What is a Multi-Modeling Language?

Artur Boronat[1], Alexander Knapp[2], José Meseguer[3], and Martin Wirsing[4]

[1] University of Leicester
aboronat@le.ac.uk
[2] Universität Augsburg
knapp@informatik.uni-augsburg.de
[3] University of Illinois
meseguer@uiuc.edu
[4] Ludwig-Maximilians-Universität München
wirsing@pst.ifi.lmu.de

Abstract. In large software projects often multiple modeling languages are used in order to cover the different domains and views of the application and the language skills of the developers appropriately. Such "multi-modeling" raises many methodological and semantical questions, ranging from semantic consistency of the models written in different sublanguages to the correctness of model transformations between the sublanguages. We provide a first formal basis for answering such questions by proposing semantically well-founded notions of a multi-modeling language and of semantic correctness for model transformations. In our approach, a multi-modeling language consists of a set of sublanguages and correct model transformations between some of the sublanguages. The abstract syntax of the sublanguages is given by MOF meta-models. The semantics of a multi-modeling language is given by associating an institution, i.e., an appropriate logic, to each of its sublanguages. The correctness of model transformations is defined by semantic connections between the institutions.

1 Introduction

In an idealized software engineering world, development teams would follow well-defined processes in which one single modeling language is used for all requirements and design documents; but in practice "multi-modeling" happens: in a large software project entity-relationship diagrams and XML may be used for domain modeling, BPEL for business process orchestration, and UML for design and deployment. In fact, UML itself can be seen as a multi-modeling language comprising several sublanguages such as class diagrams, OCL, and state machines; each sub-modeling language provides a particular view of a software system. Such views have the advantage of complexity reduction: a software engineer can concentrate on a particular aspect of the system such as the domain architecture or dynamic interactions between objects.

On the other hand, multi-modeling raises a host of methodological and semantical questions: are the different modeling sublanguages semantically consistent with each other? How can we correctly transform an abstract model in one modeling language into a more concrete one in another language? How can we detect semantic inconsistencies between heterogeneous models expressed in different modeling sublanguages? More generally, is there a notion of "multi-modeling language" which provides more insight than just an ad-hoc collection of modeling languages put together? Is it possible to give a semantics to multi-modeling languages which allows one to deal with consistency, validation and verification but that retains the advantages of multiple views by providing a local semantics and local reasoning capabilities for each modeling language?

The methodological use of views and viewpoints in software modelling is a long standing research topic [17]. In the literature, there are three main complementary approaches for interrelating modeling notations: the "system model approach", the "model-driven architecture approach", and the "heterogeneous semantics and development approach". In the system model approach the different modeling languages are translated into a common (formally defined) modeling notation called system model [9] which serves as unique semantic basis and for analyzing consistency of software engineering models. In the "model-driven architecture approach" [25] model transformations and consistency issues are typically dealt with at the syntactic level of the modeling notation. In the third approach different modeling languages are interrelated by semantic-preserving mappings [23,12]; a mathematical semantics is given locally for each modeling language and the consistency between different languages is analyzed semantically through the semantic-preserving mappings. All three approaches have been applied to several modeling languages including UML, but to the best of our knowledge, multi-modeling languages in the software engineering sense have never been systematically studied. However, research within the theory of institutions [18] on institution morphisms and comorphims [19], and on "heterogeneous institutions" [23] is directly relevant to this problem.

We combine ideas from model-driven architecture and heterogeneous semantics and propose a new, semantically well-founded notion of a multi-modeling language and a new notion of semantic correctness for model transformations. In particular, our formal definition of a multi-modeling language L: (i) uses the Meta-Object Facility MOF and its algebraic semantics [8] for describing the metamodels and models of the sublanguages of L; (ii) associates an institution to each sublanguage S of L and gives a mathematical semantics to each software engineering model[1] of S by a corresponding (logical) theory in the institution of S; (iii) defines the links between different sublanguages of S by model transformations and provides a notion of semantic correctness for such transformations; and (iv) provides a notion of consistent heterogeneous (software engineering) model in the multi-modeling language L, which is derived from a notion of a class of heterogeneous mathematical models at the institution level.

The approach is illustrated in Fig. 1: There are three sublanguages S_1, S_2, and S_3 of a common multi-modeling language L, software engineering models M_1, M_2, and M_3 conforming to (the meta-model representations of) the sublanguages, and having a formal semantics in the institutions $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$. The model transformations $trans_{12}$ and

[1] For distinguishing semantic models from the models of a modeling language we write "software engineering model (SE-model)" for a (syntactic) model defined in a modeling language such as UML. In contrast to this, "(semantic) models" are part of the mathematical semantics of a modeling language so that a semantic model corresponds to a model of a theory in a suitable logic; here, we will use institutional models (Ins-models).

$$\begin{tikzcd}
 & & & S_2 \\
\textit{multi-modeling language } L & S_1 \arrow[urr, "trans_{12}"] \arrow[dr, "trans_{13}"] & & \\
 & & S_3 & \\
 & & & M_2 \arrow[uuu, dashed, "\text{conforms to}"] \arrow[ddd, dashed, "\text{semantics in}"] \\
\textit{multimodel} & M_1 \arrow[uuu, dashed, "\text{conforms to}"] \arrow[ddd, dashed, "\text{semantics in}"] \arrow[urr, mapsto, "trans_{12}"] \arrow[dr, mapsto, "trans_{13}"] & & \\
 & & M_3 \arrow[uuu, dashed, "\text{conforms to}"] \arrow[ddd, dashed, "\text{semantics in}"] & \\
 & & & \mathcal{I}_2 \\
\textit{heterogeneous semantics} & \mathcal{I}_1 \arrow[urr, "conn_{12}"] \arrow[dr, "conn_{13}"] & & \\
 & & \mathcal{I}_3 &
\end{tikzcd}$$

Fig. 1. Relations between metamodels, models, and semantic domains.

$trans_{13}$ between the sublanguages S_1 and S_2, and S_1 and S_3, respectively, are applied to M_1 yielding (sub-models of) M_2 and M_3. These model transformations are backed by semantic connections $conn_{12}$ and $conn_{13}$ between $\mathcal{I}_1$ and $\mathcal{I}_2$, $\mathcal{I}_3$ which make it possible to show that these model transformations are correct.

In addition to make these concepts precise, we illustrate them by a case study involving (UML) class diagrams and relational database schema diagrams as modeling languages. Based on earlier work [12] we show that class diagrams and schema diagrams form a multi-modeling language where class diagrams are related to schema diagrams by a semantically correct model transformation.

The paper is organized as follows: In Sect. 2 we briefly recall the necessary background from the theory of institutions. Section 3 shows how MOF metamodels and model transformations are algebraically formalized as membership equational theories. In Sect. 4 we present the institutional semantics of metamodels and in Sect. 5 our formal notions of semantic connections between institutions and of correct model transformations. The notions of multi-modeling languages and consistent multimodels are introduced in Sect. 6. In Sect. 7 we discuss related work and future work.

2 Preliminaries: Institutions and Institution (Co-)Morphisms

We briefly recall basic notions on institutions and their morphisms and comorphisms which form the framework for our institutional semantics of multi-modeling languages. We assume familiarity with the most elementary notions of category theory: category, functor, and natural transformation (see, e.g., [20]).

An *institution* [18] $\mathcal{I}$ is a tuple $\mathcal{I} = (Sign_{\mathcal{I}}, Sen_{\mathcal{I}}, Mod_{\mathcal{I}}, \models_{\mathcal{I}})$, with: (i) $Sign_{\mathcal{I}}$ a category whose objects are called *signatures*; (ii) a functor $Sen_{\mathcal{I}} : Sign_{\mathcal{I}} \to Set$, called the *sentence functor*, from $Sign_{\mathcal{I}}$ to Set, the category of sets; (iii) a contravariant functor

$Mod_{\mathcal{I}} : Sign_{\mathcal{I}}^{\mathrm{op}} \to Cat$, called the *model functor*, from $Sign_{\mathcal{I}}$ to Cat, the category of categories; and (iv) a family $\models_{\mathcal{I}} = \{\models_{\mathcal{I},\Sigma}\}_{\Sigma \in Sign_{\mathcal{I}}}$ of *satisfaction relations* between Σ-models $M \in Mod_{\mathcal{I}}(\Sigma)$ and Σ-sentences $\varphi \in Sen_{\mathcal{I}}(\Sigma)$, such that for each $H : \Sigma \to \Sigma'$ in $Sign_{\mathcal{I}}$, $M' \in Mod_{\mathcal{I}}(\Sigma)$, and $\varphi \in Sen_{\mathcal{I}}(\Sigma)$, we have the equivalence

$$Mod_{\mathcal{I}}(H)(M') \models_{\mathcal{I},\Sigma} \varphi \iff M' \models_{\mathcal{I},\Sigma'} Sen_{\mathcal{I}}(H)(\varphi) \ .$$

An institution provides a categorical semantics for the model-theoretic aspects of a logic, focusing on the satisfaction relation between models and sentences, and emphasizing that satisfaction is invariant under changes of syntax by signature morphisms. Note that, given an institution $\mathcal{I}$, we can always define an associated category $Th_{\mathcal{I}}$ of *theories* (theory *presentations* to be more exact, see, e.g., [21]), where theories are pairs (Σ, Γ) with $\Gamma \subseteq Sen_{\mathcal{I}}(\Sigma)$, and theory morphisms $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ are signature morphisms $H : \Sigma \to \Sigma'$ such that $\Gamma' \models_{\Sigma'} Sen_{\mathcal{I}}(H)(\Gamma)$, where the satisfaction relation is extended to a semantic consequence relation between sets of sentences in the usual way (see [18]). There is then an obvious functor $sign : Th_{\mathcal{I}} \to Sign_{\mathcal{I}}$ defined on objects by the equation $sign(\Sigma, \Gamma) = \Sigma$.

