On Squeezing M0, M1, M2, and M3 into a Single Object Diagram

Martin Gogolla, Jean-Marie Favre, Fabian Büttner

University of Bremen (D), University of Grenoble (F), University of Bremen (D)

Abstract. We propose an approach for the integrated description of a metamodel and its formal relationship to its models and the model instantiations. The central idea is to use so-called layered graphs permitting to describe type graphs and instance graphs. A type graph can describe a collection of types and their relationships whereas an instance graph can represent instances belonging to the types and respecting the relationships required by the type graph. Type graphs and instance graphs are used iteratively, i.e., an instance graph on one layer can be regarded as a type graph of the next lower layer. Our approach models layered graphs with a UML class diagram, and operations and invariants are formally characterized with OCL and are validated with the USE tool. Metamodeling properties like strictness or well-typedness and features like potency can be formulated as OCL constraints and operations. We are providing easily understandable definitions for several metamodeling notions which are currently used in a loose way by modelers. Such properties and features can then be discussed on a rigorous, formal ground. This issue is also the main purpose of the paper, namely, to provide a basis for discussing metamodeling topics.
Keywords: System, Model, Metamodel, Meta-Metamodel, Class, Instance, InstanceOf, RepresentedBy, ConformsTo, Well-Typedness, Strictness, Potency, Layered Graph.

1 Motivation

Recent research activities and results in software engineering indicate that metamodeling is becoming more and more important [Sei03,Tho04,Bez05]. There are a lot of discussions about properties and notions of metamodels like the strictness of a metamodel or the potency of elements in it [AKHS03,AK03]. There are special sessions at scientific events on metamodeling. Standardized (e.g., by the OMG) and scientific, alternative metamodels have been developed for important languages like UML [ESW+05], MOF, OCL [WK02,RG99], and CWM, to name only a few. A book on metamodeling is currently under development [CESW04]. Metamodeling is important within the Model Driven Architecture [Fra03,KWB03,MSUW04], and metamodeling is beginning to be broadened to megamodeling [Fav05b,Fav05a].

However, notions within the metamodeling area are often loose due to a lack of formalization. This has been recently referred to as the *meta-muddle*. Let us mention some examples. (A) A recent nice paper [Bez05] distinguishes between *System*, *Model*, *Metamodel*, and *Meta-Metamodel* whereas the conventional OMG approach uses the notion *User Objects* (*User Data*) instead of *System*. (B) In the same paper the author calls something a metamodel what would be called a model in the OMG terminology. (C) There are continuing discussions on whether the metamodels for UML 1 and UML 2 are strict or not.

The aim of this paper is to present a framework for discussing such notions and properties of metamodels by formalizing them. We will use a graph-based approach. Within metamodels one usually has different layers of abstractions, and each layer is more or less formally described. However, the relationship between the different layers is usually not explicitly discussed and described only implictly. Our approach allows to formally describe the different layers as well as the connection between the layers in an abstract form. Each layer will build a graph with nodes and edges, and also the connections between the layers will be formally described by edges. Thus we will obtain a comprehensive single graph covering the metamodel layers and their connections.

The structure of the rest of the paper is as follows. Section 2 will introduce the basic idea by means of a simple example. Section 3 formally shows our approach with its underlying class diagram including invariants and operations. Section 4 discusses further examples. The paper is finished with a concluding section which also contains open questions.

2 Describing Layered Graphs as Object Diagrams

Our layered graphs allow to organize complex metamodel structures into several abstraction layers. As an example, consider the graph in the left part of Fig. 1 possessing three layers: On the bottom layer one identifies nodes like ada and edges like ada_ibm; in the middle layer there are the nodes Person and Company and the edge Job; the top layer consists of the node Thing and the edge Connection. We discuss features of layered graphs by considering, for the example, the three layers one after the other, starting with the middle layer.

Middle layer: The middle layer can be thought of as representing a class diagram with two classes and one association.

Bottom layer: The bottom layer can be regarded as an object diagram with respect to the middle layer. The nodes and edges on this layer are all typed by dashed edges going to the middle layer. For example, the node ada is typed with a dashed edge going to node Person and the edge ada_ibm is typed with a dashed edge to the edge Job. Both typing elements, i.e., Person and Job, belong to the next higher layer.

