On Metamodel Superstructures Employing UML Generalization Features

Martin Gogolla, Matthias Sedlmeier, Lars Hamann, Frank Hilken

Database Systems Group, University of Bremen, Germany
{gogolla|ms|lhamann|fhilken}@informatik.uni-bremen.de

Abstract. We employ UML generalization features in order to describe multi-level metamodels and their connections. The basic idea is to represent several metamodel levels in one UML and OCL model and to connect the metamodels with (what we call) a superstructure. The advantage of having various levels in one model lies in the uniform handling of all levels and in the availability of OCL for constraining models and navigating between them. We establish the connection between the metamodel levels by typing links that represent the instance-of relationship. These typing links are derived from associations that are defined on an abstraction of the metamodel classes and that are restricted by `redefines` and `union` constraints in order to achieve level-conformant typing. The abstraction of the metamodel classes together with the connecting associations and generalizations constitutes the superstructure.
Keywords. UML, OCL, Model, Metamodel, Metamodel constraint, Generalization, Redefines constraint, Union constraint.

1 Introduction

Software engineering research activities and results indicate that metamodeling is becoming more and more important [3, 4, 9]. However, there are a lot of discussions about notions in connection with metamodels like *potency* or *clabject* where no final conceptual definition has been achieved. On the other hand, software tools for metamodeling are beginning to be developed [5, 2].

Here, we propose to join the metamodels of several levels into one model (as in our previous work [7] without the use of `redefines` constraints) and to connect the levels with associations and generalizations. Typing conformance and strictness can be achieved through particular UML and OCL generalization constraints. General restrictions between the metamodel levels can be specified through the power of OCL. Restrictions can be built on metamodels and on the metamodel architecture. The metamodel architecture is the connection between (what we call) the metamodel superstructure and the contributing metamodels.

Our work has links to other metamodeling approaches. The tool Melanie [2] is designed as an Eclipse plug-in supporting strict multi-level metamodeling and support for general purpose as well as domain specific languages. Another tool is MetaDepth [5] allowing linguistic as well as ontological instantiation with an arbitrary number of metalevels supporting the potency concept. In [9] the authors describe an approach to flatten metalevel hierarchies and seek for a level-agnostic metamodeling style in contrast to the OMG four-layer architecture.

The structure of the rest of the paper is as follows. Section 2 gives a first, smaller example for a metamodel superstructure. Section 3 discusses a larger example. Section 4 shows other metamodel superstructures. The contribution is closed with a conclusion and future work in sect. 5.

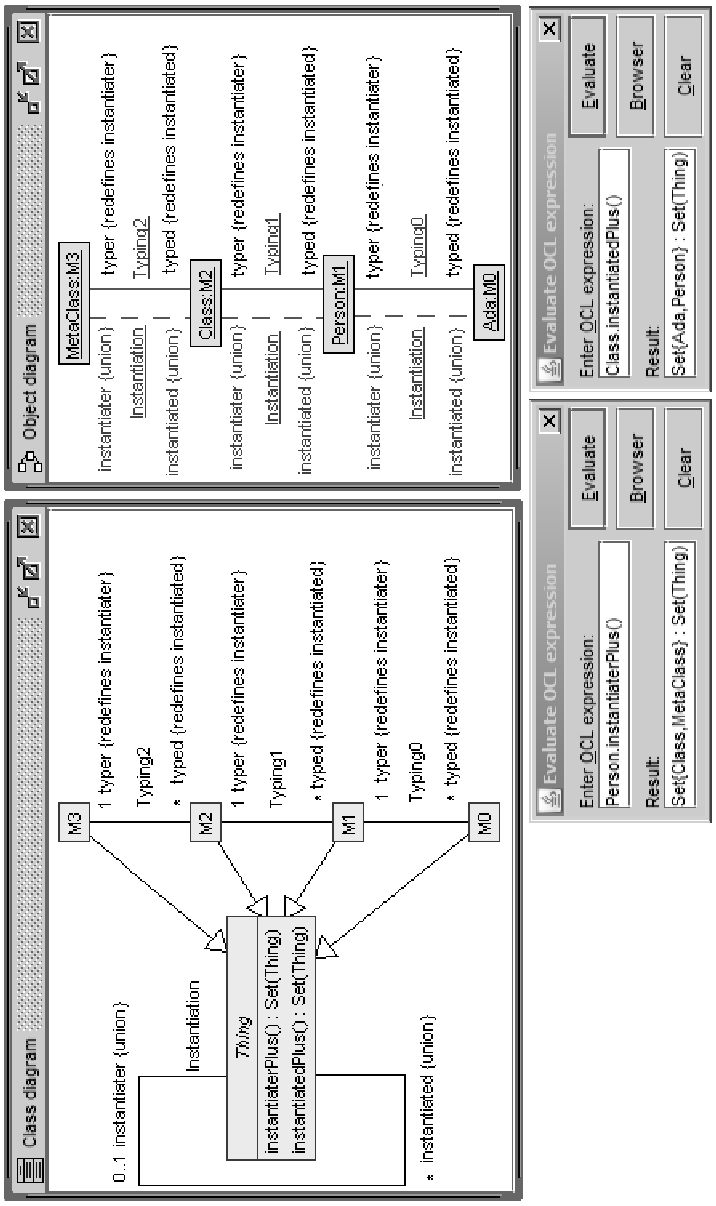
2 Superstructure Example with Ada, Person, Class, and MetaClass

