

# Unifying Approach for Model Transformations in the MOF Metamodeling Architecture

Ivan Kurtev, Klaas van den Berg

Software Engineering Group, University of Twente

P.O. Box 217, 7500 AE, Enschede, the Netherlands

{kurtev, vdberg}@cs.utwente.nl

## Abstract

In the Meta Object Facility (MOF) metamodeling architecture a number of model transformation scenarios can be identified. It could be expected that a metamodeling architecture will be accompanied by a transformation technology supporting the model transformation scenarios in a uniform way. Despite the fact that current transformation languages have similarities they are usually focused only on a particular scenario. In this paper we analyse the problems that prevent the usage of a single language for different transformation scenarios. The problems are rooted in the current organization of MOF and especially in its inability to define explicitly the mechanism of model instantiation. This causes a tight coupling between a transformation language and the instantiation mechanism specific at the level it operates upon. We propose an organization of the MOF architecture based on a simple and uniform representation of all model elements no matter at which level they are defined. In this framework different instantiation mechanisms are defined as transformations between model elements. We present a transformation language based on that paradigm which is independent of the instantiation mechanism specific for a given level.

## 1. Introduction

A key characteristic of the MDA (Model Driven Architecture) [8] is the notion of model transformations. A model transformation is a set of transformation rules and techniques operating on a source model to produce another model, the target model. A number of model transformation scenarios can be identified in current OMG standards and other publications [11][9][12][13]. Figure 1 shows four transformation scenarios in the context of the MOF (Meta Object Facility) [10] metamodeling architecture.

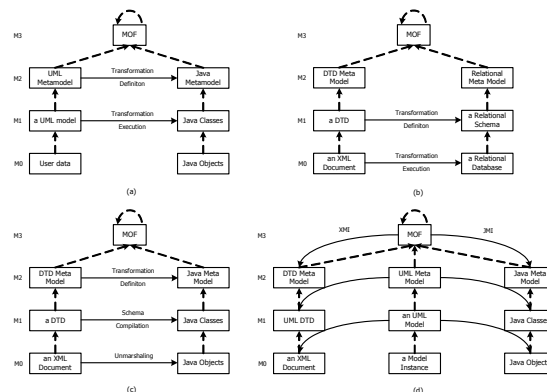


Figure 1 Model Transformation Scenarios

Figure 1a shows a transformation specified between two MOF meta models (UML and Java metamodels). This is the context of the Query/Views/Transformation (QVT) Request for Proposals issued by OMG [11]. It involves transformation specification at level M2 and execution of the transformation at level M1. Figure 1b shows similar scenario shifted one level down. It involves transformation over data at level M0. The diagram shows a transformation specified between a concrete DTD (Document Type Definition) and a concrete relational schema. The execution of the transformation converts an XML document to a relational database. This scenario is common in data warehousing and addressed in the Common Warehouse Metamodel (CWM) [9]. Figure 1c shows the Data Binding approach for XML processing [13]

from the perspective of the MOF architecture. In that case a transformation is specified at level M2 and executed at the two lower levels. The execution at level M1 is known as schema compilation. The correspondence derived between the constructs in the models at level M1 serves as a specification of the transformation executed at level M0 known as unmarshaling. In current data binding tools transformation rules applied during the unmarshaling are usually not powerful enough to express the correspondence between an arbitrary schema and an arbitrary set of application classes. In this respect XML processing can benefit from the ability of model transformation languages to express complex transformations [7].

The three scenarios shown so far may be regarded as intra-level transformations. Models being transformed reside at the same level. Another scenario is shown in Figure 1d. We refer to this scenario as inter-level transformation. Figure 1d shows two standard mappings in MOF: XML Metadata Interchange (XMI) [12] and Java Metadata Interchange (JMI). Both map the MOF Model at level M3 to a meta model at level M2 (the DTD meta model and the Java meta model respectively). These transformations are executed on models at level M2 (for example, the UML meta model) and the result is a model at level M1. Furthermore, a UML model at level M1 may be transformed to an XML document or to a set of Java objects residing at level M0.

