Surviving the Heterogeneity Jungle with Composite Mapping Operators*

M. Wimmer[1], G. Kappel[1], A. Kusel[2],
W. Retschitzegger[2], J. Schoenboeck[1], and W. Schwinger[2]

[1] Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
{firstname.lastname}@jku.at

Abstract. Model transformations play a key role in the vision of Model-Driven Engineering. Nevertheless, mechanisms like abstraction, variation and composition for specifying and applying reusable model transformations – like urgently needed for resolving recurring structural heterogeneities – are insufficiently supported so far. Therefore, we propose to specify model transformations by a set of pre-defined mapping operators (MOps), each resolving a certain kind of structural heterogeneity. Firstly, these MOps can be used in the context of arbitrary metamodels since they abstract from concrete metamodel types. Secondly, MOps can be tailored to resolve certain structural heterogeneities by means of black-box reuse. Thirdly, based on a systematic set of kernel MOps resolving basic heterogeneities, composite ones can be built in order to deal with more complex scenarios. Finally, an extensible library of MOps is proposed, allowing for automatically executable mapping specifications since every MOp exhibits a clearly defined operational semantics.

Key words: Executable Mappings, Reuse, Structural Heterogeneities

1 Introduction

Model-Driven Engineering (MDE) places models as first-class artifacts throughout the software lifecycle [2] whereby model transformations play a vital role. In the context of transformations between different metamodels and their corresponding models, the overcoming of *structural heterogeneities*, being a result of applying different modeling constructs for the same semantic concept [9, 12] is a challenging, recurring problem, urgently demanding for reuse of transformations. Building and applying such *reusable transformations* requires (i) *abstraction mechanisms*, e.g., for dealing with arbitrary metamodels, (ii) *variation mechanisms*, e.g., for tailoring a reusable transformation to certain metamodels, and (iii) *composition mechanisms*, e.g., for assembling a whole transformation specification from reusable transformations. As a backbone, such reusable transformations are required to be offered by means of an *extensible library* being a prerequisite for reducing the high effort of specifying model transformations.

* This work has been funded by the Austrian Science Fund (FWF) under grant P21374-N13.

We therefore propose to specify horizontal model transformations by means of *abstract mappings* using a set of reusable transformation components, called *mapping operators* (MOps) to resolve recurring structural heterogeneities. Firstly, to reuse these MOps for mappings between arbitrary metamodels, i.e., *metamodel independence*, MOps are typed by the core concepts of meta-metamodeling languages, being classes, attributes, and relationships [6], instead of concrete metamodel types. Secondly, to resolve certain structural heterogeneities, MOps can be tailored by means of black-box reuse. Thirdly, based on a set of kernel MOps resolving basic heterogeneities, composite ones can be built in a simple plug & play manner in order to deal with more complex scenarios. Finally, a set of MOps is proposed, providing an initial library of reusable transformations encapsulating *recurring transformation logic* for the resolution of structural heterogeneities. The rationale behind is to follow an MDE-based approach, since abstract mappings can be seen as platform-independent transformation models abstracting from the underlying execution language. These abstract mappings can then be automatically translated to different transformation languages by means of higher-order transformations (HOTs) [17] since the MOps exhibit a clearly defined operational semantics thereby achieving *transformation language independence*. Please note that the presented kernel MOps supersede the MOps presented in our previous work [8], since the initial MOps suffered from two main problems. Firstly, the initial MOps were too fine-grained resulting in scalability problems. Thereby, they neglected the fact that a whole transformation problem can be partitioned into coarse-grained recurring sub-problems demanding for coarse-grained MOps too. Secondly, the initial MOps represented hard-coded patters and were too inflexible to form the basis for arbitrary composite MOps.

The remainder is structured as follows. Section 2 introduces an example exhibiting several structural heterogeneities, which are resolved by means of composite MOps in Section 3. Section 4 presents a MOps kernel providing the basic building blocks for composite MOps. Related work is surveyed in Section 5, a prototypical implementation is proposed in Section 6, and a critical discussion of the proposed approach with an outlook on future work is given in Section 7.

2 Motivating Example

Structural heterogeneities between different metamodels occur due to the fact that semantically equivalent concepts can be expressed by different metamodeling concepts. The `ClassDiagram` shown on the left side of Fig. 1, only provides unidirectional references to represent relationships, thus bidirectionality needs to be modeled by a pair of opposite references. In contrast, the `ERDiagram` explicitly represents bidirectionality, allowing to express relationships in more detail.

In the following, the main correspondences between the `ClassDiagram` and the `ERDiagram` are described. On the level of classes, three main correspondence types can be recognized, namely *1:1*, *1:n* and *n:1*, indicated by dotted lines in Fig. 1. 1:1 correspondences can be found (i) between the root classes `ClassDiagram` and `ERDiagram` and (ii) between `Class` and `Entity`. Regarding 1:n correspondences, again two cases can be detected, namely (i) between the