An *institution morphism* [18] $\mu : \mathcal{I} \twoheadrightarrow \mathcal{I}'$ from an institution $\mathcal{I}$ to another institution $\mathcal{I}'$ is given by: (i) a functor $\mu^{Sign} : Sign_{\mathcal{I}} \to Sign_{\mathcal{I}'}$; (ii) a natural transformation $\mu^{Sen} : \mu^{Sign}; Sen_{\mathcal{I}'} \Rightarrow Sen_{\mathcal{I}}$; and (iii) a natural transformation $\mu^{Mod} : Mod_{\mathcal{I}} \Rightarrow \mu^{Sign^{\mathrm{op}}}; Mod_{\mathcal{I}'}$, such that for each $M \in Mod_{\mathcal{I}}(\Sigma)$ and each sentence $\varphi' \in Sen_{\mathcal{I}'}(\mu^{Sign}(\Sigma))$ we have

$$M \models_{\mathcal{I},\Sigma} \mu_{\Sigma}^{Sen}(\varphi') \iff \mu_{\Sigma}^{Mod}(M) \models_{\mathcal{I}',\mu^{Sign}(\Sigma)} \varphi' \ .$$

Dually, an *institution comorphism* [19] (called a map of institutions in [21]) $\rho : \mathcal{I} \to \mathcal{I}'$ is given by: (i) a functor $\rho^{Sign} : Sign_{\mathcal{I}} \to Sign_{\mathcal{I}'}$; (ii) a natural transformation $\rho^{Sen} : Sen_{\mathcal{I}} \Rightarrow \rho^{Sign}; Sen_{\mathcal{I}'}$; and (iii) a natural transformation $\rho^{Mod} : \rho^{Sign^{\mathrm{op}}}; Mod_{\mathcal{I}'} \Rightarrow Mod_{\mathcal{I}}$, such that for each $M' \in Mod_{\mathcal{I}'}(\rho^{Sign}(\Sigma))$ and each sentence $\varphi \in Sen_{\mathcal{I}}(\Sigma)$ we have

$$M' \models_{\mathcal{I}',\rho^{Sign}(\Sigma)} (\varphi) \iff \rho_{\Sigma}^{Mod}(M') \models_{\mathcal{I},\Sigma} \varphi \ .$$

Note that, given an institution comorphism $\rho : \mathcal{I} \to \mathcal{I}'$, the functor ρ^{Sign} extends naturally to a functor $\rho^{Th} : Th_{\mathcal{I}} \to Th_{\mathcal{I}'}$ with $\rho^{Th}(\Sigma, \Gamma) = (\rho^{Sign}(\Sigma), \rho_{\Sigma}^{Sen}(\Gamma))$.

3 Algebraic Semantics of MOF and of Model Transformations

We briefly explain how a MOF metamodel defines a modeling language, how it is formalized by means of a membership-equational logic theory, and how model transformations are formalized as equationally-defined functions in MOMENT2.

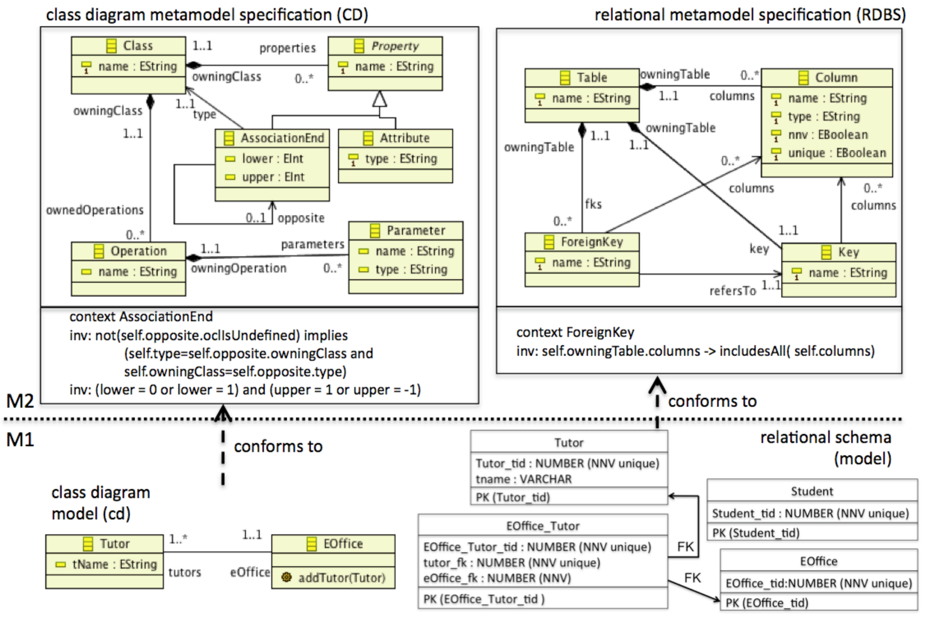
3.1 MOF

MOF [26] is a semiformal approach to define modeling languages. It provides a four-level hierarchy, with levels M_0, M_1, M_2 and M_3, where level M_{i+1} serves as the *metalevel* for level M_i. The entities populating level M_i are *collections* of a certain *type*, which is defined by means of an entity at level M_{i+1}. Level M_0 contains collections of structured data that are defined by using a specific model in a modeling space, e.g., tuples in a database or class instances of a class diagram. Level M_1 contains *models*, which are used to represent a specific reality by using a well-defined language for computer-based interpretation such as class diagrams or relational schemas. Level M_2 contains *metamodels*. A metamodel is a model specifying the types that can be used in a modeling language, such as the metamodel CD for defining class diagrams and the metamodel RDBS for defining relational schemas, as shown in Fig. 2. An entity at level M_3 is a *meta-metamodel* enabling the definition of metamodels at the level M_2.

For a model M at level M_1 and a metamodel $\mathscr{M}$ at level M_2, we write $M : \mathscr{M}$ to denote the *metamodel conformance* relation. In addition, a metamodel $\mathscr{M}$ can be enriched with a set $\mathscr{C}$ of OCL constraints constituting a *metamodel specification* $(\mathscr{M}, \mathscr{C})$ [7] so that a model M conforms to $(\mathscr{M}, \mathscr{C})$ when it conforms to the metamodel $\mathscr{M}$ *and* satisfies the constraints $\mathscr{C}$. In Fig. 2, the OCL constraints over the CD metamodel defines the concept of opposite association ends and restricts the set of possible cardinalities. The OCL constraint over the RDBS metamodel indicates that the columns of a foreign key should be contained in the same table where the column is defined.

3.2 Algebraic Semantics of Metamodel Specifications and MOMENT2

The goal of the algebraic semantics of metamodel specifications in [8,7] is to give a precise semantics to the conformance relation $M : (\mathscr{M}, \mathscr{C})$ between a model M and



```
context AssociationEnd
inv: not(self.opposite.oclIsUndefined) implies
        (self.type=self.opposite.owningClass and
        self.owningClass=self.opposite.type)
inv: (lower = 0 or lower = 1) and (upper = 1 or upper = -1)
```

```
context ForeignKey
inv: self.owningTable.columns -> includesAll( self.columns)
```

Fig. 2. Levels M_2 and M_1 of the MOF hierarchy: metamodel and model examples.

a metamodel specification $(\mathscr{M},\mathscr{C})$ (this subsumes $M : \mathscr{M}$ using $M : (\mathscr{M},\emptyset)$). This semantics is achieved as follows. First of all, the set of MOF-conformant metamodel specifications $(\mathscr{M},\mathscr{C})$ is a syntactically well-defined set *MetamodelSpecs*. Second, the set of equational theories in the institution of membership equational logic [22] is another well-defined set Th_{MEL}. The algebraic semantics is then defined as a function

$$\mathbb{A} : \mathit{MetamodelSpecs} \to \mathit{Th}_{\text{MEL}} \ : \ (\mathscr{M},\mathscr{C}) \mapsto \mathbb{A}(\mathscr{M},\mathscr{C}) \ .$$

The key point of this algebraic semantics is that the set of models M conformant with $(\mathscr{M},\mathscr{C})$, which we denote $[\![(\mathscr{M},\mathscr{C})]\!]$, is precisely axiomatized as the carrier of the sort *CModel* in the *initial algebra* $\mathcal{T}_{\mathbb{A}(\mathscr{M},\mathscr{C})}$ of the MEL theory $\mathbb{A}(\mathscr{M},\mathscr{C})$. That is, we have the definitional equality $[\![(\mathscr{M},\mathscr{C})]\!] = \mathcal{T}_{\mathbb{A}(\mathscr{M},\mathscr{C}),\mathit{CModel}}$, and hence

$$M : (\mathscr{M},\mathscr{C}) \iff M \in [\![(\mathscr{M},\mathscr{C})]\!] \iff M \in \mathcal{T}_{\mathbb{A}(\mathscr{M},\mathscr{C}),\mathit{CModel}} \ .$$

Intuitively, the elements of sort *CModel* are models algebraically represented as sets of objects with an associative, commutative union operation with identity (ACU), corresponding to an algebraic description of graphs. MEL is used in an essential way to impose the OCL constraints $\mathscr{C}$ by means of a conditional membership.