How do the current model transformation techniques support these scenarios? The QVT initiative is aimed at the definition of a standard transformation language for the first scenario. CWM solves problems in the second scenario for a number of commonly used data sources. The third scenario is supported by proprietary data binding tools that do not consider transformations in the context of MOF. The mappings in the fourth scenario are described in a semi-formal notation using grammars, templates and textual descriptions. Although the transformation approaches taken in QVT and CWM share a number of similar concepts it is not possible to use a single transformation language for the first two scenarios. The scenario in Figure 1c executes a transformation at level M1 and also derives a new transformation executed at the lower level. Current model transformation languages do not address the problem of automatic derivation and execution of a transformation over more than two consecutive levels. Finally, both QVT and CWM do not consider inter-level transformations.

One would expect that the outlined scenarios within the MOF architecture will be addressed in a uniform way, that is, the organization of MOF will allow a transformation language to operate between any levels. In this paper we analyze the problems that prevent the usage of a single language for all the scenarios outlined above. In our opinion the problems are rooted in the current organization of MOF and especially its inability to define explicitly the mechanism of model instantiation. This causes a tight coupling between a transformation language and the instantiation mechanism specific to the level it operates upon. We propose an organization of the MOF architecture based on a simple and uniform representation of all the model elements no matter at which level they are defined. In this framework different instantiation mechanisms are defined as transformations between model elements. We present a transformation language based on that paradigm which is independent of the instantiation mechanism specific for a given level. Thus, the language is able to express transformations between models at arbitrary level.

The paper is organized as follows. Section 2 gives detailed description of the problems we want to tackle. Section 3 describes our approach for representing model elements in the MOF architecture. Section 4 illustrates how instantiation mechanisms can be specified as transformations and used in the context of our transformation language. Section 5 analyses related work and section 6 gives the conclusions.

## 2. Problem Statement

The languages proposed as an answer to the QVT RFP are based on the instantiation mechanism used to create MOF metamodels. A transformation selects instances of MOF classes in a source model at level M1 and produces instances in a target model at the same level. The definition of a transformation language that works on every model at level M1 is possible because all model elements conform to the MOF semantics. It defines precisely which constructs at level M2 may be instantiated (instances of MOF *Classifier* that are not abstract) at level M1 and the structure of these instances (having identity, slots and links). Since the constructs at levels M3, M2 and M1 conform to a common structure and share the same instantiation mechanism it is possible to define a language that works on any model at level M1. The MOF specification, however, does not mention the structure of the instances at M0 level and how they are related to their meta constructs at level M1. This is specific to a particular user-defined modeling language and is outside the scope of the MOF specification. Since the QVT transformation languages are tightly coupled with the mechanism of MOF *instanceOf* relation it is not possible to use them to define transformations between models at level M1 and to execute them on data at level M0.

The *instanceOf* relation between a model construct in M1 and its instances in level M0 may differ from the *instanceOf* relation described in MOF. This observation has been made in [4] where it is argued that the actual number of levels is 3 instead of the widely accepted view of 4 levels. The lack of standard way to describe the instantiatable constructs and instantiation mechanism for the model elements at level M1 prevents the usage of QVT languages for the M0 level. If a transformation is defined between two user models in M1 then the transformation language has to know which model elements are instantiatable and how the instances are populated with data.

How does the CWM solve that problem in dealing with a variety of data sources such as XML, relational, and record based? It takes the concepts of classes and instances defined in UML and reuses them in the CWM metamodel. Then a concrete metamodel that would be separately defined at level M2 is now defined as a specialization of the CWM metamodel. Constructs that specialize the Class construct can be instantiated and their instances conform to constructs that specialize Object. The problem here is the inability to handle data conforming to metamodels at level M2 if they are not defined as specializations of the CWM metamodel.

If a transformation language is capable of transforming models residing at arbitrary level then it will require a common representation of the model constructs no matter the level they reside in and a uniform way of treating the different *instanceOf* relationships between levels. The discussion above showed that MOF does not provide these mechanisms. As a result current transformation languages are coupled with particular instantiation mechanisms. It is not possible to parameterise a language with respect to other instantiation mechanisms. This is due to the fact that in MOF there is no standard way to define *instanceOf* relations.

### 3. Approach

The approach we take to solve the problems explained in the previous sections is based on two ideas. First, we represent the model elements in MOF architecture according to a simple generic model no matter the level they reside in. We define a transformation language that operates on instances of that generic model. Second, we consider three main operations that occur in transformations: instantiation of an element from a type, querying the values of the features of elements and setting values to the features. We define these operations as transformations over model elements. The transformation language may be configured with definitions of these operations. In that way we achieve decoupling between the language and the instantiation mechanism specific to a certain metamodel.

