A Unifying Semantic Framework for Multilevel Metamodeling

Dániel Varró[1,2] and András Pataricza[2]

[1] SRI International, Computer Science Laboratory
333 Ravenswood Ave., Menlo Park, CA, U.S.A.
[2] Budapest University of Technology and Economics
Dept. of Measurement and Information Systems
Magyar tudósok k. 1/D, 1117 - Budapest, Hungary
{varro, pataric}@mit.bme.hu

Abstract. As the revision process of UML attempts to re-architecture the single and imprecise language into a family of languages with well–defined semantics, the nature of the core metamodeling framework is of growing importance. In the current paper, a formal mathematical background is presented for a uniform representation of multilevel metamodels and static well–formedness constraints supported by a verification method for checking the correctness of model refinement steps. We demonstrate that the mathematical framework is rich enough to formally capture many state–of–the–art metamodeling techniques including deep instantiation, structural extension, type restriction, package inheritance and multilevel metamodeling, which makes it feasible for providing a common mathematical basis for comparing and developing metamodeling approaches of engineering interest.

1 Introduction

The term *metamodeling* usually refers to the definition of language families for various domains of interest. Such a definition must include the specification of its abstract syntax (for instance, by a metamodel), the concrete syntax (preferably by using a graphical language), and its semantics.

The Unified Modeling Language (UML) [12], which is the standard object–oriented modeling language with a wide range of application, is probably the most relevant domain of metamodeling. However, several shortcomings have been revealed concerning, especially, its imprecisely described semantics and lack of flexibility in domain-specific applications. Recent initiatives (UML 2.0 RFP) have aimed at evolving UML into a core kernel language, and a family of distinct languages (called profiles) where each profile has its own semantics, which architecture fundamentally requires an appropriate and precise metamodeling technique.

State–of–the–art metamodeling approaches typically use a meaningful subset of UML Class diagrams to specify the abstract syntax, and a constraint language to capture the static semantics with a different level of formality.

– Meta Object Facility – MOF [7] uses the Object Constraint Language for static semantics, but lacks any formality when concerning dynamic issues.

- MetaModeling Framework – MMF [3] uses a combination of OCL and Class diagrams with a clear distinction between classes and instances, a formal background originating from object theory [1].
- General Modeling Environment — GME [6] uses the same combination for static semantics but the mixing of types and instances is allowed.
- Business Object Notation – BON [9] has its own notation for both constraints and metamodels, and the concepts are formalized in PVS [5].
- BOOM [8] is a framework for formal specification of object–oriented modeling languages using the textual description language ODAL.

Unfortunately, as pointed out in [2], all the current metamodeling approaches have fundamental problems concerning their instantiation mechanism. These problems originate in the fact that all these approaches evolved from object–oriented modeling, which traditionally had only two metalevels, the object level (M_0) and the class level (M_1) where every element at the M_0 level is an instance of exactly one M_1, element. However, when a metamodeling level M_2 is introduced, the current solutions are unable to carry information concerning attributes and associations across more than one metalevel (e.g., instantiating an M_2 class on the M_0 instance level), resulting in obvious syndromes such as replication of concepts (defining each concept twice, on the class and on the instance level as well).

A possible solution is to consider the duality of classes and objects and allow classes to be instantiated on any metalevel, which method is referred to as deep instantiation in [2]. However, if deep instantiation is adopted for metamodeling, previous attempts based on traditional formal approaches are inadequate for formalizing the semantic foundations of metamodels in a satisfying way. In addition to such a formal foundation, specific guidelines must also be explicitly provided to facilitate the development of verification tools that reason about the correctness of models and their development process.

In the current paper, we propose (i) *a unifying formal mathematical background for multilevel metamodeling with deep instantiation* (Section 2), (ii) *a verification method for checking the correctness of model refinement steps* (Section 3), and (iii) *a uniform treatment of models and static structure–oriented well–formedness constraints* (Section 4). The (mathematical) expressiveness and feasibility of our proposal will be demonstrated by formalizing some state–of–the–art metamodeling techniques such as structural extension, type restriction, package inheritance, and model restriction by static constraints.

The main contribution of the paper is it that proposes (up to our knowledge) the *first formal background for multilevel metamodeling with deep instantiation* satisfying the methodological requirements described in [2]. Moreover, this mathematical framework allows a *uniform treatment* of such properties that traditionally require different underlying mathematical models (thus different tools) to be verified (including e.g., consistent view integration, correctness of refinement steps, and validity of well–formedness constraints).

2 A Uniform Representation of Metamodels

For a uniform representation of models in a multilevel metamodeling architecture, our fundamental idea is *to handle classes and instances uniformly in a virtually flat model structure without the loss of type information*. We achieve this goal by *grouping the type information of the model into the elements* that constitute the model. In this sense, the notion of metalevels becomes obsolete; however, a particular metamodeling approach based upon this theoretical framework is able to introduce these concepts explicitly.

