# Leveraging Model-Based Tool Integration by Conceptual Modeling Techniques

Gerti Kappel[1], Manuel Wimmer[1],
Werner Retschitzegger[2], and Wieland Schwinger[3]

[1] Business Informatics Group
Vienna University of Technology, Austria
{kappel,wimmer}@big.tuwien.ac.at
[2] Department of Business Information Systems
University of Vienna, Austria
werner.retschitzegger@univie.ac.at
[3] Department of Telecooperation
Johannes Kepler University Linz, Austria
wieland.schwinger@jku.ac.at

**Abstract.** In the context of model-based tool integration, model transformation languages are the first choice for realizing model exchange between heterogenous tools. However, the lack of a conceptual view on the integration problem and appropriate reuse mechanisms for already existing integration knowledge forces the developer to define model transformation code again and again for certain recurring integration problems in an implementation-oriented manner resulting in low productivity and maintainability of integration solutions.

In this chapter, we summarize our work on a framework for model-based tool integration which is based on well-established conceptual modeling techniques. It allows to design integration models on a conceptual level in terms of UML component diagrams. Not only the design-time is supported by conceptual models, but also the runtime, i.e., the execution of integration models, is represented by conceptual models in terms of Coloured Petri Nets. Furthermore, we show how reusable integration components for resolving structural metamodel heterogeneities, which are one of the most frequently recurring integration problems, can be implemented within our framework.

## 1 Introduction

With the rise of Model-Driven Engineering (MDE) [25] 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 transformation, and code generation. Seamless exchange of models among different modeling tools increasingly becomes a crucial prerequisite for effective MDE. 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

Kaschek/Delcambre (Eds.): The Evolution of Conceptual Modeling, LNCS 6520, pp. 254–284, 2011.

exchange, current best practices (cf., e.g. [28]) 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.

We have followed the aforementioned approach in various projects such as the ModelCVS[1] project [13] focusing on the interoperability between legacy case tools (in particular CA's AllFusion Gen) with UML tools and the MDWEnet[2] project [29] 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 federated and multi database systems [2,14]. 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 Object Management Group (OMG) standard Query/View/Transformation (QVT) [21], provide no appropriate abstraction mechanisms or libraries for resolving recurring kinds of structural heterogeneities [18]. 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.

In this work, we summarize our work on a framework for realizing model-based tool integration which is based on well-established conceptual modeling techniques. In particular, the framework allows to build so-called metamodel bridges on a conceptual level by using UML component diagrams [19]. Furthermore, such a metamodel bridge allows the automatic transformation of models since for each mapping operator, which is expressed as a component, the operational semantics is specified on basis of Colored Petri Nets [7]. 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. 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 Meta Object Facility (MOF) [20]. Finally, we present an inheritance mechanism for reusing applied mappings within a single metamodel bridge. The framework has been applied in several modeling tool integration projects where structural modeling languages as well as behavioral modeling languages have been integrated.

The rest of the paper is structured as follows. In Section 2 we introduce our framework for defining mapping operators for establishing metamodel bridges. In Section 3, the mapping language CAR is presented for which an inheritance mechanism for reusing mappings within a single metamodel bridge is introduced

[1] ModelCVS—Concurrent Versioning System for Models.
[2] MDWEnet—Model-Driven Web Engineering Network.

in Section 4. While an evaluation of the framework is presented in Section 5, Section 6 discusses related work. Finally, in Section 7 a conclusion and a discussion on future research issues 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.

On the mapping view level, the user defines mappings between elements of two metamodels (cf. M2 in Figure 1). Thereby a mapping expresses also a relationship between model elements, i.e., the instances of the metamodels [3]. 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 (cf. M1 in Figure 1) from the LHS to the RHS on basis of Colored Petri Nets [7].

### 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 (i.e., the component's externals) and *white-box* view (i.e., the component's internals) of components allows switching between a high-level mapping view

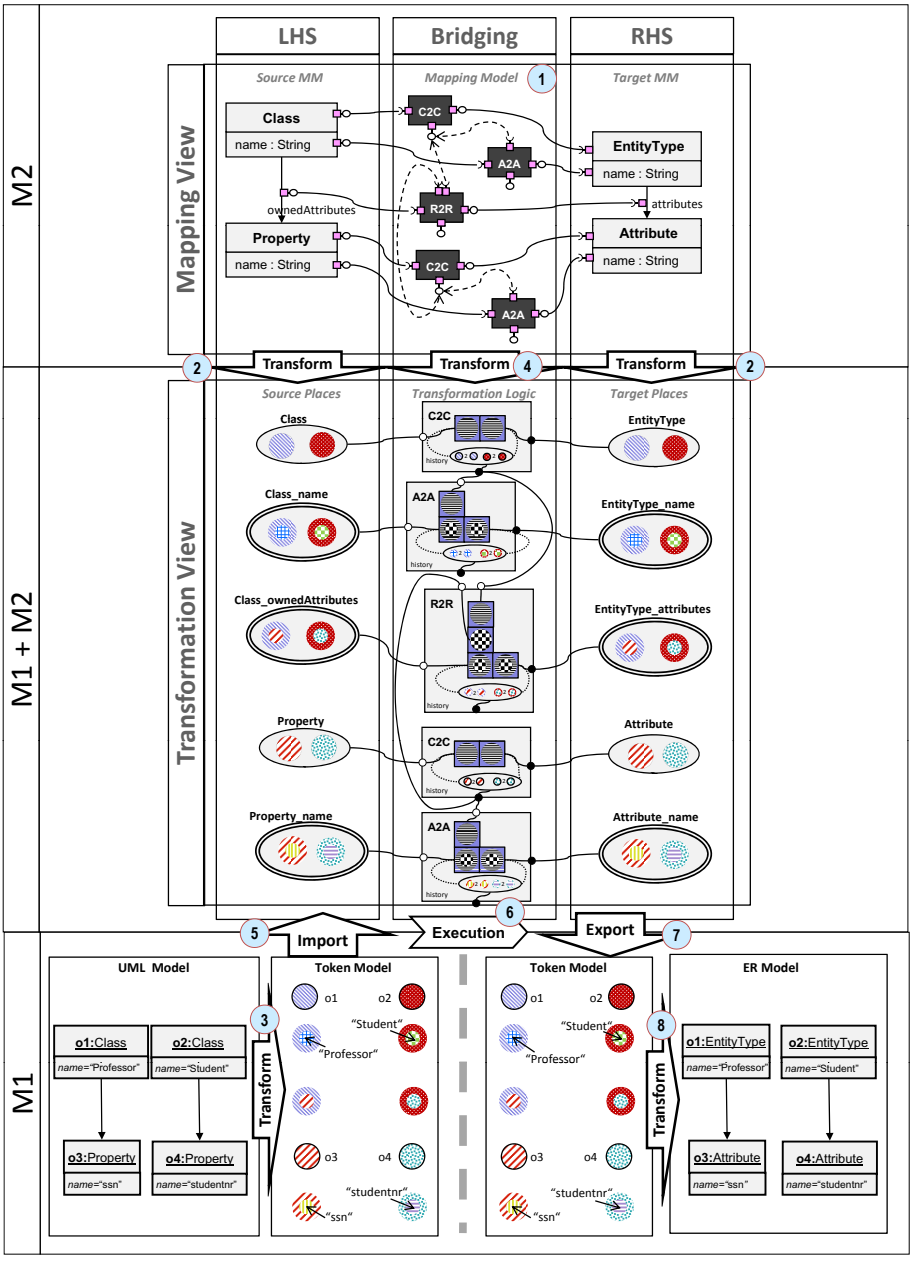



**Fig. 1.** Metamodel Bridging Framework by-example

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*, 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* [12,23,34] 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 underscore 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 consists 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 exactly the same 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, dependent 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 [10,11,35,36] 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.

