Partial Evaluation in Meta Modeling

Manfred A. Jeusfeld
Tilburg University, Department of Information Systems and Management,
Warandelaan 2, 5037 AB Tilburg, The Netherlands
jeusfeld@uvt.nl
WWW home page: http://infolab.uvt.nl/~jeusfeld

Abstract. Meta modeling is a well-established technique to describe the structure modeling languages. Method engineering environments utilize the technique to provide a flexible environment for defining and adapting modeling environments. We show that basing meta modeling strictly on first-order logic provides not only clean semantics but also the ability to define high-level constructs such as transitivity at the meta model, or even meta meta model level and to efficiently map the constructs to lower levels by partial evaluation. We show that it applies both to universally and existentially quantified expressions. Examples are included to demonstrate the usefulness. A full implementation is available in the ConceptBase meta modeling environment.

1 Introduction

A model is a structured representation of statements about some world, be it real or imagined. A meta model is a model about models, i.e. it contains some statements about some set of models, in particular models that conform to the same modeling language.

Typically, meta models are denoted in a style similar to models. Graphical notations are dominant providing rather few features to encode the desired meaning of constructs in the meta model. In this paper, we are concerned about extending the usefulness of meta models by enriching them with first -order logical expressions. Such expressions can both be used for defining some syntax rules and for defining the logic-based semantics of the use of the constructs in models conforming to the meta models.

This paper shall first recap the use of models and meta models as inspired by the model-driven architecture. The analogy of instantiation to variable substitution allows for a simple partial evaluation technique borrowed from deductive databases that translates high-level logical expressions, i.e. expressions ranging over objects at more than two meta modeling layers, to lower-level expressions down to simple expressions that relate just two meta modeling layers.

We demonstrate some generic examples to show its applicability to various meta modeling scenarios, e.g. the definition of required attributes and transitivity of attributes, which can be used to specify the semantics of the PartOf concept as well as the IsA concept. Throughout the paper, we assume Herbrand interpretations of the logical formulas. Even more, we restrict ourselves to those first order formulas that can be translated to Datalog with negation, i.e. to logical theories that have a unique minimal Herbrand interpretation. The technique has been implemented in the ConceptBase system (Jarke et al., 1995).

We claim that the incorporation of a sound meta modeling component is essential for method engineering, in particular in cases where dedicated modeling languages have to be constructed.

2 Meta Modeling Layers

The OMG meta object facility MOF [OMG 2006] organizes expressions in models with respect to their abstraction level. The lowest level M0 contains expressions that are such concrete that they do not have examples. They are representations of *examples* or *example objects*. The next level M1 contains expressions that classify or constrain the expressions at the M0 level. Expressions at the M1 level are also called *classes*. The M2 level organizes the classes of the M1 level into so-called meta classes. Meta classes are used to make statements about classes and we associate the term meta model to this level. Finally, the M3 level classifies meta classes into meta meta classes (or meta models). From a formal language point of view, the M2 level contains definitions of modeling languages, and the M3 level contains facilities to define modeling languages. The layer hierarchy can in principle continue to M4, M5 etc. but these levels are rarely used in the literature. Apparently four abstraction levels are regarded as sufficient by most authors.

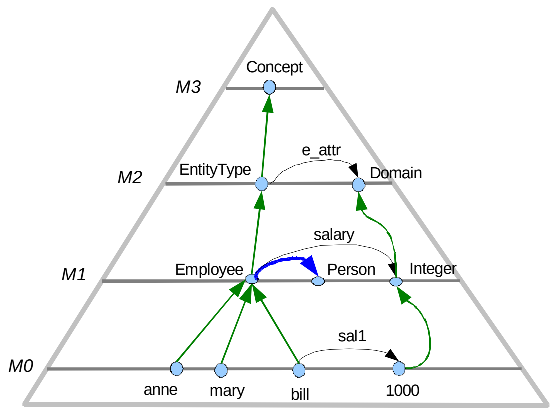

Figure 1: An interpretation of the MOF abstraction levels

Figure 1 motivates the four MOF layers. The triangular display symbolizes the expectation that the number of concepts decreases with the layer index. Intuitively, an M1 model (e.g. an entity-relationship diagram) has less elements than a M0 object model that conforms to it (e.g. the tuples of a database). This numerical relation continues with the other layers. Since each layer constrains the subsequent layer, the scope of an expression increases with the layer at which it is defined. For example, if we define the meaning of ERD cardinality constraints at the M2 layer, then it will range in principal over all databases that are conforming to some ERD model at the M1 layer.