2.1 Basic definitions and data structures

Before going into the mathematical details, we introduce a simple metamodel in Figure 1 as a running example. It describes the structure of state transition systems stating that *Transitions* may lead *from* and *to States*. On the left-hand side, the UML notation is used, in the middle, the top-level model object and implicit association inheritance is depicted explicitly, while the right-hand side is a formal representation of models to be introduced in the sequel.

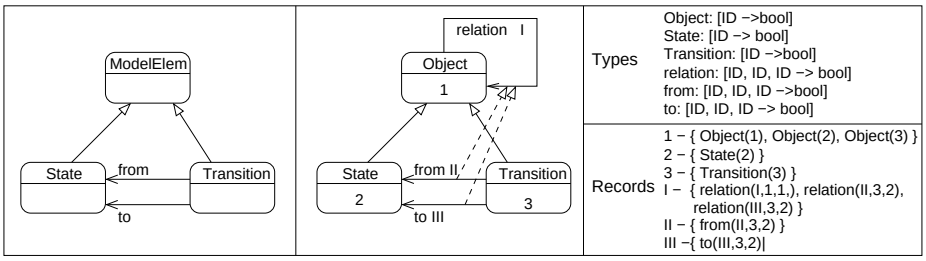


Fig. 1. A simple metamodel of state transition systems

In the following, a higher-order logic specification is used to formalize our mathematical framework.

Definition 1 (Identifier space). *The set of object identifiers is denoted by* $ObjID$*, while let* $RelID$ *be the set of relation identifiers. Let* $top\_obj \in ObjID$ *and* $top\_rel \in RelID$*.*

An object or relation identifier can be informally interpreted as a (unique) name. In our sample metamodel, arabic and roman numbers are used as identifiers of objects and relations, respectively.

Definition 2 (Type space). *Let* $ObjType \subseteq ObjID$ *be a unary relation and let* $RelType \subseteq RelID \times ObjID \times ObjID$ *be a tertiary relation.*

The type space of the running example depicted in the top right field of Fig. 1 consists of three unary (*Object, State, Transition*) and three tertiary relations (*relation, from, to*).

Definition 3 (Most general object and relation). *Let* $object \in ObjType$ *and* $relation \in RelType$ *such that the following three conditions hold.*

1. $\forall(x : ObjID) : object(x)$ *(every object identifier is an object)*
2. $\forall(r : RelID, o_1, o_2 : ObjID) : relation(r, o_1, o_2)$ *(every relation identifier is a relation)*
3. $\forall(r : ObjID, o_1, o_2 : ObjID) : relation(r, o_1, o_2) \supset object(o_1) \wedge object(o_2)$ *(the "source" and "target" of a relation is an object)*

These axioms formulate the well–formedness of the most general object and relation. One can easily check that our running example meets these axioms. Up to now, the class and instance levels were kept separated. By introducing records, which glue an identifier and its type into a single construct, the type (instantiation) information of metalevels is encapsulated into a uniform (and virtually "typeless") mathematical structure.

Definition 4 (Record space). *Let* $ObjRec$ *be a set of records with the structure* $(class : ObjType, inst : ObjID)$*, and* $RelRec$ *be a set of records of* $(class : RelType, inst : RelID, src : ObjRec, trg : ObjRec)$*. Accessors of record fields are interpreted as a corresponding function mapping from object (relations) records to the type of the corresponding field (such as, e.g.,* $class : ObjRec \rightarrow ObjType$*). However, to improve legibility, we will write* $obj.class$ *instead of the official* $class(obj)$ *notation.*

For each valid $obj \in ObjRec$ *and* $rel \in RelRec$*, the following axioms hold.*

1. $(obj.class)(obj.inst)$ *(an object record is reflective)*
2. $(rel.class)(rel.inst, (rel.src).inst, (rel.trg).inst)$ *(a relation record is reflective)*
3. $\forall(r : RelID, o_1, o_2 : ObjID) : (rel.class)(r, o_1, o_2) \supset ((rel.src).class)(o_1) \wedge ((rel.trg).class)(o_2)$ *(a relation record preserves source and target objects)*

Corollary 1. *There exists a valid object and relation record.*

1. $top\_objRec = (class := object, inst := top\_obj)$ *is a valid object record.*
2. $top\_relRec = (class := relation, inst := top\_rel, src := top\_objRec, trg := top\_objRec)$ *is a valid relation record.*

The records corresponding to our metamodel are also shown in the bottom right field of Fig 1. Obviously, all the records are well–formed with respect to our requirements.

2.2 Object and relation inheritance

Up to this point, object and relation records existed on their own. However, in a metamodel, certain relationships between the evaluations of their *class* function can easily be observed. For instance, as a *State* is supposed to be a subclass of an *Object* in our example, that fact guarantees that all the subclasses of *States* are also subclasses of *Objects*.