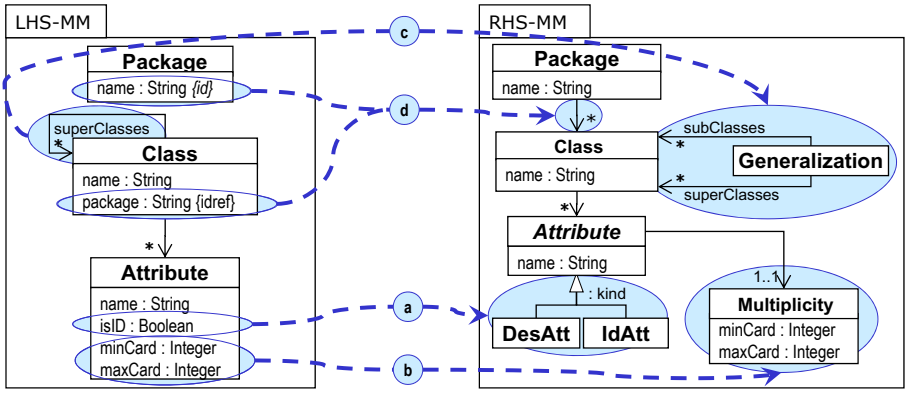



**Fig. 2.** Structural Heterogeneities Between Metamodels - Example

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 (cf. Subsection 3.3 - 3.6) 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

For resolving structural metamodel heterogeneities, we provide nine different core mapping operators as depicted in Figure 2. These nine mapping operators result from the possible combinations between the core concepts of metametamodels, 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* | *Reference* |
|---|---|---|---|
| *Class* | C2C | C2A | C2R |
| *Attribute* | A2C | A2A | A2R |
| *Reference* | R2C | R2A | R2R |

Legend
... *Copier*
... *Peeler*
... *Linker*

**Fig. 3.** CAR Mapping Operators and their categorization

**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"[3] 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)

[3] 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.

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 besides a variation of the C2C operator mainly mapping operators falling into the above mentioned peeler and linker category. The solution for the integration example is shown in Figure 4 and Figure 5 on the mapping view and on the transformation view, respectively.