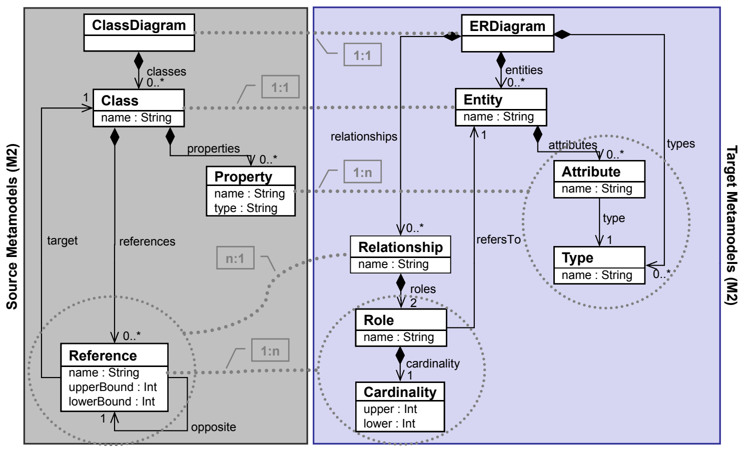

Fig. 1. Metamodels of the Running Example

class `Property` and the classes `Attribute` and `Type` and (ii) between the class `Reference` and the classes `Role` and `Cardinality`. Although these are two occurrences of a 1:n correspondence, there is a slight difference between them, since in the first case only for distinct values of the attribute `Property.type`, an instance of the class `Type` should be generated. Finally, there is one occurrence of a n:1 correspondence, namely between the class `Reference` and the class `Relationship`. It is classified as n:1 correspondence, since for *every pair* of `References` that are opposite to each other, a corresponding `Relationship` has to be established. Considering attributes, only 1:1 correspondences occur, e.g., between `Class.name` and `Entity.name`, whereas regarding references, 1:1 and 0:1 correspondences can be detected. Concerning the first category, one example arises between `ClassDiagram.classes` and `ERDiagram.entities`. Regarding the latter category, e.g., the relationship `ERDiagram.types` exists in the target without any corresponding counterpart in the source.

Summarizing, for the resolution of structural heterogeneities the fulfillment of the following requirements is desirable. Firstly, for allowing a transformation designer to focus on the correspondences in terms of abstract mappings, i.e., without having to cope with implementation details, abstraction from a concrete transformation language is preferable (cf. *abstraction by simplification* in Section 5). Moreover, for being able to cope with large transformation problems, a transformation designer should be empowered to focus on a specific structural heterogeneity at a certain point in time (cf. *abstraction by selection* in Section 5). Furthermore, even this simple example depicts that there are recurring kinds of correspondences demanding for reusable transformations between arbitrary metamodels (cf. *abstraction by generalization* in Section 5). Secondly, since equal correspondences (like the 1:n correspondences in the example) can be resolved in different forms, *tailoring mechanisms* without having to know the internals (black-box reuse) are desired. Thirdly, in order to be able to solve a transformation problem by a divide-and-conquer strategy adequate *composition mechanisms* are required. Finally, to ease the tedious task of specifying transformations, reusable transformations should be offered in an *extensible library.*

3 Composite Mapping Operators to the Rescue

The identified correspondences between metamodels (cf. Fig. 1) need to be refined to a declarative description of the transformation, denoted as *mapping*, which abstracts from a concrete transformation language achieving *abstraction by simplification*. To support transformation designers in resolving structural heterogeneities in the mapping, we provide a library of composite MOps. In this respect, reuse is leveraged since the proposed MOps are generic in the sense that they abstract from concrete metamodel types, thereby achieving *abstraction by generalization*. Thus, MOps can be applied between arbitrary metamodels since they are typed by the core concepts of current meta-modeling languages like Ecore or MOF. To further structure the mapping process we propose to specify mappings in two steps. Firstly, composite MOps, describing mappings between classes are applied, providing an abstract *blackbox-view* (cf. Fig. 2). Composite MOps select specific metamodel extracts to focus on when refining the mapping, providing *abstraction by selection*. Although attributes and references might be necessary to identify the correspondences in the blackbox-view, the actual mapping thereof is hidden in the *whitebox-view*. Secondly, in this whitebox-view the composite MOps can further be refined to specify the mappings between attributes and relationships using so-called *kernel MOps* (cf. Section 4).

We propose composite MOps (cf. Table 1), which have been inspired by a mapping benchmark in the area of data engineering [1], describing recurring mappings between relational and hierarchical schemas. Thereby we identified typical mapping situations being 1:1 copying, 1:n partitioning, n:1 merging, and 0:1 generating of objects, for which different MOps are provided. Since the mapping is executed on instance level, the actually transformed instance set should be configurable by additional conditions. Finally, inverse operators are defined, which allow to re-construct the original source object set.

1:1 Correspondences. MOps handling 1:1 correspondences map exactly one source class to one target class. Currently, two MOps are provided, being a `Copier` and a `ConditionalCopier`. A `Copier` simply creates one target object for every source object, applied in our example to map, e.g., the class `Class` to the class `Entity` (cf. Fig. 2(b)). Furthermore, it is often desired that a target object should only be created if a source object fulfills a certain condition, e.g., only if the `Class.name` attribute is not null. Therefore, the functionality of a `Copier` is extended to a `ConditionalCopier` that requires the specification of a condition, reducing the generated object set. Please note that there is no inverse MOp for the `ConditionalCopier`, since it would require knowing the filtered instances in order to re-construct the original object set.

1:n Correspondences. MOps handling 1:n correspondences connect one source class with at least two target classes and therefore allow to split concepts into finer-grained ones. There are three MOps belonging to this category, being the `VerticalPartitioner`, the `HorizontalPartitioner`, and the `CombinedPartitioner`. In this respect, a `VerticalPartitioner` deals with the problem when attributes of one source class are part of different classes in the target, e.g., the attributes of the class `Reference` are part of the classes `Role` and `Cardinality` in the running example (cf. Fig. 2(d)). Besides this default behavior, aggregation functionality is sometimes needed. Concerning the running example, this is the case when splitting the `Property` concept into the `Attribute` and `Type` concepts, since a `Type` should only be instantiated for distinct `Property.type` values (cf. Fig. 2(c)). In contrast, a `HorizontalPartitioner` splits the object set to different classes by means of a condition, e.g., splitting `References` into unbounded (upperBound=-1) and bounded (upperBound≠-1) ones. As the name implies, a `CombinedPartitioner` combines the functionality of both operators, e.g., if a `Property.type` attribute should additionally be split into numerical and non-numerical types.