In the terms of our data structure, the *class* relation of *State* implies the *class* relation of *Object*. Intuitively, one can say that *State* is more detailed than *Object*, or alternatively, *Object* is more general than *State*. Similarly, the truth of the class relation of the *from* relation simultaneously implies the truth of the class relations of *Object*, *Transition*, and *State*. Below, we define the notions of *object* and *relation inheritance* to capture these aspects.

Definition 5 (Object inheritance). *A relation $\rhd_o \subseteq ObjRec \times ObjRec$ is an **object inheritance**, iff, for each pair of valid object records $(abst, ref) \in \rhd_o : \forall x : ObjID : (ref.class)(x) \supset (abst.class)(x)$ (the truth of the inherited object implies the truth of a more abstract object).*

(We use an infix notation $abst \rhd_o ref$ for inheritance relations.)

Definition 6 (Relation inheritance). *A relation $\rhd_r \subseteq RelRec \times RelRec$ is a **relation inheritance**, iff for each pair of valid relation records $(abst, ref) \in \rhd_r$:*

1. $\forall(r : RelID, o_1, o_2) : ObjID : (ref.class)(r, o_1, o_2) \supset (abst.class)(r, o_1, o_2)$ *(the truth of an inherited relation implies the truth of a more general relation)*
2. $\forall(x : ObjID) : ((ref.src).class)(x) \supset ((abst.src).class)(x)$ *(the truth of the source object of the inherited relation implies the truth of the source object of the general one)*
3. $\forall(x : ObjID) : ((ref.trg).class)(x) \supset ((abst.trg).class)(x)$ *(the truth of the target object of the inherited relation implies the truth of the target object of the general one)*

Corollary 2. *There exists a most general object (and relation) record, of which every object (relation) is an inheritance.*

1. $\forall(obj : ObjRec) : top\_objRec \rhd_o obj$
2. $\forall(rel : ObjRec) : top\_relRec \rhd_r rel$

As for our running example, the intuitive meaning of object and relation inheritance is fulfilled by the records, as for object inheritance we established (i) $State(x) \supset Object(x)$ (ii) $Transition(x) \supset Object(x)$, while for relation inheritance (i) $from(r, t, s) \supset relation(r, t, s)$ (ii) $from(r, t, s) \supset transition(t)$ (iii) $from(r, t, s) \supset state(s)$ (and the similar holds for $to(r, t, s)$).

2.3 Models

In the previous section special inheritance relationships have been established between object and relation records. However, without further constraints on object and relation records, inconsistencies might easily be introduced. Such inconsistencies include name clashes (using the same identifier for more than a single object or relation record), and records without a more general "super-record". Therefore, *models* must be restricted to avoid such problems.

Definition 7 (Model). *A **model** $M = (Objs, Rels)$ is a set $Objs$ of object records and a set $Rels$ of relation records satisfying the following conditions.*

1. $top\_objRec \in Objs$ *and* $top\_relRec \in Rels$ *(the top records are in the model)*
2. $\forall(obj_1, obj_2 \in Objs) : obj_1 \neq obj_2 \iff obj_1.inst \neq obj_2.inst$ *and*
   $\forall(rel_1, rel_2 \in Rels) : rel_1 \neq rel_2 \iff rel_1.inst \neq rel_2.inst$
   (all records are uniquely identified by their instance field)
3. $\forall(ref_o \in Objs)\ \exists(abst_o \in Objs) : abst_o \rhd_o ref_o$ *and*
   $\forall(ref_r \in Rels)\ \exists(abst_r \in Rels) : abst_r \rhd_r ref_r$
   (each object and relation record has a more general record in the model)

One can easily observe that the object and relation records in our running example constitute a well–formed model.

2.4 Model graphs

Although, in the previous sections, the concepts of models have been formalized precisely, an equivalent representation (called model graphs) will be introduced in the sequel as a canonic form. Model graphs eliminate the distinction between objects and relations, however, type information and the casual links of inheritance are preserved. After that, we show that the nodes of this graph are partially ordered by the edges, moreover, they form a lattice structure. The importance of this representation originates in the fact that it is capable of reasoning simultaneously about models and their refinements (and restrictions).

Definition 8 (Model identifiers). *The* ***object identifiers*** *of a model* $M = (Objs, Rels)$ *are defined as* $objID(M) = \{x : ObjID | \exists(obj : Objs) : x = obj.inst\}$. *The* ***relation identifiers*** *of the same model are* $relID(M) = \{x : RelID | \exists(rel : Rels) : x = rel.inst\}$. *The* ***model identifiers*** *of the model are* $modID(M) = objID(M) \cup relID(M)$.