The example in Fig. 1 shows a substantially reduced and abstracted version of the OMG four-level metamodel architecture with modeling levels M0, M1, M2, and M3. Roughly speaking, the figure states: `Ada` is a `Person`, `Person` is a `Class`, and `Class` is a `MetaClass`. The figure does so by formally building an object diagram for a precisely defined class diagram including an OCL invariant that requires cyclefreeness when constructing instance-of connections. The distinction between `MetaClass` and `Class` is that when `MetaClass` is instantiated something is created that can be instantiated on two lower levels whereas for `Class` instantiation can only be done on one lower level. The model has been formally checked with the tool USE [6]. In particular, we have employed the features supporting UML generalization constraints as discussed in [1, 8].

Concepts on a respective level M_x are represented in a simplified way as a class M_x. All classes M_x are specializations of the abstract class `Thing` whose objects cover all objects in the classes M_x. On that abstract class `Thing` one association `Instantiation` is defined that is intended to represent the instance-of connections between a higher level object and a lower level: an object of a lower level is intended to be an instance of an object on a higher level. The association `Instantiation` on `Thing` (with role names `instantiater` and `instantiated`) is employed for the definition of the associations `Typing0`, `Typing1`, and `Typing2` between M_x and M_{x+1} all having roles `typer` and `typed`. The role `typer` is a redefinition of `instantiater`, and `typed` is a redefinition of `instantiated`. The multiplicity `1` of `typer` narrows the multiplicity `0..1` of `instantiater`.

In the abstract class `Thing` the transitive closure `instantiatedPlus()` of `instantiated` is defined by means of OCL. Analogously, `instantiaterPlus()` is defined for `instantiater`. The closure operations are needed to define an invariant in class `Thing` requiring `Instantiation` links to be acyclic.

```
abstract class Thing
operations
  instantiatedPlus():Set(Thing)=
    self.instantiated->closure(t|t.instantiated)
  instantiaterPlus():Set(Thing)= ...
```


Class diagram
0..1 instantiater {union}
Instantiation
* instantiated {union}
Thing
instantiaterPlus() : Set(Thing)
instantiatedPlus() : Set(Thing)
M3
M2
M1
M0
1 typer {redefines instantiater}
Typing2
* typed {redefines instantiated}
1 typer {redefines instantiater}
Typing1
* typed {redefines instantiated}
1 typer {redefines instantiater}
Typing0
* typed {redefines instantiated}
Object diagram
MetaClass:M3
Class:M2
Person:M1
Ada:M0
instantiater {union}
Instantiation
instantiated {union}
typer {redefines instantiater}
Typing2
typed {redefines instantiated}
instantiater {union}
Instantiation
instantiated {union}
typer {redefines instantiater}
Typing1
typed {redefines instantiated}
instantiater {union}
Instantiation
instantiated {union}
typer {redefines instantiater}
Typing0
typed {redefines instantiated}
Evaluate OCL expression
Evaluate
Browser
Clear
Evaluate OCL expression
Evaluate
Browser
Clear