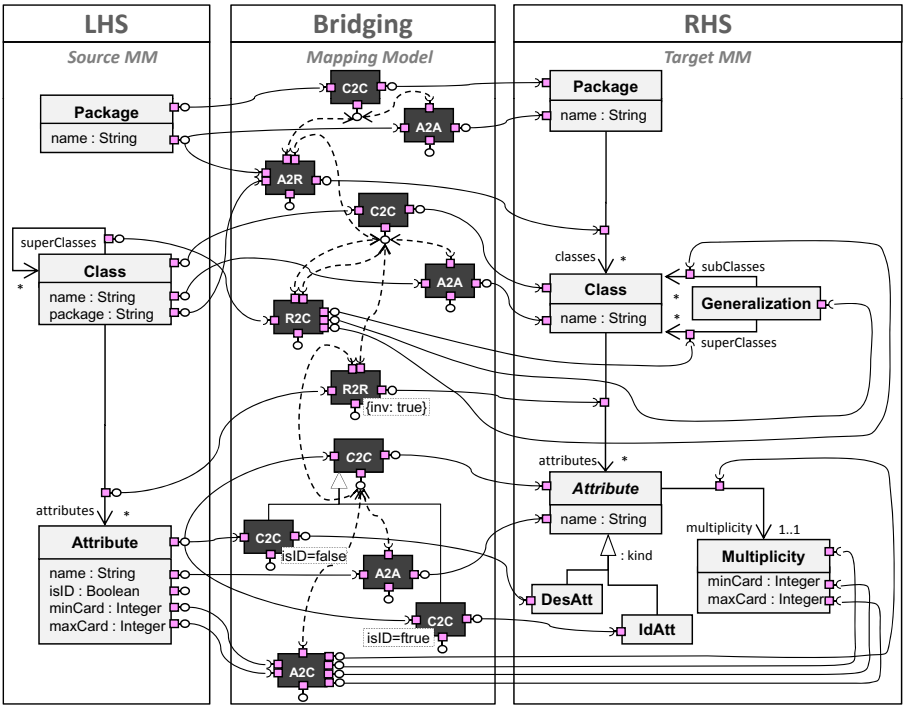



**Fig. 4.** Integration Example solved with CAR (Mapping View)

### 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.

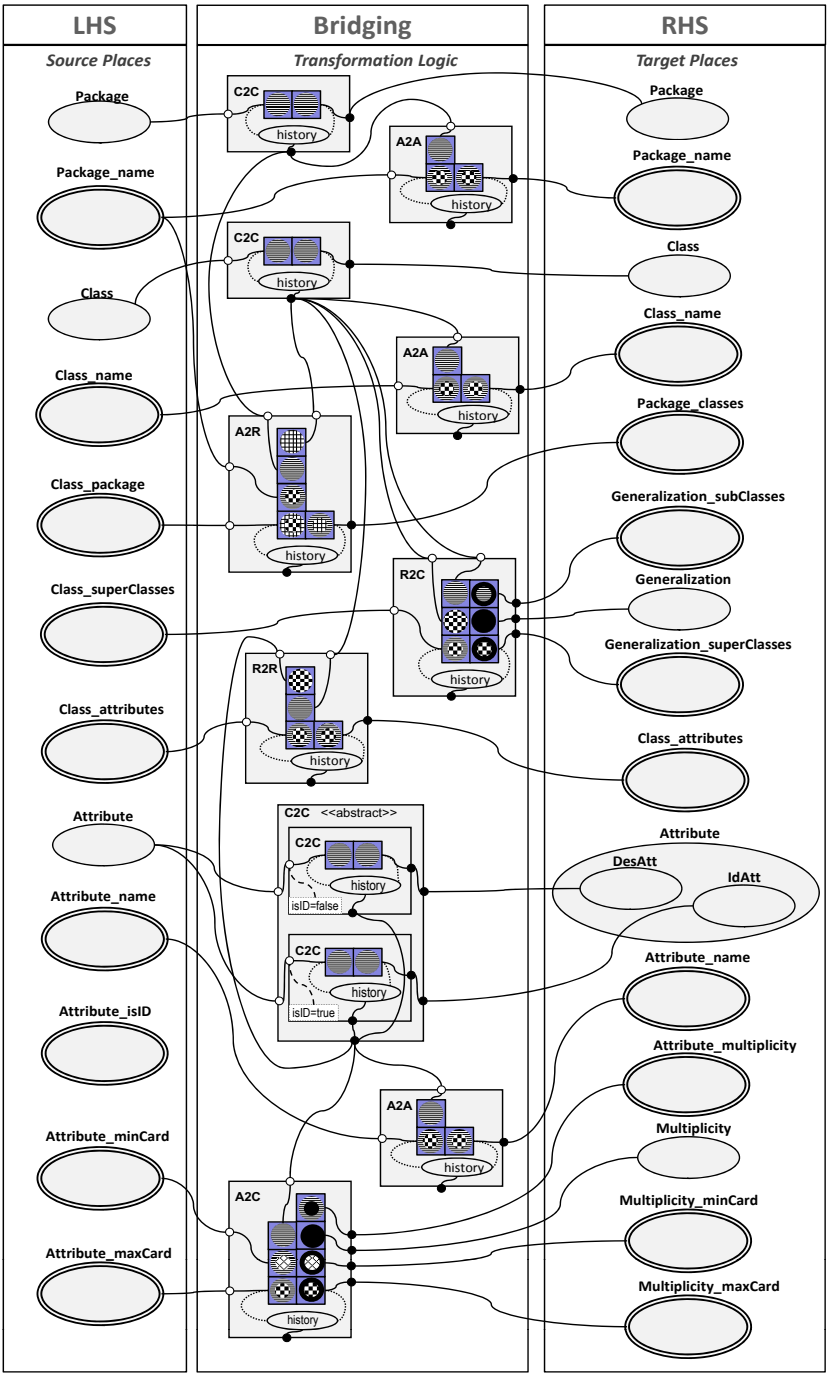



**Fig. 5.** Integration Example solved with CAR (Transformation View)

**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 6(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 look into the white-box view of the operator, realizing its basic functionality.

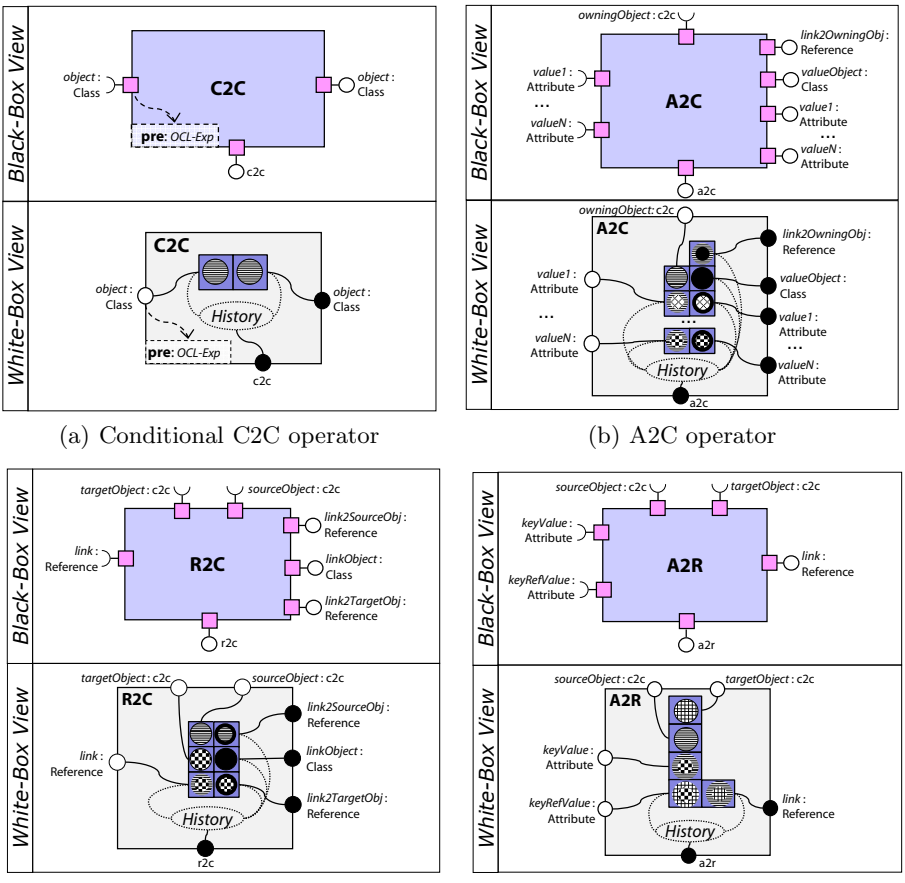
**Example Application.** In the example shown in Figure 4, we can apply two C2C mapping operators with OCL conditions, one for mapping *Attribute* to *DesAtt* with the precondition *Attribute.isID = false*, and one for mapping *Attribute* to *IdAtt* with the precondition *Attribute.isID = true*. In addition, 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), as described in more detail in Section 4.

### 3.4 A2C Mapping Operator

**Problem.** In Figure 2(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 6(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 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 a C2C operator which maps the LHS class to its corresponding target RHS class.

The white-box view of the A2C operator shown in Figure 6(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




(a) Conditional C2C operator

(b) A2C operator

(c) R2C operator

(d) A2R operator

**Fig. 6.** Black-box and white-box views of CAR mapping operators

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 a 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 4, 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 2(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 6(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 6(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 objects of the LHS.

**Example Application.** In Figure 4, 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.

### 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 6(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 4, 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.

After presenting how common structural metamodel heterogeneities are resolved with CAR mapping operators, we proceed with discussing how mapping reuse within a single mapping model may be achieved. This is especially needed for large metamodels which consists of extensive class hierarchies such as the UML 2 metamodel. For example, as reported in [16], the number of single inheritance meta-classes in UML 2 is 209. Therefore, in the next section, an inheritance mechanism for CAR mapping operators for achieving reuse within a single mapping model is presented.

## 4 An Inheritance Mechanism for CAR Mappings

**Why is an inheritance mechanism needed?** Metamodeling languages such as MOF allow to define class hierarchies in metamodels through generalization which is frequently used in practice. Consequently, when one defines mappings between metamodels which heavily use generalizations leading to huge taxonomies, it should be possible to reuse previously defined mappings between general classes for mappings between subclasses.

**Required reuse mechanisms.** To facilitate the definition of mapping models between MOF-based metamodels, reuse may be achieved in three ways:

(1) *Reuse of feature mappings*: Mappings between features of superclasses may be defined once for the superclasses and may be inherited by mappings between subclasses which may define additional feature mappings between the subclasses. This reuse mechanism is comparable with code reuse through implementation inheritance supported by common object-oriented programming languages.

(2) *Reuse for indirect instances*: The following mapping situation frequently occurs when two metamodels have to be integrated. Assume, we have on the LHS a superclass which has a multitude of subclasses. In contrast, on the RHS the class hierarchy on the LHS is collapsed into a single class inhibiting all features of the LHS class hierarchy at the RHS, which is equivalent to the LHS

superclass and also to its subclasses. To avoid that for each subclass a mapping to the RHS class must be defined, it should be possible to apply the mapping between the LHS superclass and the RHS class also for indirect instances of the LHS superclass.