Definition 9 (Model graph). *Let* $M = (Objs, Rels)$ *be a model. A* ***model graph*** $MG = (Nodes, Edges)$ *is a graph where*

- *A* ***model node*** $n \in Nodes$ *is either an object or a relation identifier:* $n : modID(M)$.
- *A* ***model edge*** $e \in Edges$ *(that can be interpreted informally as implications) leads from node* n_1 *to node* n_2 *(denoted as* $n_1 \overset{e}{\mapsto} n_2$*, or simply* $n_1 \mapsto n_2$*) when*
  1. *Both* n_1 *and* n_2 *are node identifiers, and the class predicate of* n_1 *implies the class predicate of* n_2 *i.e.* $\exists(obj_1, obj_2 : Objs) : n_1 = obj_1.inst \wedge n_2 = obj_2.inst \wedge \forall(x : objID(M)) : obj_1.class(x) \supset obj_2.class(x)$ *(in other words,* obj_1 *is derived by object inheritance from* obj_2*)*
  2. *Both* n_1 *and* n_2 *are edge identifiers, and the class predicate of* n_1 *implies the class predicate of* n_2 *i.e.* $\exists(rel_1, rel_2 : Rels) : n_1 = rel_1.inst \wedge n_2 = rel_2.inst \wedge \forall(r : relID(M), s, t : objID(M)) : rel_1.class(r, s, t) \supset rel_2.class(r, s, t)$
  3. n_1 *is an edge identifier and* n_2 *is a node identifier, and the class predicate of* n_1 *implies the class predicate of* n_2 *i.e.* $\exists(rel : Rels, srcObj : Objs) : n_1 = rel.inst \wedge n_2 = srcObj.inst \wedge \forall(r : relID(M), s, t : objID(M)) : rel.class(r, s, t) \supset srcObj.class(s)$

4. *n_1 is an edge identifier and n_2 is a node identifier, and the class predicate of n_1 implies the class predicate of n_2 i.e.* $\exists(rel : Rels, trgObj : Objs) : n_1 = rel.inst \wedge n_2 = srcObj.inst \wedge \forall(r : relID(M), s, t : objID(M)) : rel.class(r, s, t) \supset srcObj.class(t)$

– ***Extensions****: Let n_0 be a node that stands for an imaginary unique identifier, and that is linked to all the nodes having only outgoing edges.*

Informally, an object node has an outgoing edge from all its "super" object nodes (generalized nodes), while a relation node has an outgoing edge to all its "super" edges, plus all the "super"objects of its source and target object node. The truth of a class predicate is restricted to those elements that constitute the model, which closes the model space (by disregarding objects and relations outside the model).

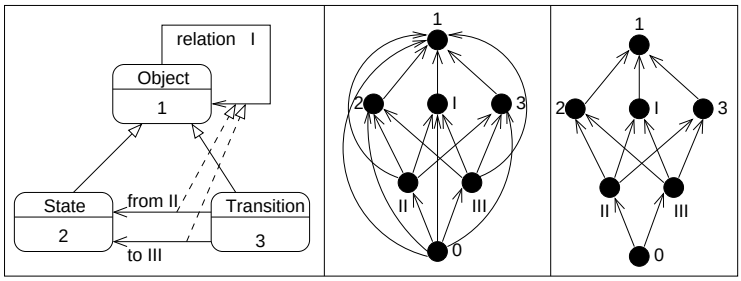


Fig. 2. Model graph and model lattice of transition systems

In Fig. 2, the model graph of the running example can be observed. The figure in the middle explicitly depicts all the edges of the model graph. The figure in the right depicts (which will be our standard notation for the rest of the paper to improve clarity) only the *direct* implication edges. The entire set of edges can be calculated by the transitive (and reflexive) closure of the implicitly depicted edges.

2.5 The lattice of a single model graph

In the following, we prove that a model graph forms a lattice. Let $MG = (Nodes, \mapsto)$ be a model graph.

Proposition 1. *$\mapsto$ is a partial order ($\sqsubseteq_N$) on $Nodes$.*

Proof. The sketch of the proof is as follows

1. $\mapsto$ is reflective: a trivial consequence of (1) and (2) in the definition of edges.
2. $\mapsto$ is transitive: (i) the implication relation is transitive, (ii) the construction of the model graph ensures that object nodes never supersede relation nodes
3. $\mapsto$ is antisymmetric: all the identifiers are unique in the model □

Least upper bound The least upper bound $\sqcup_N$ of a set $N \subseteq Nodes$ is the least node that is implied by all elements $n \in N$. This can be determined by, e.g., a reachability analysis: (i) the top node is set to the singleton *current set* as it is an upper bound (reachable) from all the nodes $n \in N$; (ii) all the predecessors of all nodes in the current set are tested whether they are still an upper bound of N. Those that satisfy this condition become the next *current set*, and this process is iterated until a fixpoint is reached. Also note that *top_obj* is the supremum ($\top_N$) of $Nodes$. Formally,

Proposition 2. *The set $N \subseteq Nodes$ has a supremum $\top_N = top\_obj$ and the least upper bound $\sqcup_N N = u \iff \forall n \in N : n \mapsto u \wedge \forall u' \in N : u \mapsto u'$.*

Greatest lower bound The calculation of the greatest lower bound $\sqcap_N$ is similar. This time one should start from the fictive n_0 element and perform the dual steps. Similarly, n_0 turns out to be the infimum of $Nodes$.

