An algebraic semantics for MOF

Artur Boronat, José Meseguer

▶ To cite this version:

Artur Boronat, José Meseguer. An algebraic semantics for MOF. Formal Aspects of Computing, 2010, 22 (3), pp.269-296. 10.1007/s00165-009-0140-9 . hal-00567269

HAL Id: hal-00567269
https://hal.science/hal-00567269

Submitted on 20 Feb 2011

Under consideration for publication in Formal Aspects of Computing

An Algebraic Semantics for MOF

Artur Boronat[1] and José Meseguer[2]

[1]Department of Computer Science, University of Leicester, UK,
[2]Department of Computer Science, University of Illinois at Urbana-Champaign, USA

Abstract. In model-driven development, software artifacts are represented as models in order to improve productivity, quality, and cost effectiveness. In this area, the Meta-Object Facility (MOF) standard plays a crucial role as a generic framework within which a wide range of modeling languages can be defined. 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 the MOF framework, and so on. However, most of these concepts are not yet fully formally defined in the current MOF standard. In this paper we define a reflective, algebraic, executable framework for precise metamodeling based on membership equational logic (MEL) that supports the MOF standard. Our framework provides a formal semantics of the following notions: *metamodel*, *model*, and *conformance* of a model to its metamodel. Furthermore, by using the Maude language, which directly supports MEL specifications, this formal semantics is *executable*. This executable semantics has been integrated within the Eclipse Modeling Framework as a plugin tool called MOMENT2. In this way, formal analyses, such as semantic consistency checks, model checking of invariants and LTL model checking, become available within Eclipse to provide formal support for model-driven development processes.

Keywords: MOF, model-driven development, membership equational logic, metamodeling semantics, reflection, formal analysis.

1. Introduction

Model-driven development is a software engineering field 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 [OMG06] offers a generic framework in which the abstract syntax of different 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 the MOF framework, and so on. However, most of these concepts are not yet fully formally defined 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, focusing on the subset *Essential MOF* (EMOF) that is commonly used for metamodeling and that has close implementations in mainstream metamodeling environments such as the

Correspondence and offprint requests to: Artur Boronat, Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, UK. e-mail: aboronat@mcs.le.ac.uk

Eclipse Modeling Framework (EMF) [Emf09]. 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 by formalizing MOF metamodels as MEL theories. Furthermore, such MEL theories are executable by rewriting. This means that useful MEL deduction capabilites become available as decision procedures to check properties about the models of a metamodel so formalized; and that all this can be achieved by executing the MEL theory formalizing a metamodel in an algebraic language such as Maude [CDE07]. In this way, the executable formal semantics of a metamodel can be used to automatically analyze the conformance of its model instances. Such model instances are characterized algebraically as terms modulo structural axioms of associativity, commutativity and identity, and have an equivalent topological characterization 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, as we explain below, it makes possible a number of very useful model management and model analysis tasks.

Our framework provides not only an algebraic semantics, but also an executable environment called MOMENT2 [Mom09], that is plugged into the Eclipse Modeling Framework (EMF) [Emf09] and that constitutes the kernel of an algebraic model management framework supporting model transformations and formal analysis techniques. In this work, we illustrate the basic principles on which MOMENT2 is based by showing how the executable formalization of MOF metamodels as MEL theories provides automated, semantics-based support for very useful model-based tasks, such as the definition of domain specific languages, model transformations, model traceability and model management operators; and formal analysis techniques, such as reachability analysis and LTL model checking of model-based systems. EMF-based technology is used for developing modeling environments and model-based software, including automated support for textual and graphical syntax and programming facilities. Our approach smoothly complements the aforementioned technology, so that already existing fully-fledged EMF-based modeling languages can be formalized and analyzed in MOMENT2 in an automatic manner.

Philosophically, we can view this work as exploiting for the area of software modeling languages the kinds of benefits that the rewriting logic semantics project [MeR07] has already demonstrated in the areas of programming languages (see [MeR04, MeR07, SRM09] and references there), and of formal specification and verification languages (see, e.g., [CDE99, StM04, CeS04]). What is common to all these efforts is the use of a flexible logical framework (either rewriting logic [Mes92], or, as done in this paper, its MEL equational sublogic [Mes98]) to formalize the semantics of either a programming language, a formal specification language, or, as done in this work, a wide range of software modeling languages. By exploiting the fact that the framework's logic is executable and has a high-performance implementation, one obtains much more than just precise semantic definitions: one obtains useful (and quite efficient) semantics-based tools, such as programming language interpreters and model checkers, theorem provers of various kinds, and, in this paper, a model management tool like MOMENT2 that is metamodel-generic and can support a wide range of model management and model analysis tasks.

The paper is structured as follows: Section 2 briefly describes the underlying formal background; Section 3 identifies important concepts that are not formally defined in the MOF standard and are usually left unspecified in most of the MOF implementations; Section 4 presents the foundations of our algebraic framework, explaining how the algebraic semantics of MOF metamodels is defined; Section 5 illustrates several applications of our framework in different model-driven development scenarios; Section 6 discusses related work; and Section 7 summarizes the main contributions of this work and discusses future work.

2. Membership Equational Logic, Rewriting Logic, and Maude

The logical framework in which we give an algebraic semantics to MOF is membership equational logic (MEL) [Mes98]. A key feature of MEL is its strong support for types (called sorts), subtypes, and partiality; and the closely-related feature that types can be very expressive. In MEL, membership in a type is not just a syntactic matter, but may depend on the satisfaction of *semantic conditions*. This MEL feature is crucially exploited in our semantics of MOF, where the set of models that conform to a given metamodel is precisely characterized by a type satisfying suitable semantic conditions. Another key feature of MEL is that it has *initial algebras* [Mes98]; therefore, our MOF semantics is an *initial algebra semantics.* Furthermore, under very reasonable assumptions MEL theories are *executable* by rewriting.

