C mapping operator, as depicted in Figure 4(c), has one required interface on the left side for pointing to a reference. On the right side it has three provided interfaces, one for the class which stands for the concept expressed as reference on the LHS and two for selecting the references which are responsible to connect the object which has been peeled out of the link of the LHS into the RHS model. To determine the objects to which the peeled object should be linked, two additional required interfaces on the top of the R2C operator are needed for determining the corresponding objects of the source and target objects of the LHS.

The white-box view of the R2C mapping operator, as illustrated in Figure 4(c), consists of a pre-condition comprising three query-tokens. The input link is connected to a two-colored query-token, the fromColor corresponds to the query-token standing for the source object and the toColor corresponds to a query-token standing for the target object. The post-condition of the transition introduces a new color and is therefore responsible to generate a new object. Furthermore, two links are produced by the other generator-tokens for linking the newly generated object with the corresponding source and target the objects of the LHS.

Example Application. In Figure 5, the reference *superClasses* in the LHS metamodel is mapped to the class *Generalization* by an R2C operator. In addition, the references *subClasses* and *superClasses* are selected for establishing an equivalent structure on the RHS as existing on the LHS. For actually determining the *Class* objects which should be connected via *Generalization* objects, the R2C operator has two dependencies to C2C mappings. This example can be seen as a special case, because the reference *superClasses* is a reflexive reference, therefore both requiredContext ports of the R2C operator point to the same C2C operator.

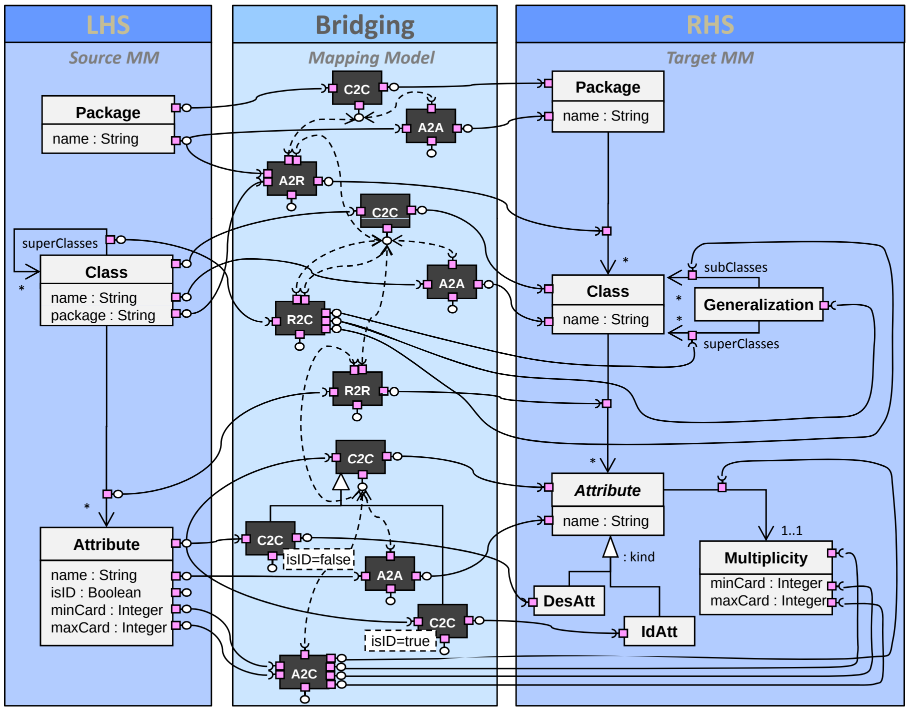


Fig. 5. Example resolved with CAR (Mapping View)

3.6 A2R Mapping Operator

Problem. The attribute vs. reference heterogeneity shown in Figure 2(d) resembles the well-known difference between value-based and reference-based relationships, i.e, corresponding attribute values in two objects can be used to "simulate" links between two objects. Hence, if the attribute values in two objects are equal, a link ought to be established between them.

Solution. For bridging the value-based vs. reference-based relationship heterogeneity, the A2R mapping operator as shown in Figure 4(d) provides on the LHS two interfaces, one for marking the *keyValue* attribute and another for marking the *keyRefValue* attribute. On the RHS, the operator provides only one interface for marking the reference which corresponds to the *keyValue/keyRefValue* attribute combination.