#### 3.1. Structure of Model Elements

Our model to represent the model elements is shown in Figure 2. MOF architecture is viewed as a homogeneous space populated with model elements. The level in which a model element resides does not affect its representation.

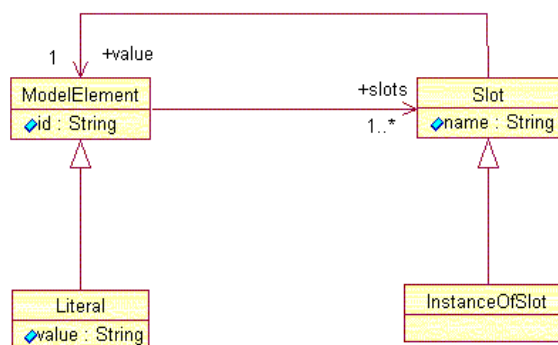
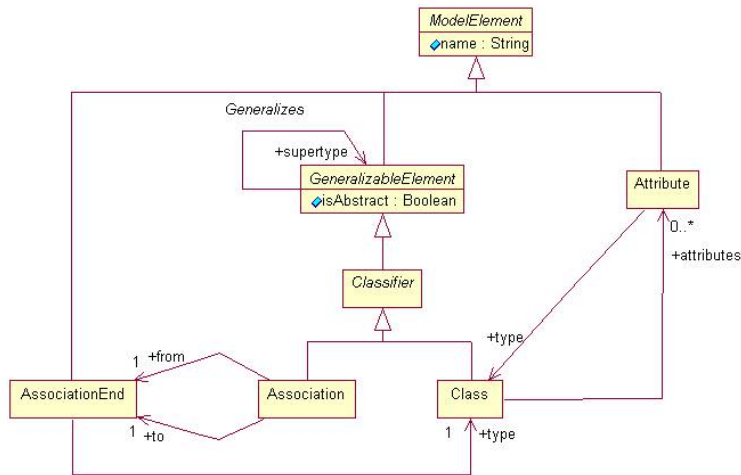


Figure 2 Generic Model of Model Elements

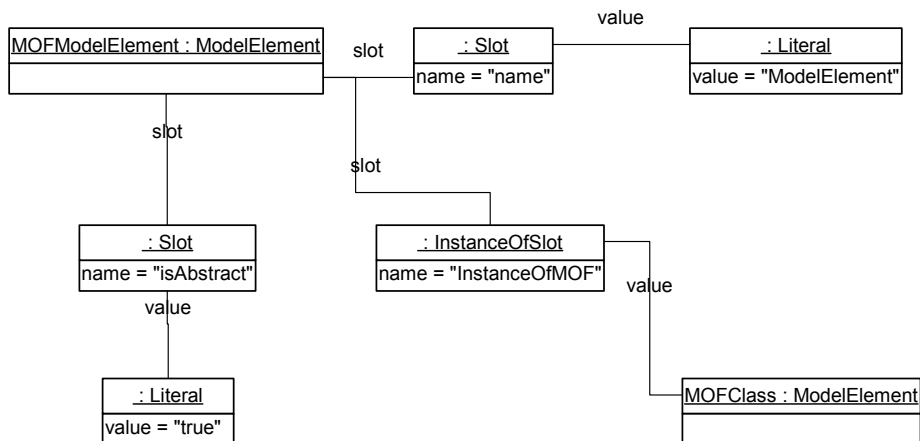
Every model element has a unique identity and a number of named slots. Every slot has a value. A slot value may be another model element or a literal. Slot value may also be a collection. There is at least one slot that indicates *instanceOf* relationship with another model element. We denote the set of all model elements with *ME*. If *me* is a model element then *me.slots* returns the set of all slots of *me* excluding the *instanceOf* slots. For a particular slot with name ‘n’, the expression *me.n* evaluates to the value of the slot.

Figure 3 shows a simplified MOF Model. Simple data types and the multiplicity of attributes and association ends are omitted for simplicity. We assume that all the associations are unidirectional.