2.1. The Syntax and Semantics of MEL

A MEL *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 of function symbols, 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 $[s]$. Intuitively, membership in a sort is exactly *definedness*; whereas membership in just a kind without membership in any of the sorts of that kind means *undefinedness* or *error*. For example, let *Numeric* be a kind having sorts *Nat*, *Int*, and *Rat*. Then, in a suitable MEL specification of the number hierarchy we may have expressions like $4 - 7$ of sort *Int*, and $2/7$ of sort *Rat*. But the expression $2/0$ has kind *Numeric* but has no sort and is therefore interpreted as an undefined or error expression.

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.* In particular, T_Σ is the Σ-algebra of ground Σ-terms, and for X a set of K-kinded variables, $T_\Sigma(X)$, is the Σ-algebra of Σ-terms with variables in X. $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ denote, respectively, the set of ground Σ-terms of kind k and of Σ-terms of 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. The novel feature with respect to other equational logics is the support for conditional memberships of the form $(\forall X)\ t : s\ \ if\ \ \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$. This makes membership in a sort not just a syntactic matter, but a semantic one, since the semantic conditions $\bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$ must be satisfied.

A MEL theory is a pair (Σ, E) with Σ a MEL signature and E a set of Σ-sentences. The paper [Mes98] 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 $T_{(\Sigma/E)}$. As usual for any Σ-algebra, given a sort $s \in S$, the set $T_{(\Sigma/E),s}$ is the set of terms of sort s in $T_{(\Sigma/E)}$, which by construction is precisely the set of E-equivalence classes of ground terms $t \in T_\Sigma$ such that $E \vdash t : s$, where $\vdash$ denotes the provability relation in MEL (see [Mes98]).

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 $(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s\ \ if\ \ \bigwedge_{1\leq i\leq n} x_i : s_i$. We write $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ in place of $(\forall x_1 : k_1, \ldots, x_n : k_n)\ t = t'\ \ if\ \ \bigwedge_{1\leq i\leq n} x_i : s_i$.

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.

2.2. Rewriting Logic

Rewriting logic [Mes92, BrM06] is a flexible logic to specify concurrent systems. Any such system is specified by a rewrite theory $\mathscr{R} = (\Sigma, E, R)$, where (Σ, E) is a MEL-theory, and R is a collection of rewrite rules of the form

$$t \longrightarrow t'\ \ if\ C$$

where $t, t' \in T_{\Sigma}(X)_k$ for some kind k, and C is a condition or guard. For the purposes of this paper we assume that C is a conjunction of Σ-equations $u_1 = v_1 \wedge \ldots \wedge u_n = v_n$ (for a more general notion of condition see [BrM06]).

The intuition about $\mathscr{R}$ is that the *states* of the sytem that $\mathscr{R}$ specifies are formalized as elements of the initial algebra $T_{(\Sigma,E)}$, whereas the local state *transitions* are applications of rules in R that satisfy their condition C. Such rewrite rules transform each state containing a fragment that is an instance of the rule's left-hand side t to a new state where that fragment is replaced by the corresponding instance of t'. More complex concurrent transitions in such a system are then formalized as *proofs* in the theory $\mathscr{R}$ (see [Mes92, BrM06]).

2.3. Executable MEL Theories and Maude

The point of the decomposition $E = E_0 \cup A$ is that, under appropriate executability requirements explained in [BJM00, CDE07], such as confluence, termination, and sort-decreasingness modulo A, a 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.

An important consequence of $(S, <, \Sigma, E_0 \cup A)$ satisfying the confluence, termination and sort-decreasingness requirements is that both term equality and term membership become *decidable* by rewriting [BJM00]. Furthermore, MEL theories satisfying these executability properties can be executed in high-performance languages such as Maude [CDE07] that support execution of MEL theories modulo any combination of associativity and/or commutatitvity and/or identity axioms.

The syntax of Maude follows very closely the mathematical syntax for MEL and rewriting logic described above, including support for order-sorted notation. In the rest of the paper we will at times present fragments of MEL and rewriting logic specifications in Maude syntax. Specifically, as explained in [CDE07], MEL theories are specified in Maude as *functional modules,* whereas rewrite theories are specified as *system modules.* Such syntax is essentially self-explanatory, but we give here a few explanations to help the reader. First of all, basic notions such as sorts, subsorts, operators, equations, rules and memberships are declared with respective keywords `sort`, `subsort`, `op`, `eq` (or `ceq` for conditional equations), `rl` (or `crl` for conditional rules) and `mb` (or `cmb` for conditional memberships). Second, associativity and/or commutativity and/or identity axioms are not declared explicitly as equations, but are instead declared as attributes of the binary operator enjoying them in the corresponding `op` declaration with the `assoc`, `comm`, and `id:` keywords. Third, the syntax of an operator need not be prefix syntax: it can also be *user-definable* "mixfix" syntax, where the argument places are indicated by underbars. For example, a user can define an if-then-else operator with syntax `if_then_else_fi`.

3. MOF and its Semantic Issues

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. We also discuss several semantic issues that are then addressed by the formal algebraic semantics presented in subsequent sections.