Our goal is to allow for the efficient management of logical expressions at any MOF abstraction layer. The higher the degree of abstraction, the higher will be the degree of reuse of the expression and ultimately the more efficient will be the design of new modeling languages.

3 Models and Logic

The higher a concept is located in the MOF abstraction layers, the more abstract is also its meaning. For reasons of simplicity, we will interpret all objects at any abstraction level by itself, i.e. we assume a Herbrand interpretation where each object as shown in figure 1 is a constant being its own interpretation.

Rather than defining a predicate c(x) to denote that object x is an instance of concept c, we introduce a binary predicate In(x,c) . In figure 1, the instantiations are displayed in green color. They correspond to the following facts being a possible Herbrand interpretation of the predicate In:

```
{In(anne,Employee),
In(mary,Employee), In(1000,Integer),
In(Employee,EntityType),
In(Integer,Domain), In(EntityType,Concept),...}
```

We introduce two further predicates Isa(c,d) for declaring c as specialization of d, and AL(x,m,n,y) for declaring an attribution link (x has an attribute labelled n to y and this attribute has the category m). Applied to the example of figure 1, we would get the following Herbrand interpretation of these two predicates

```
{Isa(Employee,Person),
AL(bill,salary,sal1,1000),
AL(Employee,e_attr,salary,Integer),
AL(EntityType,attribute,e_attr,Domain)}
```

The three predicates are capable to represent concepts at *any abstraction layer* and are the basis for defining the meaning of abstract features such as inheritance, transitivity of partOf, cardinality constraints, and so forth.

To represent the meaning of concepts, we need to refer to attributes as being concepts. This is also called *reification*. If x is concept and n is the label of some attribute of x, then x!n is the constant denoting the attribute as a concept. In our running example, we have the attribute concepts `bill!sal1`, `Employee!salary`, `EntityType!ent_attr`. Like any other concept, attribute concepts can occur in the In, Isa, and AL predicates. For example `In(bill!sal1,Employee!salary)` states that the sal1 attribute of bill is an instance of the salary attribute of Employee. If o is an attribute, then the predicate P(o,x,l,y) returns the source x, the label l and the destination y of the attribute o. The following examples are true:

```
P(Employee!salary,Employee,salary,Integer)
P(bill!sal1,bill,sal1,1000)
```

Let us recall some implications of basing the semantics on Herbrand models. We demand that any model is finite we also have a finite number of constants. This is an important restriction. While finiteness for M1, M2 and M3 models is intuitive, the M0 layer might be regarded as infinite, e.g. containing all *possible* database states. For our purpose however, we strictly demand finiteness. It implies that any Herbrand interpretation of the three base predicates is also finite.

First order logic can be used to provide additional information about the concepts in a model. They are statements over the three base predicates. For example, we can express that each employee must have a salary:

[Formula 1] `∀ e In(e,Employee) ⇒ ∃ s,n In(s,Integer) ∧ AL(e,salary,n,s)`

At the M2 level, we can demand that any entity type has at least one entity attribute:

[Formula 2] `∀ et In(et,EntityType) ⇒ ∃ t,n In(t,Domain) ∧ AL(et,e_attr,n,t)`

The first formula contains constants from the M1 level and the variables are substitutable by constants from the M0 layer. The second formula contains constants from the M2 layer and the variable range over constants from the M1 layer. We call such formulas type 1 formulas since they relate two neighbouring abstractions layers. Apparently, both formulas have the same structure. Instead of copying the same formula code, we aim for a facility were we only code a *meta formula* once and re-use it wherever required. There more of such formulas are defined, the richer is the meta modeling environment since the meaning of modeling constructs can be recombined from the meta formulas.

Definition 1: A variable occurrence x1 in a predicate P is called a *meta variable* iff P=In(x,x1), or P=AL(x,x1,n,y). A *meta formula* is a first order formula with at least one meta variable.

There are plenty of examples for meta formulas. We use the following example for discussing the method (meta variables are c,d and m):

[Formula 3] `∀ x,a,c,m,d In(a,required) ∧ P(a,c,m,d) ∧ In(x,c) ⇒ ∃ y,n In(y,d) ∧ AL(x,m,n,y)`

4 Partial Evaluation of Meta Formulas

A meta formula has no peculiar property except that it has constants and variables ranging over more than 2 abstraction layers. To understand this, we show how a meta formula can be compiled to a type 1 formula by means of partial evaluation. There are a few reasons why partial evaluation is useful for meta modeling:

1. By translating a meta formula to a type 1 formula, one can understand its meaning in terms of the context in which it is used. For example, the formulas 1 and 2 are partially evaluated from formula 3. They are more understandable than formula 3 because they use constants from the appropriate abstraction layer.
2. The partially evaluated formulas have less variables than the corresponding meta formula. Since the computational complexity grows exponentially in the number of variables, the partially evaluated formulas are more efficient to evaluate.
3. View maintenance on the basis of meta formulas is virtually intractable since the predicate occurrences In(x,c) will match facts of any model base update. An important example of view maintenance is integrity checking.

