An Algebraic Semantics for MOF

Artur Boronat[1] and José Meseguer[2]

[1] Department of Computer Science, University of Leicester
aboronat@le.ac.uk
[2] Department of Computer Science, University of Illinois at Urbana-Champaign
meseguer@uiuc.edu

Abstract. Model-driven development is a field within software engineering in which software artifacts are represented as models in order to improve productivity, quality, and cost effectiveness. In this field, the Meta-Object Facility (MOF) standard plays a crucial role by providing a generic framework where the abstract syntax of different modeling languages can be defined. In this work, we present a formal, algebraic semantics of the MOF standard in membership equational logic (MEL). By using the Maude language, which directly supports MEL specifications, this formal semantics is furthermore *executable*, and can be used to perform useful formal analyses. The executable algebraic framework for MOF obtained this way has been integrated within the Eclipse Modeling Framework as a plugin. In this way, formal analyses, such as semantic consistency checks, become available within Eclipse to provide formal support for model-driven development processes.

Key words: MOF, model-driven development, membership equational logic, metamodeling semantics, reflection.

1 Introduction

Model-driven development is a field in software engineering in which software artifacts are represented as models in order to improve productivity, quality, and cost-effectiveness. Models provide a more abstract description of a software artifact than the final code of the application. The Meta-Object Facility (MOF) standard [1] describes a generic framework in which the abstract syntax of modeling languages can be defined. This is done by specifying within MOF different metamodels for different modeling languages. Models in a modeling language are then conforming instances of their corresponding metamodel. The MOF standard aims at offering a good basis for model-driven development, providing some of the building concepts that are needed: what is a model, what is a metamodel, what is reflection in a MOF framework, etc. However, most of these concepts lack at present a formal semantics in the current MOF standard. This is, in part, due to the fact that metamodels can only be defined as data in the MOF framework.

In this paper, we define a reflective, algebraic, executable framework for precise metamodeling that supports the MOF standard. On the one hand, our formal framework provides a formal semantics of the following notions: *metamodel*,

model and *conformance* of a model to its metamodel. We clearly distinguish the different roles that the notion of *metamodel* usually plays in the literature: as *data*, as *type*, and as *theory*. In addition, we introduce two new notions: (i) *metamodel realization*, referring to the mathematical representation of a metamodel; and (ii) *model type*, allowing models to be considered as first-class citizens. In particular, our executable algebraic semantics for MOF generates in an automatic way the algebraic semantics of any MOF metamodel. This is a powerful and very useful form of *reflection*, in which metamodel MOF reflection is systematically related to logical reflection in MEL. The executable formal semantics of a metamodel obtained in this reflective way can then be used to automatically analyze the conformance of its model instances, which are characterized either as terms modulo structural axioms or, equivalently, as graphs. This makes the formal semantics particularly useful, since models can be directly manipulated as graphs in their term-modulo-axioms formal representation. Furthermore, our framework provides an executable environment that is plugged into the Eclipse Modeling Framework (EMF) [2] and that constitutes the kernel of a model management framework, supporting model transformations and formal analysis techniques.

The paper is structured as follows: Section 2 briefly describes the underlying formal background; Section 3 identifies important concepts that are not defined in the MOF standard, which are usually left unspecified in most of the MOF implementations; Section 4 gives a high level view of our algebraic framework, indicating how the algebraic semantics of MOF metamodels is defined; Section 5 presents some related work; and Section 6 summarizes the main contributions of this work and discusses future work.

2 Preliminaries: Membership Equational Logic

A membership equational logic (MEL) [3] *signature* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded signature and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A MEL Σ-algebra A contains a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1 \cdots k_n, k}$ and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that the elements in sorts are well-defined, while elements without a sort are *errors*. $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ denote, respectively, the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of kinded variables.

Given a MEL signature Σ, *atomic formulae* have either the form $t = t'$ (Σ-equation) or $t : s$ (Σ-membership) with $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$; and Σ-*sentences* are conditional formulae of the form $(\forall X)\ \varphi \ \ if \ \ \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$, where φ is either a Σ-equation or a Σ-membership, and all the variables in φ, p_i, q_i, and w_j are in X.

A MEL theory is a pair (Σ, E) with Σ a MEL signature and E a set of Σ-sentences. The paper [3] gives a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, and initial and free algebras. In particular, given

a MEL theory (Σ, E), its initial algebra is denoted $T_{(\Sigma/E)}$; its elements are E-equivalence classes of ground terms in T_Σ.

Order-sorted notation $s_1 < s_2$ can be used to abbreviate the conditional membership $(\forall x : k)\ x : s_2 \ \ if \ \ x : s_1$. Similarly, an operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s \ \ if \ \ \bigwedge_{1 \leqslant i \leqslant n} x_i : s_i$. We write $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ in place of $(\forall x_1 : k_1, \ldots, x_n : k_n)\ t = t' \ \ if \ \ \bigwedge_{1 \leqslant i \leqslant n} x_i : s_i$.

We can use order-sorted notation as syntactic sugar to present a MEL theory (Σ, E) in a more readable form as a tuple $(S, <, \Sigma, E_0 \cup A)$ where: (i) S is the set of sorts; (ii) $<$ is the subsort inclusions, so that there is an implicit kind associated to each connected component in the poset of sorts $(S, <)$; (iii) Σ is given as an order-sorted signature with possibly overloaded operator declarations $f : s_1 \times \ldots \times s_n \to s$ as described above; and (iv) the set E of (possibly conditional) equations and memberships is quantified with variables having specific sorts (instead than with variables having specific kinds) in the sugared fashion described above; furthermore, E is decomposed as a disjoint union $E = E_0 \cup A$, where A is a collection of "structural" axioms such as associativity, commutativity, and identity. Any theory $(S, <, \Sigma, E_0 \cup A)$ can then be desugared into a standard MEL theory (Σ, E) in the way explained above.