The algebraic semantics supports the notion of *submodel* (see [6] for details). From a graph-theoretic point of view, given $M_1, M_2 \in [\![(\mathscr{M},\mathscr{C})]\!]$, we say that M_1 is a *submodel* of M_2, written $M_1 \subseteq M_2$ iff it is a *subgraph*, so that all the nodes (objects with attribute values) and edges (association ends) of M_1 are included in M_2 up to name and edge order isomorphism. The submodel relation is a partial order, endowing $[\![(\mathscr{M},\mathscr{C})]\!]$ with a poset structure $([\![(\mathscr{M},\mathscr{C})]\!], \subseteq)$. The notion of submodel will be very useful to obtain a flexible notion of multimodel in a multimodeling language.

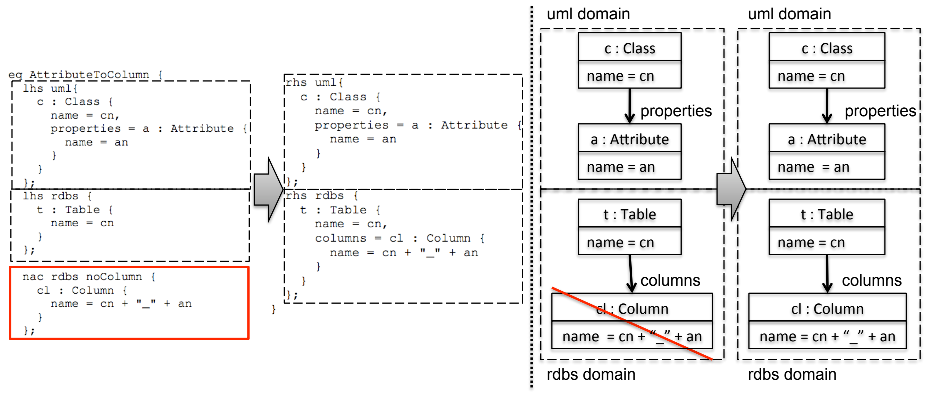
These notions are implemented in Maude and integrated within the Eclipse Modeling Framework (EMF) in the MOMENT2 tool [8,7,6].

3.3 Model Transformations

In this work we consider functional model transformations that map input models M, such that $M : (\mathscr{M},\mathscr{C})$, to output models $\beta(M)$ so that $\beta(M) : (\mathscr{M}',\mathscr{C}')$, where in general $(\mathscr{M},\mathscr{C}) \neq (\mathscr{M}',\mathscr{C}')$.

Definition 1. *Given metamodel specifications $(\mathscr{M},\mathscr{C})$ and $(\mathscr{M}',\mathscr{C}')$, a* functional model transformation *from $(\mathscr{M},\mathscr{C})$ to $(\mathscr{M}',\mathscr{C}')$ is a function $\beta : [\![(\mathscr{M},\mathscr{C})]\!] \to [\![(\mathscr{M}',\mathscr{C}')]\!]$. The transformation β is called* monotonic*, if, in addition, it is a monotonic function $\beta : ([\![(\mathscr{M},\mathscr{C})]\!], \subseteq) \to ([\![(\mathscr{M}',\mathscr{C}')]\!], \subseteq)$.*

MEL theories $\mathbb{A}(\mathscr{M},\mathscr{C})$ associated to MOF metamodel specifications $(\mathscr{M},\mathscr{C})$ are by construction executable by rewriting in Maude [13]; in fact by confluent and terminating equations modulo ACU. Therefore, the initial algebra $\mathcal{T}_{\mathbb{A}(\mathscr{M},\mathscr{C})}$ is computable [4]. Furthermore, any computable function $\beta : [\![(\mathscr{M},\mathscr{C})]\!] \to [\![(\mathscr{M}',\mathscr{C}')]\!]$ can in such a case be specified by a finite set of confluent and terminating equations modulo ACU. This is exactly the approach taken in MOMENT2, where a model transformation β can be specified as a set of recursive model equations, which are automatically translated into ordinary MEL equations, as detailed in [6].



```
eq AttributeToColumn {
  lhs uml{
    c : Class {
      name = cn,
      properties = a : Attribute {
        name = an
      }
    }
  };
  lhs rdbs {
    t : Table {
      name = cn
    }
  };
  nac rdbs noColumn {
    cl : Column {
      name = cn + "_" + an
    }
  };
```

```
rhs uml{
  c : Class {
    name = cn,
    properties = a : Attribute {
      name = an
    }
  }
};
rhs rdbs {
  t : Table {
    name = cn,
    columns = cl : Column {
      name = cn + "_" + an
    }
  }
};
}
```

Fig. 3. Model equation: MOMENT2 format (left) and graphical representation (right).

Let us consider a model transformation between the metamodel specifications for class diagrams (CD) and for relational database schemas (RDBS) in Fig. 2: $\beta_{\text{cd2rdbs}} : [\![(\mathscr{M}_{\text{CD}}, \mathscr{C}_{\text{CD}})]\!] \longrightarrow [\![(\mathscr{M}_{\text{RDBS}}, \mathscr{C}_{\text{RDBS}})]\!]$. The transformation is defined by several model equations that specify the translation process: classes are transformed into tables with primary keys, class attributes are transformed into table columns, and bidirectional associations are transformed into auxiliar tables that contain foreign keys that point to the tables that correspond to the associated classes. The complete specification of the model transformation is given in App. A using the concrete syntax of MOMENT2. In Fig. 3 we show a simplified version of the model equation that generates columns in a table from attributes of a class. The model equation is specified by a left-hand side (LHS) model pattern, a right-hand side (RHS) model pattern, and a negative application condition (NAC), which is applied over a LHS instance. The NAC ensures that the rule is applied only once for each attribute. MOMENT2 formalizes this model transformation as the function β_{cd2rdbs}, which is internally defined by equations that are generated from the user defined model equations of the transformation (see [6] for further details).

β_{cd2rdbs} maps the class diagram model cd in Fig. 2 to the relational schema $\beta_{\text{cd2rdbs}}(cd) \subseteq rs$, where rs is the relational schema shown in Fig. 2. This model transformation is *monotonic* by considering the submodel relation in both source and target metamodel specifications. In particular, we consider the submodel $tutor$ of cd that is constituted only by the class Tutor, i.e., $tutor, cd \in [\![(\mathscr{M}_{\text{CD}}, \mathscr{C}_{\text{CD}})]\!]$ and $tutor \subseteq cd$. We have then that $\beta_{\text{cd2rdbs}}(tutor) \subseteq \beta_{\text{cd2rdbs}}(cd)$.

4 Institutional Semantics of Metamodels

In order to capture the semantics of models conforming to a given metamodel, we use the mathematical framework of institutions.

Definition 2. *Given a MOF-compliant metamodel specification* $(\mathscr{M}, \mathscr{C})$*, an* institutional semantics *for* $(\mathscr{M}, \mathscr{C})$ *is specified by: (i) an institution* $\mathcal{I}$*; and (ii) a functor* $\sigma : ([\![(\mathscr{M}, \mathscr{C})]\!], \subseteq) \to Th_{\mathcal{I}}$.

Therefore, an SE-model $M \in [\![(\mathscr{M}, \mathscr{C})]\!]$ is interpreted as a *theory* $\sigma(M) \in \mathit{Th}_{\mathcal{I}}$ in the corresponding institution. This definition highlights a crucial difference between "models" in the software engineering sense, which we call *SE-models*, and semantic models in the institution $\mathcal{I}$, which we call *Ins-models*, and in this case $\mathcal{I}$-models. The key point is that an SE-model of a system is only a *partial specification* of such a system, allowing many possible implementations. For example, in a UML class diagram the semantics of the methods involved is typically only partially specified. By contrast, an Ins-model is typically much closer to an actual implementation, and may fully constrain various relevant aspects of such an implementation: for example, the full semantic specification of the methods in a class diagram.