The last reason is the most relevant one: if we want to efficiently check the integrity of a set of models in an incremental way, then we have to restrict to type 1 formulas.

Our partial evaluation technique is inspired by the ’simplification’ method for deductive integrity checking. The simplification method generates from an update and a formula that matches some facts in the update a new formula. The matching binds variables to constants. In our case, the new formula is not just evaluated against the database but it becomes part of the logical theory that represents our model base. Assume that MF is a meta formula and C is the list of some meta variables in MF.

Step 1: Rearrange MF into one of the two possible normalized forms (called input formula subsequently)

```
∀ C E(C) ⇒ F(C)
∃ C E(C) ∧ F(C)
```

where E(C) is a predicate and F(C) is the rest of the formula. It is allowed to define auxiliary deductive rules for E(C) in order to match one of the two forms.

The normalized forms are ensuring that the meta variables in C are restricted to those values V for which E(V) is in the interpretation of the E-predicate. The syntactic form is now as ’range-restricted’ or ’domain-independent’ in deductive database literature (Nicolas, 1979; Bry, 1989). To continue the example, step 1 rearranges formula 3 to

[Formula 4] `∃ c,d E1(c,d) ⇒ (∀ x In(x,c) ⇒ ∃ y,n In(y,d) ∧ AL(x,m,n,y))`

with the auxiliary deductive rule

```
∀ a,c,m,d In(a,required) ∧ P(a,c,m,d) ⇒ E1(c,d,m)
```

In the next step, we compute the interpretation of the E-predicate (also called the extension). The goal is to replace the E-predicate by its extension.

Step 2: Compute the Herbrand interpretation for the E-predicate, say I_E={E(V_1),E(V_2),..,E(V_K)} and replace the predicate E(C) in the normalized meta formula by the disjunction ((C=V_1) ∨ (C=V_2) ∨ ... ∨ (C=V_K)).

The Herbrand interpretation shall be finite because the base predicates are finite. If C has more than one variable, then C=V is a the pairwise equality of variables in C with values in V, i.e. ((c1=v1) ∧ (c2=v2) ∧ ...).

Example: Assume that `In(EntityType!e_attr,required)` is true. Then, `E1(EntityType,Domain,e_attr)` becomes derivable via the auxiliary deductive rule. As a consequence, the partially evaluated formula is:

```
[Formula 5] ∀ c,d ((c=EntityType ∧ (d=Domain) ∧ (m=e_attr) ⇒ ∀ x
(In(x,c) ⇒ ∃ y,n In(y,d) ∧ AL(x,m,n,y))
```

If the meta formula is universally quantified, then each entry (C=V) in the value disjunction leads to a substituted subformula F(C)[V/C], i.e. the formula F(C) where all occurences of variables of C are replaces by the corresponding values of V.

Lemma 1: If the input formula is a universally quantified meta formula, then the conjunction F(C)[V1/C] ∧ F(C)[V2/C] ∧ ... ∧ F(C)[VK/C] of all such substituted subformulas is equivalent to it.

Proof: The follows directly from the fact that ∀ x (x=v) ⇒ F(x) is equivalent to F(v) and the finiteness assumption.

Example: Let I_{E1}=`{E1(EntityType, Domain,e_attr), E1(Employee, Integer,salary)}`. By lemma 1, formula 4 is equivalent to the conjunction

```
[Formula 6]
∀ x In(x,EntityType) ⇒ ∃ y,n In(y,Domain) ∧ AL(x,e_attr,n,y))
∧
∀ x In(x,Employee) ⇒ ∃ y,n In(y,Integer) ∧ AL(x,salary,n,y))
```

Except variable naming, these two formulas are exactly formulas 1 and 2 of our initial example!

Lemma 2: If the input formula is an existentially quantified meta formula, then the disjunction F(C)[V1/C] ∨ F(C)[V2/C] ∨ ... ∨ F(C)[VK/C] of all such substituted subformulas is equivalent to it.

The proof is analogous to lemma 1. Not any meta formulas can be partially evaluated. Some meta formulas can simply not transformed into the normalized form of step 1. An example is

```
∃ c ∀ x In(x,c)
```

Step 3: Generate the target formula as specified in Lemma 1 (universal quantification) and Lemma 2 (existential quantification).