Top layer: The top layer can be thought of as showing a class diagram such that the middle layer diagram becomes an object diagram for this top layer

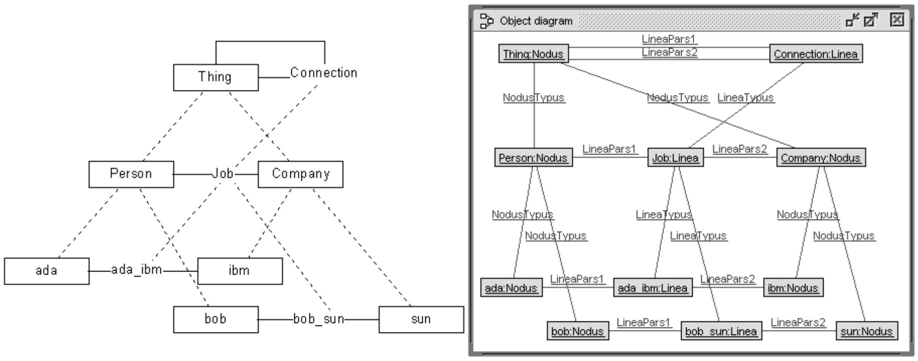

Fig. 1. Job Example as a Layered Graph and as an Object Diagram

diagram. The nodes Person and Company are typed as being nodes of type Thing and Job is an edge of type Connection.

The aim of the paper is to propose a general model for graphs like the one depicted in the left part of Fig. 1 but where not only three layers but an arbitrary number of layers can be captured. Each layer may have (solid) nodes and (solid) edges which can be typed by (dashed) edges to the next higher layer. We call such a graph consisting of several layers and (dashed) typing edges between the layers a *layered graph*.

The USE [RG01] screenshot in right part of Fig. 1 shows how the graph from the left part is represented as a UML object diagram. We will first explain the basic structure of the object diagram and introduce the respective class diagram later. One basic observation is that the nodes and the solid (non-dashed), named edges from the left part are represented as objects, but the dashed, unnamed edges are represented as links. The objects belong to (A) the class Nodus which realizes the nodes from the left part or (B) the class Linea which realizes solid edges from the left part. In order to have new, neutral names we have chosen the respective latin words as class names (see [Fav05b,Fav05a] for a discussion of the importance of distinguishing metamodel levels). Objects can be typed by Typus links. Nodus objects are typed by NodusTypus links, and Linea objects by LineaTypus links. The Typus links have been shown in the left part of Fig. 1 by dashed edges. Linea objects indicate their participating Nodus objects with links labelled LineaPars1 and LineaPars2. The latin word pars means part.

A layered graph does not induce a unique object diagram: For example, instead of having the link labelled LineaPars1 from Job to Person and the link labelled LineaPars2 from Job to Company, we could exchange 1 and 2 and have a link LineaPars2 from Job to Person and a link LineaPars1 from Job to Company.

3 Class Diagram, Operations, and Invariants

The class diagram in Fig. 2 shows our modeling for layered graphs by introducing the classes Nodus and Linea, the associations NodusTypus, LineaTypus, LineaPars1, and LineaPars2, and the names of the invariants.