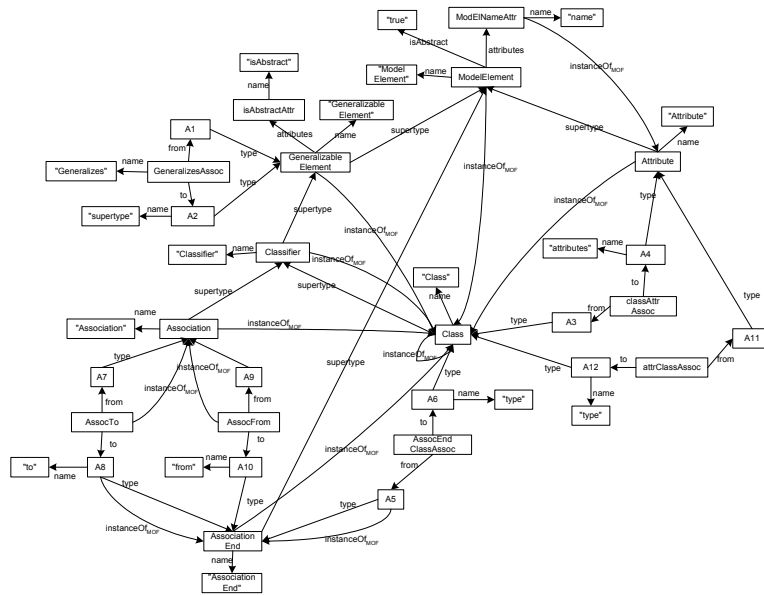
**Figure 3 Simplified MOF Model**

This simplified MOF Model may be represented as an instance of the model in Figure 2. All instances of MOF Class construct and also their instances are represented as a model element. Instances of attributes and associations are represented as slots and are used to connect the model elements. A simple object diagram that shows part of the representation of the MOF abstract class ModelElement is shown in Figure 4. The slot named “InstanceOfMOF” shows the instantiation relation to the MOF Class construct.



**Figure 4 Object diagram representing MOF abstract class ModelElement as instance of the generic model**

In this paper a more concise notation will be used for showing models represented as sets of model elements. Figure 5 shows part of the simplified MOF Model represented as a set of model elements. Model elements are shown as rectangles that contain an identifier of the element. Literals are also shown as rectangles containing the literal value enclosed by quotes. Slots are represented as arrows labelled with the name of the slot. Arrows are pointing to the slot values. The slots that represent *instanceOf* relationship are shown as arrows from an element to its type labelled ‘instanceOf<sub>MOF</sub>’. To improve readability we do not show the full representation. Simple types and their relations to the literal elements and attributes are skipped. Slots ‘isAbstract’ and ‘instanceOf<sub>MOF</sub>’ are also skipped for the most elements.



**Figure 5 Representation of the MOF Model (Figure 3) as a set of model elements**

### 3.2. The MOF *instanceOf* relation

A simplified definition of MOF *instanceOf* relation is given below. It does not take into account all the constraints defined in the specification and does not treat multiplicity and generalization.

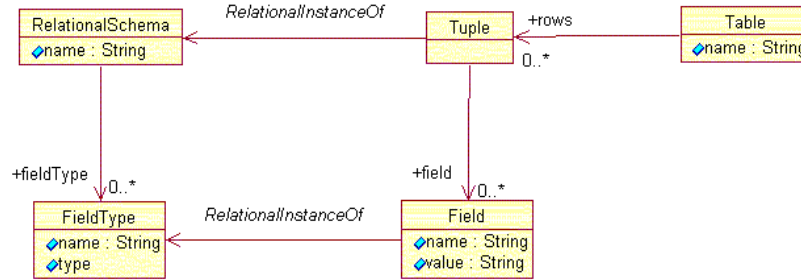
Let  $mI$  and  $mT$  are model elements from MOF and  $mI.instanceOf=mT$ . The following holds:

- $mT.instanceOf=Class$ . Only Class instances may be further instantiated into model elements.
- For each  $a \in mT.attributes$  there exist exactly one slot  $s \in mI.slots$  such that  $s.name=a.name$  and  $s.value.instanceOf=a.type$ . The intuitive meaning is that for every attribute in the type there is exactly one slot in the instance with name equal to the name of the attribute.
- For each model element  $assoc$  for which  $assoc.instanceOf=Association$  and  $assoc.from.type=mT$  there exist exactly one slot  $s \in mI.slots$  such that  $s.name=assoc.to.name$  and  $s.value.instanceOf=assoc.to.type$ . The intuitive meaning is that for every outgoing association in the type there is exactly one slot in the instance with name the name of the target association end.