n:1 Correspondences. Since MOps handling n:1 correspondences merge several source objects, they require at least two source objects (not necessarily from different classes) to create a single target object, thus representing in fact the inverse operators of 1:n MOps. In this respect, a `VerticalMerger` merges several source objects that are related to each other by references into a single target object. The `VerticalMerger` in our example (cf. Fig. 2 (e)) has two connections to the class `Reference` and a condition expressing that two objects have to be opposites to each other to generate a single instance of the target class `Relationship`. In contrast, a `HorizontalMerger` builds the union of the source objects. Finally, the `CombinedMerger` again combines the functionality of the two before mentioned MOps.

0:1 Correspondences. MOps handling 0:1 correspondences are applied if the target metamodel contains classes without any equivalent source classes. For this reason we provide the `ObjectGenerator` MOp. Therefore mechanisms for generating objects (and its contained values and links) are needed, which is the main contribution of the following section.

Table 1. Composite Mapping Operators

Correspondence	MOp	Description	Condition	Inverse MOp
1:1 - copying	Copier	creates exactly one target object per source object		Copier
	ConditionalCopier	creates one target object per source object if condition is fullfilled	✓	n.a.
1:n - partitioning	VerticalPartitioner	splits one source object into several target objects		VerticalMerger
	HorizontalPartitioner	splits the source object set to different target object sets	✓	HorizontalMerger
	CombinedPartitioner	combines behavior of VerticalPartitioner and HorizontalPartitioner	✓	CombinedMerger
n:1 - merging	VerticalMerger	merges several source objects to one target object	✓	VerticalPartitioner
	HorizontalMerger	creates union of the source object set		HorizontalPartitioner
	CombinedMerger	combines behavior of VerticalMerger and HorizontalMerger	✓	CombinedPartitioner
0:1 - generating	ObjectGenerator	generates a new target object without corresponding source object		n.a.

4 Specification of Composite MOps with Kernel MOps

In the previous section we introduced composite MOps to resolve structural heterogeneities in-the-large by using their blackbox-view. In this section we introduce so-called *kernel MOps* for mapping classes, attributes, and references in all possible combinations and mapping cardinalities which is the basis for resolving structural heterogeneities in-the-small. Subsequently, we discuss how kernel MOps are used to form composite MOps and show exemplarily how the whitebox-view is specified by means of kernel MOps.

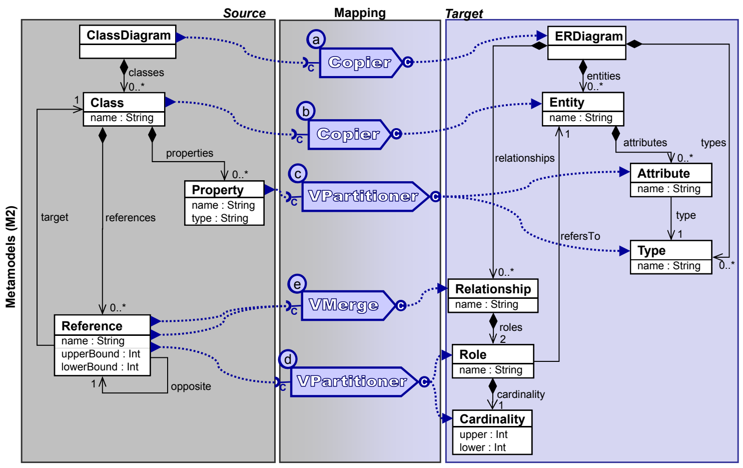

Fig. 2. Solution of the Mapping Example (Blackbox-view of MOps)

4.1 Kernel MOps

In order to provide a MOps kernel, i.e., a minimal set of required MOps to overcome structural heterogeneities in-the-small, we systematically combined the core concepts of metamodeling languages, being (i) classes, (ii) attributes, and (iii) references with different mapping cardinalities, thereby complementing the work of Legler and Naumann [12] focusing on attributes only. Based on this rationale, the kernel MOps allow mapping source elements to target classes (`2ClassMOps`), target attributes (`2AttributeMOps`), and target relationships (`2RelationshipMOps`) (cf. first inheritance level of the kernel MOps layer in Fig. 3). On the second inheritance level we distinguish the following kernel MOps, according to different cardinalities.

1:1 Mappings. $\mathtt{C_2C}$, $\mathtt{A_2A}$, and $\mathtt{R_2R}$ MOps are provided for copying exactly one object, value, and link from source to target, respectively. Therefore, each of their *source* and *target* references point to the same Ecore concept in Fig. 3.

n:1 Mappings. To resolve the structural heterogeneity that concepts in the source metamodel are more fine-grained than in the target metamodel, MOps are needed for merging objects, values, and links. For this, the MOps $\mathtt{C_2^nC}$, $\mathtt{A_2^nA}$, and $\mathtt{R_2^nR}$ are provided that require a merge condition expressed in OCL, specifying how to merge elements, e.g., a function to concatenate several attribute values.

0:1 Mappings. Whereas source and target elements of kernel MOps dealing with 1:1 and n:1 mappings are all of the same type, i.e., pointing to the same Ecore concept in Fig. 3, 0:1 MOps ($\mathtt{0_2C}$, $\mathtt{0_2A}$, and $\mathtt{0_2R}$), bridge elements of different concepts, whereby the 0 indicates that there is no source element of equivalent type. Therefore, these MOps are intended to solve structural heterogeneities resulting from expressing the same modeling concept with different meta-modeling concepts – a situation which often occurs in metamodeling practice – and are therefore a crucial prerequisite to "survive the heterogeneity jungle".