Proposition 3. *The set $N \subseteq Nodes$ has an infimum $\bot_N = n_0$ and the greatest lower bound $sqcap_N N = u \iff \forall n \in N : u \mapsto n \wedge \forall u' \in N : u' \mapsto u$.*

Proposition 4. *$\sqsubseteq_N = \mapsto$ is a lattice $L_N = (\sqsubseteq_N, \bot_N, \top_N, \sqcup_N, \sqcap_N)$ on a model graph $MG = (Nodes, \mapsto)$.*

3 Capturing Model Refinement

The lattice of a model graph is a precise means to reason about the well–formedness of a single model. In the following, we introduce the lattice representing the *set* of model graphs to provide a verification mechanism for the evolution of models, e.g., to decide whether a specific model is a proper refinement of another one. Moreover, lower and upper bound operations provide means to integrate different versions or different aspects of a model (e.g., created by different designers) into a safe and consistent global viewpoint of the system.

3.1 The lattice of the sets of model graphs

Definition 10 (Directed graph). *A **directed graph** $G = (Nodes, Edges, src^G, trg^G)$ consists of a finite set $Nodes$ of **nodes**, a finite set E_G of **edges**, two **mappings** src^G and trg^G assigning the source and the target node, respectively, to each edge.*

Definition 11 (Subgraph relation). *Let $\sqsubseteq$ be the traditional subgraph relation on graphs, i.e. $G_1 \sqsubseteq G_2 \iff G_1.nodes \subseteq g_2.nodes \wedge G_1.edges \subseteq g_2.edges$ and G_1 is a well–formed graph (i.e., the mappings relating the edges to their sources and targets coincide with the respective mappings of G_2 restricted to G_1).*

Proposition 5. *The $\sqsubseteq$ relation is a partial order on the set $S(MG)$ of model graphs (well–known result from graph theory).*

Greatest lower bound When we try to maintain the property that any operation performed on model lattices is a model lattice, the calculation of greatest lower bound and least upper bound of a subset $X \subseteq S(MG)$ needs some precautions.

The calculation of the *greatest lower bound* $\sqcap X$ of a subset $X \subseteq S(MG)$ *takes the intersection of all the lattices* in X, and the *result is expected to be a common abstract model* of all the models in X. For that reason, at most those nodes and edges are included in the result lattice that appear in all $x \in X$ lattices.

However, different models that share the same nodes may be in a confliction. Suppose that there is a model lattice M_1 where $n_1 \mapsto n_2$ but in another model lattice M_2, it is just the opposite $n_2 \mapsto n_1$. Since $\mapsto$ is a partial order, $n_1 \mapsto n_2 \wedge n_2 \mapsto n_1 \supset n_1 = n_2$, which also fulfills our expectations, as both configurations can be refined from an abstract model where the two nodes have not been distinguished yet. As there are no refined models where these conditions could also be satisfied, we showed that this method derives the greatest lower bound of X. Note that the model graph consisting of the single top_obj node is the infimum of the entire set $S(MG)$. Thus we have

Proposition 6. *The set $S(MG$ of model graphs has an infimum $\bot_G$ (which is the graph consisting of the single node top_obj), and a greatest lower bound $\sqcap_G X = M(Nodes, Edges)$ such that*

1. $\forall n \in Nodes : n \in x_1.Nodes \wedge \ldots x_n.Nodes, x_1, \ldots, x_n \in X$ *(all the common nodes)*
2. $\forall e \in Edges : e \in x_1.Edges \wedge \ldots \wedge x_n.Edges, x_1, \ldots, x_n \in X \wedge e.from \in Nodes \wedge e.to \in Nodes$ *(all the common edges)*
3. $\forall n_1, n_2 \in Nodes : n_1 \mapsto n_2 \wedge n_2 \mapsto n_1 \supset n_1 = n_2$ *(conflicting nodes are reduced to a single node as a consequence of $\mapsto$ being a partial order)*

Corollary 3. *For all $G_1, G_2 \in S(MG) : G_1 \sqcap_G G_2 \sqsubseteq G_1; G_1 \sqcap G_2 \sqsubseteq G_2$.*

Least upper bound When calculating the *least upper bound* of some $X \subseteq S(MG)$, one has to *take the union of all the graphs $x \in X$*, and, according to our informal expectations, this *union should be a refinement of all individual model graphs*. Unfortunately, in the case of contradicting models, this property cannot be established.

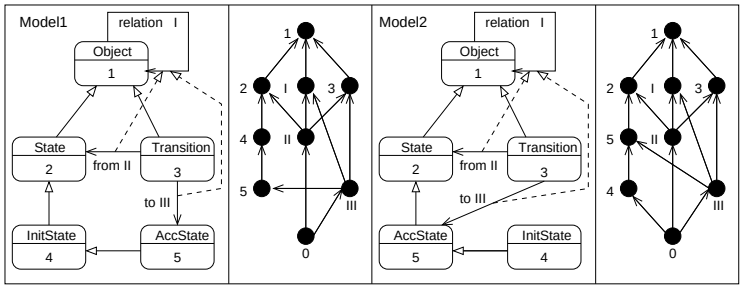
When taking the union of graphs $x \in X$, a merging is required along the nodes that appear in more than a single model. After that, the calculation of the union of the edges adds an edge from node n_i to node n_j if there is at least one model x_k with such an edge but there are no models with an edge leading to the opposite direction (i.e., from n_j to n_i). In the latter case, no edges are established in the result lattice between n_i and n_j in accordance with the properties of implication $n_i \mapsto n_j \vee n_j \mapsto n_i = \top$.