This is captured by the above definition which gives semantics to an SE-model $M \in [\![(\mathscr{M}, \mathscr{C})]\!]$ as a logical theory $\sigma(M) \in \mathit{Th}_{\mathcal{I}}$. That is, $\sigma(M)$ is a "partial" specification describing not a single Ins-model, but a class (actually a category) of Ins-models in the institution $\mathcal{I}$, viz. the class $\mathit{Mod}_{\mathcal{I}}(\sigma(M))$. Such $\mathcal{I}$-models typically fully constrain some relevant aspects of the system partially specified by the SE-model M. For example, if we choose for $\mathcal{I}$ a *computational logic*, some of the $\mathcal{I}$-models associated to M may be *executable* as programs. Therefore, an institutional semantics for a metamodel specification may support program generation methods that are correct by construction. For relational database schemas as SE-models, e.g., the Ins-models may be relational models of actual databases conformant with the given schema.

The functoriality condition in the definition of institutional semantics is very natural. Intuitively, if $M \subseteq M'$, then any implementation of the system partially specified by the SE-model M' should *a fortiori* give us an implementation of the system partially specified by M, essentially by disregarding the implementation of the extra features in $M' \setminus M$. Mathematically, this is captured by the fact that the submodel inclusion $M \subseteq M'$ induces a theory morphism $\sigma(M) \to \sigma(M')$, which, in turn, by the contravariance of the functor $\mathit{Mod}_{\mathcal{I}} : \mathit{Sign}_{\mathcal{I}}^{\mathrm{op}} \to \mathit{Cat}$, induces a forgetful functor $\mathit{Mod}_{\mathcal{I}}(\sigma(M')) \to \mathit{Mod}_{\mathcal{I}}(\sigma(M))$, corresponding to the intuition that from an implementation of M' we can always obtain an implementation of M. This functoriality condition will be useful to arrive at a proper notion of Ins-models for an SE-multimodel.

Let us exemplify the definition by explaining the institutional semantics for our running examples of (simplified) UML class diagrams and relational database schemas. The first is described in more detail in [11], the second builds on the traditional semantics of relational database schemas [14].

Institutional semantics for class diagrams. Signatures of the class diagram institution $\mathcal{I}_{\mathrm{CD}}$ declare class names, typed attributes, operations, and association names with corresponding properties as association ends. On the signature part, the functor $\sigma_{\mathrm{CD}} : ([\![\mathscr{M}_{\mathrm{CD}}, \mathscr{C}_{\mathrm{CD}}]\!], \subseteq) \to \mathit{Th}_{\mathcal{I}_{\mathrm{CD}}}$ maps, for instance, the class diagram in Fig. 2 to the $\mathcal{I}_{\mathrm{CD}}$-signature

$$\begin{aligned}
&(\{\mathsf{EOffice}, \mathsf{Tutor}, \mathsf{EString}, \mathsf{Void}\},\\
&\ \{\mathsf{tName} : \mathsf{Tutor} \to \mathsf{EString}\},\\
&\ \{\mathsf{addTutor} : \mathsf{EOffice} \times \mathsf{Tutor} \to \mathsf{Void}\},\\
&\ \{\mathsf{tuteof} \subseteq \mathsf{tutors} : \mathsf{Tutor} \times \mathsf{eOffice} : \mathsf{EOffice}\})\ .
\end{aligned}$$

Sentences associated with a signature of $\mathcal{I}_{\mathrm{CD}}$ declare multiplicities of form $0..1, 0..\star, 1..1, 1..\star$ for associations. Applying σ_{CD} to the class diagram of Fig. 2 yields a theory with a single sentence

$$\mathit{association}(\mathsf{tuteof}, \mathsf{tutors} : \mathsf{Tutor} : 1..\star, \mathsf{eOffice} : \mathsf{EOffice} : 1..1)\ .$$

Models of a class diagram signature are given as sets of object states. Object states are sets of created object identifiers of the declared class names, together with functions that interpret attributes and methods, as well as relations that interpret associations. Moreover, models of a presentation are required to satisfy the constraints put on associations. In our example, for each e-office, there is at least one tutor, and for each tutor there is exactly one e-office, such that if we navigate from a tutor to his e-office, then we can navigate back to the tutor, and vice versa.

A signature morphism between two class diagram signatures consistently maps class names, properties, methods, and association names. For example, there is an embedding signature morphism from the signature induced by our sample model without the method addTutor to the signature of the sample model above. The reduct of any model along a signature morphism simply "forgets" all additional information of the target. Signature morphisms canonically extend to sentences.

Institutional semantics for relational database schemas. Signatures of the relational database schema institution $\mathcal{I}_{\mathrm{RDBS}}$ declare the primitive types, the table names, the columns names, the typing of columns, and the primary keys of tables where each primary key of a table has to be a column of that table. On the signature part, the functor $\sigma_{\mathrm{RDBS}} : ([\![\mathscr{M}_{\mathrm{RDBS}}, \mathscr{C}_{\mathrm{RDBS}}]\!], \subseteq) \to \mathit{Th}_{\mathcal{I}_{\mathrm{RDBS}}}$ maps, for instance, the relational schema in Fig. 2 to the $\mathcal{I}_{\mathrm{RDBS}}$-signature

$$\begin{array}{l}
(\{\mathsf{NUMBER}, \mathsf{VARCHAR}\},\\
\ \{\mathsf{EOffice}, \mathsf{EOffice\_Tutor}, \mathsf{Tutor}\},\\
\ \{\mathsf{Tutor\_tid}, \mathsf{tname}, \mathsf{tutor\_fk}, \mathsf{eOffice\_fk}, \mathsf{EOffice\_tid}\},\\
\ \{(\mathsf{Tutor}, \mathsf{Tutor\_tid}) \mapsto \mathsf{NUMBER}, (\mathsf{Tutor}, \mathsf{tname}) \mapsto \mathsf{VARCHAR},\\
\quad (\mathsf{EOffice\_Tutor}, \mathsf{tutor\_fk}) \mapsto \mathsf{NUMBER}, (\mathsf{EOffice\_Tutor}, \mathsf{eOffice\_fk}) \mapsto \mathsf{NUMBER},\\
\quad (\mathsf{EOffice}, \mathsf{EOffice\_tid}) \mapsto \mathsf{NUMBER}\},\\
\ \{\mathsf{Tutor} \mapsto \mathsf{Tutor\_tid}, \mathsf{EOffice\_Tutor} \mapsto \mathsf{tutor\_fk}, \mathsf{EOffice} \mapsto \mathsf{EOffice\_tid}\})
\end{array}$$

Sentences associated with a signature of $\mathcal{I}_{\mathrm{RDBS}}$ declare constraints on tables and columns: all column entries in a table that do not correspond to either primary keys or foreign keys can be null (not nnv) and not unique, all columns that correspond to primary keys shall be nnv and unique, and all columns that correspond to foreign keys (fk) encode cardinality constraints as nnv and unique statements. Applying σ_{RDBS} to the example in Fig. 2 yields a theory with the following sentences:

$$\begin{array}{ll}
\mathit{nnv}(\mathsf{Tutor}, \mathsf{Tutor\_id}) & \mathit{unique}(\mathsf{Tutor}, \mathsf{Tutor\_id})\\
\mathit{nnv}(\mathsf{EOffice\_Tutor}, \mathsf{tutor\_fk}) & \mathit{unique}(\mathsf{EOffice\_Tutor}, \mathsf{tutor\_fk})\\
\mathit{nnv}(\mathsf{EOffice\_Tutor}, \mathsf{eOffice\_fk}) & \\
\mathit{fk}(\mathsf{EOffice\_Tutor}, \mathsf{tutor\_fk}, \mathsf{Tutor}) & \mathit{fk}(\mathsf{EOffice\_Tutor}, \mathsf{eOffice\_fk}, \mathsf{EOffice})
\end{array}$$

$nnv(\textsf{EOffice}, \textsf{EOffice\_id})$ $unique(\textsf{EOffice}, \textsf{EOffice\_id})$

Models of a relational database signature are given by relations over interpretations of the primitive types such that the typings of the columns in tables are satisfied. The interpretation of primitive types introduces special null-values. A model satisfies a clause $nnv(t, c)$ if the projection corresponding to the column c of the interpretation of the table t does not contain non-null values; and it satisfies clause $unique(t, c)$, if the projection of t to c does not show duplicated entries. A model satisfies a clause $fk(t_1, c_{i_1} \dots c_{i_k}, t_2)$ if for all tuples in t_1 projected to $c_{i_1} \dots c_{i_k}$, there exists a tuple in t_2 projected over its primary key columns.

A signature morphism between two relational database signatures consistently maps the primitive types, the table names, and the columns names such that this mapping can be extended to the typing of columns and the primary keys of tables.