Steps 1 to 3 constitute a term rewriting system where a meta formula MF is transformed to a representation with less meta variables. Note that the rewriting is also applicable to sub-formulas of a meta formula MF. As noted above, the term rewriting system is not complete, i.e. there are meta formulas that can't be rewritten. If a meta formula is range-restricted, then there is always a rewriting to a formula without meta variables (proof pending).

5 Complexity Considerations

The above method has been implemented in the ConceptBase system. The crucial problem is the transformation in step 1, i.e. the selection of the E-predicate. In general, there is more than one candidate. So the question is, which candidate is the best one. We realized a strategy where the candidate is chosen that binds the maximum number of meta variables.

A second criterion in the selection is the size of the interpretation I_E of the meta predicate. The larger the size, the more subformulas `F(C)[V/C]` will be generated. One can easily think of scenarios where the number of generated subformulas grows to the size of the model base itself. In such a case, partial evaluation is intractable. We have to demand that the abstraction layers of figure 1 are indeed *decreasing in size*, i.e. layer 0 has many more objects than layer 1, layer 1 has many more objects than layer 2, etc. Since the concepts become more and more abstract, this is true in most meta modeling scenarios, in particular in the scenario of specifying modeling languages (layer 2). If one has a large number of concepts in layer 2 (e.g. an elaborated ontology of concepts occurring in information systems development), and only few concepts in layer 1 and 0, then it makes less sense to apply the technique. It would be analogous to run a large set of queries against a tiny database.

Another issue is the incremental maintenance of the partially generated formulas. When an update to the model base changes the extension of some E-predicate, then step 3 has to be executed again. If the interpretation I_E gets more entries $\{E(V_{K+1}),E(V_{K+2},...\}$, then one only has to re-apply incrementally step 3 to the new entries. If the interpretation shrinks, then one has to remove the corresponding subformulas. ConceptBase attaches triggers to the E-predicate to achieve this type of formula maintenance. If the majority of updates to the model base include updates to the interpretation of E-predicates, then the partial evaluation method is rather expensive. Fortunately, the 'triangular' nature of layers in figure 1 suggests that this is not the case in 'normal' applications of meta modeling.

6 Application to Meta Modeling Cases

The benefit of meta formulas is that they encode the meaning of abstract concepts such as the concept of 'required' attributes encoded in formula 3. It was possible to partially evaluate this formula to a conjunction of type 1 formulas by a single E-predicate. We call such a meta formula a type 2 formula. If the meta formula has predicate occurrences `In(x,c),In(c,mc)` where c and mc are meta variables, one has to apply the partial evaluation method successively until the result is a type 1 formula. The first iteration eliminates mc as a variable, and subsequently c is eliminated. Such a meta formula is called a type 3 formula. If we have a predicates like `In(x,c)`, `In(c,mc)`, and `In(mc,mmc)` with all c, mc, mmc being meta variables, then we speak of a level 4 formula. A type 4 formulas has variables ranging over 4 different abstraction levels.

6.1 Meta level instantiation and attribution

It is useful to define some formulas that describe the relative instantiation between layers:

[Formula 7] $\forall$ x,c,mc In(mc,Concept) $\wedge$ In(c,mc) $\wedge$ In(x,c) $\Rightarrow$ In2(x,mc)

There are two meta variables in this example: c and mmc. The E-predicate is In(mc,Concept) where Concept is some constant denoting the class of all concepts in the model base. The normalized form is

[Formula 8] $\forall$ mc In(mc,Concept) ==> ($\forall$ x,c In(c,mc) $\wedge$ In(x,c) $\Rightarrow$ In2(x,mc))

Let In(EntityType,Concept) be in the interpretation of the E-predicate. Then, the partial evaluation yields

[Formula 9] $\forall$ x,c In(c,EntityType) $\wedge$ In(x,c) $\Rightarrow$ In2(x,EntityType))

Formula 9 happens to be again a meta formula (type 3). The meta variable is c and the E-predicate is `In(x,c)`. Let `In(Employee,EntityType)` be in the interpretation. The partial evaluation will then yield

[Formula 10] $\forall$ x In(x,Employee) $\Rightarrow$ In2(x,EntityType)

The derived predicate `In2(x,mc)` has an important contribution to meta modeling. It defines the relation of a concept `x` to its meta class `mc`. It can be used to define the meaning of being an entity or being a value:

[Formulas 11] $\forall$ x In2(x,EntityType) $\Rightarrow$ In(x,Entity)
[Formulas 12] $\forall$ x In2(x,Domain) $\Rightarrow$ In(x,Value)

Note that the variable x ranges over concepts at the M0 abstraction layer. Thus, formulas 11 and 12 really separate entities from values. It is defined independently from the M1 level and works for any M1 model instantiated to the M2 model. One can now easily express a condition that an entity may never be a value and vice versa. We leave this exercise to the reader. The In2 predicate can be accompanied by a similar predicate AL2 on attribution.