Enter OCL expression:
```
Person.instantiaterPlus()
```
Result:
```
Set{Class,MetaClass} : Set(Thing)
```

Enter OCL expression:
```
Class.instantiatedPlus()
```
Result:
```
Set{Ada,Person} : Set(Thing)
```

Fig. 1. Ada, Person, Class, MetaClass within Single Object Diagram.

```
constraints
  inv acyclicInstantiation: self.instantiatedPlus()->excludes(self)
end
```

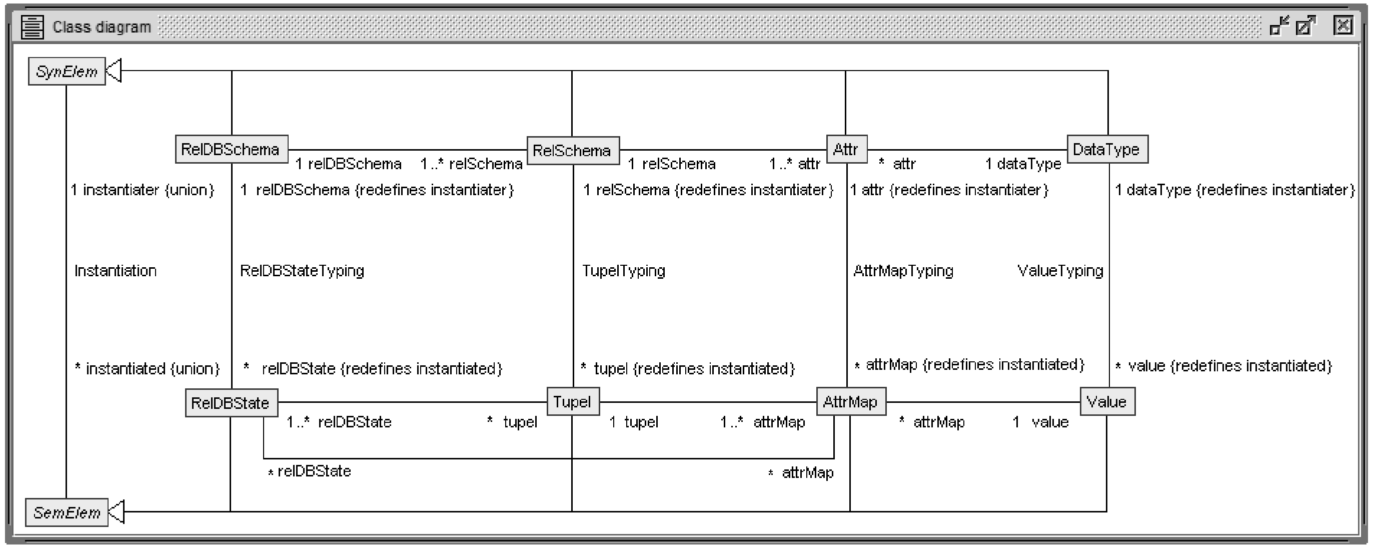
The class diagram from the left side of Fig. 1 is made concrete with an object diagram on the right side. The fact that the three associations `Typing0`, `Typing1`, and `Typing2` are all redefinitions of association `Instantiation` is reflected in the object diagram by the three dashed links for association `Instantiation` with common role names `instantiater` and `instantiated` (dashed links in contrast to continuous links for ordinary links). Viewing `Instantiation` as a generalization (in terms of redefinition) of all $\texttt{Typing}_x$ associations allows to use the closure operations from class `Thing` on objects from classes `M0`, `M1`, `M2` or `M3`. Thus the displayed OCL expressions and their results reflect the following facts: object `Person` is a (direct resp. indirect) instantiation of objects `Class` and `MetaClass`; objects `Ada` and `Person` are (direct resp. indirect) instantiations of object `Class`.