3.1. The MOF Modeling Framework

MOF is a semiformal approach to define modeling languages usually presented as a four-level hierarchy, with levels M0, M1, M2 and M3. Each entity M at level i represents a *model*[1]. M consists of a collection of *typed* elements e, which can be simple data values or objects. Types T for the elements e that constitute a model M at level i are defined as collections of elements in a *metamodel* $\mathscr{M}$ at level $i+1$. There are two different kinds of types T: data types and object types. We indicate that an element e is *typed* with a type

[1] 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.

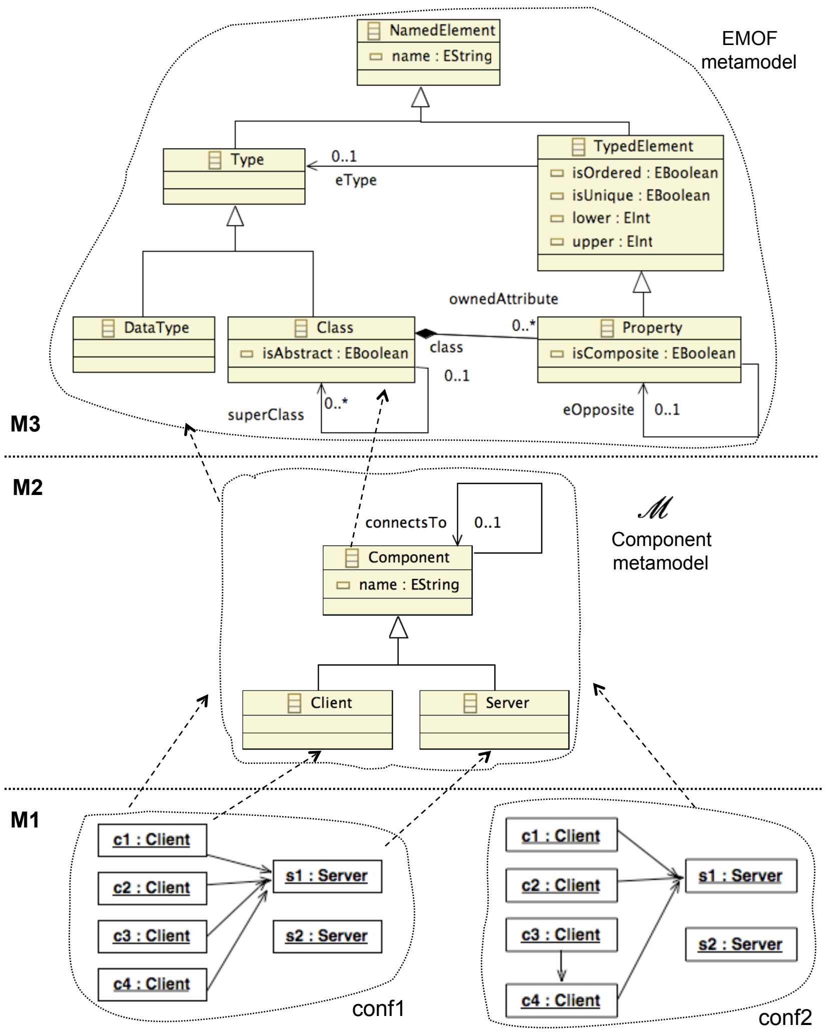


Fig. 1. The MOF framework.

T as $e : T$. This relation is specialized for simple data values as the *isValueOf* relation, and for objects as the *isInstanceOf* relation. In addition, a model M is said to *conform* to a metamodel $\mathscr{M}$ [Bez05], when the elements that constitute the model are properly typed. We call this relation *conformance relation* between a model M and a metamodel $\mathscr{M}$ and we denote it by $M : \mathscr{M}$.

Fig. 1 illustrates models at each level M1-M3 of the MOF framework. Each model is encircled by a boundary and tagged with a name. For example, `conf1` at level 1, which is a model corresponding to a configuration of components. The *isInstanceOf* relation between objects of a model M and the metarepresentation of object types in a metamodel $\mathscr{M}$, and the *conformance* relation between a model M and the corresponding metamodel $\mathscr{M}$ are depicted with dashed arrows. We consider levels M1–M3 out of the MOF hierarchy in this work, which are:

M1 level. This level contains metarepresentations of *models*. A model is a collection of *objects* 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, the model `conf1` in Fig. 1 defines a configuration of client/server components describing how they are interconnected. We use object diagram notation for representing models, making their graph structure explicit. Each object corresponds to a component indicating whether it is a client or a server.

M2 level. This level contains metarepresentations of *metamodels*, usually given as class diagrams. A metamodel is a model specifying the abstract syntax of a modeling language and defines the types that can be used for defining objects in a model by means of UML-like classes. As an example, we consider a metamodel for defining configurations of components as shown in Fig. 1 in UML notation. The types of a configuration are defined by the classes `Component`, `Server` and `Client`. The `Component` class is *abstract* so that it cannot be instantiated. This class is specialized into a `Client` and a `Server` classes. The model `conf1` at level 1 conforms to the component metamodel $\mathscr{M}$ at level 2.