The point of the decomposition $E = E_0 \cup A$ is that, under appropriate executability requirements explained in [4], such as confluence, termination, and sort-decreasingness modulo A, an MEL theory $(S, <, \Sigma, E_0 \cup A)$ becomes *executable* by rewriting with the equations and memberships E_0 *modulo* the structural axioms A. Furthermore, the initial algebra $T_{(\Sigma/E)}$ then becomes isomorphic to the *canonical term algebra* $Can_{\Sigma/E_0,A}$ whose elements are A-equivalence classes of ground Σ-terms that cannot be further simplified by the equations and memberships in E_0.

3 Presentation of the Problem

In this section, we give an informal description of the MOF standard by describing the MOF architecture and the main concepts in the MOF metamodel, which are then given a formal semantics in subsequent sections.

3.1 The MOF Modeling Framework

MOF is a semiformal approach to define modeling languages. It provides a four-level hierarchy, with levels M0, M1, M2 and M3. The entities $\widetilde{m}$ populating each level M_i, written $\widetilde{m} \in M_i$ are always *collections*, made up of constituent *data elements* $\widetilde{e}$. Each entity $\widetilde{M} \in M_{i+1}$ at level i+1 metarepresents a *model*[3] M and is viewed as the metarepresentation of a collection of *types*, i.e., as a

[3] In the MOF framework, the concept of a *model* M is conceptually specialized depending on the specific metalevel, in which a model is located: *model* at level M1, *metamodel* at level M2 and *meta-metamodel* at level M3; as shown below.

metadata collection that defines a specific collection of types. Each *type* T is metarepresented as $\widetilde{T} \in \widetilde{M}$ and characterizes a collection of data elements, its *value domain*. We write that a data element $\widetilde{e} \in \widetilde{m}$ is *a value of* type $\widetilde{T} \in \widetilde{M}$ as $\widetilde{e} \,\widetilde{:}\, \widetilde{T}$. A metarepresentation at level *i + 1* of a collection $\widetilde{M} \in M_{i+1}$ of types characterizes collections of data elements $\widetilde{m} \in M_i$ at level i. A specific data collection $\widetilde{m} \in M_i$ is said to *conform to* model M, which is metarepresented by its collection of types $\widetilde{M} \in M_{i+1}$, iff for each data element $\widetilde{e} \in \widetilde{m}$ there exists a type $\widetilde{T} \in \widetilde{M}$ such that $\widetilde{e} \,\widetilde{:}\, \widetilde{T}$. We write $\widetilde{m} \,\widetilde{:}\, \widetilde{M}$ to denote this conformance relation for model M, which we call the *structural conformance relation*. The *isValueOf* relation $\widetilde{e} \,\widetilde{:}\, \widetilde{T}$ and the *structural conformance relation* $\widetilde{m} \,\widetilde{:}\, \widetilde{M}$ are summarized in Fig. 1.

$$\begin{array}{ccc} M_i & & M_{i+1} \\ \in & & \in \\ \widetilde{m} & \widetilde{:} & \widetilde{M} \\ \in & & \in \\ \widetilde{e} & \widetilde{:} & \widetilde{T} \end{array}$$

Fig. 1. *isValueOf* and *structural conformance* relations.

Fig. 2 illustrates example collections at each level M1-M3 of the MOF framework. Each collection is encircled by a boundary and tagged with a name. For example, $rsPerson \in M_1$, which is a model corresponding to a relational schema. The *isValueOf* relation between elements $\widetilde{e}$ of a data collection and the metarepresentation of types $\widetilde{T}$ of a type collection, and the *structural conformance* relation between a data collection $\widetilde{m}$ and the metarepresentation $\widetilde{M}$ of a model M are depicted with dashed arrows. We consider levels M1–M3 out of the MOF hierarchy in this work, as illustrated in Fig. 2, which are:

M1 level. The M1 level contains metarepresentations of *models*. A model is a set of types that describe the elements of some physical, abstract or hypothetical reality by using a well-defined language. In addition, a model is suitable for computer-based interpretation, so that development tasks can be automated. For example, a model can define a relational schema describing the concepts, i.e., types, of *Person*, *Invoice* and *Item*. The type of *Person* is a table *Person*, with columns *name* and *age*; similarly, there is a table *Invoice*, with columns *date* and *cost*; and a table *Item*, with columns *name* and *price*; a foreign key *Invoice_Person_FK*; and a foreign key *Item_Invoice_FK*.

M2 level. The M2 level contains metarepresentations of *metamodels*. A metamodel is a model specifying a modeling language. As an example, we take a simple relational metamodel from the example of the QVT standard that contains the main concepts to define relational schemas, as shown in Fig. 2 in UML notation. The types of a relational schema are called *table*, *column*, *foreign key*, etc. Our example model, the relational schema with tables *Per-*