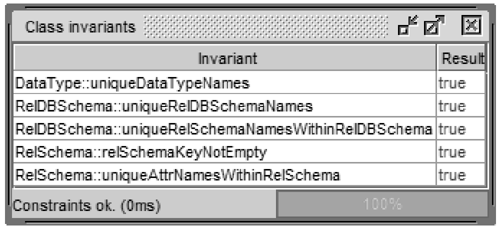
Summary: Metamodeling means to construct models for several levels. The metamodels on the respective level should be described and modeled independently (e.g., as M0, M1, M2, and M3). The connection between the models should be established in a formal way by a typing association (e.g., `Typing0` gives a type object from `M1` to a typed object from `M0`). The Typing associations are introduced as redefined versions of the association `Instantiation` from (what we call) a multi-level *superstructure*. This superstructure contains the abstract class `Thing` which is an abstraction of all metamodel elements across all levels and additionally contains the association `Instantiation` and accompanying constraints. Because `Instantiation` is defined as `union`, an `Instantiation` link can only connect elements of adjacent levels, i.e., the $\texttt{Typing}_x$ links are level-conformant and strict. The aim of the devices in the superstructure is to establish the connection between metamodel levels in a formal way and to provide support for formally restricting the connections.

3 Superstructure Example for Relational Database Model

In Fig. 2 we show two metamodels, one for the syntax and one for the semantics part of the relational database model. The upper part catches the syntax, i.e., relational database schemas, relational schemas, attributes, and data types. With regard to the class names, please recall that in the database field a *relational database schema* consists of possibly many *relational schemas*.[1] The lower part deals with the semantics (or runtime interpretation), i.e., database states, tuples, attribute mappings (for short attribute maps), and values. One also identifies two collections of invariant names, one collection for the syntax,

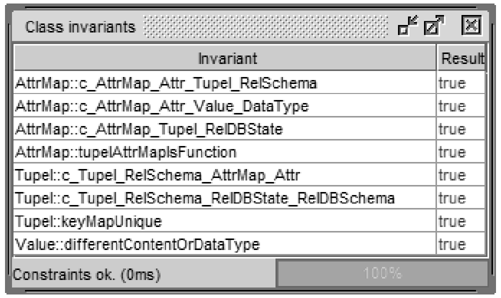
[1] A relational schema is called a table in SQL.




Class invariants

Invariant	Result
DataType::uniqueDataTypeNames	true
RelDBSchema::uniqueRelDBSchemaNames	true
RelDBSchema::uniqueRelSchemaNamesWithinRelDBSchema	true
RelSchema::relSchemaKeyNotEmpty	true
RelSchema::uniqueAttrNamesWithinRelSchema	true

Constraints ok. (0ms) 100%


Class invariants

Invariant	Result
AttrMap::c_AttrMap_Attr_Tupel_RelSchema	true
AttrMap::c_AttrMap_Attr_Value_DataType	true
AttrMap::c_AttrMap_Tupel_RelDBState	true
AttrMap::tupelAttrMapIsFunction	true
Tupel::c_Tupel_RelSchema_AttrMap_Attr	true
Tupel::c_Tupel_RelSchema_RelDBState_RelDBSchema	true
Tupel::keyMapUnique	true
Value::differentContentOrDataType	true

Constraints ok. (0ms) 100%

Fig. 2. Metamodels for Relational Database Schemas and States.

and one for the semantics. For example, RelSchema::relSchemaKeyNotEmpty requires that each relational schema must have at least one key attribute, and Tupel::keyMapUnique requires that two different tuples[2] must be distinguishable by at least one key attribute in each database state. The constraints starting with c_ are (what we call) commutativity constraints which require that the evaluation of two different paths through the class diagram coincide. Both paths start in one class and typically end in one different class. For example, the constraint AttrMap::c_AttrMap_Attr_Tupel_RelSchema requires that for an object am:AttrMap the paths am.attr.relSchema and am.tupel.relSchema coincide.