MOps for 0:1 Mappings. In the following, we shortly elaborate on each concrete 0:1 MOp depicted on the third inheritance level in Fig. 3.

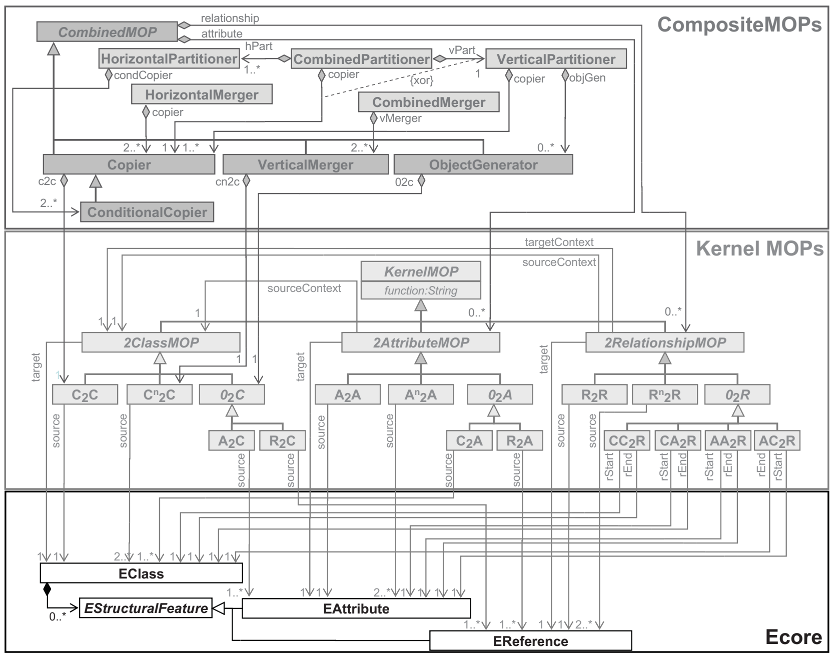

Fig. 3. Metamodel of Kernel and Composite MOps

0_2C MOps. This kind of MOp is used to create objects out of values and links. In particular, the $\texttt{A}_2\texttt{C}$ MOp is used for resolving the structural heterogeneity that in one metamodel a concept is expressed by an attribute, whereby the other metamodel defines the concept by an additional class. In the running example, this kind of heterogeneity occurs between the attribute `Property.type` and the class `Type`. Analogously, it might be the case that one metamodel has a reference between two classes whereas the other metamodel exhibits an additional class in between the corresponding classes. In such a case, for every link an object has to be created which is realized by the $\texttt{R}_2\texttt{C}$ MOp.

0_2A MOps. Besides MOps for explicating source model concepts by means of classes in a target model, MOps are needed to generate values from objects and links. Concerning the first case, a target class may require a certain attribute, for which no corresponding attribute in the source class exists, e.g., an additional `id` attribute. In this case a value has to be set for every generated target object by a $\texttt{C}_2\texttt{A}$ MOp since for every source object a value has to be generated. Concerning the second case, if an object has been generated from a link (cf. $\texttt{R}_2\texttt{C}$ MOp) and the class of the generated object requires an attribute, we provide an $\texttt{R}_2\texttt{A}$ MOp, which sets a value of an attribute of the generated object for every source link.

0_2R MOps. Finally, links may be generated on the target side which have no corresponding links on the source side, but are computable by analyzing the source model. Since links need a source and a target object, it is not sufficient to use $\texttt{A}_2\texttt{R}$ and $\texttt{C}_2\texttt{R}$ MOps, only, instead we need two elements on the left hand side of such MOps. By systematically combining the possible source elements used to generate target objects which are additionally linked, the following MOps are supported: $\texttt{CC}_2\texttt{R}$, $\texttt{AC}_2\texttt{R}$, $\texttt{CA}_2\texttt{R}$, and $\texttt{AA}_2\texttt{R}$ whereby the first letter identifies the element used to generate the source object of the link and the second letter identifies the element used to generate the target object.

In contrast to composite MOps, we do not provide 1:n kernel MOps as they can be reduced to $n \times 1{:}1$ correspondences and thus be again realized as a composite MOp. Furthermore, MOps handling 1:0 correspondences are not needed, since this means that there is a source concept without any corresponding target concept, i.e., no transformation action is required (in the forward direction).

4.2 Composition Model for MOps

To assemble the presented kernel MOps to composite MOps and to bind them to specific metamodels, every MOp has *input ports* with required interfaces (left side of the component) as well as *output ports* with provided interfaces (right side of the component), typed to classes (C), attributes (A), and relationships (R) (cf. Fig. 4). Since there are dependencies between MOps, e.g., a link can only be set after the two objects to be connected are available, every `2ClassMOp` and every composite MOp (which contains `2ClassMOps`) additionally offers a trace port (T) at the bottom of the MOp, providing `context information`, i.e., offering information about which output elements have been produced from which input elements. This port can be used by other MOps to access context information via *requiredContext* ports (C) with corresponding interfaces on top of the MOp, or in case of `2RelationshipsMOps` via two ports, whereby the top port depicts the required source context and the bottom port the required target context (cf. Fig. 4). Since MOps are expressed as components, the transformation designer can apply them in a plug & play manner by binding their interfaces.