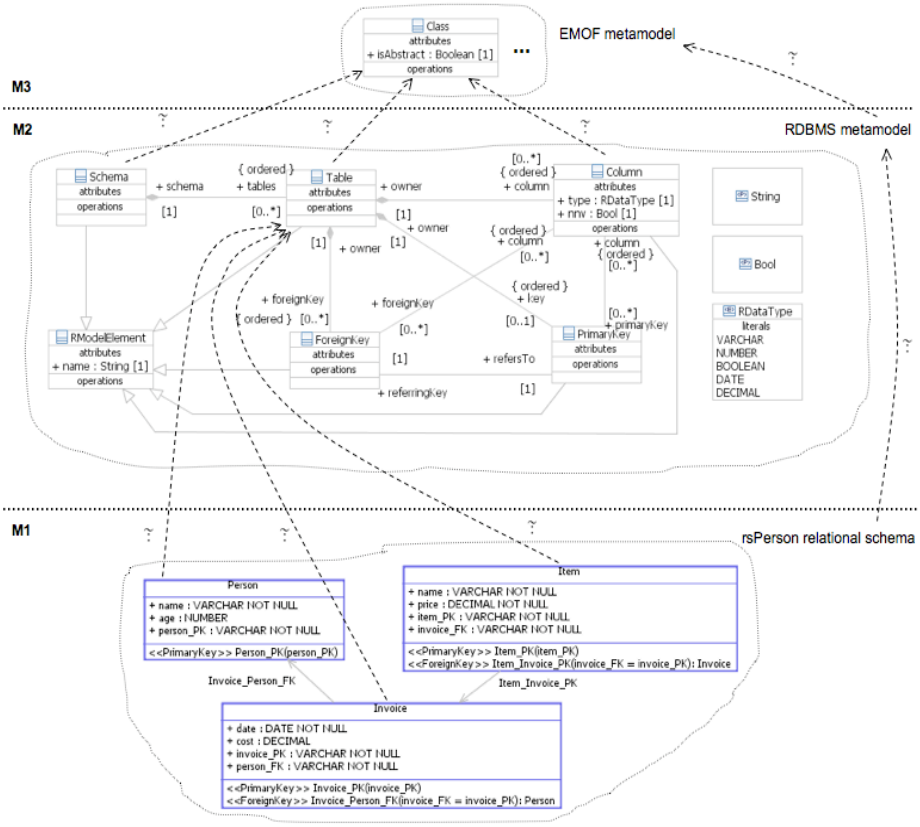

Fig. 2. The MOF framework

son, *Invoice* and *Item* can be represented as a collection at level M1 that conforms to the relational metamodel at level M2.

M3 level. An entity at the M3 level is the metarepresentation of a *meta-metamodel*. A meta-metamodel specifies a *modeling framework*, which could also be called a *modeling space*. In MOF, there is only one such meta-metamodel, called the MOF meta-metamodel. Within the MOF modeling framework one can define many different metamodels. Such metamodels, when represented as data, must conform to the MOF meta-metamodel. In particular, the relational metamodel conforms to the MOF meta-metamodel. But in MOF one can likewise define many other metamodels, for example the UML metamodel to define UML models, the OWL metamodel to define ontologies, the AADL metamodel, and so on. The fact that all these meta-models are specified within the single MOF framework greatly facilitates systematic model/metamodel interchange and integration.

3.2 Discussion and Open Problems

At present, important MOF concepts such as those of metamodel, model and conformance relation do not have an explicit, *syntactically characterizable* status in their data versions. For example, we can syntactically characterize the correctness of the data elements in $\widetilde{\mathcal{M}}$ for a metamodel $\mathcal{M}$, but there is no explicit type that permits defining $\widetilde{\mathcal{M}}$ as a well-characterized value, which we call a *model type*. In addition, in the MOF standard and in current MOF-like modeling environments, such as Eclipse Modeling Framework or MS DSL tools, a metamodel $\mathcal{M}$ does not have a precise *mathematical* status. Instead, at best, a metamodel $\mathcal{M}$ is realized as a program in a conventional language, which may be generated from $\widetilde{\mathcal{M}}$, as, for example, the Java code that is generated for a metamodel $\widetilde{\mathcal{M}}$ in EMF. This informal implementation corresponds to what we call a *metamodel realization*. In these modeling environments, the *conformance relation* between a model definition $\widetilde{M}$ and its corresponding metamodel definition $\widetilde{\mathcal{M}}$ is checked by means of indirect techniques based on XML document validation or on tool-specific implementations in OO programming languages. Therefore, metamodels $\widetilde{\mathcal{M}}$ and models $\widetilde{M}$ cannot be explicitly characterized as first-class entities in their data versions, and the semantics of the *conformance relation* remains formally unspecified. This is due to the lack of a suitable reflective formal framework in which software artifacts, and not just their metarepresentations, can acquire a formal semantics.

In this work, we formalize the notions of: (i) model type, (ii) metamodel realization and (iii) conformance relation, by means of a reflective semantics that associates a mathematical metamodel realization to each metamodel definition $\widetilde{\mathcal{M}}$ in MOF.

4 An Algebraic Semantics for MOF

The practical usefulness of a formal semantics for a language is that it provides a rigorous standard that can be used to judge the correctness of an implementation. For example, if a programming language lacks a formal semantics, compiler writers may interpret the informal semantics of the language in different ways, resulting in inconsistent and diverging implementations. For MOF, given its genericity, the need for a formal semantics that can serve as a rigorous standard for any implementation is even more pressing, since many different modeling languages rely on the correctness of the MOF infrastructure. In this section, we propose an algebraic, mathematical semantics for MOF in membership equational logic (MEL).