[Formula 13] `∀ c,d,x,m,mm,n,y In(x,c) ∧ In(y,d) ∧ AL(c,mm,m,d) ∧ AL(x,m,n,y) ⇒ AL2(x,mm,m,y)`

Here AL(c,mm,m,d) can serve as E-predicate. For AL(Employee,e_attr,salary,Integer) it is partially evaluated to

[Formula 14] `∀ x,m,n,y In(x,Employee) ∧ In(y,Integer) ∧ AL(x,salary,n,y) ⇒ AL2(x,e_attr,salary,y)`

With AL2, we can now define a predicate A2 as follows:

[Formula 15] `∀ x,mm,y AL2(x,mm,m,y) ⇒ A2(x,mm,y)`

This predicate is using an attribute label from the meta class layer (e.g. M2) while x,y are ranging over concepts two levels below (e.g. M0). Applied to our running example, `A2(x,e_attr,y)` subsumes all attribute links between objects x and y from the M0 level. Again, the A2 predicate is independent from the middle layer M1. If we extend our running example at the M2 level by an attribute 'key' between EntityType and Domain, then the predicate A2(x,key,y) precisely defines the key values y for a given entity x. The formula

[Formula 16] `∀ x1,x2,k A2(x1,key,k) ∧ A2(x2,key,k) ⇒ (x1 = x2)`

axiomatizes the key property based on the A2 predicate. Figure 2 illustrates this application. It implies that `A2(bill,key,130606)` is true.

The two predicate In2 and A2 may also be used to query the M0 level from the M2 level, i.e. to formulate queries to a database that are independent from the database schema. For example, one can find those entities that are identified by a key that occurs as normal attribute value (e_attr) in another entity.

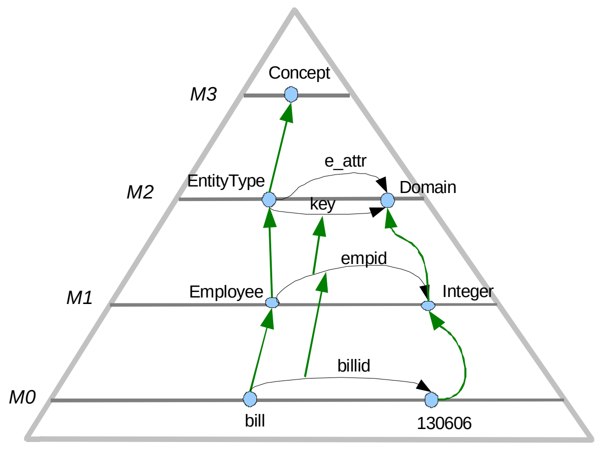

Figure 2: Formalizing the key property

Analogous to In2 and A2, one can define In3 and A3 predicate that relate concepts from the M3 layer and the M0 layer. We do not provide their definition but state that it allows to express properties of concepts from the M0 layer that are not only independent from the M1 layer (e.g. database schema) but also independent from the M2 layer, i.e. the modeling language. For example, one can define the fact that two concepts are linked to each other regardless of the type of the link.

We have applied meta formulas to provide a complete definition of the ERD modeling language including specialization (ISA), key attributes, and cardinalities. Apparently data modeling languages with their static semantics are affine to predicate logic with Herbrand interpretation. The same is not true for dynamic modeling languages like Petri nets[1]. While the semantics of ERDs can be explained in terms of the finite M0 models, dynamic modeling languages are defined on M0 layers encoding potentially infinitely many states of the execution of the dynamic model.

6.2 Relation properties

Some constructs in meta modeling are defined by re-usable patterns. For example, both the IsA relation between classes and subclasses and the PartOf relation between classes are transitive. However, while the IsA relation is reflexive, the partOf relation is anti-symmetric. Meta formulas easily cope with this partial semantic overlap of concepts. All that one has to do is to define the appropriate meta formulas and then instantiate the modeling constructs to those meta formulas that are applicable to them. Let us first define the relation properties by adapting their textbook definitions to our base predicates. To do so, we introduce the A predicate in terms of the AL predicate.