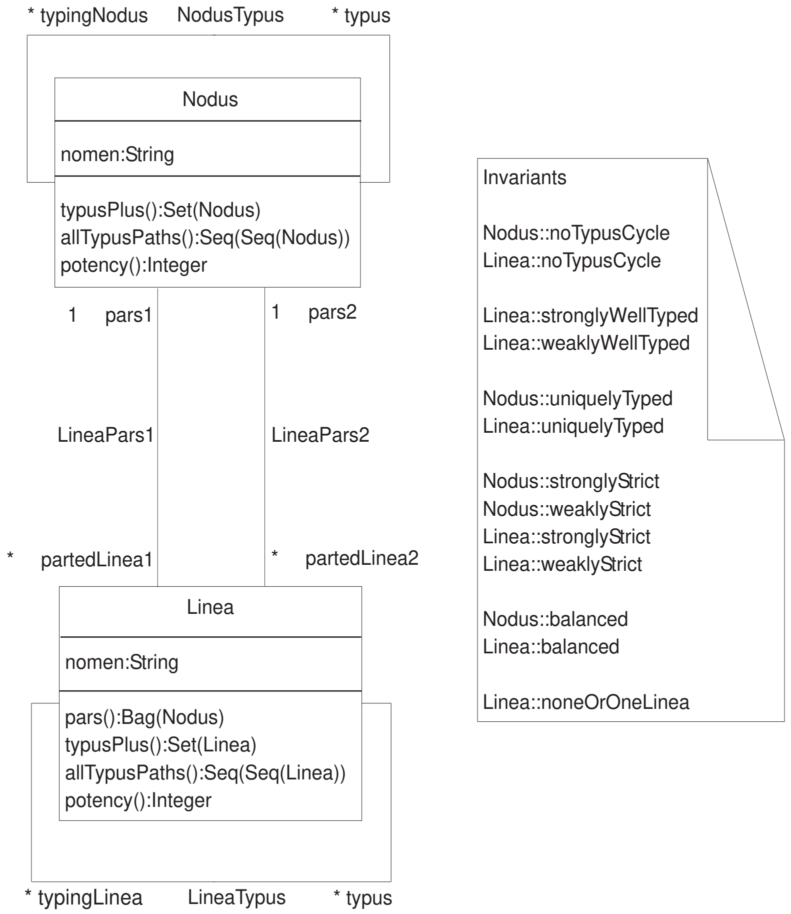

Fig. 2. UML Class Diagram for Nodus and Linea

- The class Nodus describes nodes. Nodus objects as well as Linea objects possess a string-valued name attribute nomen. The operation typusPlus is the transitive closure of the role name typus which will be explained below. The operation potency provides one definition for the potency of Nodus objects which basically indicates the layer to which the Nodus object belongs. The operation allTypusPaths delivers all paths without node repetitions consisting of Nodus objects starting in the respective Nodus object and using typing edges (dashed edges) only.

– The class Linea describes edges. Apart from having analogous operations mentioned already for the class Nodus, it possesses the operation pars yielding the bag of Nodus objects the respective Linea object is connected to.
– The association NodusTypus represents the typing of Nodus objects. Its role names are typus and typingNodus. typus yields the types of the current nodus object in the next higher layer (dashed edges upwards), whereas typingNodus yields the objects in the next lower layer which are typed through the current nodus object (dashed edges downwards). The association LineaTypus represents the typing of Linea objects. This association has analogous role names as NodusTypus. We emphasize that the multiplicities for NodusTypus and LineaTypus do not require a unique typing mechanism: Nodus and Linea objects can have multiple types.
– The associations LineaPars1 and LineaPars2 give the two Nodus objects the Linea object is connected to. These two Nodus objects have not to be distinct as, for example, the edge Connection in Fig. 1 shows. The choice between what is LineaPars1 and what is LineaPars2 is not important.

We now turn to the invariants. Not all these invariants are expected to be true in all system states which we discuss. The invariants are used to display whether certain properties hold in the current system state or not. Our USE tool allows with the invariant window to display such properties in a compact way, and therefore, we have formulated most properties as invariants. Alternatively, we could have formulated the invariants with observer operations, but then we could not have displayed the resuls in a compact way. However, the interesting discussion points will occur when certain invariants are not satisfied. Our introductory example however satisfies all invariants.

Most invariants are formulated for the class Nodus as well as for the class Linea. Therefore, we only show the invariants for class Nodus because the ones for class Linea are formulated analogously.[1]

–
```
context self:Nodus inv noTypusCycle: -- also for Linea
  self.typusPlus()->excludes(self)
```
Nodus::noTypusCycle requires that dashed edges between nodes do not include a cycle: The association NodusTypus constitutes a directed, acyclic graph (dag).

–
```
context self:Linea inv stronglyWellTyped: -- only for Linea
  self.typus->notEmpty implies
    self.pars().typus=self.typus.pars()
```
Linea::stronglyWellTyped demands that the types of the nodes of a (solid) edge are equal to the nodes of the types of the edge (types of the nodes vs. nodes of the types). In other words, for any (solid) edge, typing and building edge components are interchangeable.