M3 level. An entity at level 3 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 (usually also called the metamodel MOF). Within the MOF standard, we focus on the *Essential MOF* (EMOF) specification that describes the meta-metamodel, which appears simplified in Fig. 1. In the MOF modeling framework one can define many different metamodels. Such metamodels, when represented as data, must conform to the MOF meta-metamodel. That is, a metamodel, such as the component metamodel, is a model that conforms to the meta-metamodel MOF. Other metamodels that are likewise specified in MOF are the UML2 metamodel to define UML models, the OWL metamodel to define ontologies [Lac05], the AADL metamodel to specify real-time distributed computer systems [SAE07], and so on. The fact that all these metamodels are specified within the single MOF framework greatly facilitates systematic model/metamodel interchange and integration.
The MOF hierarchy is closed at its top level, because the MOF meta-metamodel itself is also defined as a model that conforms to itself. The class diagram notation is indeed used as a convenient concrete graphical syntax at levels 2 and 3 for hidding the verbose representation of a model, which could likewise be given as an object diagram. Fig. 2 in Section 4.3 shows the model that constitutes the component metamodel as an object diagram.

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 of a metamodel $\mathscr{M}$, but there is no explicit type that permits defining $\mathscr{M}$ 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 $\mathscr{M}$ does not have a precise *mathematical* status. Instead, at best, a metamodel $\mathscr{M}$ is realized as a program in a conventional language as, for example, the Java code that is generated for a metamodel $\mathscr{M}$ in EMF. This informal implementation corresponds to what we call a *metamodel realization.* In these modeling environments, the *conformance relation* between a model definition M and its corresponding metamodel $\mathscr{M}$ is checked by means of indirect techniques based on XML document validation or on tool-specific implementations in OO programming languages. Therefore, metamodels $\mathscr{M}$ and models 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) metamodel realization, (ii) model type, and (iii) conformance relation, by means of a reflective semantics that associates a mathematical metamodel realization to each metamodel $\mathscr{M}$ 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 possibly 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 $\mathscr{M}$ describes a metamodel realization that contains a model type. What this metamodel describes is, of course, a *set* of models. We call this the *extensional* semantics of $\mathscr{M}$, and denote this semantics by $[\![\mathscr{M}]\!]$. Recall that we use the notation $M : \mathscr{M}$ for the conformance relation. Using this notation, the extensional semantics can be informally defined as follows:

$$[\![\mathscr{M}]\!] = \{M \mid M : \mathscr{M}\}.$$

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

$$\mathbb{A} : [\![\mathrm{MOF}]\!] \longrightarrow \mathit{SpecMEL}$$

that associates to each MOF metamodel $\mathscr{M}$ a corresponding MEL specification $\mathbb{A}(\mathscr{M})$. The function $\mathbb{A}$ is defined in detail in Sections 4.2–4.4. Our informal semantics $[\![\mathscr{M}]\!]$ is thus 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 $\mathbb{A}(\mathscr{M})$, there is always a sort called *Model*, whose data elements in the initial algebra are precisely the data representations of those models that conform to $\mathscr{M}$. That is, *Model* is the syntactical representation of the *model type* $[\![\mathscr{M}]\!]$ associated to a metamodel $\mathscr{M}$. Therefore, after $\mathbb{A}$ is defined, we can give a precise mathematical semantics to our informal MOF extensional semantics by the defining equation

$$[\![\mathscr{M}]\!] = T_{\mathbb{A}(\mathscr{M}),Model}.$$

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 $[\![\mathscr{M}]\!]$, (ii) metamodel realization $\mathbb{A}(\mathscr{M})$, and (iii) conformance relation $M : \mathscr{M}$. Specifically, we associate to a metamodel $\mathscr{M}$ a precise mathematical object, namely, the MEL theory $\mathbb{A}(\mathscr{M})$, constituting its *metamodel realization.* The *structural conformance* relation between a model and its metamodel is then defined mathematically by the equivalence

$$M : \mathscr{M} \quad \Leftrightarrow \quad M \in T_{\mathbb{A}(\mathscr{M}),Model}.$$

We will summarize again our formal semantics of all the key MOF concepts in Section 4.4.1 after all details of the function $\mathbb{A}$ have been made precise in Sections 4.2–4.4.

4.2. Reflective Bootstrapping of the Algebraic Semantics

The algebraic semantics that we propose exploits the reflective features of both MOF and MEL. This allows a modular, stepwise approach in the definition of the semantic function $\mathbb{A}$. This has many advantages, both theoretically and in the practical realization of $\mathbb{A}$ in the MOMENT2 tool.

The key observation about reflection in MOF is that the MOF meta-meta-model at level 3 is also a meta-model at level 2, which can be treated just as any other metamodel. In particular this means that

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

It also means that our algebraic semantics function $\mathbb{A} : [\![\mathrm{MOF}]\!] \longrightarrow \mathit{SpecMEL}$ applies in particular to MOF, that is, that there is a MEL theory $\mathbb{A}(\mathrm{MOF})$.

The MEL theory $\mathbb{A}(\mathrm{MOF})$ is enormously useful to bootstrap our algebraic semantics because of the following remarkable property. Since for any metamodel $\mathscr{M}$ we have the identity $[\![\mathscr{M}]\!] = T_{\mathbb{A}(\mathscr{M}),\mathit{Model}}$, in particular for the meta-metamodel MOF we have the identity

$$[\![\mathrm{MOF}]\!] = T_{\mathbb{A}(\mathrm{MOF}),\mathit{Model}}.$$

This suggests the following bootstrapping strategy to define the semantic function $\mathbb{A} : [\![\mathrm{MOF}]\!] \longrightarrow \mathit{SpecMEL}$.

1. First define the MEL theory $\mathbb{A}(\mathrm{MOF})$. This automatically gives us the domain $[\![\mathrm{MOF}]\!]$ of our desired semantic function $\mathbb{A} : [\![\mathrm{MOF}]\!] \longrightarrow \mathit{SpecMEL}$ as the algebraic data type $[\![\mathrm{MOF}]\!] = T_{\mathbb{A}(\mathrm{MOF}),\mathit{Model}}$.
2. Once the domain $[\![\mathrm{MOF}]\!]$ of $\mathbb{A}$ is thus algebraically represented, proceed to give a recursive definition of the semantic function $\mathbb{A}$ for any $\mathscr{M} \in [\![\mathrm{MOF}]\!]$.