[Formula 17] `∀ x,m,n,y AL(x,m,n,y) ⇒ A(x,m,y)`

There is nothing special about the A predicate. It is simply a projection on AL. Analogous to the AL predicate, we regard a variable in `A(x,m,y)` to be a meta variable.

[Formula 18: transitivity] `∀ AC,x,y,z,M,C In(AC,transitive) ∧`
`P(AC,C,M,C) ∧ In(x,C) ∧ In(y,C) ∧ In(z,C) ∧ A(x,M,y) ∧ A(y,M,z) ⇒ A(x,M,z)`

[Formula 19: symmetry]
`∀ AC,x,y,M,C In(AC,symmetric) ∧ P(AC,C,M,C) ∧ In(x,C) ∧ In(y,C) ∧ In(z,C) ∧ A(x,M,y) ⇒ A(y,M,x)`

[Formula 19: antisymmetry] `∀ AC,x,y,M,C In(AC,antisymmetric) ∧`
`P(AC,C,M,C) ∧ In(x,C) ∧ In(y,C) ∧ In(z,C) ∧ A(x,M,y) ∧ A(y,M,x)`
`⇒(x = y)`

[Formula 20: reflexivity]
`∀ AC,x,M,C In(AC,reflexive) ∧`
`P(AC,C,M,C) ∧ In(x,C) ∧ In(y,C) ∧ In(z,C) ⇒ A(x,M,x)`

[1] The ConceptBase system is capable to model the dynamic semantics of languages such as Petri nets by so-called active rules. They are however of a procedural nature and therefore beyond the scope of this paper.

Figure 3 shows how the meta formulas are applied to a the `IsA` and `PartOf` relations of classes. It is sufficient to declare

```
{In(Class!IsA,transitive), In(Class!IsA,reflexive),
In(Class!PartOf,transitive),
In(Class!PartOf,antisymmetric)}
```

for encoding the desired meaning of the two constructs. Hence, the more meta formulas are available, the higher are the chances of re-use for multiple cases.

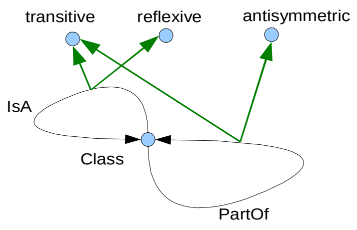

Figure 1: Configuring semantics via instantiation

7 Implementation

The method of partially evaluating meta formulas has been fully implemented in ConceptBase since 2003 for universally quantified formulas. The generated formulas are incrementally maintained when an update to an E-predicate occurs. This works both for additions and deletions. The bulk of the implementation work was not the partial evaluator but the code that selects the best E-predicate out of multiple candidates.

Except for pathological cases with large extensions of the E-predicate, there are no performance penalties when using meta formulas like the one on transitivity. We did tests with large model bases where version I defined transitivity 'by hand' and the version II used the meta formula. Both versions exhibit virtually the same evaluation times for queries over the transitive attribute. In the table below, reponse times of four queries are reported. Query Q1 is computing the dead ends of a network, i.e. those nodes that are connected only to nodes that lead to leave nodes. Query Q2 is an incomplete test on cliques and Q3 computes the transitive closure. Finally, Q4 returns the inverse of the transitive closure, i.e. nodes that are not reachable to each other. The size of the network is about 1400 nodes with about 5000 links. The table shows that there is no performance penalty to use the partially evaluated meta formulas.

Version	**Q1**	**Q2**	**Q3**	**Q4**
I	0.084	0.812	4.31	1.417
II	0.083	0.820	4.39	1.418

The universally quantified meta formulas are slightly easier to handle since they produce conjunctions of subformulas. Hence, each individual subformula can be maintained as a separate formula in the (deductive) theory. Existentially quantified meta formulas produce disjunctions of subformulas that must remain in one formula. ConceptBase currently does not support this case.

We used the MOF layers to motivate meta models and meta meta models. The implementation does not require this view. In fact by abandoning the strict association of concepts to MOF layers, one gets a more expressive meta modeling framework. Consider the concept 'Concept' in figure 1. It is located at the M3 layer and classifies concepts of the M2 level. ConceptBase allows objects like 'Concept' to have any another concept as instance regardless at which MOF layer we prefer to locate it. By this, any feature defined for Concept becomes available to any defined concept. For example, transitivity can be applied to the PartOf relation (M2 layer), but also to some domain model construct like 'hasAncestor' (M1 layer). In the first case, the partially evaluated formulas allow to compute the transitive closure between model elements (M1 layer). In the second case, they operate on the M0 layer. So, concepts like transitivity should not be assigned to strictly one abstraction layer. They constitute a relation between pairs of abstraction layers: M2 to M0, M3 to M1, M4 to M2, and so forth. Once defined, meta formulas can be used in ConceptBase by just making sure that the corresponding E-predicate is populated.