[1] OCL invariants without explict variables possess an implicit variable `self` typed by the context class. We here prefer to name variables explicitly because we need a second context class variable for this constraint.

- ```
  context self:Linea inv weaklyWellTyped: -- only for Linea
    self.typus->notEmpty implies
      self.pars().typus->includesAll(self.typus.pars())
  ```
  Linea::weaklyWellTyped claims that the types of the nodes of a (solid) edge are a superset of the nodes of the types of the edge (again, types of the nodes vs. nodes of the types). In other words, for any (solid) edge, calculating first the type and then the edge components is compatible with calculating first the edge components and then the type, but not the other way round.
  The last two well-typedness properties can be formulated only for the class Linea.
- ```
  context self:Nodus inv uniquelyTyped: -- also for Linea
    self.typus->notEmpty implies self.typus->size=1
  ```
  Nodus::uniquelyTyped means that all nodes except the nodes in the top layer have exactly one type.
- ```
  context self:Nodus inv stronglyStrict: -- also for Linea
    self.typus->notEmpty implies
      self.typus->forAll(n|self.potency()+1=n.potency())
  ```
  Nodus::stronglyStrict demands that the potency of a node lies exactly under all the potencies of the types of the node being on the next higher layer.
- ```
  context self:Nodus inv weaklyStrict: -- also for Linea
    self.typus->notEmpty implies
      self.typus->forAll(n|self.potency()+1<=n.potency())
  ```
  Nodus::stronglyStrict requires that the potency of a node lies under, but not necessarily exactly under all the potencies of the types of the node being on the next higher layer.
- ```
  context self:Nodus inv balanced: -- also for Linea
    Nodus.allInstances->forAll(self2|
      self<>self2 and self.potency()=self2.potency() implies
      self.typingNodus->notEmpty=self2.typingNodus->notEmpty)
  ```
  Nodus::balanced states that all layers are balanced in the sense that two different nodes with the same potency also both possess typing nodes.
- ```
  context self:Linea inv noneOrOneLinea: -- only for Linea
    Linea.allInstances->forAll(self2|
      self<>self2 implies self.pars()<>self2.pars())
  ```
  Linea::noneOrOneLinea requires that between two nodes there can be at most one (solid) edge. This invariant could also be formulated in a restricted way for particular layers only.
- The overall aim of operation[2] potency shown in Fig. 3 is to return the layer number in which the respective Nodus object lies. The numbering starts with zero on the lowest layer. The formal definition of potency is rather complex, partly because potency should yield a result even in cases when the underlying NodusTypus structure is cyclic. The operation potency uses the helper operations max and allTypusPaths. allTypusPaths in turn needs the helper operation oneStep. allTypusPaths computes all paths consisting of nodes and (dashed) edges going upwards but a single node is allowed to occur only once. So, if there are cycles in the dashed edges between nodes,

[2] We employ the USE syntax for operation definitions.

i.e., invariant Nodus::noTypusCycle is not valid, allTypusPaths and with this potency will yield no usable result.

```
oneStep(aSeq:Sequence(Sequence(Nodus))):Sequence(Sequence(Nodus))=
  if aSeq->isEmpty or aSeq->isUndefined then
    oclEmpty(Sequence(Sequence(Nodus)))
  else
    aSeq->iterate(s:Sequence(Nodus);
      r1:Sequence(Sequence(Nodus))=oclEmpty(Sequence(Sequence(Nodus)))|
        s->last.typus->iterate(n:Nodus;r2:Sequence(Sequence(Nodus))=r1|
          if s->excludes(n)
            then r2->append(s->including(n)) else r2 endif))
  endif
allTypusPaths():Sequence(Sequence(Nodus))=
  Nodus.allInstances->iterate(n:Nodus;
    r:Sequence(Sequence(Nodus))=Sequence{Sequence{self}}|
    let new=oneStep(r)->reject(s|r->includes(s)) in r->union(new))
max():Integer=
  Nodus.allInstances->collect(n|n.allTypusPaths())->flatten->
    collect(size)->iterate(i:Integer;r:Integer=0 |
      if i>r then i else r endif)
potency():Integer=
  max()-self.allTypusPaths()->collect(p|p->size)->iterate(
    i:Integer;r:Integer=0 | if i>r then i else r endif)
```

Fig. 3. Operation potency

In order to give a simple example, how the operation potency works, we show its results for the introductory example:

	potency
ada,bob,ibm,sun,ada_ibm,bob_sun	0
Person,Company,Job	1
Thing,Connection	2

After having formulated these abstract properties let us turn to some more examples in order to see how the invariants behave in concrete situations.

4 Further Examples

The first example is about the poodle fido where poodle in turn is regarded as a breed. The situation is displayed in Fig. 4 as a layered graph and in Fig. 5 as a USE object diagram. The (solid) edges express an InstanceOf association. fido is typed as an Object and as a Poodle, Poddle is typed as a Breed, as an Object and as a Class, and Breed is typed as a Class. Thus the invariant Nodus::uniquelyTyped is invalid. This formally reveals that Poodle is a clabject. A clabject is a cross between class and object [AKHS03,AK03]. In formal terms this can be captured as Poodle.typus = Set{Breed,Class,Object}. The typing is also not strongly strict, i.e., the invariant Nodus::stronglyStrict is invalid, because Poodle.potency()=1 but, e.g., Object.potency()=3. Last, the Linea objects are not strongly well-typed because, e.g., Poodle_fido.pars().typus = Bag{Breed,Class,Object,Object,Poodle} but Poodle_fido.typus.pars() = Bag{Class,Object}. However, Linea objects are weakly well-typed, because the second mentioned bag is included in the first bag.