The white-box view of the operator comprises a transition which has four query-tokens. The first two ensure that the objects which are referencing each other on the LHS have been already transformed. The last two are the *keyValue* and *keyRefValue* query-tokens whereby the inner-color (representing the attribute values) is the same for both tokens. The generator-token of the transition produces one two-colored token by using the outer-color of the *keyRefValue* query-token as the outer-color and the outer-color of the *keyValue* query-token as the inner-color.

Example Application. In Figure 5, the A2R operator is used to map the *Package.name* attribute as the key attribute and the *Class.package* attribute as the keyRef attribute of the LHS metamodel to the reference between *Package* and *Class* on the RHS metamodel.

3.7 Inheritance for C2C Mappings

For reusing existing mappings, we introduce the possibility to define generalization relationships between C2C mappings. This means, the user can define general mappings between superclasses called *supermappings* and more specific mappings between *subclasses* called *submappings* which can be used to refine (i.e., the source and target types of the supermappings) and extend (i.e., new feature mappings can be introduced) the supermappings. As concrete syntax for generalization relationships between C2C mappings, we reuse the notation of UML generalization relationships between classes, i.e., a line with a hollow triangle as an arrowhead.

Note that we allow generalization relationships only for C2C mappings for inheriting feature-mappings which are dependent on C2C mappings such as symmetric mappings (A2A, R2R), or asymmetric mappings (A2C, R2C, A2R, and their inverse operators). This is due to the fact that C2C operators are responsible for providing the context information for all other CAR mapping operators.

One important constraint for generalization relationships between C2C operators is that if a generalization between two C2C operators is defined, the participating LHS classes of the supermappings and the submappings must be either in a generalization relationship or it must be actually the same class. Of course, the same constraint must hold on the RHS. These two constraints must be ensured, because the submappings inherit the feature mappings of the supermappings and therefore, the features of the superclasses must be also available on instances which are transformed according to the submappings.

Representing Inheritance within Transformation Nets. In the following, we discuss how C2C generalization relationships influence the generation of transformation nets and consequently the execution of the transformation logic. On overall design goal is naturally to express new language concepts at the black-box view – such as mapping generalizations in this case – as far as possible by means of existing transformation net mechanisms.

When we take a closer look on supermappings, we see that these mappings must provide the context, i.e., the trace model information, for all dependent mappings. This means, the supermappings must also provide context information about the transformation of indirect instances, e.g., for assigning attribute values of indirect instances when the attribute is contained by the superclass. Consequently, a supermapping is derived into a transformation component which contains the union of its own trace model for logging the transformation of direct instances of the superclass and the trace models of its submappings for logging the transformation of indirect instances. Therefore, the corresponding transformation components of the submappings are nested into the transformation component of the supermapping. For constructing the union of trace models of nested transformation components, each nested component gets an arc from its own trace model to the union trace model of the outer component. Mappings which depend on the supermapping are connected to the union trace model available on the outer component and mappings which are dependent on submappings are directly connected to the individual trace models of the nested components.

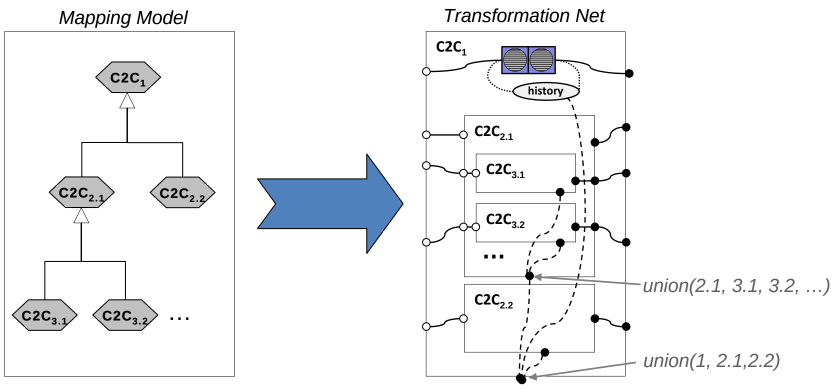


Fig. 6. Representing Inheritance Structures with Nested Transformation Components

Figure 6 illustrates the derivation of generalization relationships into transformation net components. The mapping $C2C_1$ of the Mapping Model shown on the LHS of Figure 6 is transformed into the outer component $C2C_1$, which consists of a transition for transforming direct instances and of two subcomponents

$C2C_{2.1}$ and $C2C_{2.2}$. In addition, the outer component provides a union trace model of the transformation components $C2C_1$, $C2C_{2.1}$, and $C2C_{2.2}$. Because the mapping $C2C_{2.1}$ has two submappings, the corresponding transformation component has also two sub-components $C2C_{3.1}$ and $C2C_{3.2}$. In addition, the component $C2C_{2.1}$ provides a union trace model of itself and the subcomponents $C2C_{3.1}$ and $C2C_{3.2}$.