4.1 A High-Level View of the MOF Algebraic Semantics

A metamodel definition $\widetilde{\mathcal{M}}$ describes a metamodel realization that contains a model type $\mathcal{M}$. What this metamodel definition describes is, of course, a *set* of models. We call this the *extensional* semantics of $\widetilde{\mathcal{M}}$, and denote this semantics

by $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$. Recall that we use the notation $\widetilde{M} : \mathcal{M}$ for the conformance relation. Using this notation, the extensional semantics can be informally defined as follows:

$$[\![\mathcal{M}]\!]_{\mathrm{MOF}} = \{\widetilde{M} \mid \widetilde{M} : \mathcal{M}\}.$$

We make the informal MOF semantics just described mathematically precise in terms of the *initial algebra semantics* of MEL. As already mentioned in Section 2, a MEL specification (Σ, E) has an associated initial algebra $T_{(\Sigma,E)}$. We call $T_{(\Sigma,E)}$ the *initial algebra semantics* of (Σ, E), and write

$$[\![(\Sigma, E)]\!]_{IAS} = T_{(\Sigma,E)}.$$

Let $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ denote the set of all MOF metamodel definitions $\widetilde{\mathcal{M}}$, and let *SpecMEL* denote the set of all MEL specifications. The reason why we define $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ as a set of metamodel definitions $\widetilde{\mathcal{M}}$, instead than as a set of model types $\mathcal{M}$ is because, as already mentioned, the mathematical status of $\mathcal{M}$ is, as yet, undefined, and is precisely one of the questions to be settled by a mathematical semantics. Instead, well-formed metamodel definitions $\widetilde{\mathcal{M}}$ *are* data structures that can be syntactically characterized in a fully formal way. Therefore, the set $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, thus understood, is a well-defined mathematical entity. Our algebraic semantics is then defined as a function

$$\mathit{reflect}_{\mathrm{MOF}} : [\![\mathrm{MOF}]\!]_{\mathrm{MOF}} \longrightarrow \mathit{SpecMEL}$$

that associates to each MOF metamodel definition $\widetilde{\mathcal{M}}$ a corresponding MEL specification $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. Our informal semantics $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$ is now made mathematically precise. Recall that any MEL signature Σ has an associated set S of sorts. Therefore, in the initial algebra $T_{(\Sigma,E)}$ each sort $s \in S$ has an associated set of elements $T_{(\Sigma,E),s}$. The key point is that in any MEL specification of the form $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, there is always a sort called $\mathit{ModelType}\{\mathcal{M}\}$, which we also denote as $\mathcal{M}$ for short, whose data elements in the initial algebra are precisely the data representations of those models that conform to $\mathcal{M}$. That is, $\mathcal{M}$ is the *model type* associated to a metamodel definition $\widetilde{\mathcal{M}}$. Therefore, we can give a precise mathematical semantics to our informal MOF extensional semantics by means of the equation

$$[\![\mathcal{M}]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{ModelType}\{\mathcal{M}\}}.$$

Note that our algebraic semantics gives a precise mathematical meaning to the entities lacking such a precise meaning in the informal semantics, namely, the notions of: (i) model type $\mathcal{M}$, (ii) metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, and (iii) conformance relation $\widetilde{M} : \mathcal{M}$. Specifically, we associate to a metamodel definition $\widetilde{\mathcal{M}}$ a precise mathematical object, namely, the MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, constituting its *metamodel realization*. The *structural conformance* relation between a model and its metamodel is then defined mathematically by the equivalence

$$\widetilde{M} : \mathcal{M} \quad \Leftrightarrow \quad \widetilde{M} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{ModelType}\{\mathcal{M}\}}.$$

4.2 Algebraic Semantics of MOF Metamodels

As introduced above, the $\mathit{reflect}_{\mathrm{MOF}}$ function maps a MOF metamodel definition $\widetilde{\mathcal{M}}$ to a MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ that constitutes its metamodel realization. In this section, we provide a high-level summary of the definition of the $\mathit{reflect}_{\mathrm{MOF}}$ function, whose complete definition is available in [5].

MOF as a MEL theory. We denote the metamodel definition that constitutes the meta-metamodel of the MOF framework by $\widetilde{\mathrm{MOF}}$. $\widetilde{\mathrm{MOF}}$ is itself a MOF metamodel definition, since $\widetilde{\mathrm{MOF}} : \mathrm{MOF}$. We first define a MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$, that is, we first define $\mathit{reflect}_{\mathrm{MOF}}$ for a *single* metamodel, namely $\widetilde{\mathrm{MOF}}$. The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory defines the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ type as the set of metamodel definitions $\widetilde{\mathcal{M}}$, which can be viewed as both graphs and terms. This theory has been manually defined as a first step in the bootstrapping process needed to define the $\mathit{reflect}_{\mathrm{MOF}}$ function in general.

The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory provides the algebraic representation for object types and model types for defining metamodel definitions $\widetilde{\mathcal{M}}$. In this theory, object types are used to describe a metamodel definition $\widetilde{\mathcal{M}} : \mathrm{MOF}$ as a set of objects. Objects are defined by using the following sorts: `Oid#MOF` for object identifiers; `Cid#MOF` for class names; and `PropertySet#MOF` for multisets of comma-separated pairs of the form `(property : value)`, which represent property values. Objects in a metamodel definition $\widetilde{\mathcal{M}}$ are then syntactically characterized by means of an operator

```
<_:_|_> : Oid#MOF Cid#MOF PropertySet#MOF -> Object#MOF.
```

These sorts, subsorts and operators are defined, in Maude notation, as follows:

```
sorts Oid#MOF Cid#MOF Property#MOF PropertySet#MOF Object#MOF .
subsort Property#MOF < PropertySet#MOF .
op noneProperty : -> PropertySet#MOF .
op _‘,_ : PropertySet#MOF PropertySet#MOF -> PropertySet#MOF
    [assoc comm id: noneProperty] .
op <_:_|_> : Oid#MOF Cid#MOF PropertySet#MOF -> Object#MOF .
```