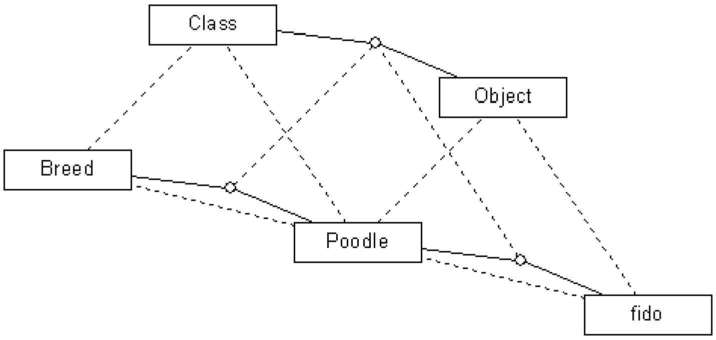

Fig. 4. The Fido Example as a Layered Graph

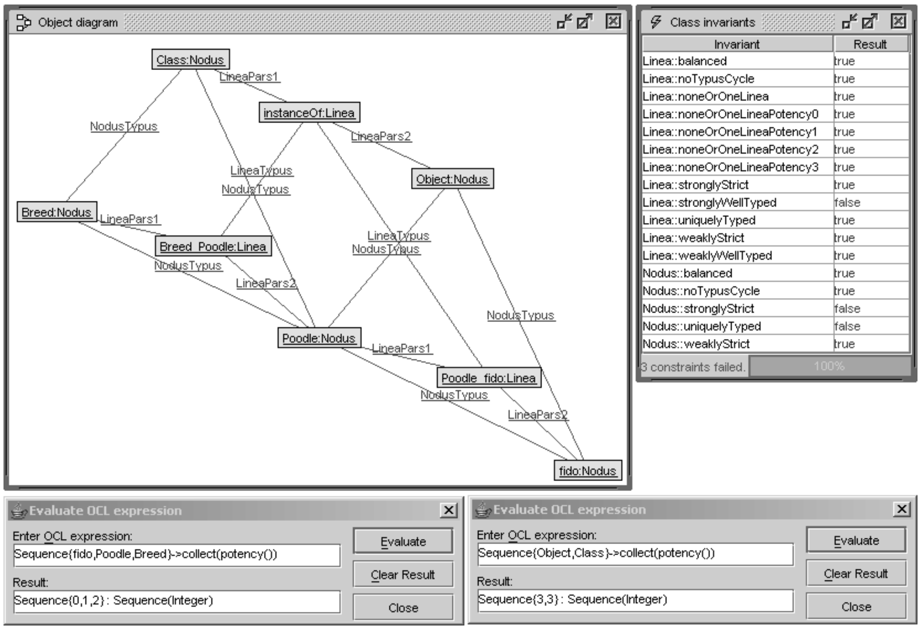

Fig. 5. The Fido Example as a USE Object Diagram

The second example shows in Fig. 6 and Fig. 7 parts of a modeling for the Entity-Relationship model, the Relational data model and their translation. The left part expresses that the instance adaInstance is typed by PersonEntity and PersonEntity in turn is typed by Entity. The right part shows that the tuple adaTuple is typed by PersonRelSchema which in turn is typed by a Relational schema (RelSchema). Both elements, Entity and RelSchema, are typed as classes. The (solid) edges express a Reification connection between the elements. All invariants are satisfied and the potencies work in the hopefully expected way.