It is possible to check that the graph in Figure 5 satisfies these constraints. The definition given above also defines an instantiation mechanism for every MOF class, i.e. how to identify the names and types of the slots of instances. From now on we will refer to that relation as *instanceOf<sub>MOF</sub>*.

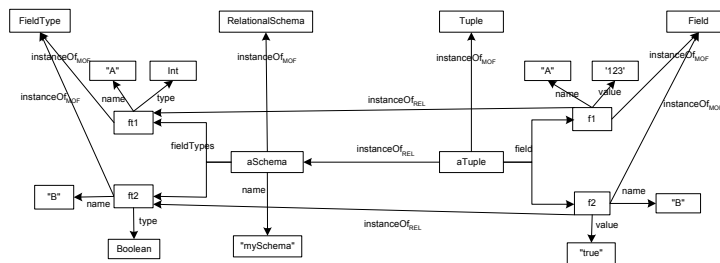
### 3.3. Example: the *instanceOf* relation for the relational model

Some approaches [2][4] reduce the number of metalevels to 3 by defining the instance model of a given metamodel at the same level M2. Therefore both the concrete models and their instances are situated in level M1 and instantiated with the standard MOF mechanism. However, this does not remove the presence of the second *instanceOf* relation defined within the metamodel. The authors of [2] identify the existence of these distinct *instanceOf* relations and distinguish between linguistic and ontological instantiations. We illustrate this approach by modeling relational databases. The relational model shown in Figure 6 contains both the metamodel of relational schema (*RelationalSchema*, *FieldType*) and the model of its instances (*Table*, *Tuple* and *Field*).



**Figure 6 Relational metamodel and its instance model both at level M2**

The model is instantiated in a standard way through the  $instanceOf_{MOF}$  and one example instance is shown in Figure 7. The data (the model elements  $aTuple$ ,  $f1$  and  $f2$ ) that would reside in level M0 are now at level M1 and may be queried according to the types  $Tuple$  and  $Field$ . For instance, to access the value of field B one has to write the expression  $aTuple.field(name="B").value$  that returns 'true'. However, the more natural way is to use the relational schema of the tuple (represented by  $aSchema$  model element) and to write the expression  $aTuple.B$  that reflects instantiation relation between  $aTuple$  and  $aSchema$ . This querying is not possible because  $aTuple$  does not have slot with name "B".



**Figure 7 A particular relational schema and relational data both at level M1**

This illustrates the need that both instantiation mechanisms should be available. A single element may conform to more than one model element and the  $instanceOf$  relations are defined differently. In the rest of the paper we will refer to the conformance between schemas and tuples as  $instanceOf_{REL}$ .

## 4. Specifying *instanceOf* relations as transformations

### 4.1. Transformation Language

In this section we briefly describe a transformation language capable of specifying transformations between models at any level. The language is a generalization of the one applied to XML processing described in [7]. Because of the limited space we do not give full language description. Example syntax with short explanation is given further in this section.

A transformation language usually selects elements from the source model and instantiates elements in the target model. Selection may be based on the type of elements and the values of their features. Instantiation of a target element is usually followed by assigning values to its features. The names and types of the available features are constrained by the type of the element. In our framework a source model and a target model are comprised of model elements according to our generic model. Since all the model elements share the same representation it does not matter at which level they reside. The list below explains how our language deals with four common operations found in model transformations:

- *Selection of source model elements on the base of their type.* An  $instanceOf$  relation is represented as a special slot of a model element. Moreover, because multiple  $instanceOf$  slots are allowed we may select on the base of more than one type. This helps in dealing with both linguistic and ontological instantiations;
- *Accessing the values of the slots.* The possible slot names are defined by the type and the instantiation mechanism. For example, in  $instanceOf_{MOF}$  a slot name directly corresponds to a concrete slot of the model element. In the case of  $instanceOf_{REL}$ , however, slot names ('A' and 'B' in the example above), do not correspond to real slots used in the representation of model element. This raises two questions: first, what is the instantiation mechanism used to generate the model element and second, if the slots are not in the representation how the values are

obtained. The answer of the first question is that one instantiation mechanism is always chosen as default and it determines the slot names. We can think about it as a linguistic *instanceOf* in the spirit of [2]. In our approach we choose *instanceOf<sub>MOF</sub>* as default mechanism. To answer the second question we define a translation mechanism that maps the slot names implied by a given type to the slots obtained through the basic instantiation.