In the left of the class diagram we identify the metamodel superstructure established by the abstract classes SynElem, SemElem, and the association Instantiation. Instantiation is specialized through redefinition to RelDBStateTyping, TupelTyping, AttrMapTyping, and ValueTyping. We regard the syntax model, i.e., SynElem and its specializations together with the associations, as a metamodel of the semantics model, i.e., SemElem and its specializations together with the associations. We take this view because each higher level class (in the syntax part) serves to instantiate a lower level class (in the semantics part), and thus each lower level object has exactly one type that is defined in the higher level. Another argument supporting the view that we here have two connected metamodels is the factor that the relationship between RelSchema and Tupel is the same as the relationship between Class and Object in the OMG four-level architecture. The same holds for the other (SynElem,SemElem) class pairs: (RelDBSchema,RelDBState), (Attr,AttrMap), and (DataType,Value).

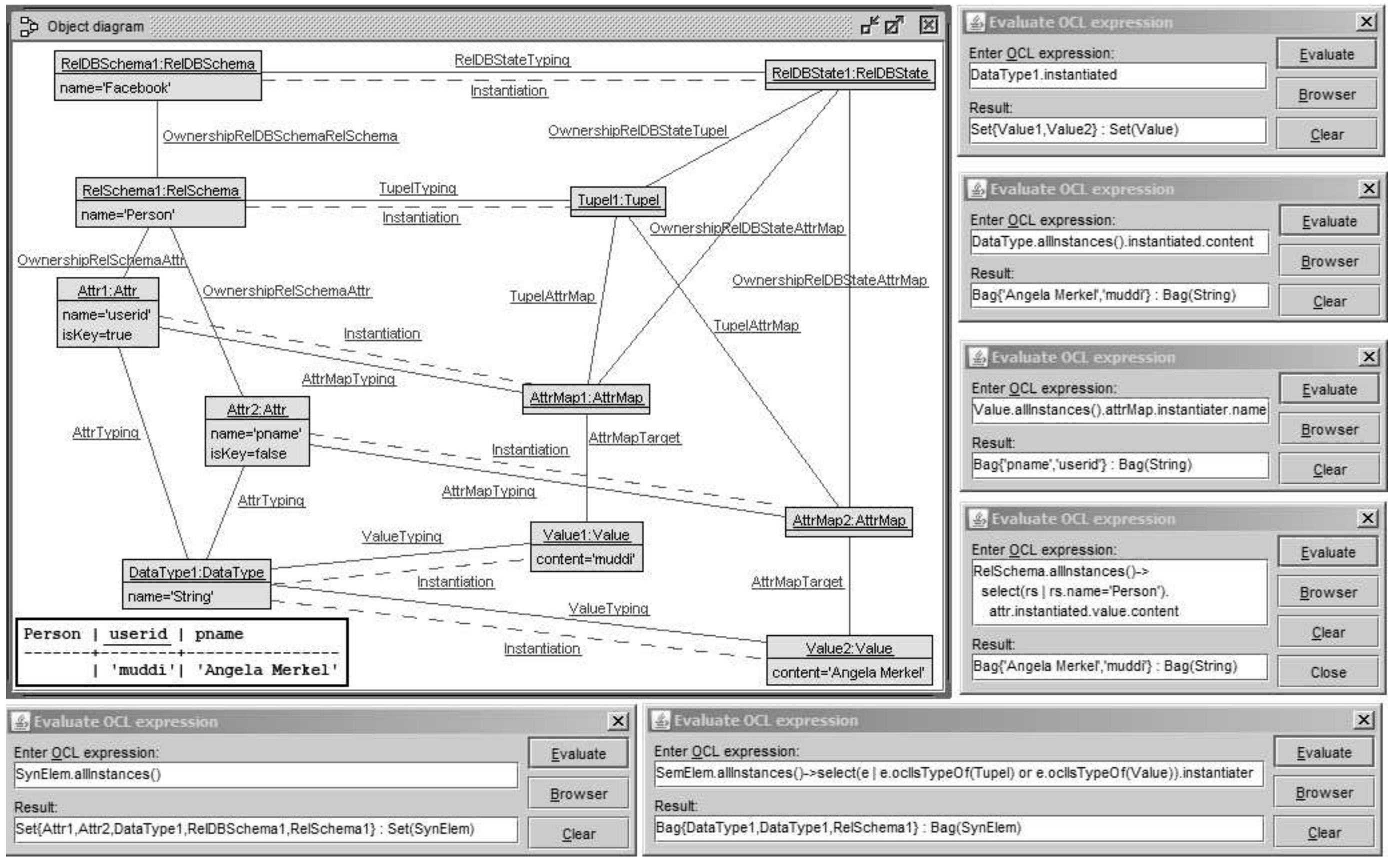
Interestingly, some invariants span across metamodel boundaries, i.e., an invariant from the semantics part sometimes uses elements from the syntax part. For example, the mentioned uniqueness requirement for tuples with regard to their key attributes is only required, if the tuples under consideration belong to the same relational schema. Thus the invariants of the semantics part rely on or use information from the syntax part.