For composing kernel MOps we provide the abstract class `CombinedMOp`, aggregating `2AttributeMOps` and `2RelationshipMOps` (cf. Fig. 3). As we identified correspondences between classes in a first step, those kernel MOps dealing with the mapping of classes are an obligatory part of every composite MOp. Therefore, every composite MOp consists of one concrete refinement of the abstract class `2ClassMOp`, dependent on the composite MOps' number of correspondences, being a `Copier` for 1:1 mappings using a $\texttt{C}_2\texttt{C}$ kernel MOp, a `VerticalMerger` for n:1 mappings using a $\texttt{C}^n_2\texttt{C}$ kernel MOp, and an `ObjectGenerator` for 0:1 mappings using a $\texttt{0}_2\texttt{C}$ kernel MOp, as depicted in Fig. 3. Furthermore, composite MOps can again be combined to more complex composite MOps, e.g., a `VerticalPartitioner` consists of several `Copiers` and `ObjectsGenerators`.

In this respect, the presented metamodel (cf. Fig. 3) makes the relationships between kernel MOps and composite MOps explicit thereby representing a conceptual model of heterogeneities, being a main advance compared to our previous work in [8] where no statements about interrelationships have been made.

4.3 Whitebox-View of Composite MOps

In Section 3 we only specified mappings between source and target metamodel classes in a first step, leaving out mappings for attributes and references. This is done in the second step, by switching to the *whitebox-view* of the composite MOps. Because each composite MOp sets the focus by marking involved metamodel classes for a specific mapping, switching to the whitebox-view allows to show only the metamodel elements which are relevant for this mapping. The transformation designer has to complete the mappings by adding appropriate kernel MOps to the composite MOp and by specifying their bindings to attributes and references. Thereby bindings of `2ClassMOps` are automatically set by analyzing the class bindings afore defined in the blackbox-view. To exemplify this, we elaborate on the whitebox-view of the `Copier` (cf. Fig. 4) and the `VerticalPartitioner` (cf. Fig. 5) by means of our running example.

Whitebox-View of Copier. The intention of a `Copier` is to transform one object and its contained values and links to a corresponding target object. Thus, a `Copier` generates one target object for every source object by the $\texttt{C}_2\texttt{C}$ kernel MOp, representing the *fixed part* of the `Copier` (cf. both `Copiers` in Fig. 4) for which the bindings are automatically derived. The bindings of the *variable part*, i.e., arbitrary number of `2AttributeMOps` and `2RelationshipMOps`, are dependent on the specific metamodels. For example, the transformation designer has to use an $\texttt{A}_2\texttt{A}$ kernel MOp to specify the correspondence between the attribute `Class.name` and `Entity.name`. The inclusion of attribute and relationship mappings results in additional ports compared to the blackbox-view (Fig. 2), which only shows class ports. As attributes only exist in the context of an object, the $\texttt{A}_2\texttt{A}$ MOp depends on the $\texttt{C}_2\texttt{C}$ MOp for acquiring context information.

For mapping references, `2RelationshipMOps` are employed in both `Copiers`. For example, the reference `ClassDiagram.classes` is mapped to the reference `ERDiagram.entities` using a $\texttt{R}_2\texttt{R}$ MOp. As links require a source and a target object, the $\texttt{R}_2\texttt{R}$ MOp is linked to the trace port of the $\texttt{C}_2\texttt{C}$ MOp on top and to the trace port provided by the second copier at the bottom. Since the remaining references on the target side, i.e., `ERDiagram.types` and `ERDiagram.relationships`, do not have a counterpart reference on the source side, they have to be mapped by 0:1 MOps. Concerning links instantiated from the reference `ERDiagram.type`, the links' source object is an `ERDiagram` object which is created from a `ClassDiagram` object and the target object is a `Type` object created from a value of the `Property.type` attribute. These source elements are related by the path `ClassDiagram.classes.properties`. In order to reflect this in the target model, the user has to specify this path explicitly since otherwise only the cross product between the involved objects could be built. As the link's source was created on basis of a class and the link's target on basis of an attribute, a $\texttt{CA}_2\texttt{R}$ MOp is applied. Finally, since `ERDiagram.relationships` links have to reflect the context of instances from class `ClassDiagram` and from class `Reference`, described by the path `ClassDiagram.classes.references`, we apply a $\texttt{CC}_2\texttt{R}$ MOp.

Whitebox-View of VerticalPartitioner. Since in the general case the `VerticalPartitioner` only splits attributes of the source class to attributes in several different target classes, `Copiers` can be reused. As shown in Example Application 1 in Fig. 5, the first `Copier` creates `Role` objects and sets the value of the `Role.name` attribute and the second one creates `Cardinality` objects and

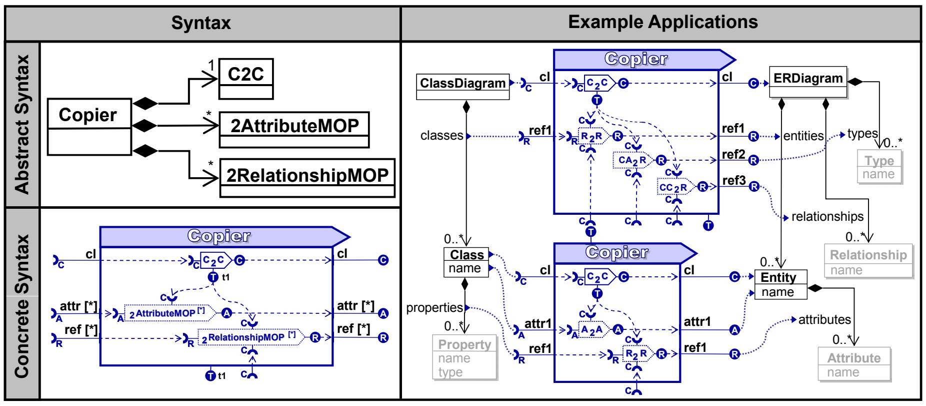
Fig. 4. Whitebox-View of Copier

sets the attributes Cardinality.upper and Cardinality.lower. Again the attribute mappings are specified by the transformation designer in the whitebox-view. Since there is no equivalent for the Role.cardinality link in the source model available, a link for every object pair has to be generated by the CC_2R MOp, which is automatically added to the Copier since it is always required. As the Copiers do not affect the instance set, there has to be a 1:1 relationship between the target classes. In case that the target classes are related by a 1:* relationship, the intention typically is to generate only one object per distinct value of a dedicated source attribute, e.g., only one Type object per distinct Property.type value, which can not be achieved by a Copier. In order to gen-