- *Instantiating a model element of a given type.* An instantiation mechanism (no matter basic or non-basic) is always treated as a transformation from the model element that serves as a type to another model element (the instance) with empty slots.
- *Setting values of slots.* This problem is similar to the problem of accessing values of slots. The same two cases are presented here. If the slot exists in the representation of the element the value is set directly on the slot. If, however, the slot name comes from a type not used for the default instantiation a translation mechanism is required. Setting the slot value is treated as an in-place transformation over the model element whose slot is being set;

In summary, the four basic operations in model transformations are highly dependent on the instantiation mechanism used for a particular model element. In our framework we associate every instantiation mechanism with three transformations:

- *Instantiation:* a transformation from the model element that plays the role of a type to another model element (the instance) with empty slots;
- *Slot accessor:* a transformation from the model element being queried to the value of the slot that is queried;
- *Slot mutator:* an in-place transformation over a model element that changes it by setting a value to a slot;

All transformations are written in the transformation language and operate over the generic representation based on model elements and slots. During the transformation execution the transformation engine executes these transformations.

#### 4.2. Example of defining *instanceOf<sub>REL</sub>*

First we will define the *instanceOf<sub>MOF</sub>* relation since it will be used as default instantiation mechanism (or the linguistic one). The following rules will be explained below.

```
MOFClassInstantiation ModelElementRule{
  source [c:Class, condition {c.isAbstract=false}]
  target [ModelElement {slots=instSlot},
         instSlot:InstanceOfSlot {name='instanceOfMOF', value=c}]

  SlotRules {
    attributeSlots
    source [a:Attribute=c.attributes]
    target [slots]

    associationSlots
    source [assoc:Association, condition{assoc.from.type=c}]
    target [slots]
  }
}

MOFAttributeToSlot ModelElementRule{
  source [a:Attribute]
  target [Slot {name=a.name}]
}

MOFAssociationToSlot ModelElementRule{
  source [assoc:Association]
  target [Slot {name=assoc.to.name}]
}
```

The language defines two types of rules: model element rules and slot rules. Model element rules create new model elements and their slots in the target model. In the example the rule *MOFClassInstantiation* creates a model element for every MOF class that is not abstract and sets that class as the type of the element by instantiating a special slot *instSlot* of type *InstanceOfSlot*. Slot rules calculate the slot values of a model element. Two rules *attributeSlots* and *associationSlots* set the slots of the created model

element. The other two model element rules *MOFAttributeToSlot* and *MOFAssociationToSlot* create slots from the class attributes and associations respectively.

Assume that the schema *aSchema* (see Figure 7) exists and we want to instantiate a tuple that conforms to it. The tuple must have exactly two fields named ‘A’ and ‘B’ respectively. That instantiation will be based on *instanceOf<sub>REL</sub>* and the following transformation with two model element rules will be used:

```

RelSchemaInstantiation ModelElementRule{
  source [s:RelationalSchema]
  target [Tuple(field, instanceOfREL =s)]

  SlotRules{
    Fields
    source [f:FieldType=s.fieldTypes]
    target [field]
  }
}

FieldTypeToField ModelElementRule{
  source [ft:FieldType]
  target [Field{name=ft.name}]
}

```

In this transformation the targets are not the generic classes *ModelElement* and *Slot* as in the previous transformation. Instead *Tuple* and *Field* are used. The transformation engine will use the default *instanceOf<sub>MOF</sub>* transformation to instantiate *Tuple* and *Field* and will build the underlying representation.