In Fig. 3 we make the metamodels from Fig. 2 concrete by presenting a simple relational database schema consisting of one relational schema and a very simple accompanying database state with only one tuple. The presentation is done in form of an object diagram. The figure shows also OCL queries and their result that demonstrate how one can bridge the boundary between the metamodels. All queries either use the roles instantiater or instantiated which cross a metamodel boundary. For example, the fourth query from the top (RelSchema.allInstances()->select(rs | rs.name='Person').attr.instantiated.value.content) retrieves all values that are present in one of the attributes of the relational schema Person.

In Fig. 4 we show a larger object diagram[3] with two relational database states, two relational schemas and three tuples. The object diagram satisfies all invari-

[2] We have used the German spelling Tupel because Tuple is a keyword in OCL.

[3] In order to make the figure easier to grasp some links are hidden.



```
Person | userid | pname
-------+--------+----------------
       | 'muddi'| 'Angela Merkel'
```

Evaluate OCL expression
Enter OCL expression:
```
DataType1.instantiated
```
Result:
```
Set{Value1,Value2} : Set(Value)
```

Evaluate OCL expression
Enter OCL expression:
```
DataType.allInstances().instantiated.content
```
Result:
```
Bag{'Angela Merkel','muddi'} : Bag(String)
```

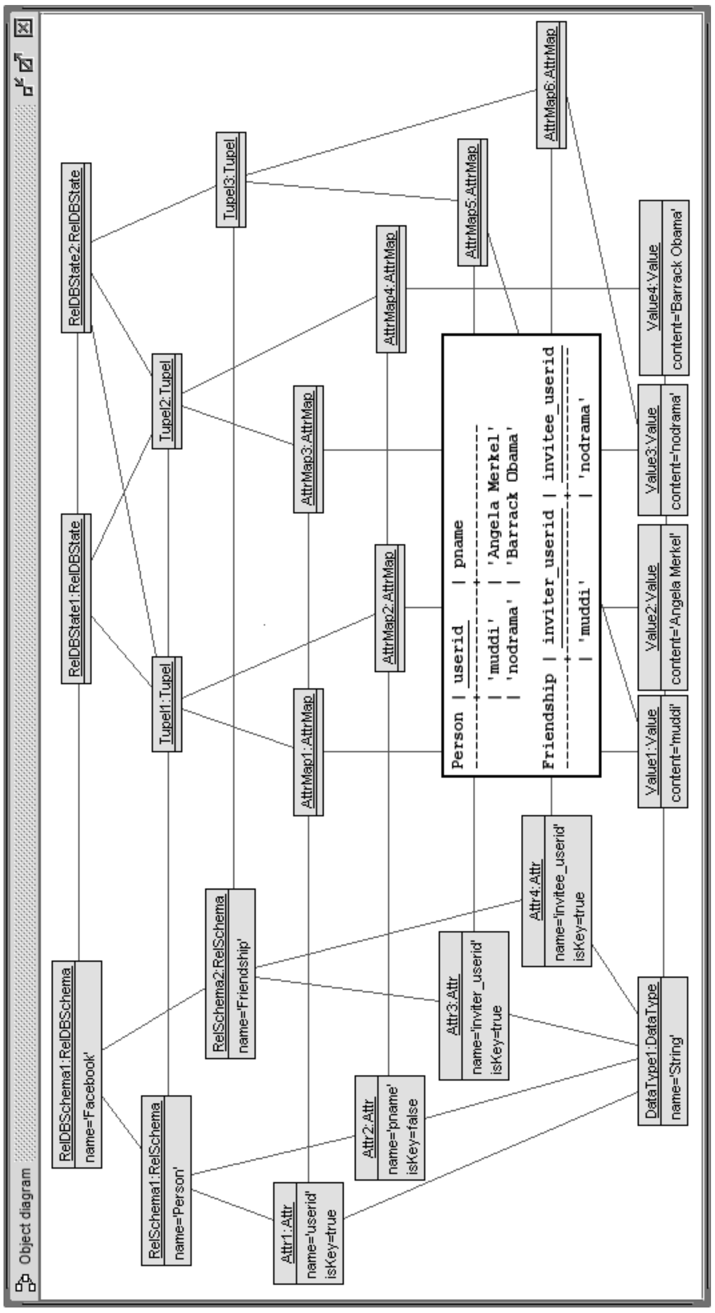
Evaluate OCL expression
Enter OCL expression:
```
Value.allInstances().attrMap.instantiater.name
```
Result:
```
Bag{'pname','userid'} : Bag(String)
```