When a least upper bound of the set X is not a refinement of one or more models, one can easily return to a consistent global state in the past by taking the greatest lower bound of X, which is inevitably a proper abstraction, and the refinement process can be redone.

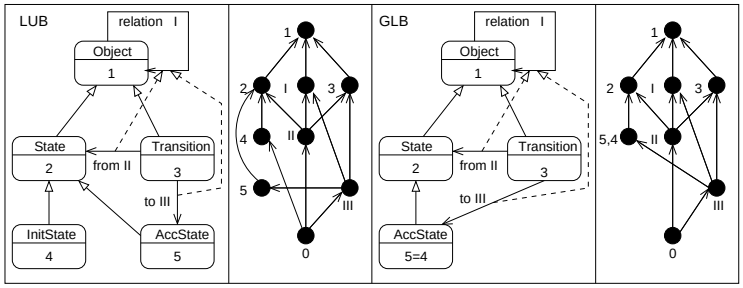
Proposition 7. *The set $S(MG)$ of model graphs has an supremum $\top_G$ and a least upper bound $\sqcup_G X = M(Nodes, Edges)$ such that*

1. $\forall n \in Nodes, x_1, \ldots, x_n \in X : n \in x_1.Nodes \vee \ldots \vee x_n.Nodes$ *(all the common nodes)*
2. $\forall e \in Edges, x_1, \ldots, x_n \in X : e \in x_1.Edges \vee \ldots \vee x_n.Edges \wedge e.from \in Nodes \wedge e.to \in Nodes$ *(all the common edges)*
3. $\forall n_1, n_2 \in Nodes : \neg(n_1 \mapsto n_2 \wedge n_2 \mapsto n_1)$ *(conflicting edges are removed as a consequence of $\mapsto$ being a partial order)*

In Fig. 3 the greatest lower bound and least upper bound of a set consisting of two models (introducing accepting and initial states in a deliberately contradicting way) is calculated. For better understanding, the visual representations of the models are also depicted.



(a) Two contradictory models



(b) The least upper bound and greatest lower bound

Fig. 3. Calculating upper and lower bounds

– In the case of the least upper bound (lub), the examination of the model lattice detects that nodes 4 and 5 are contradicting. For this reason, the lub supposes that both were introduced on purpose and keeps both of them, but the ordering relation between them is removed. As a result, we have an *AccState* and *InitState* derived by object inheritance from *State*, which is not a refinement of the two models.
– In the case of the greatest lower bound (glb), the examination of the model lattice detects that nodes 4 and 5 are equal. For this reason, they are treated as if all the edges entering one of them ends in this common state. As a result, we have an *AccState* derived by object inheritance from *State* but *InitState* must be reintroduced later in the design.

Proposition 8. *The subgraph relation is a lattice* $L_G = (\sqsubseteq_G, \bot_G, \top_G, \sqcup_G, \sqcap_G)$ *on the set* $S(MG)$ *of model lattices.*

Although we showed that the lattice structure of a single model is preserved when calculating lower and upper bounds of sets of model graphs, the result might be irregular at the diagram level. For instance, if a relation is redirected from one object to another, the intersection of the two models might contain only the edge between the source object and the relation while the target of the relation is not specified. However, this situation can easily be detected by comparing the degrees of each relation node with its original model lattice. When taking the union of model lattices, a relation node may have additional outgoing edges, which fact can be detected similarly within the semantic domain.

The final definition captures the notion of model refinement in accordance with the previous results.

Definition 12. *Let* $M_1, M_2 \in S(MG)$*. If* $M_1 \sqsubseteq_G M_2$ *then* M_1 *is an* ***abstraction*** *of* M_2*, or conversely saying,* M_2 *is a* ***refinement*** *of* M_1*.*

3.2 Practical uses of model refinement

We demonstrate how some major concepts of object–oriented metamodeling can be captured uniformly by model refinement on our running example (see Fig. 4).