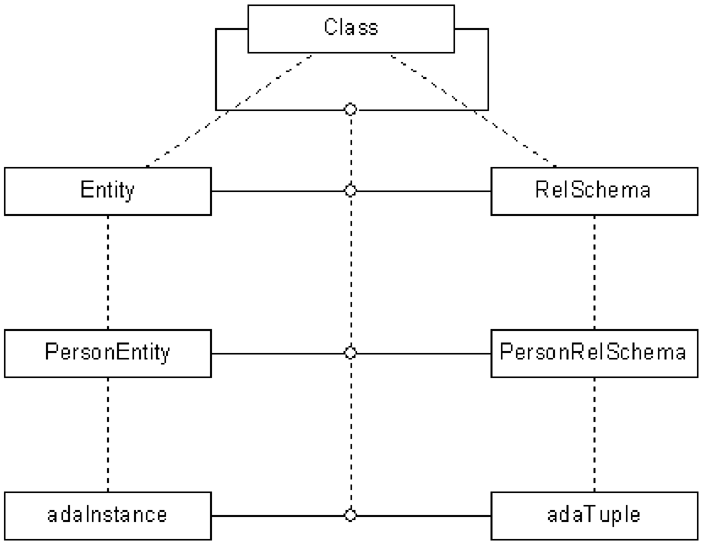

Fig. 6. The ER-RE Example as a Layered Graph

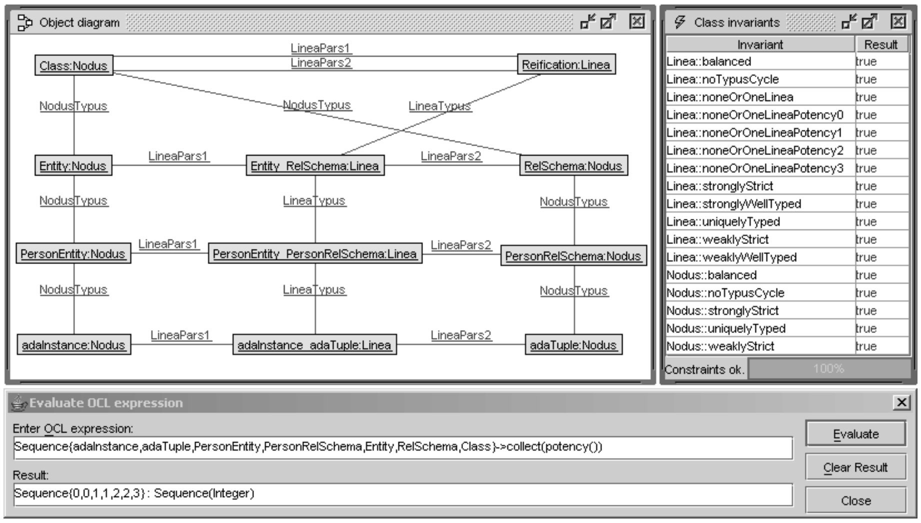

Fig. 7. The ER-RE Example as a USE Object Diagram

The third example in Fig. 8 and Fig. 9 shows a variation of the previous ER and Relational data model example where clabjects are used. As said already, interesting points come up where invariants are not satisfied. As in the first example of this section, Nodus::stronglyStrict and Nodus::uniquelyTyped fail. In this example, the Linea objects are not uniquely typed, but they are strongly well-typed. And the Linea object potencies are not strongly strict, because the (solid) edge potencies violate the requirements: PersonEntity_PersonRelSchema.potency()=1, but PersonEntity_PersonRelSchema.typus.potency()=Bag{2,3}.

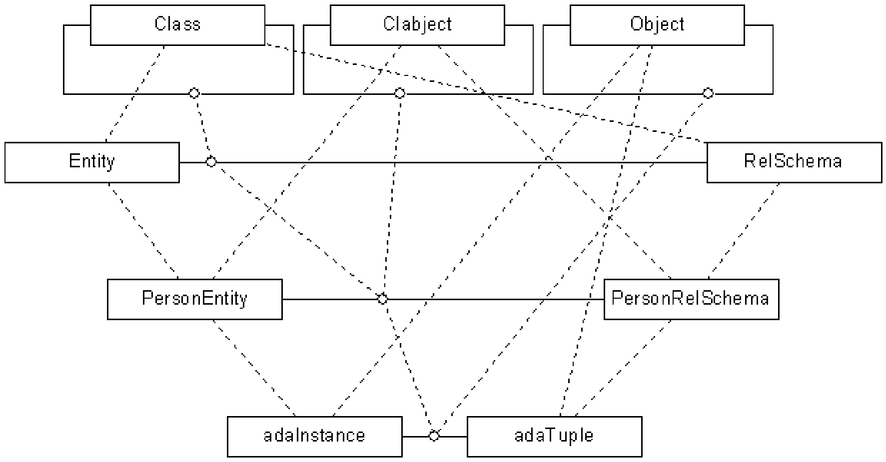

Fig. 8. Alternative ER-RE Example as a Layered Graph

5 Conclusion and Open Questions

This paper proposes to describe metamodeling notions and relationships with OCL invariants and operations. Such rigorous characterizations allow a precise and sharper discussion of such notions, the different metamodelling layers and their relationships. We have defined two versions of the notion *strict metamodeling* (weakly and strongly strict metamodeling), and we have formally defined the notion potency. An overview of our approach is given in Fig. 10. All four OMG metamodel layers are represented in a single object diagram which conforms to the Nodus-Linea class diagram. This is shown by the outer CD-OD bracket (CD-OD: Class Diagramm - Object Diagram). As indicated by the inner CD-OD brackets within this object diagram, further object and class diagrams with relationships between the layers are present. Note that in Fig. 10 the association Job and the link ada_ibm are displayed in the Nodus-Linea compliant style as nodes.