5 Semantic Connections and Correct Model Transformations

The institutional semantics of metamodel specifications provides a formal framework without which the following burning question in software engineering cannot be given any precise meaning: *When is a model transformation $\beta : [\![(\mathscr{M}, \mathscr{C})]\!] \to [\![(\mathscr{M}', \mathscr{C}')]\!]$ correct?*

The point is that, although model transformations can be very useful to leverage model building efforts in one modeling language to be used in another modeling language, we can in principle define many such βs, but some of them may be *disastrous*. Given an SE-model $M \in [\![(\mathscr{M}, \mathscr{C})]\!]$, which gives us a partial specification of a system, we want the transformed model $\beta(M) \in [\![(\mathscr{M}', \mathscr{C}')]\!]$ to be a model of the *same* system from a different perspective. In particular, $\beta(M)$ should *never* have implementations that are *incompatible* with those allowed by M. However, when modeling languages do not have any precise mathematical semantics, this very real problem can be painfully experienced in practice, but there is no way to systematically understand and prevent it.

5.1 Semantic Connections

Institution (co-)morphisms provide relations between different institutions which can be used to reflect model transformations semantically. Intuitively, institution comorphisms map a "poorer" institution into a "richer" one, whereas institution morphisms forget logical structure by mapping a "richer" institution into a "poorer" one. Sometimes, however, we have situations in which two institutions cannot be naturally related by either an institution morphism or an institution comorphism. In the example, $\mathcal{I}_{\mathrm{CD}}$ shows operations which have no counterpart in $\mathcal{I}_{\mathrm{RDBS}}$; on the other hand, $\mathcal{I}_{\mathrm{RDBS}}$ allows the uniqueness constraint to be stated while this cannot be mimicked in $\mathcal{I}_{\mathrm{CD}}$. However, we may choose a "lowest common denominator" institution $\mathcal{I}_{\mathrm{PCD}}$, which is poorer than both $\mathcal{I}_{\mathrm{CD}}$ and $\mathcal{I}_{\mathrm{RDBS}}$: the institution of "poor man's class diagrams", which is defined like $\mathcal{I}_{\mathrm{CD}}$ but does not show operations in its signature. We can then use this $\mathcal{I}_{\mathrm{PCD}}$ to relate $\mathcal{I}_{\mathrm{CD}}$ and $\mathcal{I}_{\mathrm{RDBS}}$ by what we call a *semantic connection*.[2]

[2] In recent discussions with A. Tarlecki we have learned that the same idea is also contemplated in his upcoming paper with T. Mossakowski [24].

Definition 3. *A* semantic connection *between an institution $\mathcal{I}$ and another institution $\mathcal{I}'$ is a pair (μ, ρ) of the form $\mathcal{I} \overset{\mu}{\twoheadrightarrow} \mathcal{I}_0 \overset{\rho}{\rightarrow} \mathcal{I}'$, where $\mathcal{I}_0$ is a third institution, μ is an institution morphism, and ρ is an institution comorphism.*

Using $\mathcal{I}_{\mathrm{PCD}}$, we may define a semantic connection $\mathcal{I}_{\mathrm{CD}} \overset{\mu_{\mathrm{C2R}}}{\twoheadrightarrow} \mathcal{I}_{\mathrm{PCD}} \overset{\rho_{\mathrm{C2R}}}{\rightarrow} \mathcal{I}_{\mathrm{RDBS}}$ between $\mathcal{I}_{\mathrm{CD}}$ and $\mathcal{I}_{\mathrm{RDBS}}$ as follows: The signature part $\mu^{\mathit{Sign}}_{\mathrm{C2R}}$ of the institution morphism $\mu_{\mathrm{C2R}} : \mathcal{I}_{\mathrm{CD}} \rightarrow \mathcal{I}_{\mathrm{PCD}}$ forgets all operations and the sentence part $\mu^{\mathit{Sen}}_{\mathrm{C2R}}$ is the identity. On the other hand, the institution comorphism ρ_{C2R} is defined along the lines of the model transformation β_{cd2rdbs} in Sect. 3.3, adding primary keys and encoding $\mathit{association}$-clauses as nnv and unique properties of the columns that are involved in foreign keys; here the model part $\rho^{\mathit{Mod}}_{\mathrm{C2R}}$ is the identity.

A semantic connection $\mathcal{I} \overset{\mu}{\twoheadrightarrow} \mathcal{I}_0 \overset{\rho}{\rightarrow} \mathcal{I}'$ also allows us to relate models in $\mathcal{I}$ and $\mathcal{I}'$ by viewing them both as models in the "common semantic ground" $\mathcal{I}_0$:

Definition 4. *Given institutions $\mathcal{I}$, $\mathcal{I}'$, and a semantic connection $\mathcal{I} \overset{\mu}{\twoheadrightarrow} \mathcal{I}_0 \overset{\rho}{\rightarrow} \mathcal{I}'$, a pair of models $(\overline{M}, \overline{M}')$, with $\overline{M} \in \mathit{Mod}_{\mathcal{I}}(\Sigma)$, $\overline{M}' \in \mathit{Mod}_{\mathcal{I}'}(\rho^{\mathit{Sign}}(\mu^{\mathit{Sign}}(\Sigma))$, and $\Sigma \in \mathit{Sign}_{\mathcal{I}}$, is called* (μ, ρ)-consistent, *if $\mu^{\mathit{Mod}}_{\Sigma}(\overline{M}) = \rho^{\mathit{Mod}}_{\mu^{\mathit{Sign}}(\Sigma)}(\overline{M}')$.*

For the semantic connection $\mathcal{I}_{\mathrm{CD}} \overset{\mu_{\mathrm{C2R}}}{\twoheadrightarrow} \mathcal{I}_{\mathrm{PCD}} \overset{\rho_{\mathrm{C2R}}}{\rightarrow} \mathcal{I}_{\mathrm{RDBS}}$, e.g., two models for the class diagram and the relational database schema in Fig. 2 which have a different number of Tutors would be inconsistent.

5.2 Correctness of Model Transformations

Based on semantic connections, we may go on to define a notion of semantic correctness for a model transformation.

Definition 5. *Given metamodel specifications $(\mathscr{M}, \mathscr{C})$ and $(\mathscr{M}', \mathscr{C}')$ with corresponding institutional semantics $(\mathcal{I}, \sigma : (\llbracket(\mathscr{M}, \mathscr{C})\rrbracket, \subseteq) \rightarrow \mathit{Th}_{\mathcal{I}})$ and $(\mathcal{I}', \sigma' : (\llbracket(\mathscr{M}', \mathscr{C}')\rrbracket, \subseteq) \rightarrow \mathit{Th}_{\mathcal{I}'})$, and given a semantic connection $\mathcal{I} \overset{\mu}{\twoheadrightarrow} \mathcal{I}_0 \overset{\rho}{\rightarrow} \mathcal{I}'$, a model transformation $\beta : \llbracket(\mathscr{M}, \mathscr{C})\rrbracket \rightarrow \llbracket(\mathscr{M}', \mathscr{C}')\rrbracket$ is called* (μ, ρ)-correct, *if the following two conditions hold:*

1. *For each $M \in \llbracket(\mathscr{M}, \mathscr{C})\rrbracket$ we have*

$$\rho^{\mathit{Sign}}(\mu^{\mathit{Sign}}(\mathit{sign}(\sigma(M))) = \mathit{sign}'(\sigma'(\beta(M))) \ .$$

This condition can be visualized as the commutativity of the diagram:

$$\begin{tikzcd}
\llbracket(\mathscr{M}, \mathscr{C})\rrbracket \arrow[rr, "\beta"] \arrow[d, "\sigma"'] & & \llbracket(\mathscr{M}', \mathscr{C}')\rrbracket \arrow[d, "\sigma'"] \\
\mathit{Th}_{\mathcal{I}} \arrow[d, "\mathit{sign}"'] & & \mathit{Th}_{\mathcal{I}'} \arrow[d, "\mathit{sign}'"] \\
\mathit{Sign}_{\mathcal{I}} \arrow[r, "\mu^{\mathit{Sign}}"] & \mathit{Sign}_{\mathcal{I}_0} \arrow[r, "\rho^{\mathit{Sign}}"] & \mathit{Sign}_{\mathcal{I}'}
\end{tikzcd}$$

where if β is not monotonic this is just a commuting diagram of functions, but if β is monotonic we further require it to be a commuting diagram of functors.

2. *For each $M \in [\![(\mathscr{M},\mathscr{C})]\!]$ we have the containment:*

$$\rho^{Mod}_{\Sigma'}(Mod_{\mathcal{I}'}(\sigma'(\beta(M))) \subseteq \mu^{Mod}_{\Sigma}(Mod_{\mathcal{I}}(\sigma(M)))$$

where $\Sigma = sign(\sigma(M))$, $\Sigma_0 = \mu^{Sign}(\Sigma)$, and $\Sigma' = \rho^{Sign}(\Sigma_0)$.