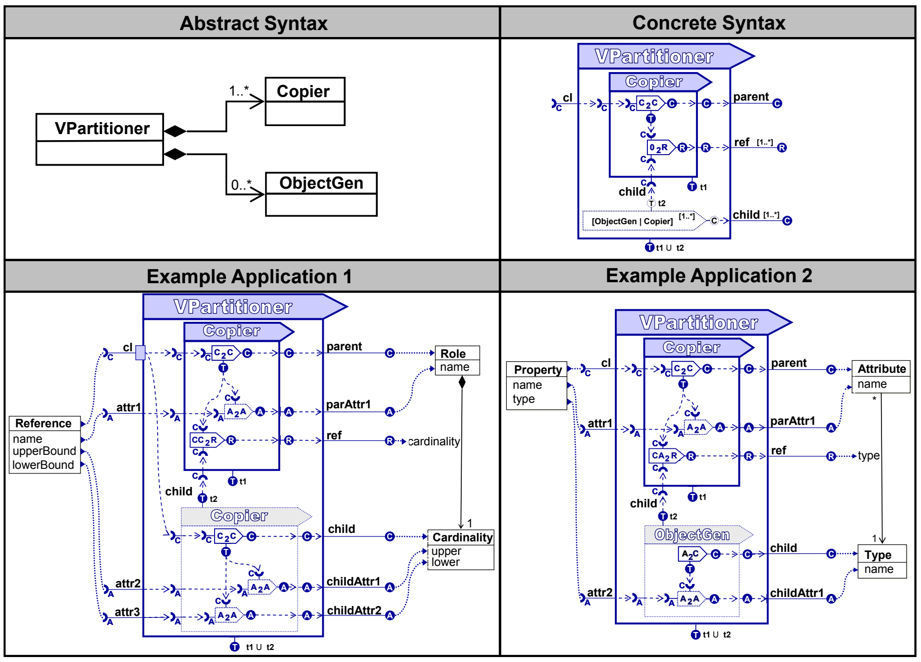
Fig. 5. Whitebox-View of VerticalPartitioner

erate the distinct objects the `ObjectGenerator` is applied, using an `A`$_2$`C` MOp to generate the according objects based on the `Property.type` attribute (cf. Example Application 2 in Fig. 5). In order to decide whether a new object should be created, i.e., an attribute value for which no object has been created up to now, the `A`$_2$`C` MOp queries its own context information.

5 Related Work

In the following, related transformation approaches are considered stemming not only from the area of model engineering, but also from data and ontology engineering since models can be treated as data, metamodels as schemas and transformations as a means for realizing data exchange. A similar analogy can be drawn for the area of ontology engineering. The comparison to our MOps is done in the following according to the introduced criteria (i) abstraction, (ii) variation, (iii) composition, and (iv) library of pre-defined MOps (cf. Table 2).

Table 2. Comparison of Transformation Approaches

	Transformation Approach	**Abstraction**			**Variation**			**Composition**			**Library**				
		By Simplification	By Selection	By Generalization	White-Box Reuse		Black-Box Reuse	Implicit	Explicit		Mapping Cardinality				Extensibility
					Copy-Paste	Inheritance			Internal	External	1:1	1:n	n:1	0:1	
Model Eng.	QVT Relations	control flow	✗	✗	✓	✓	✗	✗	when, where clauses	✗	✗	✗	✗	✗	✗
	TGG (Moflon)	control flow	corres-pondence nodes set focus	genericity (proposed)	✗	✓	✗	✗	✗	layers	✗	✗	✗	✗	✗
	VIATRA	control flow	✗	genericity	✓	✓	✗	✗	✗	ASMs	✗	✗	✗	✗	✗
	ATL (delcarative part)	control flow	✗	✗	✓	✓	✗	✓	lazy rules	✗	✗	✗	✗	✗	✗
	AMW	execution language	~	genericity	✗	✗	✗	✗	nesting	✗	✓	✗	✗	~	~
	MOps	execution language	Composite MOps set focus	genericity	✗	✗	✓	✗	context passing	✗	✓	✓	✓	✓	✓
Data Eng.	Clio/Clip	execution language	~	genericity	✗	✗	✗	✗	context passing	✗	✓	✗	✗	~	✗
	Mapforce	execution language	~	genericity	✗	✗	~	✓	✗	✗	✓	✗	~	~	✓
Ontology Eng.	MAFRA	execution language	✗	genericity	✗	✗	✗	✗	context passing	✗	✓	✗	✗	~	✗

Concerning *abstraction*, different kinds of mechanisms have been investigated, namely (i) *abstraction by simplification*, (ii) *abstraction by selection* and (iii) *abstraction by generalization*. These terms are inspired by the general discussion of abstraction in [11]. In the context of model transformations *abstraction by simplification* denotes the removal of elements from a transformation language, abstracting language-specific details, e.g., for describing correspondences between classes. *Abstraction by selection* is the process of focussing on a certain part of the metamodels only, i.e., the transformation designer is enabled to specify a whole transformation in a divide-and-conquer manner, e.g., partitioning a certain Class into several others. *Abstraction by generalization* is interpreted in the model transformation context, i.e., allowing to focus on the generic transformation logic. This is currently mostly supported by generic types.