(3) *Refinement of mappings*: A refinement mechanism is needed to define for certain subclasses specific mappings, which refine the mapping between the superclasses. For example, instances of a LHS subclass should be transformed into instances of a RHS subclass and not into instances of the RHS superclass. Of course, the feature mappings between the superclasses should be also applied for such submappings.

In the following subsections, we elaborate on how the CAR mapping language is extended in order to provide these three reuse features.

### 4.1 Inheritance for C2C Mappings

For reusing existing feature mappings (first reuse mechanism), we introduce the possibility to define generalization relationships between C2C mappings. This means, the user defines general mappings between superclasses called *supermappings* and more specific mappings between *subclasses* called *submappings* which may be used to refine the supermappings (third reuse mechanism). 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. Concerning the second reuse mechanism, it has to be noted that there are also cases in that the mentioned behavior of applying a supermapping for indirect instances is not desired. Sometimes it is required that only direct instances should be transformed and not indirect instances. Therefore, certain configuration parameters for mappings are required in order to express such integration details in the mapping model.

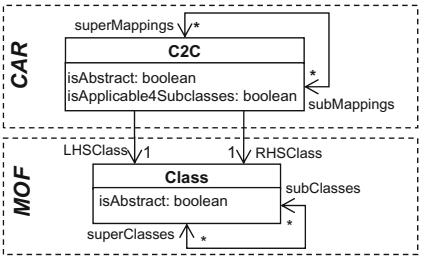



**Fig. 7.** C2C Operator Extended with Generalization Relationships

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. The introduction of generalization relationships between C2C mappings

results in an extension of the C2C operator as shown in Figure 7. In this figure, the class *C2C*, representing the C2C mapping operator, is extended for defining generalization relationships by setting the references *superMappings* and *subMappings* accordingly. Furthermore, for allowing different kinds of supermappings, i.e., if a mapping is itself executable or if it is applicable for unmapped subclasses, two additional boolean attributes, namely *C2C.isAbstract* and *C2C.isApplicable4SubClasses*, are defined for the *C2C* class. These two attributes enable the user to configure the behavior of the mappings in more detail.

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. The OCL constraints shown in Listing 1.1 validate a mapping model with respect to the correct usage of generalization relationships between C2C mappings.

**Listing 1.1.** Well-formedness Rules for C2C Generalizations

```
context C2C
 inv: self.superMappings -> forAll(supMap|supMap.LHSClass.subClasses ->
          union(supMap.LHSClass) -> contains(self.LHSClass));
 inv: self.superMappings -> forAll(supMap|supMap.RHSClass.subClasses ->
          union(supMap.RHSClass) -> contains(self.RHSClass));
```

The default configuration of the C2C operator for each mapping situation specifies that the supermapping itself is executable and applicable for indirect instances. In order to give the user more possibilities to explicitly define other interpretations of supermappings, we furthermore allow three non-default supermapping configurations, thereby the first configuration allows to define abstract supermappings with the capability to be applied for indirect instances, and the other two configurations allow reuse of depending mappings of supermappings without applying the supermappings on indirect instances.

For further explanations how to use generalization between C2C operators, we assume that the mapping problem is symmetric, i.e., the same generalization structure is available on the LHS and on the RHS, and that only single inheritance is used for defining the metamodels. In particular, we assume that on the LHS and on the RHS a superclass with various subclasses exists. For asymmetric mapping problems, i.e., one side has a taxonomy and the other has not, and integration scenarios where metamodels use multiple inheritance the interested reader is kindly referred to [32].

### 4.2 Representing Inheritance within Transformation Nets

In this subsection we discuss how C2C generalization relationships influence the generation of transformation nets and consequently the execution of the

transformation logic. An 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.

**Basic Idea.** When we take a closer look on supermappings with a standard configuration, 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, for a supermapping a transformation component is derived 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.

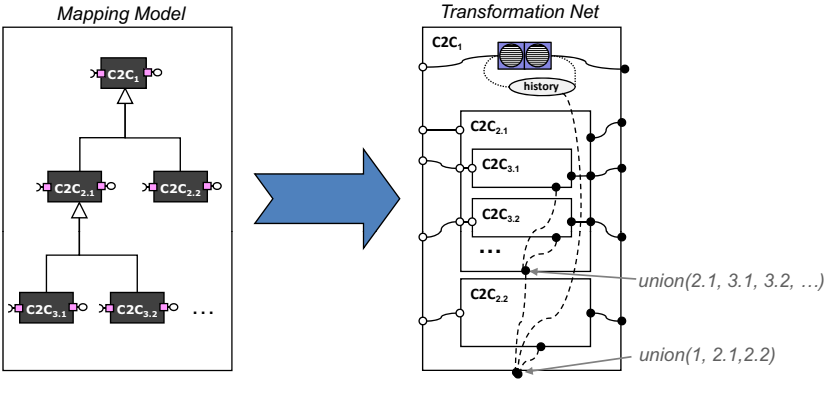



**Fig. 8.** Representing Inheritance Structures with Nested Transformation Components

Figure 8 illustrates the derivation of generalization relationships into transformation net components. For describing the basic mapping rule how generalization relationships are represented in transformation nets, it is assumed that all mappings are concrete mappings and it is not considered if a mapping is applicable for subclasses or not. The mapping $C2C_1$ of the mapping model shown on the LHS of Figure 8 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

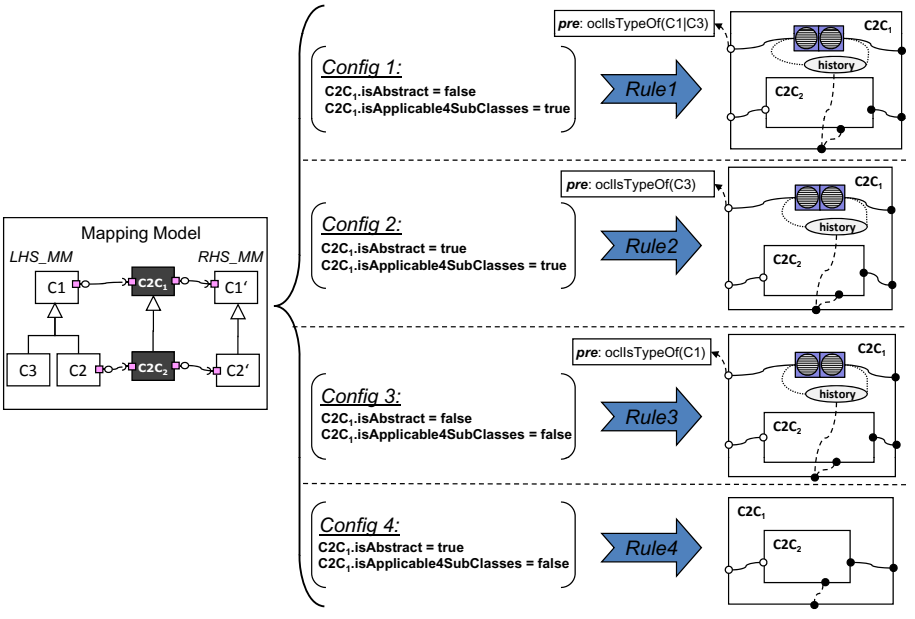



**Fig. 9.** Representing Supermapping Configurations in Transformation Components

the mapping $C2C_{2.1}$ has two submappings, the corresponding transformation component has also two subcomponents $C2C_{3.1}$ 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 $C2C_{3.2}$.