Note that condition (1) is a sanity check for the SE-models M and $\beta(M)$ to be *relatable* at the semantic level, since the signatures of their corresponding theories $\sigma(M)$ and $\sigma'(\beta(M))$ should be compatible. Condition (2) assumes condition (1) and adds the further stipulation that each $\mathcal{I}'$-model of $\beta(M)$, when brought to the common ground $\mathcal{I}_0$, should also be (the downgraded version of) an $\mathcal{I}$-model of M. This captures the crucial requirement that an implementation of $\beta(M)$ should never be incompatible with the implementations allowed by M.

The model transformation β_{cd2rdbs} is indeed correct w.r.t. the semantic connection $\mathcal{I}_{\mathrm{CD}} \overset{\mu_{\mathrm{C2R}}}{\twoheadrightarrow} \mathcal{I}_{\mathrm{PCD}} \overset{\rho_{\mathrm{C2R}}}{\rightarrow} \mathcal{I}_{\mathrm{RDBS}}$, as, given a class diagram $cd \in (\mathscr{M}_{\mathrm{CD}}, \mathscr{C}_{\mathrm{CD}})$, the "poor man's"-models of $\sigma_{\mathrm{RDBS}}(\beta_{\mathrm{cd2rdbs}}(cd))$ in $\mathcal{I}_{\mathrm{PCD}}$ still fulfill all cardinality constraints induced by associations.

6 Multimodeling Languages

At the very least, a multimodeling language should be a collection of modeling languages supporting different views of a system. But if no interactions of any kind are supported between models in the different modeling sublanguages, a multimodeling language is not very useful, since there is no way of taking advantage of model building and model analysis efforts in one sublanguage to benefit similar efforts in another sublanguage. Therefore, we assume in what follows that a multimodeling language supports model transformations between some of its sublanguages.

Definition 6. *A* multimodeling language *is specified by*

1. *A family $((\mathscr{M}_i, \mathscr{C}_i))_{i \in I}$ of metamodel specifications.*
2. *An irreflexive relation $K \subseteq I \times I$ where each pair $(i,j) \in K$ is called a* connection.
3. *For each $(i,j) \in K$ a model transformation $\beta_{ij} : [\![(\mathscr{M}_i, \mathscr{C}_i)]\!] \to [\![(\mathscr{M}_j, \mathscr{C}_j)]\!]$.*

The family $L_{\mathrm{C\&R}} = (\mathscr{M}_i, \mathscr{C}_i)_{i \in \{\mathrm{CD},\mathrm{RDBS}\}}$ with $K_{\mathrm{C\&R}} = \{(\mathrm{CD}, \mathrm{RDBS})\}$ and $\beta_{\mathrm{C\&R}} = \{\beta_{\mathrm{CD,RDBS}}\}$ with $\beta_{\mathrm{CD,RDBS}} = \beta_{\mathrm{cd2rdbs}}$ may serve as a simple example of a multimodeling language $\mathrm{C\&R} = (L_{\mathrm{C\&R}}, K_{\mathrm{C\&R}}, \beta_{\mathrm{C\&R}})$.

It is assumed that for the purposes of the multimodeling language there is, given $(i,j) \in K$ a *single* model transformation relating $(\mathscr{C}_i, \mathscr{M}_i)$ to $(\mathscr{C}_j, \mathscr{M}_j)$. This seems reasonable, since such a model transformation is used to provide a systematic "change of viewpoint" from the perspective supported by $(\mathscr{C}_i, \mathscr{M}_i)$ to that of $(\mathscr{C}_j, \mathscr{M}_j)$.

We envision teams of system designers and developers using such a multimodeling language to design and develop a given system. A useful division of labor is supported by the multimodeling language, so that some team members may concentrate their efforts on building and validating models mostly in a given sublanguage. If the team is well-coordinated and the multimodelign language has a good infrastructure, team members working in different sublanguages will benefit from the efforts of their colleagues working in other sublanguages. For example, if a model in sublanguage $(\mathscr{C}_j, \mathscr{M}_j)$ has not yet been developed, a modeler may not have to begin from scratch, but may have available model framents in $(\mathscr{C}_j, \mathscr{M}_j)$ that have been obtained by transformations from models in other sublanguages $(\mathscr{C}_i, \mathscr{M}_i)$. Or a person responsible for model analysis in sublanguage $(\mathscr{C}_j, \mathscr{M}_j)$ may be asked to verify some properties of a model in $(\mathscr{C}_j, \mathscr{M}_j)$ after it is transformed by $\beta_{i,j}$. This leads to the important question: *What is a multimodel?* for which we provide the following definition.

Definition 7. *Given a multimodeling language* $(((\mathscr{M}_i, \mathscr{C}_i))_{i \in I}, K, \beta)$ *an* I*-indexed family* $(M_i)_{i \in I}$ *is called*

1. *a* pre-multimodel*, if* $M_i \in [\![(\mathscr{M}_i, \mathscr{C}_i)]\!]$ *for all* $i \in I$*;*
2. *a* multimodel*, if it is a pre-multimodel and, furthermore, we have* $\beta_{ij}(M_i) \subseteq M_j$ *for all* $(i, j) \in K$*.*

The notion of pre-multimodel may seem chaotic, but may accurately reflect the real situation of a team at moments when different team members are actively developing different models quite independently of each other. We think of this as a hopefully *transient* but very common situation, reflecting the fact that the software team may be large and geogragraphically distributed, so that it may not be feasible for model changes in different sublanguages to be immediately taken into account across sublanguages.

However, to avoid dangerous and costly design divergences, from time to time team members should try to keep their model building efforts coordinated by freezing the current pre-multimodel $M = (M_i)_{i \in I}$ and asking some hard questions about it. A very natural question to ask is: is it the case that for each $(i, j) \in K$ we have $\beta_{ij}(M_i) \subseteq M_j$? This may not be the case, and then this may perhaps reveal that incompatible design decisions may have been made in different sublanguages.

The idea behind the model inclusion $\beta_{ij}(M_i) \subseteq M_j$ is that the model M_i, even when we transform it, may only account for part of all the information that must be modeled from the $(\mathscr{M}_j, \mathscr{C}_j)$ modeling point of view. Therefore, requiring an equality $\beta_{ij}(M_i) = M_j$ would be too restrictive. The inclusion requirement $\beta_{ij}(M_i) \subseteq M_j$ could perhaps be relaxed to a requirement that $\beta_{ij}(M_i)$ can be "mapped" to a submodel of M_j, but we do not explore this further here. For our example of tutors and e-offices in Fig. 2, the models not only form a multimodel in the multimodeling language C&R, but also $\beta_{\mathrm{cd2rdbs}}(cd) \subseteq rs$ holds.

Note that a multimodeling language as defined so far lacks an institutional semantics. This means that the requirements $\beta_{ij}(M_i) \subseteq M_j$, although useful for the coherence of the overall effort, are *primarily syntactic* and do not address the burning issue of the *semantic correctness* of the transformations β_{ij} supported by the multimodeling language. For this we need an institutional semantics.

Definition 8. *Given a multimodeling language* $(((\mathscr{M}_i, \mathscr{C}_i))_{i \in I}, K, \beta)$ *an* Ins-semantics *for it is specified by:*

1. *an Ins-semantics* $(\mathcal{I}_i, \sigma_i)$ *for each* $(\mathscr{M}_i, \mathscr{C}_i)$*,* $i \in I$*;*
2. *for each* $(i, j) \in K$ *a semantic connection* $\mathcal{I}_i \overset{\mu_{ij}}{\twoheadrightarrow} \mathcal{I}_{ij} \overset{\rho_{ij}}{\rightarrow} \mathcal{I}_j$ *such that* β_{ij} *is* (μ_{ij}, ρ_{ij})*-correct.*

Applying this definition to C&R we get that $((\mathcal{I}_i, \sigma_i)_{i \in \{\mathrm{CD},\mathrm{RDBS}\}}, \mathcal{I}_{\mathrm{CD}} \overset{\mu_{\mathrm{C2R}}}{\twoheadrightarrow} \mathcal{I}_{\mathrm{PCD}} \overset{\rho_{\mathrm{C2R}}}{\rightarrow} \mathcal{I}_{\mathrm{RDBS}})$ is an Ins-semantics for the multimodeling language C&R.

The above Ins-semantics for a multimodeling language can be very useful in several ways. First of all, it can make sure that its model transformations β_{ij} are semantically correct. Since they will be used all the time across many modeling efforts and their incorrectness would be disastrous, this is a very valuable requirement worth verifying. There is, however, a second very useful consequence, namely, that we also obtain a notion of *Ins-model* for a multimodel.