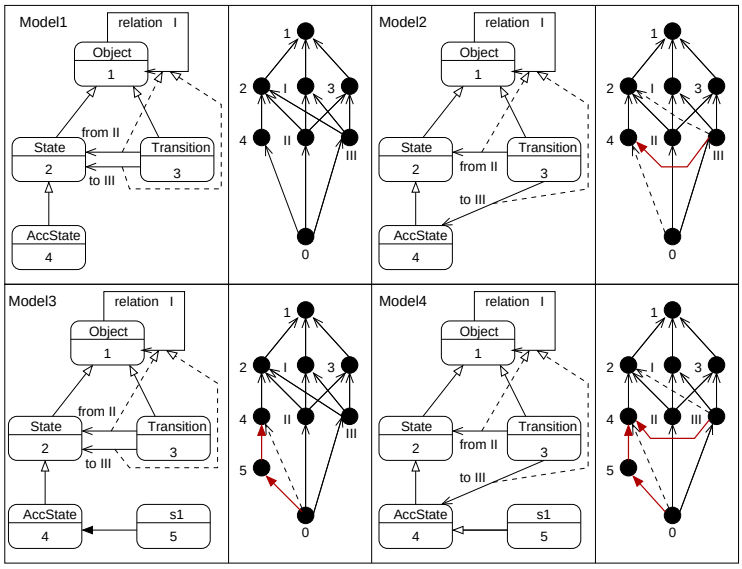


Fig. 4. Structural extension, relation restriction, instantiation

Structural extension In a structural extension, a new model element (object or relation) is added to the model by refining an existing model element (object or relation). For instance, a new class *AccState* has been derived by inheritance to the original diagram of Fig. 1 from the existing object *State*. Although this time the designer can be quite certain that the refinement step he or she performed is correct, this can be verified formally on the semantic level by comparing the new lattice $Model_1$ by the one of Fig 2 ($Model_0$). One can conclude that the refinement step is correct as $Model_0 \sqsubseteq_G Model_1$ because node 4 and its incoming and outgoing edges were introduced by the refining operation.

Moreover, altering a model by the following set of operations turns out to be formally a model refinement in our sense (not proven here).

- Inserting an object record (e.g., a UML class) consistently as a leaf element or between two existing object records (refining the object inheritance tree).
- Inserting a relation record (e.g., a UML association) consistently as a leaf element or between two existing relation records (refining the relation inheritance tree).
- Introducing multiple inheritance for both object and relation records (when the inheritance structure is no longer a tree but a directed acyclic graph).

On the other hand, several incorrectness properties (such as circularity in the inheritance structure, or invalid type refinements) can be detected on the model lattice. In fact, they are never introduced if the model refinement process is guided by the $\sqcap_G$ and $\sqcup_G$ operations.

Type restriction In a type restriction, our model is not extended but altered by redirecting the refinement of an object or a relation. This could possibly mean that

- An object inheritance relation is established between two object records having previously the same parent in the inheritance structure. However, note that the inheritance of relation records cannot always be redirected in this way as the proper inheritance of sources and targets may not be assured.
- The source (target) of a relation record is redirected to a subclass of its former source (target) object record.

Our running example covers the latter case when $Model_2$ is generated from $Model_1$ (and, additionally, when $Model_4$ is derived from $Model_3$). In Fig. 4, the dashed lines can be derived by the transitive closure of solid lines; however, they are depicted to improve clarity of the subgraph relation when verifying that $Model_2$ is a proper refinement of $Model_1$.

Instantiation Up to this point, the diagrams contained only the relationship on the class level. Now, when deriving $Model_3$ from $Model_1$, an instance *s1* of class *AccState* is introduced by inheritance. This example demonstrates that classes and instances can be treated uniformly by the lattice structure: an instance is another node of the model lattice derived from its class by inheritance. In this way, an instance is considered to be a singleton set, and not an element of a set, and the instantiation step from $Model_1$ to $Model_3$ (and, similarly, from $Model_2$ to $Model_4$) is interpreted as an *object inheritance*.

On the other hand, the uniform model lattice also allows a particular metamodeling approach to distinguish between classes and instances by introducing instantiation as a *relation* and thus deriving instances by *relation inheritance*.

Package inheritance In the MML approach [3], specialization (inheritance) is also interpreted for packages, which allows a modular construction and the reuse of information between such packages (models). The child package specializes all (and therefore contains all) the contents of the parent package, i.e., each class and association is a specialization of the corresponding concept in the parent package, where the correspondence is defined by a renaming annotation on the specialization arrow (illustrated in Fig. 5).

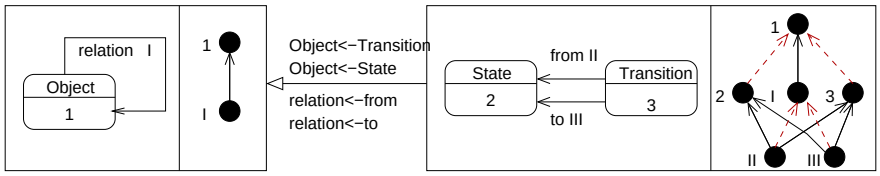


Fig. 5. Package specialization

Such a package inheritance (together with the renaming information on the object and relation inheritance structure) can easily be established on the model lattice by model refinement (i.e., calculating the least upper bound and adding the edges specified in the renaming annotation). These edges are depicted by dashed lines in Fig. 5.