8 Related Work

While the partial evaluation technique itself is an adaption of the simplification method for deductive datab9ases (Nicolas, 1979), our contribution is to apply it to meta modeling. By our technique abstract constructs like transitivity can be defined once and forever. While the simplification method generates simplified formulas for any predicate occurrence, the partial evaluation method requires determining a single so-called E-predicate to compile a meta formula. The flat representation of the model base by predicates is inspired by Telos (Mylopoulos et al, 1990). In contrast to Telos, we have no predefined axioms to make the approach generally applicable. In particular the P predicate has a purely auxiliary function in our approach whereas it is central in the Telos axiomatization.

There are other ways to represent models and meta models by flat facts. The approach by (Bezivin, 2006) uses predicate names to encode concept names. As pointed out in section 2, such representations are preventing a management of first-order meta formulas.

Meta formulas are also treated by ontology management systems, in particular Protégé (Protégé, 2006). Protégé has predefined expressions for transitivity, symmetry etc. They are however not evaluated against some database as we do.

In linguistic semantics, so-called generalized quantifiers (Barwise and Cooper, 1981) have been investigated to descripe higher-order predicates, in particular to express properties of properties. Lambda parameters expressions abstract the predicate names from the logical formula. By substituting the parameters by actual predicate names, one yields a first order expression. For example,

```
lambda.X lambda.Y (X subset Y)(Student)(Person)
```

would be reduced to

```
Student subset Person.
```

We note that our definition of meta formulas is completely embedded into first order logic and does not require additional abstractions such as the lambda operator.

The HiLog logic programming language (Chen et al, 1993) deals with properties of properties much in the spirit of generalized quantifiers. Properties like transitive closure are expressed in a higher-order syntax of logic. HiLog has a mechanism to encode such higher-order expressions to predicate calculus using the `apply` predicate that shifts the higher order predicate symbol like 'transitive' to an argument position. The difference to our approach is that HiLog has a one-to-one encoding, i.e. the higher order formula is mapped to a single first order formula. In our approach, the meta formula (expressing the same principle of transitive closure) is mapped to many first order formulas, each specialized for one element of the extension of the E-predicate. So, we stick to first-order syntax to express higher order features and we provide for an efficient evaluation of the meta formulas via partial evaluation. On the other hand, HiLog is not limited by a finiteness assumption. HiLog is a Turing-complete programming language while our approach is defined for a deductive theory with finite perfect model semantics.

9 Application to Method Engineering

Partial evaluation of meta-level formulas has a somewhat theoretic flavor as it deals with highly abstract concepts. The most obvious application of the technique is for designing new modeling languages or formally defining existing modeling languages. The abstract concepts such as transitivity and multiplicity of relationships can be employed directly to define constructs like specialization and part-of in modeling languages. The added value of the meta-level formulas is that they only have to be defined once. Instantiating them leads to the automatic generation of a formula (either a rule or a constraint) specialized for the modeling language. In essence, the meta level formulas are the building blocks for the semantics of formally defined modeling languages.

The modeling languages are the product side of a method to be engineered. There is also a production side, namely the guidelines or processes that create and manipulate the products. The two sides are dependent on each other leading to amalgamated model such as process-data diagrams (Weerd et al., 2006) or software process models (Jarke et al, 1990). The amalgamated model constitutes a method fragment subject to be incorporated in complex methods. Such models also conform to some schema, i.e. there is a meta model in the sense of MOF that describes how the product side and the production side may be linked to each other. The link types themselves are elements of a modeling language, namely the language to describe the connection between a process and its products. This language has a meaning that should be reflected in the defininition of its constructs. Not surprisingly, the abstract concepts like transitivity are applicable to define the desired semantics. As an example, we focus on the traceability of products, i.e. of models and their elements. Traceability is has been a hot topic in the requirements engineering community. It allows following the development of the products of a development process. Each product (and each element of a product) depends on other products (and their elements). This is essentially the transitivity concept. All we would need to do would be to define a predicate `A(p1,dependsOn,p2)` in the meta model of the amalgamated process/product diagrams and to declare that dependsOn is transitive, i.e. `In(Product!dependsOn,transitive)`.