In the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory, a metamodel definition $\widetilde{\mathcal{M}}$ that conforms to the metamodel MOF, that is, such that $\widetilde{\mathcal{M}} : \mathrm{MOF}$, can be represented as a collection of objects by means of a term of sort `ModelType{MOF}`. A term of sort `ModelType{MOF}` is defined by means of the following constructors, in Maude notation:

```
op __ : ObjectCollection{MOF} ObjectCollection{MOF} -> ObjectCollection{MOF} [assoc comm] .
op <<_>> : ObjectCollection{MOF} -> ModelType{MOF} .
```

where the `Object#MOF` sort is a subsort of the `ObjectCollection{MOF}` sort. That is, we first form a multiset of objects of sort `ObjectCollection{MOF}` using the associative

and commutative multiset union operator __[4] and then we *wrap* the set of objects by using the `<<_>>` constructor to get the desired term of sort `ModelType{MOF}`.

Each of the modeling primitives that constitute the MOF metamodel is specified in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory by means of sorts, subsorts and operators. As an example, we focus on the Class and Property object types of the MOF metamodel.

Class *object type.* Object types are the central concept of MOF to model entities of the problem domain in metamodels. An object type is defined in a metamodel definition $\widetilde{\mathcal{M}}$ as a Class instance $\tilde{c}$ and a set of Property instances $\tilde{p}$. The object type Class contains meta-properties like *name*, which indicates the name of the object type, *ownedAttribute*, which indicates the properties that belong to the Class instance, and *superClass*, which indicates that the object type is defined as a specialization of the object types that are referred to by means of this property. The Class object type is specified as a sort `Class`, such that `Class < Cid#MOF`, and a constant `Class : -> Class`. Each of the class properties is defined as a constructor for the sort `Property#MOF`. For example, to define the *name* property, we have the constructor `name‘:_ : String -> Property#MOF`, and to define the *package* property, we have the constructor `package‘:_ : Oid -> Property#MOF`[5].

In the relational metamodel definition, the Class instance that defines the object type Table in the metamodel $\widetilde{\mathrm{RDBMS}}$ is defined as the term

```
< 'Foo : Class | name : "Table", package : ..., ownedAttribute : ...>,
```

where `'Foo` is an object identifier.

Property *object type.* A model can be viewed as a graph where the collection of nodes is constituted by the collection of attributed objects of the model and the edges are defined by means of directed references or links between objects. A Property object in a metamodel definition $\widetilde{\mathcal{M}}$ enables the definition of an attribute in an object or a reference between objects in a model definition $\widetilde{M} : \widetilde{\mathcal{M}}$, one level down in the MOF framework. A Property object defines the type of the property, where the type can be a basic type definition, an enumeration type definition or an object type definition. Other meta-properties, such as *lower*, *upper*, *ordered* and *unique*, constitute the multiplicity metadata of a specific property.

The constructors that permit defining objects of the Property object type are defined, in Maude notation, as follows:

```
sort Property  . subsort Property < Cid#MOF .
op Property : -> Property .
op lower‘:_ : Int -> Property .   op upper‘:_ : Int -> Property .
op isOrdered‘:_ : Bool -> Property . op isUnique‘:_ : Bool -> Property .
```

The Property instance $\tilde{p}$ that defines the metaproperty *name* of the object type RModelElement in the metamodel definition $\widetilde{\mathrm{RDBMS}}$ is represented by the term

[4] This binary operator symbol has empty syntax (juxtaposition).

[5] Note that property operators can be typed with an object identifier (in the case of references) and with a data type (in the case of attributes).

```
< 'Foo : Property | name : "name", lower : 1, upper: 1, isOrdered = true, isUnique = false,
isComposite = false, type : 'PrimitiveType0, class : 'Class0 >.
```

We have taken into account the modeling primitives that constitute the Essential MOF metamodel definition, including simple data types and enumeration types. A detailed specification is provided in [5].

Reflective Algebraic Semantics of MOF Metamodels. Once the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory is defined, we focus on the $\mathit{ModelType}\{\mathrm{MOF}\}$ sort in this theory, whose carrier in the initial algebra defines the $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ type, i.e., the model type whose elements are metamodels:

$$[\![\mathrm{MOF}]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}}),\mathit{ModelType}\{\mathrm{MOF}\}}.$$

Note that, since $[\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$ is the set of all metamodel definitions $\widetilde{\mathcal{M}}$ in MOF, this means that

$$\widetilde{\mathrm{MOF}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}.$$

We then define the value of the function $\mathit{reflect}_{\mathrm{MOF}}$ on *any* metamodel $\widetilde{\mathcal{M}}$, such that $\widetilde{\mathcal{M}} \in [\![\mathrm{MOF}]\!]_{\mathrm{MOF}}$, as its corresponding MEL theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. Given a metamodel definition $\widetilde{\mathcal{M}}$, the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory defines the $[\![\mathcal{M}]\!]_{\mathrm{MOF}}$ semantics as the set of model definitions $\widetilde{M}$ that are constituted by a collection of typed objects, which can be viewed as both a graph and a term.

In the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory, the algebraic notion of *object type* is generically given by means of the sort $\mathit{Object}\#\mathcal{M}$. Terms of sort $\mathit{Object}\#\mathcal{M}$ are defined by means of the constructor

$$< \_ : \_ \mid \_ > : \mathit{Oid}\#\mathcal{M}\ \ \mathit{Cid}\#\mathcal{M}\ \ \mathit{PropertySet}\#\mathcal{M} \rightarrow \mathit{Object}\#\mathcal{M},$$

which is analogous to the constructor for objects that has been presented in the $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{MOF}})$ theory.