**Mapping Rules for Supermapping Configurations.** In addition to the derivation of inheritance structures to nested transformation components, specific derivation rules for the configuration variants of the supermappings are needed to represent *abstract* and *concrete* mappings in transformation nets as well as the *applicability* of supermappings for subclasses. In particular, the following four rules, which are summarized in Figure 9, are sufficient to generate transformation nets for all possible supermapping configurations. The mapping model shown in Figure 9 is used as an example input mapping model for describing the mapping rules and comprises a mapping between the superclasses *C1* and *C1'* of the LHS and RHS metamodels and between the subclasses *C2* and *C2'*, whereby the subclass *C3* of the LHS remains unmapped.

*Rule 1 - Concrete/Applicable Supermapping*: When a supermapping is concrete, a transition is available in the outer transformation component for transforming direct instances of the superclass and indirect instances for which no specific mappings are available. Because only direct and indirect instances of subclasses without specific mappings should be transformed by the transition of the outer component, an OCL condition is attached on the inputPort which leads to the transition in order to reject tokens for which more specific mappings are available. Such constraints can be defined with the OCL function *oclIsTypeOf*

which gets as parameters the superclass and all subclasses for which no specific mappings have been defined in the mapping model (cf. OCL condition *oclIsTypeOf(C1|C3)*). If there is a more specific mapping between subclasses, a nested component is produced and the tokens are not streamed via the superclass mapping, instead the subplace generated from the LHS subclass gets an additional arc which leads to a more specific transformation component.

*Rule 2 - Abstract/Applicable Supermapping*: Although the supermapping is abstract, a transition resides directly in the outer component, which is not applicable for direct instances but for transforming all indirect instances for which no specific mapping has been applied (cf. OCL condition *oclIsTypeOf(C3)*).

*Rule 3 - Concrete/Non-Applicable Supermapping*: If a supermapping is defined as concrete and non-applicable for unmapped subclasses then an outer component is produced which consists of a transition for transforming direct instances of the superclass (cf. OCL condition *oclIsTypeOf(C1)*).

*Rule 4 - Abstract/Non-Applicable Supermapping*: When a supermapping is abstract and non-applicable for unmapped subclasses only the outer component is generated for providing a union trace model for its submappings. This is sufficient, because neither direct instances nor indirect instances have to be transformed by such a component.

**Design Alternatives.** The following three design alternatives exist for transformation nets to model the applicability of the supermapping transition on subPlaces. First, we could extend the place modeling constructs with tags such as "superTransition is applicable". However, the introduction of such a transformation net feature would violate our design goal that the transformation net formalism should not be changed. The second possibility is to generate for each unmapped class an additional arc from the corresponding source place to the outer component generated for the supermapping. This variant would lead to much more complicated transformation nets and to a huge amount of duplicated arcs, which simply does not pay off the information gain for the user. Therefore, we decided for a third variant, namely the usage of OCL constraints as explained for Rule 1 to 3.

**Example.** To summarize this section, a concrete integration example, as shown in Figure 10, is discussed on the mapping view and on the transformation view. In the LHS metamodel, a class *Person* is specialized into *Supplier*, *Employee*, and *Customer* classes. The RHS metamodel consists also of a superclass *Person*, and of *Client*, *Staff*, and *ShareHolder* subclasses. Each LHS class can be mapped to a RHS class, except the class *Supplier*. Hence, the LHS class *Person* is mapped with a C2C mapping operator to the RHS class *Person*. The properties of this C2C are set to *isAbstract=FALSE* and *Applicable4SubClasses=TRUE*. Consequently, each instance of the LHS class *Person* is transformed into an instance of the RHS class *Person*, as well as each instance of a subclass which has no further refinement mapping is also transformed into an instance of the RHS *Person* class. For example, each instance of the class *Supplier* becomes an instance of the class *Person*. Additionally, the name attribute of the LHS class *Person* is

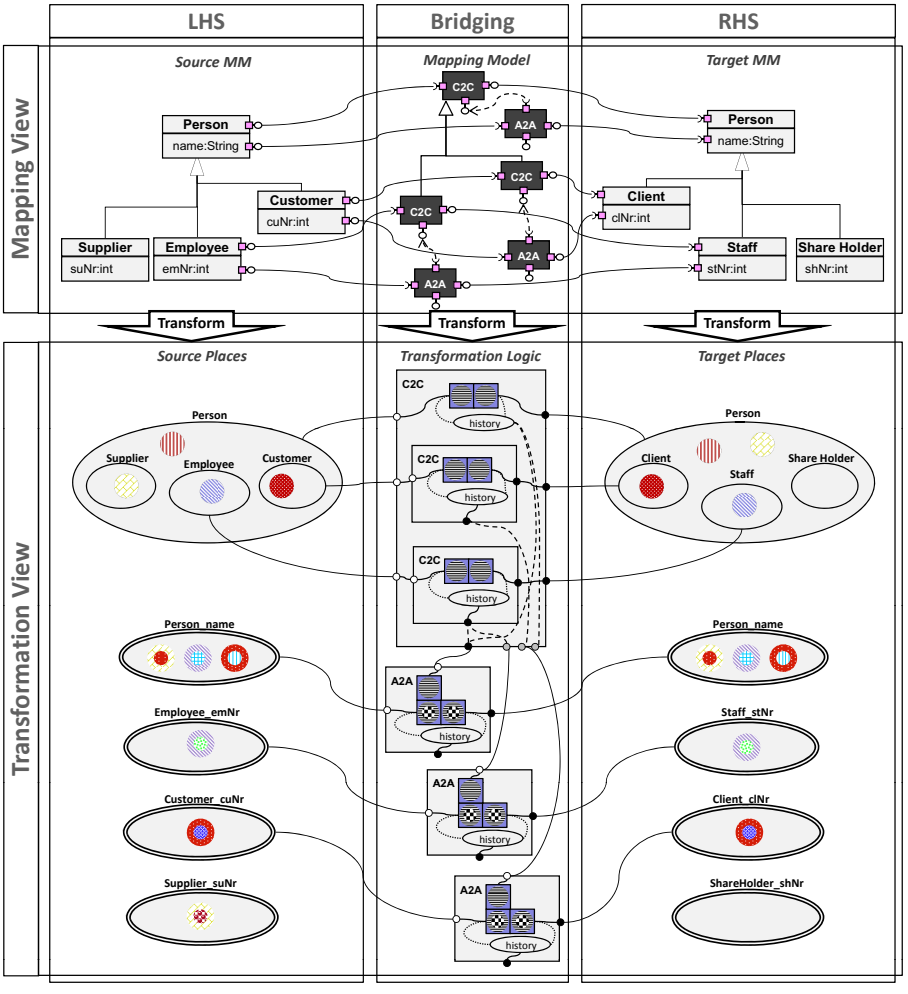



**Fig. 10.** Inheritance between C2C Mappings - Example

mapped by an A2A mapping operator to the name attribute of the RHS class *Person*.