Step (1) is described in Section 4.3, and Step (2) in Section 4.4.

There is, however, a third very important step, namely, to also exploit reflection in the target logical framework MEL. Reflection in MEL means that there is a *universal* MEL theory $\mathcal{U} \in \mathit{SpecMEL}$ that can simulate the deduction of all finitary MEL theories, including its own deduction [CMP07].

In particular, the universal theory $\mathcal{U}$ has a sort *Module* that meta-represents all finitary MEL theories, including $\mathcal{U}$ itself. This means that there is an algebraic data type $\overline{\mathit{SpecMEL}}$ whose elements are meta-representations of MEL theories, so that given any finitary MEL theory (Σ, E), we have $(\Sigma, E) \in \mathit{SpecMEL}$ iff $\overline{(\Sigma, E)} \in \overline{\mathit{SpecMEL}}$, where $\overline{(\Sigma, E)}$ is the meta-representation of (Σ, E). Specifically, the algebraic data type $\overline{\mathit{SpecMEL}}$ has the following initial algebra semantics definition:

$$\overline{\mathit{SpecMEL}} = T_{\mathcal{U},\mathit{Module}}.$$

All this means that we can take a third step in our bootstrapping process, namely, to realize our semantic function $\mathbb{A}$ as an *equationally defined function*

$$\overline{\mathbb{A}} : [\![\mathrm{MOF}]\!] \longrightarrow \overline{\mathit{SpecMEL}}$$

mapping each metamodel $\mathscr{M}$ to the metarepresentation $\overline{\mathbb{A}(\mathscr{M})}$ of the MEL theory $\mathbb{A}(\mathscr{M})$.

At the theoretical level, this third bootstrapping step means that the entire algebraic semantics of MOF can be defined *within* the semantic framework of MEL. At the practical level it also means that, because of the efficient support in Maude for key functionality of the universal MEL theory $\mathcal{U}$ by means of Maude's `META-LEVEL` module, the function $\overline{\mathbb{A}}$ can be efficiently implemented in Maude. In fact, this is exactly how the core functionality of the MOMENT2 tool has been developed, as explained in Section 4.4.

In Sections 4.3 and 4.4 we give a high level summary of the definition of the $\mathbb{A}$ semantic function, as achieved by the three reflective bootstrapping steps described above. A complete definition of $\mathbb{A}$ is available in [Bor07][2]. In Section 4.5 we explain how $\mathbb{A}$ is used in the EMF by means of MOMENT2.

4.3. The MEL Theory $\mathbb{A}$(MOF)

As already mentioned, we denote the metamodel that constitutes the meta-metamodel of the MOF framework by MOF. In particular, we focus on the subset EMOF of the MOF standard. An excerpt of the metamodel EMOF, extracted from the core specification of the standard MOF 2.0 [OMG06] is shown in Fig. 1. MOF is itself a MOF metamodel, since MOF : MOF. In this section we take the first bootstrapping step described in Section 4.2, namely, to define the MEL theory $\mathbb{A}(\mathrm{MOF})$. That is, we first define $\mathbb{A}$ for a *single* metamodel, namely MOF. The $\mathbb{A}(\mathrm{MOF})$ theory defines the $[\![\mathrm{MOF}]\!]$ type as the set of metamodels $\mathscr{M}$, which can be viewed as both graphs and terms. This theory has been manually defined in [Bor07] as a first step in the bootstrapping process needed to define the $\mathbb{A}$ function in general. The $\mathbb{A}(\mathrm{MOF})$ theory provides the algebraic representation for both object types and model types for defining metamodels $\mathscr{M}$. The full specification of $\mathbb{A}(\mathrm{MOF})$ as a membership equational theory is given in Maude notation as a functional module [Bor07]. As already explained in Section 2.3, Maude functional modules are *exactly* MEL theories with an initial algebra

[2] In the dissertation, the $\mathbb{A}$ function is denoted by $\mathit{reflect}_{\mathrm{MOF}}$.

semantics. The Maude notation for sorts, subsorts, operations, equations and memberships is self-explanatory and typewriter variant of the mathematical notation [CDE07]. In what follows we present some fragments of the theory fully specified in [Bor07], in Maude notation, with special emphasis on its sorts, subsorts and operations.

In the theory $\mathbb{A}$(MOF), object types are used to describe a metamodel $\mathscr{M}$: MOF as a collection of objects. Objects are defined by using the following sorts: `Oid` for object identifiers; `Cid` for class names; and `PropertySet` for multisets of comma-separated pairs of the form `(property : name = value)`, which represent property values. Objects in a metamodel $\mathscr{M}$ are then syntactically characterized by means of an operator

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

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

```
sorts Oid Cid Property PropertySet Object .
subsort Property < PropertySet .
op noneProperty : -> PropertySet .
op _`,_ : PropertySet PropertySet -> PropertySet
    [assoc comm id: noneProperty] .
op property`:_ : String -> Property .
op property`:_=_ : String String -> Property .
op property`:_=_ : String Int -> Property .
op property`:_=_ : String Float -> Property .
op property`:_=_ : String Bool -> Property .
op property`:_=_ : String Collection+{Oid} -> Property .
```