With respect to abstraction by simplification, the approaches can be categorized into two groups, namely the ones abstracting from control flow only and the ones abstracting from the underlying execution language at all, thus focussing on the mapping specification, also known as schema mapping approaches from the area of data engineering. Regarding abstraction by simplification, only TGGs [10] provide support, since by defining the correspondence nodes, a focus on a certain metamodel part is set. The subsequent definition of the underlying graph transformation rule therefore just concentrates on this focused part of the metamodel. AMW [4], Clio/Clip [14], and Mapforce[3] provide only basic support in the sense that they allow to collapse certain metamodel parts. Concerning abstraction by generalization, TGGs, VIATRA [18], AMW, Clio/Clip, Mapforce and MAFRA [13] provide support by *genericity.* In contrast, our approach provides all kinds of abstraction mechanisms, being (i) *simplification* through abstracting from the underlying execution language, (ii) *selection* since the composite MOps set a certain focus, which can be concentrated on in the white-box view, and (iii) *generalization* through abstraction from the concrete metamodel since MOps are based on the meta-metamodel.

With respect to *variation mechanisms*, the support for (i) white-box reuse, i.e., the implementation must be known in order to customize the reused components and for (ii) black-box reuse can be distinguished. In this context, mainly white-box reuse is supported so far by existing approaches. It is supported in the simplest form by copy-paste (QVT Relations, VIATRA, ATL [7]) as well as by inheritance (QVT Relations, TGGs, VIATRA, ATL). Regarding black-box reuse, only Mapforce provides basic support, e.g., by allowing to set parameters for string operators. On the contrary, MOps can be tailored without knowing the internals, thus realizing black-box reuse.

Concerning *composition mechanisms*, the approaches have been examined according to the criteria proposed by [3], distinguishing between (i) *implicit composition*, i.e., hard-coded mechanisms not adaptable by the transformation designer and (ii) *explicit composition*, i.e., composition can be explicitly specified by the transformation designer, further classified into (iia) *internal composition*, i.e., intermingling composition specification and rules and (iib) *external composition*, i.e., there is a clear separation between composition specification and rules. With respect to explicit composition mechanisms, all the approaches provide support except Mapforce. Regarding internal composition, most approaches follow this idea in different forms. Whereas QVT allows to describe composition by means of preconditions (when-clauses) and postconditions (where-clauses), ATL allows for the explicit calling of lazy rules. In contrast, Clio/Clip and MAFRA rely on data dependencies between the rules only, i.e., context passing. This is favorable, since the ordering of rules just depends on the data and therefore our approach also follows this idea. Concerning external composition, only TGGs and VIATRA follow this approach allowing for a clear separation between rules and composition specification in the simple form of layers (TGGs) or by the more sophisticated form of ASM (abstract state machine) programs (VIATRA).

Concerning the *library* aspect, only AMW, Clio/Clip, Mapforce and MAFRA provide a basic set of pre-defined components. In this respect, (i) the mapping

[3] http://www.altova.com/mapforce.html

cardinality, i.e., the cardinality supported by the offered MOps and (ii) the possibility to extend the pre-defined set of MOps have been investigated. With respect to mapping cardinality, only 1:1 mappings are supported by all approaches. More coarse-grained MOps, i.e., composite MOps (1:n, n:1, n:m) are neglected so far. Finally, for 0:1 mappings, i.e., the mapping between different metamodel concepts only basic support is provided. Regarding the extension of predefined MOps with user-defined ones, only Mapforce allows for the definition of new ones on the one hand by composing existing ones and on the other hand by writing a code script, i.e., the transformation designer has to define a MOp from scratch. Basically, also AMW could be extended by modifying the underlying metamodel and the HOT responsible for generating the executable ATL code. Nevertheless, this kind of extensibility is quite heavyweight and reserved to specialists. In contrast, our approach supports all kinds of mapping cardinalities and by offering a set of kernel MOps, the library can be easily extended by composing a new MOp out of kernel MOps or other composite ones.

6 Prototypical Implementation

This section elaborates on the prototypical implementation of the presented approach based on the AMMA platform[4]. In particular, we extended AMW [5] to specify mappings by using the presented MOps as well as a HOT [17] for generating executable ATL code [7] out of the mappings (cf. Fig. 6).

Extending AMW. The AMW framework provides a generic infrastructure and editor to declaratively specify weaving models between two arbitrary models. The editor is based on a generic weaving metamodel, defining generic weaving links, which can be extended to specify custom weaving links. The generic weaving links are mainly represented by the abstract classes `WLink`. For each kernel and composite MOp shown in Fig. 3, we introduced a direct or indirect subclass of `WLink` defining the properties of the MOp. In order to ensure context-sensitive mapping constraints, we provide basic verification support based on the EMF Validation Framework[5], e.g., every MOp has to be correctly connected to its source and target metamodel elements as well as to its context mappings.