The subclasses *Employee* and *Customer* of the class *Person* on the LHS are mapped by C2C mappings to *Staff* and *Client* of the RHS, respectively. Additionally, the attributes of these classes, namely *Customer.cuNr*, *Employee.emNr*, and *Client.clNr*, *Staff.stNr*, are mapped by A2A mappings, respectively. Due to the fact that each of the subclasses inherit the attributes of the superclass – the attribute *Person.name* – the A2A mapping between the superclasses is also inherited by the C2C mappings by setting the *superMappings* reference to the C2C mapping which resides between the *Person* classes.

The corresponding transformation net for the presented mapping model is depicted in the *Transformation View* of Figure 10. The *Person* classes become places which comprise for each subclass an inner place. As subclass places are nested in superclass places, the inheriting submappings are nested in the transformation component which corresponds to the supermapping. The outer transformation component, corresponding to the supermapping, contains a transition, because the *isAbstract* property of the C2C mapping is set to *FALSE*. Furthermore, due to the *isApplicable4SubClasses* property of the C2C mapping, which is set to TRUE, the outer transformation component of the transformation net owns an additional OCL constraint, namely *oclTypeOf(Person|Supplier)*. Due to readability purposes, we refrain from displaying these features in Figure 10. Consequently, each direct instance of type *Person* from the LHS is transformed into an instance of class *Person* on the RHS. Furthermore, this OCL constraint ensures that each instance of subclasses of the class *Person*, which has no refined mapping, is also transformed by the supermapping into an instance of type *Person* on the RHS.

The attribute *Person.name* can be transformed only if the containing instance which can be of type *Person*, *Employee*, or *Customer* has been already transformed. Consequently, the A2A transformation component for name values must be in the context of three C2C transformation components. This is achieved by the trace model provided by the black port in the middle of the bottom of the outer C2C transformation component. This trace model unifies the individual trace models of the C2C transformation components. The other A2A operators are connected to the gray ports which link directly to individual trace models of the nested components.

This example shows the value of using an explicit notion of trace models together with a union of trace models for automatically synchronizing transformation net components. With the automatic synchronization provided by the Petri Net semantic, the streaming of dependent elements such as attribute values and links comes for free. Thus, no additional control structures and OCL constraints are required, and the derivation of transformation nets from mapping models is straightforward.

## 5 Evaluation

In this section, we discuss the applicability of our approach by (1) elaborating on how additional tool integration requirements, namely roundtrip capabilities, may be supported by our approach and by (2) comparing the efforts of building such roundtrip transformations using the proposed mapping approach and using model transformation languages.

*Roundtrip Transformations.* In case modeling languages of two different tools are not entirely overlapping, i.e., some modeling concepts are available in one modeling language which cannot be expressed in the other modeling language, a transformation may be lossy. Thus, although transformations from tool $A$ to tool $B$ and vice versa are available, the initial model $M_a$ of tool $A$ may be

different from the roundtrip result $M'_a$ which is computed by translating $M_a$ into $M_b$ via the transformation $T_{a2b}$ and the application of $T_{b2a}$ on $M_b$ to produce $M'_a$. The main reason for not roundtripping transformations is the fact that bijective mappings are not always possible to establish between metamodels as for example reported in [27]. Here the challenging question arises, how to deal with such cases in order to ensure roundtripping transformations.

In the ModelCVS project[4], besides other integration efforts, we have bridged the Domain Specific Language (DSL) for defining database schemas of the AllFusion Gen[5] (AFG) modeling tool to the class diagram concepts of the UML Rational Software Modeler[6] tool. Thereby, the first case study was to bridge structural modeling, i.e., the AFG Data Model with the UML Class Diagram. The first attempt was to bridge AFG with plain UML. The resulting bridge was not appropriate for using the tools in combination. Because, although we have defined for each AFG modeling concept a specific mapping to UML, a lot of information was lost during roundtrip or even though after the first step when moving from the AFG to UML. Table 1 summarizes the roundtrip scenario by depicting some model metrics for each step in the roundtrip.

**Table 1.** Model Metrics for Data Model/Class Diagram RoundTrip (RT)

| Metrics | Initial AFG Model | UML Model | AFG Model after RT | Diff in % |
|---|---|---|---|---|
| **#Objects** | 156 | 165 | 156 | 0 |
| **#Values** | 1099 | 156 | 156 | 85,8 |
| **#Links** | 44 | 54 | 36 | 18,2 |
| **#Containment Links** | 155 | 164 | 155 | 0 |
| **File Size** | 32,8 KB | 16 KB | 14,6 KB | 55,5 |

The main reason for the massive loss of information was the fact that on the attribute level only a minimal overlap between the languages exists. In most cases, only the *name* attribute of the modeling concepts may be bridged, but all platform specific attributes of the AFG modeling language such as database optimization information may not. When we take a look at the model metrics in Table 1, the initial AFG model and the generated UML model have nearly the same amount of objects and containment links, only some derived objects are additionally instantiated in the UML model. This means, the same model structure can be reproduced on the UML side. However, when we compare the amount of values, we see that a huge amount of information gets lost in the first step. In particular, when comparing the number of values of the initial AFG model and the resulting AFG model after roundtrip, 85,8 % of values are lost during the roundtrip. For links which are not containment links we see that more links exist in the generated UML model compared to the initial AFG model. This is due to the fact, that also derived links are generated for the aforementioned

[4] http://www.modelcvs.org

[5] http://ca.com/us/products/product.aspx?ID=256

[6] http://www-306.ibm.com/software/awdtools/modeler/swmodeler

additionally derived objects. Therefore, even though we have more links and objects on the UML side, less information is expressed and some links cannot be reconstructed when transforming the UML models back to AFG models. Finally, the information loss has of course an effect on the file size, namely the resulting file after roundtrip has only half the size of the initial file.

**Table 2.** Model Metrics for Data Model/Class Diagram Roundtrip Revisited

| Metrics | Initial AFG Model | UML Model | AFG Model after RT | Diff in % |
|---|---|---|---|---|
| **#Objects** | 156 | 165 | 156 | 0 |
| **#Values** | 1099 | 156 | 1099 | 0 |
| **#Links** | 44 | 54 | 44 | 0 |
| **#Containment Links** | 155 | 164 | 155 | 0 |
| **File Size** | 32,8 KB | 58,5 KB | 32,8 KB | 0 |
| **#Annotations** | - | 156 | - | - |
| **#Key/Value Pairs** | - | 951 | - | - |

In Table 2, the model metrics are again presented, however, now for unmapped features (i.e., attributes and references) dedicated annotations are created on the UML side by using an enhanced transformation generation from mapping models tailored to preserving information during roundtrip. As one can see in the most right column of the table, no difference regarding the number of model elements between the initial AFG model and the AFG model after roundtrip exists. The information can be preserved by applying for each corresponding object an appropriate annotation allowing to use key/value pairs for saving attribute values which otherwise would be lost. This specific logic can be easily introduced in the transformation generation thanks to the declarative nature of the mapping models. Furthermore, a comparison of the initial AFG model and the resulting AFG model using the comparison facility of EMF Compare[7] demonstrated that both were equivalent, thus no information has been lost during roundtrip.