Future research includes the following questions.

- Fig. 10 assumes that the CD-OD brackets all obey the same rules or in other words that the class and object diagrams are UML respectively MOF diagrams. Are the inner CD-OD brackets and the outer CD-OD brackets really of the same kind?

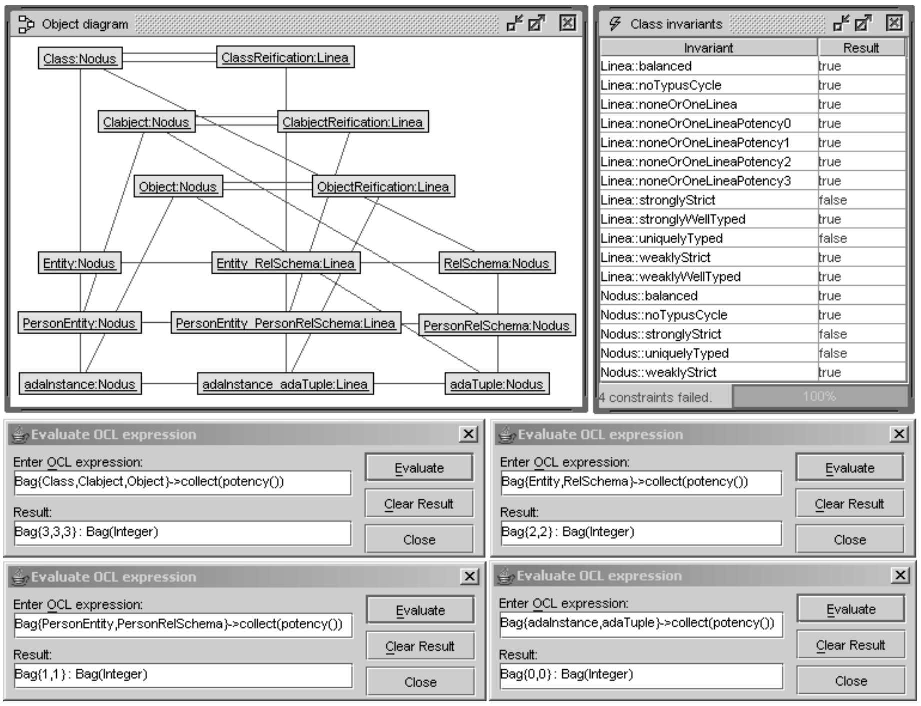

Fig. 9. Alternative ER-RE Example as a USE Object Diagram

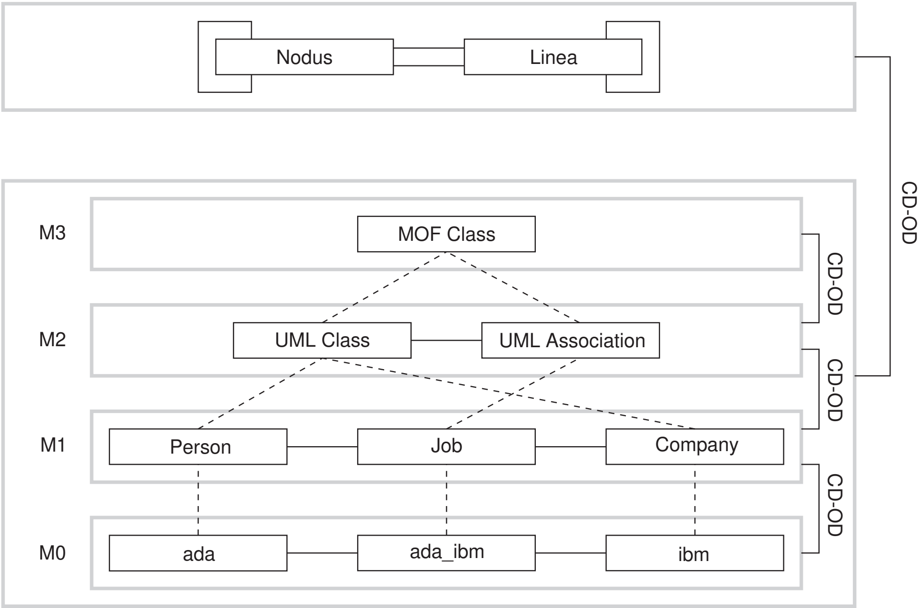

Fig. 10. Overview on the Approach

– The classes Nodus and Linea can be generalized to a common superclass GraphElement. This would make the model shorter but also even more abstract. Up to now our typing has to map nodes to nodes and edges to edges. A generalization class GraphElement could also allow, e.g., that nodes are type mapped to edges. Would this generalization condense the model or introduce more confusion?
– We have developed also an easier definition of the potency notion than the one we have shown. How does this easier definition relate to the more complex definition? Under which conditions do the two definitions coincide? How do these implicit or calculated potencies relate to an explicit assignment of potency in the class or object diagram?
– In our view a layer is collection of nodes and edges which have the same potency. And layers like M0, M1, M2, and M3 are then collections of objects and nodes with the respective potency. Is it possible to introduce particular constraints for particular layers? For example, naming conventions like *object names on the lowest layer (System) only have lower case letters*? Or, for example, uniqueness constraints for names within a particular namespace, like *Person names are unique*?
– In [Bez05] the view is taken that the *System* is *RepresentedBy* the *Model* and that the *Model* then *ConformsTo* the *Metamodel*. What are the consequences from the fact that both relationships *RepresentedBy* and *ConformsTo* are represented in our approach uniformly as dashed typing edges?
– Our approach also allows to discuss particular language feature of UML diagrams, e.g., ternary (and higher order) associations in class diagrams.

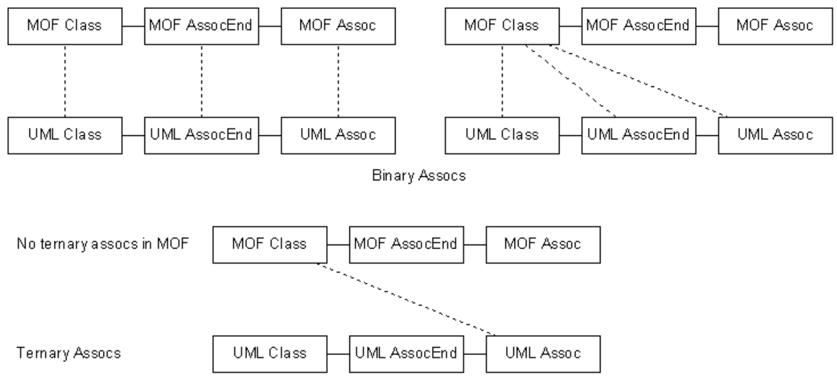

Fig. 11. UML in MOF - Ternary UML Assocs in MOF