Definition 9. *Let $M = (M_i)_{i \in I}$ be a multimodel in a multimodeling language with an Ins-semantics. Then the class of its Ins-models is defined as the set $Mod(M)$ of all families $(\overline{M}_i)_{i \in I}$ where*

1. *$\overline{M}_i \in Mod_{\mathcal{I}_i}(\sigma_i(M_i))$, that is, each $\overline{M}_i$ is an Ins-model for the model M_i.*
2. *For each $(i, j) \in K$ the models $\overline{M}_i$ and $Mod_{\mathcal{I}_j}(\sigma_j(\beta_{ij}(M_i) \subseteq M_j))(\overline{M}_j)$ are (μ_{ij}, ρ_{ij})-consistent.*

The second condition is an "obvious" semantic compatibility condition, but it is somewhat terse in its formulation, so let us unpack it. Since $M = (M_i)_{i \in I}$ is a multimodel, for $(i, j) \in K$ we must have $\beta_{ij}(M_i) \subseteq M_j$. By the functoriality of σ_j this then gives us a theory morphism $\sigma_j(\beta_{ij}(M_i) \subseteq M_j)$, which is also a signature morphism, and which when applying the contravariant functor $Mod_{\mathcal{I}_j}$ to it gives us a reduct of $\overline{M}_j$ to a model in $Mod_{\mathcal{I}_j}(sign(\sigma_j(\beta_{ij}(M_i))))$. This reduct and the model M_i are the ones that must be (μ_{ij}, ρ_{ij})-consistent.

Why are such Ins-models useful from a software engineering point of view? Because they allow us to address another burning practical question: *When is a multimodel inconsistent?* Intuitively, a multimodel is inconsistent when it has no implementation meeting all the requirements imposed by all the models of the multimodel. But since Ins-models are mathematical surrogates for implementations of the differen system aspects (and may in fact *be* implementations when the logics are computable), if a multimodel has no Ins-models, then there is no hope for it to have an implementation.

Definition 10. *In a multimodeling language with an Ins-semantics a multimodel $M = (M_i)_{i \in I}$ is called* consistent*, if $Mod(M) \neq \emptyset$.*

The point, therefore, is that if $Mod(M) = \emptyset$, then the whole software design is *inconsistent* and irrealisable: the different models M_i in M place semantic constraints on each other that cannot be simultaneously satisfied.

7 Related Work and Conclusions

Interrelating different modeling notations is a difficult task due to the variety of possible structuring mechanisms and underlying computational paradigms. In the introduction we have already shortly discussed the three main approaches: the "system model approach", the "model-driven architecture approach", and the "heterogeneous semantics and development approach".

Further system model formalisms are for example, stream-based [10], graph grammar [16] and rewrite system models [13], or the integration of different specification formalisms, like CSP and Z [32]. In the model-driven architecture approach, the MOF standard permits the syntactical definition of modeling languages by means of the *meta-model* notion. The formal semantics of the MOF standard and its use for model transformations have been studied in algebraic [8], relational [2], graph grammar [5] and type-theoretic [27,28] settings. OCL-constraints of meta-models have been added in our algebraic setting in [7] and in the relational approach for Alloy in [3]. Most of these model transformation approaches are also well supported by tools such as AGG [1], VIATRA2 [31], and MOMENT2 [6]. The heterogeneous semantics line of research concentrates on the comparison and integration of different specification formalisms, retaining the formalisms most appropriate for expressing parts of the overall problem [33]. The theory of institutions [18] and its subsequent development into a powerful framework for heterogenous specifications [29,23,15,30] provide the mathematical foundations for our approach.

This paper is aimed as a first step for developing a consistent and semantically well-founded framework for software development with multiple modeling languages. We have presented a novel notion of multi-modeling language which not only allows the developer to study the consistency of a multi-language design, but makes it also easy to integrate additional modeling languages. In our approach a multi-modeling language consists of a set of sublanguages and correct model transformations between some of the sublanguages. The abstract syntax of the sublanguages is specified by MOF meta-models. The semantics of a multi-modeling language is then given by associating an institution to each of its sublanguages. A further main result of the paper is the notion of semantic correctness of model transformations. It is defined by a so-called semantic connection between the institutions of the source and target meta-model of the transformation. The main correctness condition is given by a model inclusion which expresses the fact that a model transformation is understood as a kind of semantic refinement relation. This definition corresponds well with the use of model transformations in MDA; in other settings one may use other kinds of model transformations such as refactorings and abstraction mappings. For such cases our correctness notion may not be adequate and we may need to distinguish between different notions of correctness such as refinement correctness, abstraction correctness, and structural correctness.

A careful reader may have observed that our algebraic semantics for MOF, which has provided what might be called the "metalevel" at which the Ins-semantics for modeling languages is defined, is itself an *instance* of this Ins-semantics. Specifically, all MOF-compliant metamodels are exactly the SE-*models* of the MOF *meta-metamodel*. Therefore, our algebraic semantics $\mathbb{A}$ for MOF is just an institutional semantics for a modeling language in the general sense we have proposed. Namely, a semantics in which the (meta-)metamodel is MOF itself, and the institution in question is MEL. This suggests several important generalizations of the present work. Why restricting ourselves to MOF? Why not considering similar semantics for multimodeling languages in other modeling frameworks? More generally, why not considering *multi-framework multi-languages*? Many challenging questions remain open and will be subject of our further studies including verification and tool support for multi-language consistency.

Acknowledgements We would like to thank Maria-Victoria Cengarle and Andrzej Tarlecki for several fruitful discussions on heterogeneous institutions and their morphisms.

This work has been partially supported by the ONR Grant N00014-02-1-0715, by the NSF Grant IIS-07-20482, by the NAOMI project funded by Lockheed Martin, by the EU project SENSORIA IST-2005-016004 and by the Spanish project META TIN2006-15175-C05-01.

References

1. The AGG website, 1997. `tfs.cs.tu-berlin.de/agg/`.
2. D. H. Akehurst, S. Kent, and O. Patrascoiu. A Relational Approach to Defining and Implementing Transformations Between Metamodels. *Softw. Sys. Model.*, 2(4):215–239, 2003.
3. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In *Proc. MoDELS'07*, volume 4735 of *LNCS*, pages 436–450. Springer, 2007.
4. J. A. Bergstra and J. V. Tucker. A Characterisation of Computable Data Types by Means of a Finite Equational Specification Method. In *Proc. ICALP'80*, volume 85 of *LNCS*, pages 76–90. Springer, 1980.
5. E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *Proc. MoDELS'08*, LNCS. Springer, 2008. To appear.
6. A. Boronat, R. Heckel, and J. Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. Technical Report CS-08-004, University of Leicester, 2008.
7. A. Boronat and J. Meseguer. Algebraic Semantics of OCL-constrained Metamodel Specifications. Technical Report UIUCDCS-R-2008-2995, University of Illinois, Urbana Champaign, 2008.
8. A. Boronat and J. Meseguer. An Algebraic Semantics for MOF. In *Proc. FASE'08*, volume 4961 of *LNCS*, pages 377–391. Springer, 2008.
9. M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The Structural Data Model. Technical Report TUM-I0612, Technische Universität München, 2006.
10. M. Broy and K. Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces, and Refinement*. Springer, 2001.
11. M. V. Cengarle and A. Knapp. An Institution for UML 2.0 Static Structures. Technical Report TUM-I0807, Technische Universität München, 2008.
12. M. V. Cengarle, A. Knapp, A. Tarlecki, and M. Wirsing. A Heterogeneous Approach to UML Semantics. In *Festschrift for Ugo Montanari*, volume 5019 of *LNCS*, pages 383–402. Springer, 2008.
13. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.
14. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM*, 13(6):377–387, 1970.
15. R. Diaconescu. *Institution-Independent Model Theory*. Birkhäuser, 2008.
16. G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A Combined Reference Model- and View-Based Approach to System Specification. *Int. J. Softw. Knowl. Eng.*, 7(4):457–477, 1997.
17. A. Finkelstein, M. Goedicke, J. Kramer, and C. Niskier. Viewpoint Oriented Software Development: Methods and Viewpoints in Requirements Engineering. In *Proc. Algebraic Methods'89*, volume 490 of *LNCS*, pages 29–54, 1991.
18. J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *J. ACM*, 39(1):95–146, 1992.

19. J. A. Goguen and G. Rosu. Institution Morphisms. *Form. Asp. Comp.*, 13(3–5):274–307, 2002.
20. S. MacLane. *Categories for the Working Mathematician.* Springer, 1971.
21. J. Meseguer. General Logics. In *Logic Coll. '87*, pages 275–329. North Holland, 1989.
22. J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *Proc. WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.
23. T. Mossakowski. Heterogeneous Specification and the Heterogeneous Tool Set. Habilitationsschrift, Universität Bremen, 2005.
24. T. Mossakowski and A. Tarlecki. Heterogeneous Specification. In preparation.
25. Object Management Group. MDA Guide Version 1.0.1. Technical report, OMG, 2003. `www.omg.org/docs/omg/03-06-01.pdf`.
26. Object Management Group. MOF 2.0 Core Specification. Technical report, OMG, 2006. `www.omg.org/cgi-bin/doc?formal/2006-01-01`.
27. I. Poernomo. The Meta-Object Facility Typed. In *Proc. SAC'06*, pages 1845–1849. ACM, 2006.
28. I. Poernomo. Proofs-as-Model-Transformations. In *Proc. ICMT'08*, volume 5063 of *LNCS*, pages 214–228. Springer, 2008.
29. A. Tarlecki. Moving between Logical Systems. In *Proc. WADT'95*, volume 1130 of *LNCS*, pages 478–502. Springer, 1996.
30. A. Tarlecki. Distributed Specifications in Heterogeneous Logical Environments. In *Proc. WADT'08*, LNCS. Springer, 2008. To appear.
31. D. Varró and A. Balogh. The Model Transformation Language of the VIATRA2 Framework. *Sci. Comp. Prog.*, 68(3):187–207, 2007.
32. H. Wehrheim. Behavioural Subtyping in Object-Oriented Specification Formalisms. Habilitationsschrift, Carl-von-Ossietzky-Universität Oldenburg, 2002.
33. M. Wirsing and A. Knapp. View Consistency in Software Development. In *Proc. Monterey'02*, volume 2941 of *LNCS*, pages 341–357. Springer, 2004.