*Metrics for Mapping Model and corresponding Model Transformations.* In addition to the evaluation of the roundtrip capability of the generated bridge, the number of model elements of the manually created mapping model compared to the number of elements needed for the corresponding model transformations defined in the de-facto model transformation standard in Eclipse, namely the ATLAS Transformation Language (ATL) [8], is presented. This comparison should give an indication on how much effort requires the creation of the mapping model in contrast to building the integration artifacts manually from scratch by using a textual model transformation language such as ATL.

Table 3 summarizes some metrics for the mapping model and for the corresponding ATL model transformations. We decided to use as metrics for the mapping model, first, the number of applied MOPs, and second, how many non-default property values (e.g., for setting inheritance relationships between MOPs) have to be set by the user, because this is exactly the work the user has

[7] `www.eclipse.org/modeling/emft`

to do for creating the mapping model. For the transformation code we simply use lines of code (LOC) as metric, just to give an indication how much effort the manual implementation of the transformations for both directions (AFG2UML, UML2AFG) would cause.

**Table 3.** Metrics Overview: Mapping Model vs. ATL Code

| Mapping Model Metrics | | Model Transformation Metrics | |
|---|---|---|---|
| *Mapping Operator* | *User Actions* | *ATL File* | *Lines of Code* |
| C2C | 6 | AFG2UML | 120 overall<br>75% declarative<br>25% imperative |
| A2A | 1 | | |
| R2R | 7 | | |
| Properties | 7 | UML2AFG | 100 overall<br>86% declarative<br>14% imperative |

For realizing model exchange between AFG and UML, in total 220 lines of ATL code are necessary, where most parts are declarative rules. However, for using annotations within the transformation for saving unmapped values, imperative code is needed, e.g., for applying annotations and for setting key/value pairs when moving from AFG to UML as well as for assigning key/value pairs as attribute values when we are going back from UML to AFG. Instead, using the presented mapping approach, we are able to develop the same bridge using only 14 MOPs and setting 7 properties of the applied MOPs from which the transformations in both directions are derived. For dealing with annotations for setting unmapped values, the generator for the transformations is capable of producing this dedicated logic without requiring the user to define mappings for this aspect. Only the generator has been extended with one additional generation rule which can be turned on/off as required. Compared to the manual authoring of model transformations where this aspect is intermingled with the rest of the code, the presented mapping approach allows for a faster and more systematic development of tool integrations.

## 6 Related Work

With respect to our approach of defining reusable mapping operators for resolving metamodel heterogeneities as a kind of mapping between metamodels we distinguish between three broad categories of related work: first, related work concerning our goal to design a framework for building reusable mapping operators in the field of MDE, and second, related work concerning our solution approach in the field of ontology integration. In addition, we elaborate on related approaches which employ Petri Nets as conceptual modeling language for defining model transformations.

### 6.1 Reusable Model Transformations

**Generic Model Transformations.** Typically model transformation languages, e.g., ATL [8] and QVT [21], 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. [30] who propose a notion of specifying 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. [1]. 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. Realization 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) [6]. 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.

### 6.2 Ontology Mapping for Bridging Structural Heterogeneities

In the field of ontology engineering, several approaches exist which make use of high-level languages for defining mappings between ontologies (cf. [9] for an overview). For example, in Maedche et al. [17], 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 [24], 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 partial 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.

### 6.3 Petri Nets and Model Transformations

The relatedness of Petri Nets and graph rewriting systems has also induced some impact in the field of model transformation. Especially in the area of graph transformations some work has been conducted that uses Petri nets to check formal properties of graph production rules. Thereby, the approach proposed in [31] translates individual graph rules into a place/transition net and checks for its termination. Another approach is described in [5], which applies a transition system for modeling the dynamic behavior of a metamodel.

Compared to these two approaches, our intention to use Petri Nets is totally different. While these two approaches are using Petri Nets as a back-end for automatically analyzing properties of transformations by employing place/-transition nets, we are using Colored Petri Nets as a front-end for debuggability and understandability of transformations. In particular, we are focussing on how to represent model transformations as Petri Nets in an intuitive manner. This also covers the compact representation of Petri Nets to eliminate the scalability problem of low-level Petri nets. Finally, we introduce a specific syntax for Petri Nets used for model transformations and integrate several high-level constructs, e.g., colors, inhibitor arcs, and pages, into our language. However, it has to be noted that the higher expressivity gained from high-level constructs comes with a negative impact on the analyzability of the Petri Nets.

## 7 Conclusion and Future Research Issues

In this paper we have presented 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 integration process further enhances 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 structural metamodel heterogeneities[8]. The proposed framework is expressive enough to define advanced composition mechanisms such as inheritance between mapping operators. This is achieved by the explicit notion of trace models for mapping operators in combination with the automatic synchronization provided by the Petri Net semantic.

The presented framework has been applied in several modeling tool integration projects (for more details the interested reader is kindly referred to www.modelcvs.org and www.modeltransformation.net). It has to be mentioned that the presented approach is not only applicable for integrating structural modeling languages, but also for integrating behavioral modeling languages. For example, we have integrated the dialog flow modeling language of CA's AllFusion Gen with UML state machines as well as with UML activity diagrams. The most interesting point of this case study was that we explored nearly the same metamodel heterogeneities as we explored when integration structural modeling languages. Thus, the presented metamodel heterogeneities seem to be modeling domain independent.

The work presented in this chapter leaves several issues open for further research. In the following, we present four research issues that we believe are most important for the success of model-based tool integration.

(1) *Bridging Technical Spaces.* Several modeling languages are not described with MOF-based metamodels as proposed by the OMG. Instead, text-based languages such as EBNF, DTD, or XML schema are employed. In order to use model-based integration frameworks which require MOF-based metamodels, converters are needed. In particular, not only the language definition must be converted into a MOF-based metamodel, also the models have to be transformed into instances conforming to the generated metamodels. This raises the question of how to produce such converters for bridging different technical spaces [15] with reasonable effort also in the light of the evolution of these languages.

(2) *Automatic Creation of Mapping Models.* Another issue is the automatic creation of mapping models between two metamodels. With the rise of the semantic web and the emerging abundance of ontologies, several matching approaches and tools for automatically creating mapping models have been proposed, for an overview see [22,26]. The typical output of such tools are simple one-to-one correspondences. However, these correspondences cannot cope with structural heterogeneities between MOF-based metamodels as presented in this work. Therefore, a novel matching approach is needed which is capable to automatically generate mappings expressed with a more powerful mapping language.

[8] For more details about the implementation, we kindly refer the interesting reader to `www.modeltransformation.net`

(3) *Formal Verification of Mapping Models.* As the correctness of the automatically generated target model fully depends on the correctness of the specified mapping model, formal underpinnings are required to enable verification of mapping models by proving certain properties like confluence and termination, to ease debugging of mapping models. The formal underpinning of CPNs enables simulation of mapping models and exploration of the state space, which shows all possible firing sequences of a CPN. In the future, it has to be determined how generally accepted behavioral properties, characterizing the nature of a certain CPN, e.g., with respect to confluence or termination, as well as custom functions, e.g., to check if a certain target model can be created with the given transformation logic, can be applied for interactive debugging and automatic verification of mapping models [33].