We also specify a slot rule used for slot value access. In our example all slot rules share a common structure that allows specifying them in a single template that will be instantiated with a concrete slot name (the input parameter *slotName*).

```

TupleFieldToSchemaField SlotRule inputParameters [slotName: String, contextNode: Tuple]{
  source [f:Field=contextNode.field, condition {f.name=slotName}]
  target [Slot{name=slotName}=f.value]
}

```

This rule will be executed on an instance of *Tuple* class supplied through the input parameter *contextNode*. It will navigate over the fields and will select a field with name equal to the input parameter. The value will be obtained via the expression *f.value* and the result will be assigned to a slot with name the value of the input parameter.

To mutate slot values we use the following rule, which performs an in-place transformation over a model element that is an instance of *Tuple*:

```

SettingSlotValue ModelElementRule{
  inputParameters [slotName:String, newValue:ModelElement]
  source [s:Tuple, f:Field=s.field, condition {f.name=slotName}]
  target [update f {value=newValue}]
}

```

The rule takes two input parameters: the slot name and the value to be assigned. The source of the rule (the variable *s*) is determined from the transformation context. Variable *f* is obtained from *s* and satisfies the condition. Then the slot of *f* with name ‘value’ will be set.

## 5. Related Work

Metamodeling architectures based on a common representation of the elements in different levels can be found in other domains of computer science. RDF Schema [6] defines a three level architecture where all constructs are represented as triples according to the RDF data model [3]. The approach for metamodeling described in [5] also has three levels and 5 types of instantiation mechanisms called conformance relationships. Different data schemas may be modeled in this framework such as XML, Topic maps, relational model, etc. The framework uses a transformation language based on logical formulas. Transformations between any levels are possible.

The authors of [14] propose a multilevel metamodeling framework where instantiation and generalization are treated in a uniform way. In [1] the instantiation mechanism is explicitly defined as a function that can be applied on a model at any level. That function resembles the MOF instantiation mechanism and is reused also in the UML metamodel. The paper does not study how other instantiation mechanisms would be defined in that framework.

## 6. Conclusions

We presented an approach for defining a model transformation language that allows specifying transformations between models residing at arbitrary level in the MOF architecture. Our language treats the MOF architecture as a homogeneous modeling space consisting of model elements all with the same generic structure. Different *instanceOf* relations may be defined within that space and implemented as a transformation from a particular type to its instances. The accessor and mutator operations over model elements are also specified as transformations. In that way multiple instantiation mechanisms may be implemented with the transformation primitives thus making the transformation language independent of these mechanisms.

As a next step we plan to study the nature of the generalization relations found in different metamodels and the possibility to define them with the primitives of our transformation language.

## References

- [1] Álvarez, J., Evans, A., Sammut, P., *Mapping between Levels in the Metamodel Architecture*, In Proceedings of UML2001, Springer-Verlag Heidelberg, Toronto, Canada, 2001
- [2] Atkinson, C., Kühne, T., *Model-Driven Development: A Metamodeling Foundation*, IEEE Software 20(5), 2003, pp. 36-41
- [3] Beckett, D., *RDF/XML Syntax Specification*, W3C Document, 2003
- [4] Bézivin, J., Lemesle, R., *Ontology-Based Layered Semantics for Precise OA&D Modeling*, ECOOP'97 Workshop Reader, Finland, 1997
- [5] Bowers, S., Delcambre, L., *On Modeling Conformance for Flexible Transformation over Data Models*, In Proc. of the Workshop on Knowledge Transformation for the Semantic Web at the 15th European Conference on Artificial Intelligence (KTSW-2002), Lyon, France, 2002
- [6] Brickley, D., Guha, R. V., *RDF Vocabulary Description Language 1.0: RDF Schema*, W3C Document, 2003
- [7] Kurtev, I., van den Berg, K., *Model Driven Architecture based XML Processing*, ACM Symposium on Document Engineering, Grenoble, France, 2003
- [8] OMG. *MDA Guide*. 2003
- [9] OMG/CWM. *Common Warehouse Metamodel*, 2001
- [10] OMG/MOF. *Meta Object Facility Specification*. 2000
- [11] OMG. *MOF 2.0 QVT RFP*, 2002
- [12] OMG/XMI: *XML Metadata Interchange (XMI) Version 2*, 2001
- [13] Reinhold, M. An XML Data-Binding Facility for the Java Platform. 1999
- [14] Varró, D., Pataricza, A., *VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML*, Software and System Modeling 2(3), Springer-Verlag, 2003, pp. 187-210