In the $\mathbb{A}$(MOF) theory, a metamodel $\mathscr{M}$ that conforms to the meta-metamodel MOF, that is, such that $\mathscr{M}$: MOF, can be represented as a collection of objects by means of a term of sort `Model`. A term of sort `Model` is defined by means of the following constructors, in Maude notation:

```
sorts ObjCol Model . subsort Model < ObjCol .
op none : -> ObjCol .
op __ : ObjCol ObjCol -> ObjCol [assoc comm id: none] .
op <<_>> : ObjCol -> [Model] .
```

That is, we first form a collection of objects of sort `ObjCol` using the associative and commutative multiset union operator `__`[3] and then we *wrap* the set of objects by using the `<<_>>` constructor to get the desired term of sort `Model`. Note that membership in the sort `Model` is not defined just by the constructor `<<_>>` (only membership in the *kind* `[Model]` is implied by the above operator declaration). Instead, membership of a term in the sort `Model` is achieved by conditional membership axioms that are given below after introducing the formal definition of the object types *Class* and *Property*. Each of these modeling primitives is specified in the $\mathbb{A}$(MOF) theory by means of sorts, subsorts and operators, in Maude notation as follows.

***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 $\mathscr{M}$ as a *Class* instance and a set of *Property* instances. 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`, and a constant with the operator `op Class : -> Class .` Terms of sort `Property` in the theory $\mathbb{A}(\mathscr{M})$ enable the definition of property values in objects that form part of a metamodel.

In the component metamodel, the *Class* instance that defines the object type `Component` in the metamodel $\mathscr{M}$ is defined as the term

```
< 'class0 : Class | property : "name" = "Component",
                    property : "isAbstract" = true,
                    property : "ownedAttribute" = ... >
```

where `'class0` is an object identifier.

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

***Property* object type.** A model can be viewed as a graph where the collection of nodes consists of the collection of attributed objects of the model and the edges are defined by means of directed links between objects, defined as object-typed property values. This graph structure is usually represented with object diagram notation as in Fig. 1. Indeed, a model is an enriched graph where edges can also be defined as structural containments so that hierarchies of nested objects can also be represented.

A *Property* instance in a metamodel $\mathscr{M}$ enables the definition of an *attribute* in an object or an *association end* between objects in a model definition $M : \mathscr{M}$, one level down in the MOF framework. A *Property* instance defines the type of the property, where the type is represented as a *DataType* instance (basic data values and enumerations) or as a *Class* instance (object types). Properties that are typed with datatypes are *attributes*, and properties that are typed with object types are *association ends* or *references.* An association end defines a unidirectional association between two classes[4]. Bidirectional associations are defined by means of two *opposite* association ends. Composition associations can be defined by indicating that the association end that points to the composite class has its meta-property *isComposite* set to *true.* Other meta-properties, such as *lower*, *upper*, *isOrdered* and *isUnique*, 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[5]:

```
sort Prop . subsort Prop < TypedElement .
op Prop : -> Prop .
```

The *Property* instance that defines the metaproperty *name* of the object type `Component` in the metamodel $\mathscr{M}$ of the example is represented by the term

```
< 'prop : Prop | property : "name" = "name",
                 property : "type" = 'String,
                 property : "class" = 'class0 >.
```

where `'String` is the identifier of the object that represents the built-in String type. We have taken into account the modeling primitives that constitute the metamodel Essential MOF, including simple data types and enumeration types. A detailed specification is provided in [Bor07]. The metamodel $\mathscr{M}$ of the example is defined as a term of sort `Model` in the $\mathbb{A}$(MOF) theory in Fig. 2.

MOF model type. In the theory $\mathbb{A}$(MOF), the sort `ObjCol` denotes collections of objects that are instances of the classes in the metamodel MOF and the sort `Model` denotes the set of collections of objects that are models that conform to a given metamodel by satisfying a number of constraints. In particular, we consider that an object collection is a model if: *(*i) there are no duplicate identifiers for different objects, *(ii)* properties are only set once in an object tuple, *(iii)* there are no dangling references (that is, a model is a well-formed graph), *(iv)* an object can only have one container according to the composite association ends in the metamodel (that is, a model is a hierarchical graph), and *(v)* values in properties correspond to the type of the corresponding property definition in the metamodel.

The sort `Model` is defined with the following memberships, which provide the semantics of the model type for the metamodel MOF:

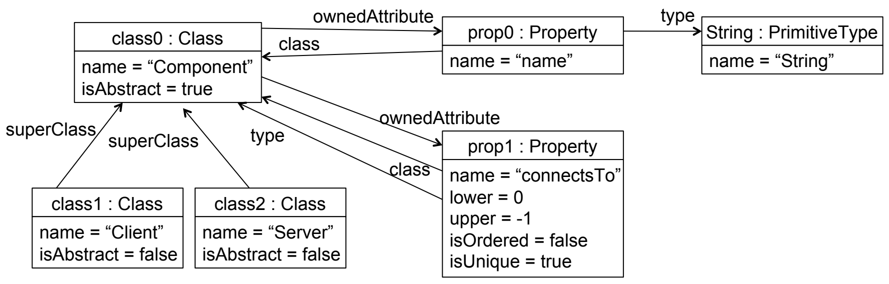
```
mb << none >> : Model  .
cmb << OC >> : Model if ( OC =/= none ) and uniqueOid( OC )
  and noDuplicateProperties( OC ) and noDanglingEdges( OC )
  and singleContainer( OC ) and validProperties( OC ) .
```

The predicates used in the condition of the second membership axiom are defined in the Appendix A as equationally defined boolean functions, where the predicates correspond to the above-mentioned constraints, in order of appearance.