What is the relationship between a UML Association and a MOF Association? In our approach, the question would be stated as: How is the typing of the Nodus object UML Association with respect to the next higher layer, i.e., with respect to the Nodus object MOF Association (as indicated in Fig. 11)? Goes the typing from UML Association to (A) MOF Class, (B) MOF Association, or (C) MOF Class and MOF Association?

- The general question behind this concrete question is: Where and how is the relationship between metamodeling layers expressed?
- Currently, our OCL tool USE (and also other tools) only allows to handle two layers: One class diagram and one object diagram layer. How can our approach help to develop *Meta-OCL tools* which support more layers where the middle layer (in a three layer setting) is at the same time an object diagram for the top layer and a class diagram for the bottom layer?

References

[AK03] C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[AKHS03] C. Atkinson, T. Kühne, and B. Henderson-Sellers. Systematic Stereotype Usage. *Software and System Modeling*, 2(3):153–163, 2003.

[Bez05] J. Bezivin. On the Unification Power of Models. *Software and System Modeling*, 4(2):171–188, 2005.

[CESW04] T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied Metamodelling: A Foundation for Language Driven Development.* Xactium, 2004.

[ESW+05] A. Evans, P. Sammut, J. S. Willans, A. Moore, and G. Maskeri. A Unified Superstructure for UML. *Journal of Object Technology*, 4(1):165–182, 2005.

[Fav05a] J.-M. Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[Fav05b] J.-M. Favre. Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In

J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[Fra03] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley and Sons, 2003.

[KWB03] A.G. Kleppe, J.B. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Pearson Education, 2003.

[MSUW04] S.J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled*. Addison Wesley, 2004.

[RG99] M. Richters and M. Gogolla. A Metamodel for OCL. In R. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, pages 156–171. Springer, Berlin, LNCS 1723, 1999.

[RG01] M. Richters and M. Gogolla. OCL - Syntax, Semantics and Tools. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

[Sei03] E. Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[Tho04] D.A. Thomas. MDA: Revenge of the Modelers or UML Utopia? *IEEE Software*, 21(3):15–17, 2004.

[WK02] J.B. Warmer and A.G. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2nd Edition edition, 2002.