Model definitions $\widetilde{M} : \mathcal{M}$ are given as collections of objects, which are instances of a specific object type OT. Object types are defined in a metamodel definition $\widetilde{\mathcal{M}} : \mathrm{MOF}$, as a multiset of objects $\widetilde{\mathrm{OT}} \subseteq \widetilde{\mathcal{M}}$. Defining the algebraic semantics of an object type involves the definition of the object identifiers and the properties that may be involved in the definition of a specific object in a model definition $\widetilde{M} : \mathcal{M}$. Object type specialization relationships must be also taken into account. Therefore, we need to define the carrier of the sorts $\mathit{Oid}\#\mathcal{M}$, $\mathit{Cid}\#\mathcal{M}$ and $\mathit{PropertySet}\#\mathcal{M}$ for a specific object type definition.

Consider, for example, the $\widetilde{\mathrm{RDBMS}}$ metamodel definition, where the TABLE object type, denoted by $\widetilde{\mathrm{TABLE}}$, is specified in Maude notation as

```
<< < 'Table : Class | name : "Table", isAbstract : false,
  ownedAttribute : OrderedSet{ 'prop0 ::'prop1 :: 'prop2 :: 'prop3},
  superClass : OrderedSet{ 'RModelElement } >
< 'prop0 : Property | name : "schema", lower : 1, upper: 1,
  isOrdered = true, isUnique = true, isComposite = true, type : 'Schema, class : 'Table >
< 'prop1 : Property | name : "column", lower : 0, upper: -1,
  isOrdered = true, isUnique = true, isComposite = false, type : 'Column, class : 'Table >
```

```
< 'prop2 : Property | name : "key", lower : 0, upper: 1,
  isOrdered = true, isUnique = true, isComposite = false, type : 'PrimaryKey, class : 'Table >
< 'prop3 : Property | name : "foreignKey", lower : 0, upper: -1, isOrdered = true,
  isUnique = true, isComposite = false, type : 'PrimaryKey, class : 'Table > >>.
```

where `'PrimaryKey` is the object identifier for the CLASS instance of the PRIMARYKEY object type in the relational metamodel $\widetilde{\text{RDBMS}}$. In subsequent paragraphs, we use this example to obtain the theory that defines the TABLE object type.

Object Type Names. In the $\mathit{reflect}_{\text{MOF}}(\widetilde{\mathcal{M}})$ theory, each CLASS instance $\widetilde{cl}$ in $\widetilde{\mathcal{M}}$ is defined as a new sort and a constant, both of them with the name of the class. *Abstract classes* are defined as those that cannot be instantiated. The name of an abstract class C is *not* specified with a constant $C : C$, so that objects in a metamodel definition $\widetilde{\mathcal{M}}$ cannot have C as their type.

In the example of the RDBMS metamodel, the $\mathit{reflect}_{\text{MOF}}$ function generates a single sort and a single constant for the object type TABLE, specified in Maude notation as follows,

```
sort Table . subsort Table < Cid#rdbms . op Table : -> Table .
```

Object Type Properties. An object type OT is defined with a collection of PROPERTY instances describing its properties. A PROPERTY instance $\widetilde{p}$ in a metamodel definition $\widetilde{\mathcal{M}}$: MOF is given by an object $\widetilde{p}$, such that $\widetilde{p}$: PROPERTY and $\widetilde{p} \in \widetilde{\mathcal{M}}$. A Property instance $\widetilde{p}$ is associated with a specific type $\widetilde{t}$ in the metamodel definition $\widetilde{\mathcal{M}}$, which is defined as an object $\widetilde{t}$: TYPE. Depending on the type $\widetilde{t}$ of a property, we can distinguish two kinds of properties:

- *Value-typed Properties or Attributes.* Properties of this kind are typed with DATATYPE instances, which can represent either a simple data type or an numeration type. Value-typed properties define the attributes of the of nodes in a graph.
- *Object-typed Properties or References.* Properties of this kind are typed with object types. Model definitions $\widetilde{M}$ can then be viewed as graphs, where objects define graph nodes and object-typed properties define graph edges. For example, we can define a CLASS instance "Table" and a PROPERTY instance "name" that are related by means of their respective *ownedAttribute* and *class* properties:

  ```
  < 'class0 : Class | name : "Table", ownedAttribute : OrderedSet{ 'prop0 } >
  < 'prop0 : Property | name : "name", class : 'class0 >
  ```

The *type* meta-property together with the multiplicity metadata define a set of specific constraints on the acceptable values for the property type. These constraints are taken into account in the algebraic type that is assigned to the property by means of OCL collection types. In the example, the TABLE object type is specified in the theory $\mathit{reflect}_{\text{MOF}}(\widetilde{\text{RDBMS}})$, in Maude notation, as follows:

```
sorts Table . subsort Table < Cid#rdbms . op Table : -> Table .
op schema : Oid -> Property#rdbms . op column : OrderedSet{Oid} -> Property#rdbms .
op key : [Oid] -> Property#rdbms .  op foreignKey : OrderedSet{Oid} -> Property#rdbms .
```

Object Type Specialization Relation. A *specialization* is a taxonomic relationship between two object types. This relationship specializes a general object type into a more specific one. In the RDBMS example, we algebraically define the specialization relationship between the object types $\widetilde{RModelElement}$ and $\widetilde{Table}$ as the subsorts `Table < RModelElement`. The supersorts of the resulting subsort hierarchy are defined as subsorts of the `Cid#rdbms` sort, for object type name sorts and object identifier sorts, respectively. In this way, we can define a table instance as `< 'Foo : Table | name : "date", ...>`, where the `name` property is defined for the RModelElement object type.