The resulting transformation net for the CAR mapping model shown in Figure 5, which uses two generalization relationships, may be found in the Appendix A.

4 Related Work

With respect to our approach we can distinguish between two kinds of related work: first, related work in the field of model-driven engineering concerning the design of reusable model transformations, and second, related work in the field of ontology engineering concerning the usage of dedicated mapping languages for bridging structural heterogeneities.

4.1 Reusable Model Transformations

Generic Model Transformations. Typically model transformation languages, e.g., ATL [7] and QVT [17], allow to define transformation rules based on types defined as classes in the corresponding metamodels. Consequently, model transformations are not reusable and must be defined from scratch again and again with each transformation specification. One exception thereof is the approach of Varró et al. [22] who define a notion of defining generic transformations within their VIATRA2 framework, which in fact resembles the concept of templates in C++ or generics in Java. VIATRA2 also provides a way to implement reusable model transformations, although it does not foster an easy to debug execution model as is the case with our proposed transformation nets. In addition, there exists no explicit mapping model between source and target metamodel which makes it cumbersome to reconstruct the correspondences between the metamodel elements based on the graph transformation rules, only.

Transformation Patterns. Very similar to the idea of generic transformations is the definition of reusable idioms and design patterns for transformation rules described by Karsai et al. [2]. Instead of claiming to have generic model transformations, the authors propose the documentation and description of recurring problems in a general way. Thus, this approach solely targets the documentation of transformation patterns. Implementation issues how these patterns could be implemented in a generic way remain open.

Mappings for bridging metamodels. Another way of reuse can be achieved by the abstraction from model transformations to mappings as is done in our approach or by the ATLAS Model Weaver (AMW) [5]. AMW lets the user extend a generic so-called weaving metamodel, which allows the definition of simple correspondences between two metamodels. Through the extension of the weaving metamodel, one can define the abstract syntax of new weaving operators which roughly correspond to our mapping operators. The semantics of weaving operators are determined by a higher-order transformation that take a weaving model as input and generates model transformation code. Compared to our approach, the weaving models are compiled into low-level transformation code in terms of ATL which is in fact a mixture of declarative and imperative language constructs. Thus, it is difficult to debug a weaving model in terms of weaving operators, because they do not explicitly remain in the model transformation code. Furthermore, the abstraction of mapping operators from model transformations expressed in ATL seems more challenging compared to the abstraction from our proposed transformation net components.

4.2 Ontology Mapping for Bridging Structural Heterogeneities

In field of ontology engineering, several approaches exist which make use of high-level languages for defining mappings between ontologies (cf. [8] for an overview). For example, in Maedche et al. [15], a framework called *MAFRA* for mapping two heterogeneous ontologies is proposed. Within this framework, the mapping ontology called *Semantic Bridge Ontology* usually provides different ways of linking concepts from the source ontology to the target ontology. In addition to the Semantic Bridge Ontology, MAFRA provides an execution platform for the defined mappings based on services whereby for each semantic bridge type a specific service is available for executing the applied bridges. In [19], Scharffe et al. describe a library of so called *Ontology Mediation Patterns* which can be seen as a library of mapping patterns for integrating ontologies. Furthermore, the authors provide a mapping language which incorporates the established mapping patterns and they discuss useful tool support around the pattern library, e.g., for transforming ontology instances between different ontology schemas.

The main difference to our approach is that ontology mapping approaches are based on Semantic Web standards, such as *OWL* and *RDFS*, and therefore contain mapping operators for typical description logic related mapping problems, e.g., *union* or *intersection* of classes. We are bridging metamodels expressed in MOF, a language which has only a partially overlap with OWL or RDFS, leading to different mapping problems. Furthermore, in contrast to the ontology mapping frameworks, we provide a framework allowing to build new mapping operators by using well-known modeling techniques not only for defining the syntax but also for the operational semantics of the operators.

5 Conclusion and Future Work

In this paper we have introduced a framework allowing the definition of mapping operators and their application for building metamodel bridges. Metamodel bridges are defined by the user on a high-level mapping view which represents the semantic correspondences between metamodel elements and are tested and executed on a more detailed transformation view which also comprises the transformation logic of the mapping operators. The close integration of these two views and the usage of models during the whole bridging process further enhances understandability and the debugging of the defined mappings in terms of the mapping operators. The applicability of the framework has been demonstrated by implementing mapping operators for resolving common structural metamodel heterogeneities.