The theory $\mathbb{A}$(MOF) is the metamodel realization of MOF. The carrier of the sort `Model`, in the theory $\mathbb{A}$(MOF), in the initial algebra $T_{\mathbb{A}(\mathrm{MOF})}$ defines the $[\![\mathrm{MOF}]\!]$ model type, i.e., the model type whose elements

[4] Association ends are defined, in the MOF standard [OMG06], as properties that are owned by the class from which they can be navigated. This notion was called *Reference* in previous versions of the MOF standard and it is called *EReference* in the EMF.

[5] We represent the name of the *Property* class as `Prop` to avoid name collisions with the aforementioned sort *Property.*



```
<< < 'class0 : Class | property : "name" = "Component",
     property : "isAbstract" = true, property : "superClass" = empty-set,
     property : "ownedAttribute" = OrderedSet{'prop0 :: 'prop1} >
   < 'prop0 : Prop | property : "name" = "name",
     property : "type" = 'String, property : "class" = 'class0 >
   < 'prop1 : Prop | property : "name" = "connectsTo",
     property : "type" = 'class0, property : "class" = 'class0,
     property : "lower" = 0, property : "upper" = -1,
     property : "isOrdered" = false, property : "isUnique" = true >
   < 'class1 : Class | property : "name" = "Client",
     property : "isAbstract" = false, property : "superClass" = Set{ 'class0 },
     property : "ownedAttribute" = empty-orderedset >
   < 'class2 : Class | property" = "name" = "Server",
     property : "isAbstract" = false, property : "superClass" = Set{ 'class0 },
     property : "ownedAttribute" = empty-orderedset >
   < 'String : PrimitiveType | property : "name" = "String" > >>
```

Fig. 2. Metamodel $\mathscr{M}$ viewed as a graph and as a term.

are metamodels:

$$[\![\text{MOF}]\!] = T_{\mathbb{A}(\text{MOF}),Model}.$$

Note that, since $[\![\text{MOF}]\!]$ is the set of all metamodels $\mathscr{M}$ in MOF, this means that

$$\text{MOF} \in [\![\text{MOF}]\!].$$

4.4. Reflective Algebraic Semantics of MOF Metamodels

Once the $\mathbb{A}(\text{MOF})$ theory is provided, we define the value of the function $\mathbb{A}$ on *any* metamodel $\mathscr{M}$, such that $\mathscr{M} \in [\![\text{MOF}]\!]$, as its corresponding MEL theory $\mathbb{A}(\mathscr{M})$. This corresponds to the second bootstrapping step described in Section 4.2. Given a metamodel $\mathscr{M}$, the $\mathbb{A}(\mathscr{M})$ theory defines the $[\![\mathscr{M}]\!]$ semantics as the set of models M that consist of a collection of typed objects, which can be viewed as a graph where objects correspond to nodes, object attributes to node attributes and object-typed properties to edges. In addition, nodes can be nested by means of object-typed properties that are defined as *isComposite*. The metamodel $\mathscr{M}$ of a model M corresponds to the type graph of the graph M, where object types constitute node types enabling node inheritance by means of class inheritance relationships.

In the $\mathbb{A}(\mathscr{M})$ theory, the algebraic notion of *object type* is generically given by means of the sort *Object*. Terms of sort *Object* are defined by means of the constructor

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

which is analogous to the constructor for objects that has been presented in the $\mathbb{A}(\text{MOF})$ theory.

Models $M : \mathscr{M}$ are given as collections of objects, which are instances of a specific class with name C. A class in a metamodel is an instance of the class *Class* in MOF. Each class may contain a number of properties, defined as instances of the class *Property* in MOF, as explained in Section 4.3. The object type $[\![C]\!]$ that corresponds to a class C is the set of objects that can be defined as instances of the class C. 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 $M : \mathscr{M}$. Class inheritance and the corresponding object type specialization relationships must be also taken into account. Therefore, we need to define the carrier of the sorts `Oid`, `Cid` and `PropertySet` for a specific object type definition.

Consider, for example, the metamodel $\mathscr{M}$ in Fig. 1. In subsequent paragraphs, we use this example to obtain the theory that defines the `Component`, `Server` and `Client` object types.

Class Names. In the $\mathbb{A}(\mathscr{M})$ theory, each *Class* definition with name C in $\mathscr{M}$ (except for abstract classes) is defined as a new sort also named C and declared to be a subsort of `Cid`, and C is also declared as a constant by giving an operator declaration `op C : -> C .` *Abstract classes* are defined as those that cannot be instantiated. For an abstract class C, only the sort C is declared as before. The name of an abstract class C is *not* specified as a constant. In this way, when C is an abstract class, objects in a model M, such that $M : \mathscr{M}$, cannot have C as their type.

In the example of the component metamodel, the $\mathbb{A}$ function generates a sort for the abstract object type `Component` in Maude notation as follows:

```
sort Component .
```

The concrete object types `Client` and `Server` are formalized as sorts and constants as follows:

```
sorts Client Server .
op Client : -> Client .
op Server : -> Server .
```

The class name for a given object can be obtained by means of the operator `class` defined as follows:

```
op class : Object -> Cid .
eq class(< O : C | PS >) = C .
```

Object Type Specialization Relation. A *specialization* is a taxonomic relationship between two object types and is represented as a class inheritance relationship between a subclass and a superclass, where multiple inheritance is allowed. This relationship specializes a general object type into a more specific one and is formalized in Maude as a subsort relationship. In the example, we algebraically define the inheritance relationships between the classes `Client`, `Server` and `Component` by means of the subsorts

```
subsorts Client Server < Component .
```

The supersorts of the resulting subsort hierarchy are defined as subsorts of the sort `Cid`.

```
subsort Component < Cid .
```

Object Type Properties. A class is defined with a collection of *Property* instances describing its meta-properties. A Property instance is associated with a specific type t in the metamodel $\mathscr{M}$, which is defined as an object $t : \mathit{Type}$. Depending on the type 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 enumeration type. Value-typed properties define the attributes of nodes in a graph.
- *Object-typed Properties or Association Ends.* Properties of this kind are typed with object types, enabling the definition of unidirectional labelled edges in a graph.