4 Model Restriction as a Constraint Language

A typical metamodeling approach provides a mechanism for *refining* a model (e.g., by inheritance, or type restriction) *and* a constraint language for expressing certain *restrictions* on valid model instances. Moreover, a model is obliged to fulfill all its static constraints for the entire life-cycle. In this section, constraints are introduced as distinct models that are specially related to the model to be restricted, therefore, a separate constraint language is not required.

Informally speaking, a constraint is regarded as a model pattern that is obliged (or forbidden) to be found in a model instance. In fact, the pattern-based manipulation of graphs has been studied thoroughly by the theory of graph transformation [11]. Moreover, several papers (e.g., [10]) address the use of graph transformation as a semantic basis for the Object Constraint Language. In the current paper, the emphasis is on expressing structural constraints by special relations on models, thus without the use of a *separate* constraint language.

Definition 13. *A **graph isomorphism*** $gi : G \to H$ *is a pair of bijective functions* $\langle f_N : G.nodes \to H.nodes, f_E : G.edges \to H.edges \rangle$ *compatible with source and*

target functions, i.e., for all edges $e \in G.edges$, $f_N(src^G(e)) = src^H(f_E(e))$ *and* $f_N(trg^G(e)) = trg^H(f_E(e))$.

A ***subgraph isomorphism*** *is an isomorphism between* G *and a subgraph* H_1 *of* H*:* $H_1 = gi(G) \subseteq H$.

Definition 14. *Let* M *and* M_C *be a model.*

- M *fulfills the* ***positive existential constraint*** *of* M_C *if there exists a graph isomorphism between* M_C *and a subgraph of* M.
- M *fulfills the* ***negative existential constraint*** *of* M_C *if a graph isomorphism cannot be established between* M_C *and any subgraph of* M.

Definition 15. *Let* M *be a model,* $M_C^1, \ldots, M_C^n$ *be the set of its (positive and negative) constraints. The model* M *is* ***syntactically correct*** *if it fulfills all of its constraints. The* ***refinement process*** *of a model is* ***correct*** *if all the individual steps fulfill all the constraints.*

Probably the main advantage of handling constraints and models uniformly is the increase in reusability, i.e., model refinement and restriction techniques can be applied to constraints as well.

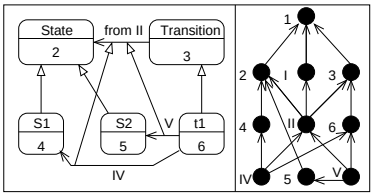


Fig. 6. Static constraints as a model

Finally, model constraints are illustrated in Fig. 6, where the constraint specifies the condition that a transition instance *t1* is connected to two states *s1* and *s2* by a *from* relation. In fact, this constraint can be either (i) a positive one stating that each transition is obliged to have (at least) two states linked by a from edge, or (ii) a negative one, which forbids the presence of the same pattern, thus allowing the user model to have less than two states linked to specific transition.

5 Conclusion and Future Work

We have presented a formal mathematical background as a potential candidate for expressing the semantic foundations of multilevel metamodeling approaches with deep instantiation. The proposed mathematical framework is unifying in the sense that models and their static well–formedness constraints, and the correctness of refinement steps, can be investigated on the same representation.

Although the treatment of dynamic semantics was out of the scope of this paper, we would like to point out that (i) our approach does not rule out any kind of definitions for dynamic semantics (such as trace, operational, denotational or axiomatic semantics); moreover, (ii) by the proposed lattice structure, abstract interpretation [4] and symbolic analysis techniques that have been widely applied for the verification of systems for several decades become accessible for the formal verification of metamodels.

An initial version of the mathematical framework of metamodels has already been formulated in the higher-order logic specification language of the PVS theorem prover

[5]. We are planning to improve the current version by proving the correctness of certain refinement operations (e.g., package inheritance, structural extension), and to reason about metamodels by deriving them automatically from some visual formalisms.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. C. Atkinson and T. Kühne. The essence of multilevel metamodelling. In M. Gogolla and C. Kobryn, editors, *Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools*, volume 2185 of *LNCS*, pages 19–33. Springer, 2001.
3. T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation semantics for UML. In H. Hussmann, editor, *Proc. Fundamental Approaches to Software Engineering, FASE 2001 Genova, Italy*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
5. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, Apr. 1995.
6. A. Ledeczi., M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Proc. Workshop on Intelligent Signal Processing*, 2001.
7. Object Management Group. *Meta Object Facility Version 1.3*, September 1999. http://www.omg.org.
8. G. Övergaard. Formal specification of object-oriented meta-modelling. In T. Maibaum, editor, *Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany*, volume 1783 of *LNCS*. Springer, 2000.
9. R. Paige and J. Ostroff. Metamodelling and conformance checking with PVS. In H. Hussmann, editor, *Proc. Fundamental Approaches to Software Engineering, FASE 2001 Genova, Italy*, volume 2029 of *LNCS*, pages 2–16. Springer, 2001.
10. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.
11. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1: Foundations. World Scientific, 1997.
12. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.