Algebraic Semantics of Object Types. The algebraic semantics of an object type is then given by the set of all the objects that can be defined either as instances of the object type, i.e., a class, or as instances of any of its subtypes. The algebraic semantics of an object type definition $\widetilde{\mathrm{OT}}$, such that $\widetilde{\mathrm{OT}} : \mathrm{MOF}$, is defined as follows:

$$[\![\mathrm{OT}]\!]_{\mathrm{MOF}} = \{\tilde{o} \mid \tilde{o} \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{Object}\#\mathcal{M}} \wedge \mathit{class}(\tilde{o}) \in T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{ClassSort}(\widetilde{\mathrm{OT}})}\}.$$

where *class* is an operator that obtains the type constant of a given object, and *ClassSort* is an operator that obtains the sort that corresponds to the object type constant in $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$. The *isValueOf* relation $\tilde{o} : \mathrm{OT}$ indicates if an object $\tilde{o}$ is instance of a given object type OT, is defined as follows:

$$\tilde{o} : \mathrm{OT} \;\Leftrightarrow\; \tilde{o} \in [\![\mathrm{OT}]\!]_{\mathrm{MOF}}.$$

Algebraic Semantics of MOF Metamodels. The $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$ theory constitutes the metamodel realization of the metamodel definition $\widetilde{\mathcal{M}}$. This theory provides the model type $\mathcal{M}$, which is represented by the sort $\mathit{ModelType}\{\mathcal{M}\}$. $\mathcal{M}$ is the type of collections of typed objects that have both a graph and a term structure. The semantics of the $\mathcal{M}$ type is defined by the equation

$$[\![\mathcal{M}]\!]_{\mathrm{MOF}} = T_{\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}}),\mathit{ModelType}\{\mathcal{M}\}},$$

and the *structural conformance relation* between a model definition $\widetilde{M}$ and its corresponding model type $\mathcal{M}$ is then formally defined by the equivalence

$$\widetilde{M} : \mathcal{M} \Leftrightarrow \widetilde{M} \in [\![\mathcal{M}]\!]_{\mathrm{MOF}}.$$

Embedding MOF Reflection into mel Logical Reflection. The logical reflective features of MEL [4], together with its logical framework capabilities, make it possible to *internalize* the representation $\Phi : \mathit{Spec}\mathcal{L} \longrightarrow \mathit{SpecMEL}$ of a formalism $\mathcal{L}$ in MEL, as an equationally-defined function $\overline{\Phi} : \mathit{Module}_{\mathcal{L}} \longrightarrow \mathit{Module}$, where $\mathit{Module}_{\mathcal{L}}$ is an equationally defined data type representing specifications in $\mathcal{L}$, and *Module* is the data type whose terms, of the form $\overline{(\Sigma, E)}$, metarepresent MEL specifications of the form (Σ, E). We can apply this general method to the case of our algebraic semantics

$$\mathit{reflect}_{\mathrm{MOF}} : \mathrm{MOF} \longrightarrow \mathit{SpecMEL}.$$

Then, the reflective internalization of the MOF algebraic semantics $\mathit{reflect}_{\mathrm{MOF}}$ becomes an equationally-defined function

$$\overline{\mathit{reflect}_{\mathrm{MOF}}} : \mathit{ModelType}\{\mathrm{MOF}\} \longrightarrow \mathit{Module}.$$

where *Module* is the sort whose terms represent MEL theories in the universal MEL theory (see [4]).

The function $\overline{\mathit{reflect}_{\mathrm{MOF}}}$ is completely defined in [5] and is implemented as $\overline{\mathit{reflect}_{\mathrm{MOF}}}$ in the prototype that is available at [6]. By using the theory $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathrm{RDBMS}})$, the table PERSON in the relational schema that appears in Fig. 2 is defined as the term

```
<< < 'column.0 : Column | nnv : true, owner : 'table.0, type : VARCHAR, name : "name" >
< 'column.1 : Column | owner : 'table.0, type : NUMBER, name : "age" >
< 'column.2 : Column | key : OrderedSet{ 'key.0 },
  nnv : true, owner : 'table.0, type : VARCHAR, name : "person_PK" >
< 'key.0 : Key | column : OrderedSet{ 'column.2 }, owner : 'tables.0, name : "Person_PK" >
< 'table.0 : Table | name : "Person", key : OrderedSet{'key.0},
  column : OrderedSet{ 'column.0 :: 'column.1 :: 'column.2 } > >>.
```

If we represent this term as the constant `model`, we can automatically check whether this model conforms to its metamodel by evaluating the following membership in Maude:

```
red model :: ModelType{rdbms} .
result Bool: true
```

5 Related work

The meaning of the *metamodel* notion has been widely discussed in the literature, see for example [7,8,9,10]. There is a consensus that a metamodel can play several roles: as *data*, as *type* or as *theory*. In this paper, we have formally expressed each of these roles by means of the notions of metamodel definition $\widetilde{\mathcal{M}}$, model type $\mathcal{M}$, and metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, respectively.

The current MOF standard does not provide any guidelines to implement a reflective mechanism that obtains the semantics of a metamodel. An informal attempt to realize MOF metamodel definitions as Java programs is provided in the Java Metadata Interface (JMI) specification [11], which is defined for a previous version of the MOF standard. A mapping of this kind has been successfully implemented in modeling environments such as the Eclipse Modeling Framework. By contrast, our $\mathit{reflect}_{\mathrm{MOF}}$ function gives us an executable formal specification of the algebraic semantics of any metamodel $\widetilde{\mathcal{M}}$ in MOF.