Execution of Mapping Operators. The extended AMW metamodel defines the abstract syntax of the mapping language, but does not provide means to specify the operational semantics needed for execution of mappings. Since we explicitly represent mappings as a model, we employ a HOT to compile the mappings into executable ATL code. For MOps dealing with 1:1, 1:n and n:1 mappings, declarative ATL code is generated in terms of matched rules. For MOps dealing with 0:1 mappings imperative code blocks are generated. Considering our running example, an $\texttt{A}_2\texttt{C}$ MOp was applied to generate `Type` objects on basis of distinct `Property.type` values, for which a lazy rule is required. To ensure that the lazy rule is called only for distinct values, according trace information is needed. Since ATL provides trace information automatically for

[4] http://wiki.eclipse.org/AMMA
[5] http://www.eclipse.org/modeling/emf/?project=validation

matched rules, only, we implemented our own, more flexible trace model for providing trace information for every rule (irresponsible of the rule type) and for providing traces of values and links in addition to objects. For example, this specific trace information is needed to omit the creation of redundant `Type` objects in our running example. For 0:1 R-MOps it has to be ensured that the source and target objects of the link to be generated have already been established, before the link is set. Therefore, those links are created in an endpoint rule, which is called by the ATL engine just before termination. For more information on our prototype, we kindly refer the interested reader to our project homepage[6].

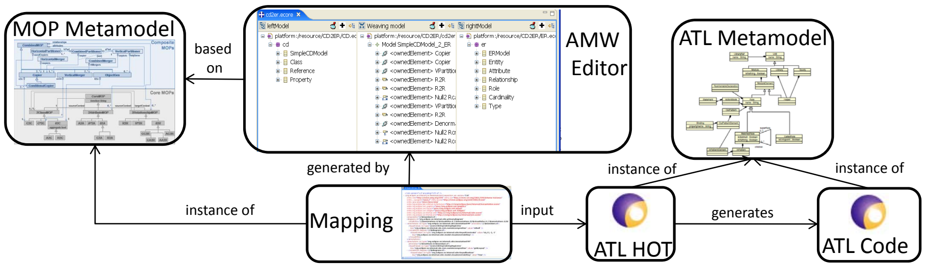

Fig. 6. Generating Transformation Code from Mapping Model

7 Critical Discussion and Future Work

A critical reflection of our MOps opens up several issues for future work.

Incorporation of Additional Modeling Concepts. Currently, only the most important concepts of modeling languages, i.e., classes, attributes and relationships have been considered. It would be highly desireable, however, to extend our MOp library to be able to deal also with concepts such as inheritance. We have already published first ideas in this direction in [8].

Effort Reduction by Matching Techniques. Our mapping approach consists of two steps being first the definition of mappings on class level which has to be further refined with attribute and relationships mappings. Since this refinement can be time-consuming, matching strategies [15, 16] may be applied to automatically derive attribute and relationship mappings. Matching in this context may be especially beneficial since through setting a certain focus in the blackbox view, the search area is already restricted.

Impedance Mismatch Reduction by Other Execution Languages. The operational semantics of our MOps is defined using a HOT to ATL. Thus, there is an impedance mismatch between the abstract mapping specification and the executable code, which hinders understandability and debugging of the generated code. Therefore, the translation to other transformation languages should be investigated, trying to identify which transformation language fits best to the mapping specification. In this respect, the applicability of our own transformation language TROPIC [19] should be investigated as well.

[6] www.modeltransformation.net

Usability Evaluation. Kernel MOps provide rather fine-grained operators for overcoming structural heterogeneities. Nevertheless, they abstract from the intricacies of a certain transformation language, e.g., 0:1 MOPs often require querying trace information or helper functions which require considerable effort in implementing manually. Therefore, the usability of our approach has to be investigated in further studies, evaluating the expected advantage.

References

1. B. Alexe, W.-C. Tan, and Y. Velegrakis. STBenchmark: Towards a Benchmark for Mapping Systems. *VLDB Endow.*, 1(1):230–244, 2008.
2. J. Bézivin. On the Unification Power of Models. *Journal on Software and Systems Modeling*, 4(2):31, 2005.
3. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
4. M. D. Fabro and P. Valduriez. Towards the development of model transformations using model weaving and matching transformations. *SoSym*, 8(3):305–324, 2009.
5. M. Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: A Generic Model Weaver. In *Proc. of IDM'05*, 2005.
6. R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.
7. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.
8. G. Kappel, H. Kargl, T. Reiter, W. Retschitzegger, W. Schwinger, M. Strommer, and M. Wimmer. A Framework for Building Mapping Operators Resolving Structural Heterogeneities. In *Proc. of UNISCON'2008*, pages 158–174, 2008.
9. V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: A context-based approach. *The VLDB Journal*, 5(4):276–304, 1996.
10. A. Koenigs. Model Transformation with Triple Graph Grammars. *Model Transformations in Practice Workshop of MODELS'05, Montego Bay, Jamaica*, 2005.
11. J. Kramer. Is abstraction the key to computing? *Com. ACM*, 50(4):36–42, 2007.
12. F. Legler and F. Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. *Proc. of BTW'07*, 2007.
13. A. Maedche, B. Motik, N. Silva, and R. Volz. MAFRA - A MApping FRAmework for Distributed Ontologies. In *Proc. of EKAW'02*, pages 235–250, 2002.
14. A. Raffio, D. Braga, S. Ceri, P. Papotti, and M. A. Hernández. Clip: a visual language for explicit schema mappings. In *Proc. of ICDE'08*, pages 30–39, 2008.
15. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.
16. R. Ramos, O. Barais, and J. M. Jézéquel. Matching Model-Snippets. In *Proc. of MoDELS'07*, 2007.
17. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *Proc. of ECMDA-FA'09*, pages 18–33, 2009.
18. D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *Proc. of UML'04*, pages 290–304. Springer, 2004.
19. M. Wimmer, A. Kusel, T. Reiter, W. Retschitzegger, W. Schwinger, and G. Kappel. Lost in Translation? Transformation Nets to the Rescue! In *Proc. of UNISCON'09*, pages 315–327, 2009.