Future work will focus on making current matching engines for automatically finding correspondences between metamodels aware of the CAR mapping operators in combination with a supervised machine learning approach [13].

References

1. A. Vallecillo et al. MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods. In *Workshop Proc. of 7th Int. Conf. on Web Engineering (ICWE'07), Italy, July*, 2007.
2. A. Agrawal, A. Vizhanyo, Z. Kalmar, F. Shi, A. Narayanan, and G. Karsai. Reusable Idioms and Patterns in Graph Transformation Languages. In *Proc. of the Int. Workshop on Graph-Based Tools (GraBaTs'04), Italy*, 2004.
3. C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.
4. P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data, China*, 2007.
5. M. D. D. Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: a generic model weaver. In *Proc. of the 1re Journe sur l'Ingnierie Dirige par les Modles (IDM'05), France*, 2005.
6. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer, 1992.
7. F. Jouault and I. Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference, Jamaica*, 2006.
8. Y. Kalfoglou and W. M. Schorlemmer. Ontology Mapping: The State of the Art. In *Dagstuhl Seminar Proceedings: Semantic Interoperability and Integration*, 2005.
9. G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer. Lifting Metamodels to Ontologies - A Step to the Semantic Integration of Modeling Languages. In *9th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'06), Italy*, 2006.
10. G. Kappel, H. Kargl, G. Kramler, A. Schauerhuber, M. Seidl, M. Strommer, and M. Wimmer. Matching Metamodels with Semantic Systems - An Experience Report. In *Workshop Proc. of Datenbanksysteme in Business, Technologie und Web (BTW'07), Germany*, 2007.
11. G. Kappel, H. Kargl, T. Reiter, W. Retschitzegger, W. Schwinger, M. Strommer, and M. Wimmer. A Framework for Buidling Mapping Operators Resolving Structural Heterogeneities - Extended Version. Technical report, Vienna University of Technology, 2008.

12. E. Kapsammer, H. Kargl, G. Kramler, G. Kappel, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer. On Models and Ontologies - A Semantic Infrastructure Supporting Model Integration. In *Proc. of Modellierung 2006, Austria*, 2006.

13. H. Kargl and M. Wimmer. SmartMatcher - How Examples and a Dedicated Mapping Language can Improve the Quality of Automatic Matching Approaches. In *1st. Int. Workshop on Ontology Alignment and Visualization (OnAV'08), Spain*, 2008.

14. V. Kashyap and A. P. Sheth. Semantic and Schematic Similarities Between Database Objects: A Context-Based Approach. *VLDB J.*, 5(4):276–304, 1996.

15. A. Maedche, B. Motik, N. Silva, and R. Volz. MAFRA - A MApping FRAmework for Distributed Ontologies. In *13th European Conf. on Knowledge Engineering and Knowledge Management (EKAW'02), Spain*, 2002.

16. OMG. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/docs/ptc/03-10-04.pdf, 2004.

17. OMG. MOF 2.0 Query/View/Transformation Specification. http://www.omg.org/docs/ptc/07-07-07.pdf, 2007.

18. T. Reiter, M. Wimmer, and H. Kargl. Towards a runtime model based on colored Petri-nets for the execution of model transformations. In *3rd Workshop on Models and Aspects, Germany*, 2007.

19. F. Scharffe and J. de Bruijn. A language to specify mappings between ontologies. In *Proc. of the 1st Int. Conf. on Signal-Image Technology & Internet-Based Systems (SITIS'05)*, 2005.