Poernomo gives a formal metamodeling framework based on Constructive Type Theory [12], where models, which are define as terms (token models), can also be represented as types (type models) by means of a reflection mechanism. In this framework, the conformance relation is implicitly provided by construction:

only valid models can be defined as terms, and their definition constitutes a formal proof of the fact that the subject belongs to the corresponding type, by means of the Curry-Howard isomorphism.

[13] describes a metamodeling framework, based on Maude, which also represents graphs as terms by taking advantage of the underlying term matching algorithm modulo associativity and commutativity. In this work, the authors address the model subtyping relation that can be defined between two model types and type inference to deal with model management scenarios. Their work is based on the data version of metamodels, i.e., over metamodel definitions. A difference compared to our work is the reflective semantics for MOF metamodels that we have defined in our framework.

6 Conclusions and Future Work

In this work we have proposed an algebraic semantics for the MOF metamodeling framework, formalizing notions not yet clear in the MOF standard. In our approach, we give an explicit formal representation for each of the different notions that may be involved in a metamodeling framework: model type $\mathcal{M}$, metamodel realization $\mathit{reflect}_{\mathrm{MOF}}(\widetilde{\mathcal{M}})$, and metamodel conformance $\widetilde{M} : \mathcal{M}$. Our work provides an algebraic executable formalization of the MOF standard that can be reused for free, in standard-compliant frameworks. At present, the prototype, available at [6], implements the MOF framework in Maude and uses the EMF as the metamodeling front-end.

We plan to use this framework as the kernel of a model management tool suite that provides support for QVT and graph-based model transformations within the EMF. For this we have relied on the experience gained in previous prototypes that gave algebraic executable specifications for OCL [14], QVT [15] and model management operators [16]. Our framework supports the application of formal analysis techniques, such as inductive theorem proving and model checking, to model-based and graph-based systems by means of the underlying Maude framework and its formal tools [4]. In addition, grammar-based software artifacts can also be related to models by specifying context-free grammars as MEL signatures. This last feature makes our framework also suitable for forward and reverse Model-Driven Engineering.

We are currently working on a graph transformation tool that provides support for OCL and QVT. This tool is being developed entirely in Maude and uses rewriting logic to support graph transformations. In future work, we plan to apply the algebraic MOF framework together with the aforementioned QVT model transformation tool and Maude-based formal verification techniques to model management scenarios, where formal verification techniques play an important role. In particular, we are considering the formal analysis of real-time embedded systems in the avionics specific domain that are developed by following a model-driven approach, by using model-based languages like the Architecture Analysis and Design Language (AADL) [17].

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

References

1. Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01) (2006) http://www.omg.org/cgi-bin/doc?formal/2006-01-01.
2. Eclipse Organization: The Eclipse Modeling Framework (2007) http://www.eclipse.org/emf/.
3. Meseguer, J.: Membership algebra as a logical framework for equational specification. In Parisi-Presicce, F., ed.: Proc. WADT'97, Springer LNCS 1376 (1998) 18–61
4. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude. Springer LNCS Vol. 4350 (2007)
5. Boronat, A., Meseguer, J.: Algebraic semantics of EMOF/OCL metamodels. Technical Report UIUCDCS-R-2007-2904, CS Dept., University of Illinois at Urbana-Champaign (2007) http://www.cs.le.ac.uk/people/ab373/papers/UIUC-TR-MOF-OCL-Boronat-Meseguer.pdf.
6. The ISSI Research Group: (The MOMENT Project) http://moment.dsic.upv.es.
7. Ludewig, J.: Models in software engineering - an introduction. Inform., Forsch. Entwickl. **18**(3-4) (2004) 105–112
8. Seidewitz, E.: What models mean. Software, IEEE **20**(5) (2003) 26–32
9. Kuhne, T.: Matters of (meta-) modeling. Software and Systems Modeling (SoSyM) **5** (December 2006) 369–385(17)
10. Rensink, A.: Subjects, models, languages, transformations. In Bézivin, J., Heckel, R., eds.: Language Engineering for Model-Driven Software Development. Volume 04101 of Dagstuhl Seminar Proceedings., Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2004)
11. Java Community Process: The Java Metadata Interface (JMI) Specification (JSR 40) (2002) http://www.jcp.org/en/jsr/detail?id=40.
12. Poernomo, I.: The meta-object facility typed. In Haddad, H., ed.: SAC, ACM (2006) 1845–1849
13. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and Tool Support for Model Driven Engineering with Maude. Journal of Object Technology **6**(9) (2007) 187–207 http://www.jot.fm/issues/issue_2007_10/paper10/.
14. Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J.A.: An Algebraic Specification of Generic OCL Queries Within the Eclipse Modeling Framework. In Rensink, A., Warmer, J., eds.: ECMDA-FA. Volume 4066 of LNCS., Springer (2006) 316–330
15. Boronat, A., Carsí, J.A., Ramos, I.: Algebraic specification of a model transformation engine. In Baresi, L., Heckel, R., eds.: FASE. Volume 3922 of Lecture Notes in Computer Science., Springer (2006) 262–277
16. Boronat, A., Carsí, J.A., Ramos, I.: Automatic Support for Traceability in a Generic Model Management Framework. In Hartman, A., Kreische, D., eds.: ECMDA-FA. Volume 3748 of LNCS., Springer (2005) 316–330
17. SAE: AADL (2007) http://www.aadl.info/.