(4) *The Role of Semantics.* An open research problem in MDE is how to explicitly and formally specify the semantics of modeling languages. In the last decade several diverse approaches inspired from programming language engineering have been proposed. However, in contrast to syntax, currently there is no commonly approved or standardized approach as well as tool support for defining the semantics of modeling languages. Thus, often the semantics are only informally specified in terms of natural language—the most prominent example is UML—or the semantics are hard-coded in code generators, simulators, or interpreters. However, for bridging modeling tools, an explicit notion of semantics would be of paramount importance. For example, when integrating tools for modeling state machines, it can happen that the modeling tools provide completely the same syntax definition, but the execution of the same model in tool *A* can differ from the execution in tool *B*, as is for example reported in [4]. Thus, one of the biggest challenges in MDE is how to provide explicit semantic definitions for modeling languages and how to use these definitions to build and verify integration solutions.

## References

1. Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G.: Reusable Idioms and Patterns in Graph Transformation Languages. In: Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004) (2004)
2. Batini, C., Lenzerini, M., Navathe, S.B.: A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Survey 18(4), 323–364 (1986)
3. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, China (2007)
4. Crane, M.L., Dingel, J.: UML vs. classical vs. rhapsody statecharts: not all models are created equal. Software and System Modeling 6(4), 415–435 (2007)
5. de Lara, J., Vangheluwe, H.: Translating Model Simulators to Analysis Models. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 77–92. Springer, Heidelberg (2008)
6. Fabro, M.D.D., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver. In: Proceedings of the 1re Journe sur l'Ingnierie Dirige parles Modles, IDM 2005 (2005)

7. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Springer, Heidelberg (1992)
8. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
9. Kalfoglou, Y., Schorlemmer, W.M.: Ontology mapping: The state of the art. In: Dagstuhl Seminar Proceedings: Semantic Interoperability and Integration (2005)
10. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 528–542. Springer, Heidelberg (2006)
11. Kappel, G., Kargl, H., Kramler, G., Schauerhuber, A., Seidl, M., Strommer, M., Wimmer, M.: Matching Metamodels with Semantic Systems - An Experience Report. In: Workshop Proceedings of Datenbanksysteme in Business, Technologie und Web (BTW 2007) (2007)
12. Kappel, G., Kargl, H., Reiter, T., Retschitzegger, W., Schwinger, W., Strommer, M., Wimmer, M.: A Framework for Building Mapping Operators Resolving Structural Heterogeneities. In: Proceedings of 7th Int. Conf. on Information Systems Technology and its Applications (2008)
13. Kapsammer, E., Kargl, H., Kramler, G., Kappel, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: On Models and Ontologies - A Semantic Infrastructure Supporting Model Integration. In: Proceedings of Modellierung 2006 (2006)
14. Kashyap, V., Sheth, A.P.: Semantic and schematic similarities between database objects: A context-based approach. VLDB Journal 5(4), 276–304 (1996)
15. Kurtev, I., Aksit, M., Bézivin, J.: Technical Spaces: An Initial Appraisal. In: Meersman, R., Tari, Z., et al. (eds.) CoopIS 2002, DOA 2002, and ODBASE 2002. LNCS, vol. 2519. Springer, Heidelberg (2002)
16. Ma, H., Shao, W.-Z., Zhang, L., Ma, Z.-Y., Jiang, Y.-B.: Applying OO Metrics to Assess UML Meta-models. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 12–26. Springer, Heidelberg (2004)
17. Maedche, A., Motik, B., Silva, N., Volz, R.: MAFRA – A MApping FRAmework for Distributed Ontologies. In: Gómez-Pérez, A., Benjamins, V.R. (eds.) EKAW 2002. LNCS (LNAI), vol. 2473, p. 235. Springer, Heidelberg (2002)
18. Olsen, G.K., Aagedal, J., Oldevik, J.: Aspects of Reusable Model Transformations. In: Proceedings of the 1st European Workshop on Composition of Model Transformations (CMT 2006) (2006)
19. OMG: UML Superstructure Specification, version 2.0 formal/05-07-04 edition (2005)
20. OMG: Meta Object Facility (MOF) 2.0 Core Specification, version 2.0 formal/2006-01-01 edition (2006)
21. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0 formal/2008-04-03 edition (2008)
22. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB Journal 10(4), 334–350 (2001)
23. Reiter, T., Wimmer, M., Kargl, H.: Towards a runtime model based on colored Petri-nets for the execution of model transformations. In: 3rd Workshop on Models and Aspects, in conjunction with ECOOP 2007 (2007)
24. Scharffe, F., de Bruijn, J.: A language to specify mappings between ontologies. In: Proceedings of the 1st International Conference on Signal-Image Technology & Internet-Based Systems (SITIS 2005) (2005)

25. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
26. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches. In: Spaccapietra, S. (ed.) Journal on Data Semantics IV. LNCS, vol. 3730, pp. 146–171. Springer, Heidelberg (2005)
27. Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
28. Tratt, L.: Model transformations and tool integration. Software and System Modeling 4(2), 112–122 (2005)
29. Vallecillo, A., Koch, N., Cachero, C., Comai, S., Fraternali, P., Garrigós, I., Gómez, J., Kappel, G., Knapp, A., Matera, M., Meliá, S., Moreno, N., Pröll, B., Reiter, T., Retschitzegger, W., Rivera, J.E., Schauerhuber, A., Schwinger, W., Wimmer, M., Zhang, G.: MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods. In: Workshop Proceedings of 7th International Conference on Web Engineering (ICWE 2007) (2007)
30. Varró, D., Pataricza, A.: Generic and Meta-transformations for Model Transformation Engineering. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 290–304. Springer, Heidelberg (2004)
31. Varró, D., Varró-Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination Analysis of Model Transformations by Petri Nets. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 260–274. Springer, Heidelberg (2006)
32. Wimmer, M.: From Mining to Mapping and Roundtrip Transformations - A Systematic Approach to Model-based Tool Integration. PhD thesis, Vienna University of Technology (2008)
33. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Right or Wrong? - Verification of Model Transformations using Colored Petri Nets. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM 2009) (2009)
34. Wimmer, M., Kusel, A., Reiter, T., Retschitzegger, W., Schwinger, W., Kappel, G.: Lost in Translation? Transformation Nets to the Rescue! In: Proceedings of 8th Int. Conf. on Information Systems Technology and its Applications (2009)
35. Wimmer, M., Schauerhuber, A., Schwinger, W., Kargl, H.: On the Integration of Web Modeling Languages: Preliminary Results and Future Challenges. In: Workshop Proceedings of 7th International Conference on Web Engineering (ICWE 2007) (2007)
36. Wimmer, M., Schauerhuber, A., Strommer, M., Schwinger, W., Kappel, G.: A Semi-automatic Approach for Bridging DSLs with UML. In: Workshop Proceedings of 7th OOPSLA Workshop on Domain-Specific Modeling (DSM 2007) (2007)