As a second example consider the versioning of product models as described in (Saeki, 2006). A product model is versioned by applying change operations on it. Let's assume that the version relation is represented by a predicate `A(p1,versionedTo,p2)` where p1 and p2 are product models. Then, one would define the versionedTo relation to be transitive and assymetric (a product model cannot be versioned to itself). The newest version of a given product model p can then be retrieved simply by querying

```
A(p,versionedTo,pv) ∧ ¬ ∃ px (pv,versionedTo,px)
```

10 Conclusions

We presented an approach to manage first order formulas defining the meaning of modeling constructs at the meta and meta meta class levels. It turned out that it is sufficient to demand range-restrictedness in order to partially evaluate the meta formulas into conjunctions or disjunctions of non-meta formulas that are more efficient to evaluate.

The flat representation of the model base with predicate facts is powerful enough to capture the MOF abstraction layers. Since instantiation is stated explicitly, our method is more general by allowing models that *link concepts* of different abstraction layers. To apply the partial evaluation approach, one simply has to encode that one concept is an instance of another concept rather than by specifying to which MOF layer a concept belongs. It is perfectly possible to define a meta class (or even a meta meta class) that has an attribute link to a concept that one would regard as M0 concept. For example, a meta class can have an attribute 'createdBy' that links it to its creator. This phenomenon systematically occurs when one superimposes a product meta model (e.g. ERD) with a process meta model, i.e. a specification of operations that manipulate instances of the product meta model. This is the common case in method engineering as it links process models (the procedural steps of the method) with product models (the input and output of the steps).

The greatest benefit of our method arises when meta formulas are defined at the most generic layer. Then, the semantics of modeling language constructs is generated by instantiating them to the meta formulas. This level of re-use of meta formulas makes meta modeling itself a more productive activity: instead of coding formulas one simply declares a construct as an instance of the abstract concepts defined by the meta formulas. As an additional bonus, the generated formulas are materialized and can be attached to the meta model describing the modeling language.

As mentioned earlier, semantics of dynamic modeling languages are not covered by our technique because they require to reason about infinite extensions. We plan to investigate whether certain principles like state transitions can be defined as an abstract concept, i.e. independently from the specific modeling language, and then be instantiated to a specific modeling language by an analogous approach. That capability would further enrich the toolbox for engineering modeling languages from pre-fabricated building blocks. For example, languages like event-process chains should be definable from building blocks that can also be used to define petri nets.

The partial evaluation technique described in this paper is fully implemented in the ConceptBase system and has been used in various meta modeling scenarios. Some details are on the web page http://conceptbase.cc.

Acknowledgements: We cordially thank René Soiron for implementing a major part of the meta formula compiler in ConceptBase.

References

Barwise, J. and Cooper, R., 1981. Generalized quantifiers and natural language. Linguistics and Philosophy 4: 159-219.

Bezivin, J., 2006. On the Unification Power of Models. Software and System Modeling (SoSym) 4(2):171--188.

Bry, F., 1989. Logical rewritings for improving the evaluation of quantified queries. Proc. 2nd Intl. Symposium on Mathematical Fundamentals of Database Systems, Visegrád, Hungary, 1989, Springer-Verlag, LNCS 364.

Chen, W., Kifer, M., Warren, D.S., 1993. HiLog: A foundation for higher-order logic programming. Journal of Logic Programming 15(3):187-230.

OMG, 2006. Meta Object Facility. Online http://www.omg.org/mof/, June 2006.

Protégé, 2006. The Protégé ontology editor and knowledge acquisition system. Online http://protege.stanford.edu/, June 2006.

Jarke, M., R. Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S, 1995.: ConceptBase - a deductive object base for meta data management. Journal of Intelligent Information Systems, 4, 2, 1995, pp. 167-192.

Jarke, M., Jeusfeld, M.A., Rose, T., 1990: A software process data model for knowledge engineering in information systems. In Information Systems, 15, 1, 1990, pp. 85-116.

Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M., 1990. Telos - a language for representing knowledge about information systems. In ACM Trans. Information Systems, 8, 4, 1990, pp. 325-362.

Nicolas, J.-M., 1979. Logical formulas and integrity constraints: the range restricted property and a simplification method. Technical report T-R CERT-LBD/79-1, Toulouse, France.

Saeki, M., 2006: Configuration management in a method engineering context. Proceedings CAiSE 2006, Springer-Verlag, LNCS 4001/2006, pp. 384-398.

Weerd, I. van de, Versendaal, J., Brinkkemper, S., 2006. A product software knowledge infrastructure for situational scpability maturation: vision and case studies in product management, Technical Report UU-CS-2006-008, Utrecht University, The Netherlands.