Evaluate OCL expression
Enter OCL expression:
```
RelSchema.allInstances()->
  select(rs | rs.name='Person').
    attr.instantiated.value.content
```
Result:
```
Bag{'Angela Merkel','muddi'} : Bag(String)
```

Evaluate OCL expression
Enter OCL expression:
```
SynElem.allInstances()
```
Result:
```
Set{Attr1,Attr2,DataType1,RelDBSchema1,RelSchema1} : Set(SynElem)
```

Evaluate OCL expression
Enter OCL expression:
```
SemElem.allInstances()->select(e | e.oclIsTypeOf(Tupel) or e.oclIsTypeOf(Value)).instantiater
```
Result:
```
Bag{DataType1,DataType1,RelSchema1} : Bag(SynElem)
```

Fig. 3. Single Tuple Represented within Metamodel.



```
Person | userid    | pname
-------+-----------+----------------
       | 'muddi'   | 'Angela Merkel'
       | 'nodrama' | 'Barrack Obama'

Friendship | inviter_userid | invitee_userid
-----------+----------------+---------------
           | 'muddi'        | 'nodrama'
```

Fig. 4. Three Tuples Represented within Metamodel.

ants. The metamodels reflect the syntactical and semantical requirements, in particular through the use of constraints. For example, if one changes in the object `Attr1:Attr` the `isKey` attribute value from `true` to `false`, the syntactical requirement that relational schemas must have at least one key attribute value would be violated and this would be indicated by a constraint violation for the respective constraint `RelSchema::relSchemaKeyNotEmpty`. As an example on the semantical side, if one changes in the object `Value1:Value` the `content` attribute value from `'muddi'` to `'nodrama'`, the semantical requirement that each two tuples must have at least one distinguishing key attribute would be violated and this would be indicated by a constraint violation for the respective constraint `Tupel::keyMapUnique`.

4 Other Metamodel Superstructures

In the two examples above we have employed different metamodel superstructures. The first example Ada-Person-Class-MetaClass used the superstructure displayed in the upper left part of Fig. 5. The second example for the relational