20. D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

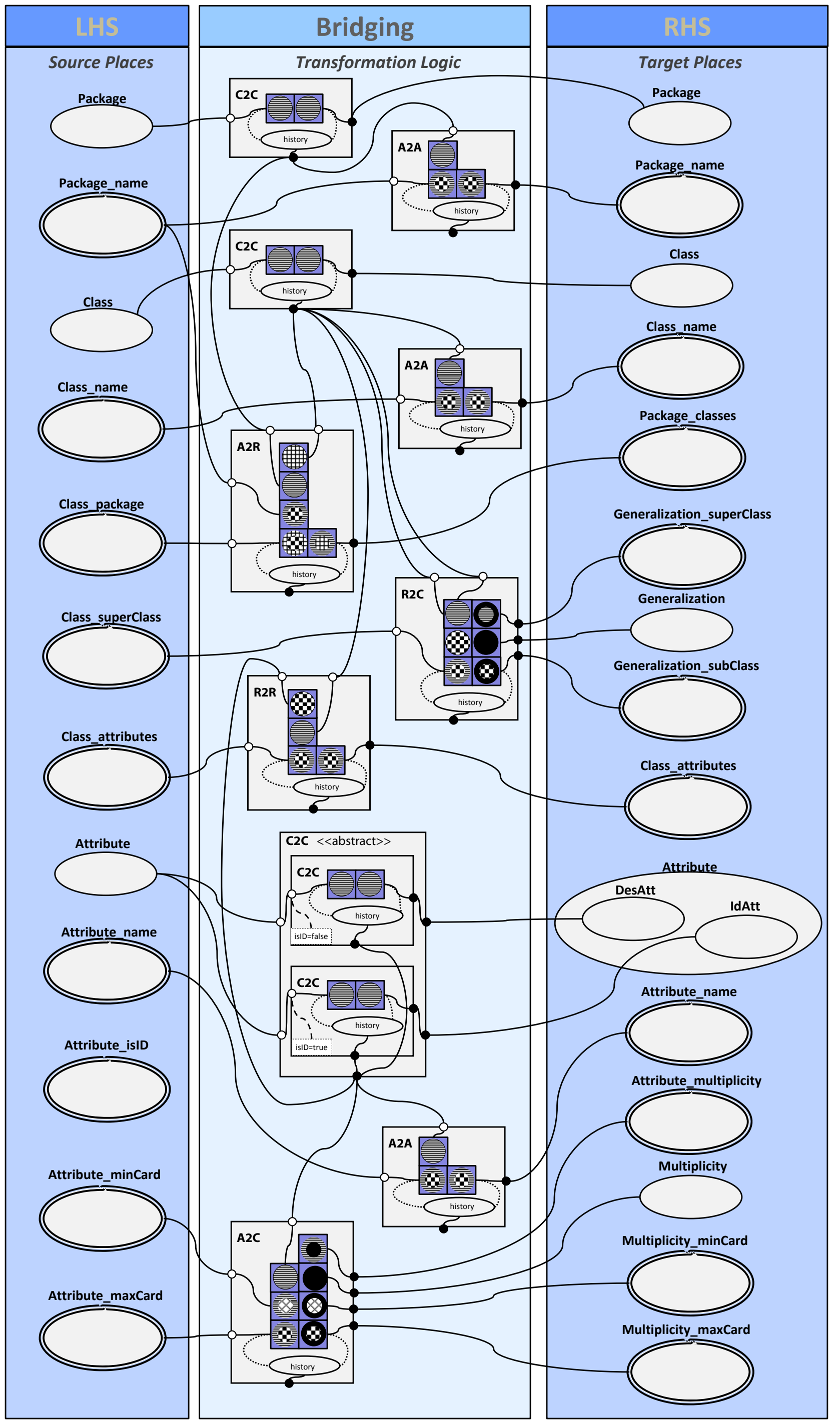
21. L. Tratt. Model transformations and tool integration. *Software and System Modeling*, 4(2):112–122, 2005.

22. D. Varró and A. Pataricza. Generic and Meta-transformations for Model Transformation Engineering. In *Proc. of the 7th Int. Conf. on the Unified Modeling Language (UML'04), Portugal*, 2004.

23. M. Wimmer, A. Schauerhuber, W. Schwinger, and H. Kargl. On the Integration of Web Modeling Languages: Preliminary Results and Future Challenges. In *Workshop Proc. of 7th Int. Conf. on Web Engineering (ICWE'07), Italy*, 2007.

24. M. Wimmer, A. Schauerhuber, M. Strommer, W. Schwinger, and G. Kappel. A Semi-automatic Approach for Bridging DSLs with UML. In *Workshop Proc. of 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Canada*, 2007.

Appendix A - Transformation View for Integration Example


LHS
Source Places
Bridging
Transformation Logic
RHS
Target Places
Package
Package_name
Class
Class_name
Class_package
Class_superClass
Class_attributes
Attribute
Attribute_name
Attribute_isID
Attribute_minCard
Attribute_maxCard
C2C
A2A
A2R
R2C
R2R
C2C <<abstract>>
isID=false
isID=true
A2C
history
Package_classes
Generalization_superClass
Generalization
Generalization_subClass
DesAtt
IdAtt
Attribute_multiplicity
Multiplicity
Multiplicity_minCard
Multiplicity_maxCard