A Model transformation: from CD to RDBS

The `uml2rdbs` model transformation obtains relational schemas that conform to the RDBS metamodel specification in Fig. 2 from class diagrams that conform to the CD metamodel specification also in Fig. 2. In particular, we consider class diagrams that contain bidirectional associations between classes, where there should be at least an association end with upper and lower bound = 1.

In MOMENT2, the notion of domain is used to refer to a model. The `uml2rdbs` transformation is defined with two domains `uml` and `rdbs` that represent the input and output model of the transformation, respectively.

```
transformation uml2rdbs ( uml : SimpleCD ; rdbs : SimpleDB) {
```

The model transformation rules are given as model equations, where there may be LHS and RHS model patterns for the `uml` and `rdbs` domains, indicating how they are manipulated in a similar way to graph transformations. We provide a brief description of each rule and its specification in the MOMENT2 model transformation language.

The `PackageToSchema` model equation generates a schema in the model of the `rdbs` domain that will contain the generated relational schema. The NAC condition avoids the recursive application of this equation in a non-terminating way.

```
  eq PackageToSchema {
    lhs uml {
      p : Package { }
    };
    lhs rdbs { };

    rhs uml {
      p : Package { }
    };
    rhs rdbs {
      s : Schema {   }
    };

    nac rdbs noSchema {
      s : Schema {  }
    };
  }
```

The `ClassToTable` model equation generates a table in the model of the `rdbs` domain for each class that is found in the `uml` domain. The new table is named with the class name. A *primary key* and its corresponding *column* are added to the generated table.

```
  eq ClassToTable {
    lhs uml{
      c : Class {
        package = p : Package {},
        name = cn
      }
    };
    lhs rdbs {
      s : Schema { }
    };

    rhs uml{
      c : Class {
```

```
      package = p : Package {},
      name = cn
    }
  };
  rhs rdbs {
    t : Table {
      schema = s : Schema { },
      name = cn,
      columns = cl : Column {
        name = cn + "_tid",
        type = "NUMBER",
        nnv = true,
        unique = true
      },
      key = k : Key {
        name = cn + "_pk",
          columns = cl : Column { }
      }
    }
  };

  nac rdbs noTable {
    t : Table {
      name = cn
    }
  };
}
```

The `AttributeToColumn` model equation generates a column in the model of the `rdbs` domain for each *attribute* of a class that is found in the `uml` domain. The column is added to the table that has been previously generated from the class that contains the attribute. The name of the column is given by the concatenation of the class name and the attribute name.

```
eq AttributeToColumn {
  lhs uml{
    c : Class {
      name = cn,
      properties = a : Attribute {
        name = an,
        type = at
      }
    }
  };
  lhs rdbs {
    t : Table {
      name = cn
    }
  };

  rhs uml{
    c : Class {
      name = cn,
      properties = a : Attribute {
        name = an,
        type = at
      }
    }
  };
  rhs rdbs {
    t : Table {
      name = cn,
      columns = cl : Column {
        name = cn + "_" + an,
        type =
          if at == "INTEGER" then
            "NUMBER"
          else
            if at == "BOOLEAN" then
              "BOOLEAN"
            else
              "VARCHAR"
            endif
          endif
      }
    }
  };

  nac rdbs noColumn {
    cl : Column {
      name = cn + "_" + an
    }
  };
}
```

The `AssociationToTable` model equation translates bidirectional associations between classes to auxiliar tables that contain *foreign keys* to the tables that correspond to the associated classes in the `uml` domain.

The name of the auxiliar table is given by the lexicographically ordered names of the involved classes. In this way, we ensure that the application of this equation will always produce the same result independently of different matches, implying confluence. In the auxiliar table, a primary key, the column that is used in the primary key, two foreign keys and their columns are generated. The *not null value* and *unique* properties of the foreign key columns are not set. This data is set in the `Multiplicity` equation as explained below.

```
eq AssociationToTable {
  lhs uml{
    ae1 : AssociationEnd {
      name = ae1name,
      owningClass = c1 : Class {
        name = c1name
      },
      opposite = ae2 : AssociationEnd {
        name = ae2name,
        owningClass = c2 : Class {
          name = c2name
        }
      }
    }
  };
  lhs rdbms {
    t1 : Table {
      name = c1name,
      key = k1 : Key { },
      schema = s : Schema { }
    }
    t2 : Table {
      name = c2name,
      key = k2 : Key { }
    }
  };

  rhs uml{
    ae1 : AssociationEnd {
      name = ae1name,
      owningClass = c1 : Class {
        name = c1name
      },
      opposite = ae2 : AssociationEnd {
        name = ae2name,
        owningClass = c2 : Class {
          name = c2name
        }
      }
    }
  };
  rhs rdbms {
    t1 : Table {
      name = c1name,
      key = k1 : Key { },
      schema = s : Schema { }
    }
    t2 : Table {
      name = c2name,
      key = k2 : Key { }
    }
    t : Table {
      name = if c1name < c2name then c1name + "_" + c2name
        else c2name + "_" + c1name endif,
      schema = s : Schema { },
      fks = fk1 : ForeignKey {
        name = ae2name + "_fk",
        refersTo = k1 : Key { },
        columns = cl2 : Column { }
      },
      columns = cl1 : Column {
        name = ae1name,
        type = "NUMBER",
        owningTable = t : Table { }
      },
      fks = fk2 : ForeignKey {
        name = ae1name + "_fk",
        refersTo = k2 : Key { },
        columns = cl1 : Column {  }
      },
      columns = cl2 : Column {
        name = ae2name,
        type = "NUMBER",
        owningTable = t : Table { }
      },
      columns = kColumn : Column {
        name = if c1name < c2name then c1name + "_" + c2name + "_tid"
          else c2name + "_" + c1name + "_tid" endif,
        type = "NUMBER",
        nnv = true,
        unique = true
      },
      key = k : Key {
        name = if c1name < c2name then c1name + "_" + c2name + "_pk"
          else c2name + "_" + c1name + "_pk" endif,
        columns = kColumn : Column { }
      }
    }
  };

  nac rdbms noColumn {
    t : Table {
      name =
        if c1name < c2name then c1name + "_" + c2name
        else c2name + "_" + c1name endif
      }
  };
}
```

The following model equation uses the multiplicity metadata of association ends to set the *not null value* and *unique* properties of the columns that are involved in the foreign keys that are generated from associations.

The multiplicity metadata of an association end, owned by a class A and typed with a class B, is used to set the column that corresponds to the foreign key that is owned by the generated auxiliary table AB and that refers to the key of the table A. If `lb` represents the lower bound of the association end, and `ub` represents its upper bound, the `nnv` and `unique` properties of the corresponding column are computed as follows:

```
eq Multiplicity {
  lhs uml{
    ae1 : AssociationEnd {
      lowerBound = lb,
      upperBound = ub,
      owningClass = c1 : Class {
        name = c1name
      },
      type = c2 : Class {
        name = c2name
      }
    }
  };
  lhs rdbms {
    fk1 : ForeignKey {
      refersTo = k1 : Key {
        owningTable = t1 : Table {
          name = c1name
        }
      },
      columns = cl1 : Column {
        name = cl1Name
      }
    }
  };

  rhs uml {
    c1 : Class {
      name = c1name
    }
    c2 : Class {
      name = c2name
    }
  };
  rhs rdbms {
    fk1 : ForeignKey {
      refersTo = k1 : Key {
        owningTable = t1 : Table {
          name = c1name
        }
      },
      columns = cl1 : Column {
        name = cl1Name,
        nnv = if (lb == 0) then false else true endif,
        unique = if (ub == 1) then true else false endif
      }
    }
  };
}
```

This transformation can be applied to the class diagram, shown in Fig. 2, that conforms to the CD metamodel specification to generate the relational schema, also shown in Fig. 2, that conforms to the RDBS metamodel specification. To apply the transformation, the *uml* domain is constituted by the input class diagram and the *rdbs* domain is constituted by an empty RDBS model. After executing the model transformation, the *rdbs* domain contains the generated relational schema.