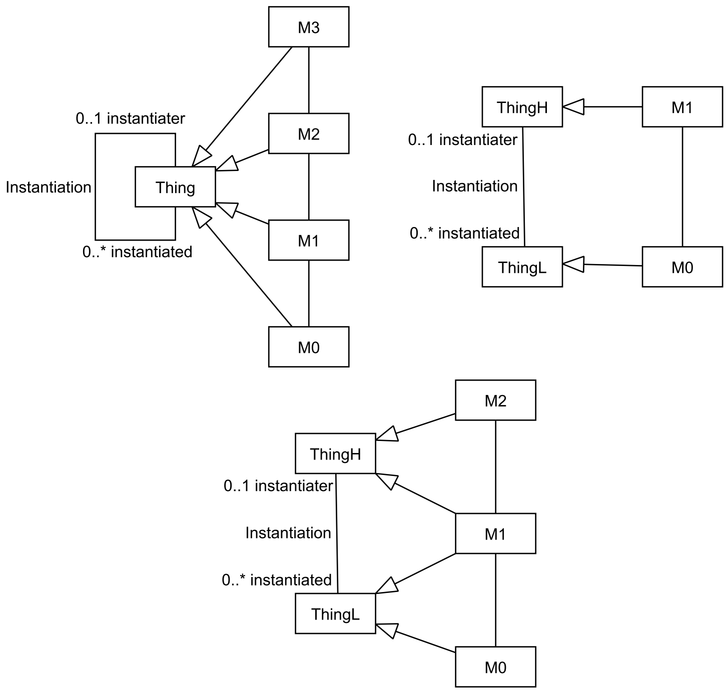


Fig. 5. Three Different Multi-Level Metamodel Superstructures.

database model used more or less the superstructure shown in the upper right part of the figure. However instead of the generic class names `ThingH[igh]` and `ThingL[ow]` the example used `SynElem` and `SemElem`, and instead of `M1` and `M0` the example used a bunch of connected classes like {`RelDBSchema`, `RelSchema`, `Attr`, `DataType`} and {`RelDBState`, `Tupel`, `AttrMap`, `Value`}. Other metamodeling superstructures could be used as well, for example the one displayed in the lower part of Fig. 5 utilizing multiple inheritance. Dependent on the actual needs for the metamodels at hand, a suitable superstructure with fitting classes, associations, and constraints can be chosen.

In our example superstructures we have been using the multiplicities `0..1` or `1` for the roles `instantiater` and `typer`. However, in principle other multiplicities like `1..*` could be used. It is an open question whether this could make sense, for example, in the context of multiple inheritance.

5 Conclusion

This paper proposed to describe different metamodels in one model and to connect the metamodels with a (so-called) superstructure consisting of generalizations and associations with appropriate UML and OCL constraints. We explained our ideas in particular with an example expressing the syntax and the semantics of the relational database model on different metamodel levels.

Future research includes the following topics. We would like to work out for our approach formal definitions for notions like potency or strictness. The notion of powertype will be given special attention in order to explore how far this concept can be integrated. Our tool USE could be extended to deal with different metamodel levels simultaneously. So far USE deals with class and object diagram. In essence, we think of at least a three-level USE (cubeUSE) where the middle level can be seen at the same time as an object and class diagram. Furthermore, larger examples and case studies must check the practicability of the proposal.

References

1. Alanen, M., Porres, I.: A Metamodeling Language Supporting Subset and Union Properties. Software and System Modeling **7**(1) (2008) 103–124
2. Atkinson, C.: Multi-Level Modeling with Melanie. Commit Workshop 2012 (2012) commit.wim.uni-mannheim.de/uploads/media/commitWorkshop_Atkinson.pdf.
3. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software **20**(5) (2003) 36–41
4. Bézivin, J.: On the Unification Power of Models. Software and System Modeling **4**(2) (2005) 171–188
5. de Lara, J., Guerra, E.: Deep Meta-Modelling with Metadepth. In Vitek, J., ed.: TOOLS (48). Volume 6141 of LNCS., Springer (2010) 1–20
6. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Comp. Prog. **69** (2007) 27–34
7. Gogolla, M., Favre, J.M., Büttner, F.: On Squeezing M0, M1, M2, and M3 into a Single Object Diagram. In Baar, T. et al., eds.: Proc. MoDELS'2005 Workshop Tool Support for OCL and Related Formalisms, EPFL (Switzerland), LGL-REPORT-2005-001 (2005)
8. Hamann, L., Gogolla, M.: Endogenous Metamodeling Semantics for Structural UML2 Concepts. In Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P.J., eds.: MoDELS. Volume 8107 of LNCS., Springer (2013) 488–504
9. Henderson-Sellers, B., Clark, T., Gonzalez-Perez, C.: On the Search for a Level-Agnostic Modelling Language. In Salinesi, C., Norrie, M.C., Pastor, O., eds.: CAiSE. Volume 7908 of LNCS., Springer (2013) 240–255