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, as indicated in Table 1. In this table, the `Set{Oid}` sort corresponds to sets of identifiers that may be empty by means of the constant `empty-set`, while the `NeSet{Oid}` sort corresponds to sets of identifiers that cannot be empty, i.e., it excludes the constant `empty-set`. In the example, the `Component` object type is specified in the theory $\mathbb{A}(\mathscr{M})$ with the above sorts and subsorts and with the operators for defining properties.

4.4.1. Summary of the MOF Algebraic Semantics

The function $\mathbb{A} : [\![MOF]\!] \longrightarrow \mathit{SpecMEL}$ specified in Sections 4.2–4.4 provides the basics for associating to each basic, informal concept about MOF metamodels a corresponding formal definition. This was already sketched out in Section 4.1, but is now made even more explicit for the following concepts: *(i)* metamodel realization, *(ii)* object type, *(iii)* model type, and *(iv)* conformance of a model to a metamodel.

Table 1. OCL types for encoding multiplicity constraints in properties.

Type	Lower Bound	Upper Bound	isOrdered	isUnique
$\emptyset$, `Oid`	0	1	-	-
`Set{Oid}`	0	*	false	true
`OrderedSetOid}`	0	*	true	true
`Bag{Oid}`	0	*	false	false
`Sequence{Oid}`	0	*	true	false
`Oid`	1	1	-	-
`NeSet{Oid}`	1	*	false	true
`NeOrderedSetOid}`	1	*	true	true
`NeBag{Oid}`	1	*	false	false
`NeSequence{Oid}`	1	*	true	false

Definition 1. Given a MOF-conformant metamodel $\mathscr{M}$: MOF, and given an object class named C in $\mathscr{M}$:

(i) The metamodel realization of $\mathscr{M}$ is the MEL-theory $\mathbb{A}(\mathscr{M})$.

(ii) The *object type* of the class C is the set

$$[\![C]\!] = \{o \mid o \in T_{\mathbb{A}(\mathscr{M}),Object} \ \wedge \ class(o) \in T_{\mathbb{A}(\mathscr{M}),C}\},$$

that is, the elements of the object type $[\![C]\!]$ are all the object instances of the class C and of all its subclasses in the inheritance relation specified in $\mathscr{M}$.

(iii) The *model type* of $\mathscr{M}$ is the set

$$[\![\mathscr{M}]\!] = T_{\mathbb{A}(\mathscr{M}),Model},$$

that is, each model M of $\mathscr{M}$ is algebraically represented as an element $M \in [\![\mathscr{M}]\!]$, i.e., as an equivalence class of terms of sort *Model* modulo the equations and memberships of $\mathbb{A}(\mathscr{M})$.

(iv) The *metamodel conformance* relation between a model M and its metamodel $\mathscr{M}$, denoted $M : \mathscr{M}$, is, by definition, the memberhip relation $M \in [\![\mathscr{M}]\!]$, that is, we have

$$M : \mathscr{M} \Leftrightarrow M \in [\![\mathscr{M}]\!].$$

4.4.2. Embedding MOF Reflection into MEL Logical Reflection.

We are now ready to discuss the third, final bootstrapping step, mentioned in Section 4.2. The logical reflective features of MEL [CDE07, CMP07], particulary its universal theory $\mathcal{U}$, make it possible to *internalize* the representation function mapping the specifications of some other formalism to MEL as an equationally defined function *within* MEL (see [CDE99]). In particular this can be done for our representation function for MOF

$$\mathbb{A} : [\![\mathrm{MOF}]\!] \longrightarrow \mathit{SpecMEL}.$$

Note that the domain of the function $\mathbb{A}$, since it is the algebraic data type $[\![\mathrm{MOF}]\!] = T_{\mathbb{A}(\mathrm{MOF}),Model}$ defined by initial algebra semantics in Section 4.3, is already internalized within MEL. However, the set of (finitary) MEL specifications *SpecMEL* is a metalevel entity and therefore *outside* the object level of MEL, that is, not *directly* definable as an algebraic data type in MEL. As explained in detail in [CMP07], this set is *representable* at the object level inside the universal theory $\mathcal{U}$ as the set of elements of sort *Module* in its initial algebra $T_{\mathcal{U}}$. That is, we have a faithful representation mapping

$$\mathit{SpecMEL} \longrightarrow \overline{\mathit{SpecMEL}} : (\Sigma, E) \mapsto \overline{(\Sigma, E)}$$

where, by definition, the set $\overline{\mathit{SpecMEL}}$ is the algebraic data type

$$\overline{\mathit{SpecMEL}} = T_{\mathcal{U},Module}$$

corresponding to the elements of sort *Module* in the initial algebra of the universal theory $\mathcal{U}$. Then, the reflective internalization of the MOF algebraic semantics $\mathbb{A}$ becomes an equationally defined function

$$\overline{\mathbb{A}} : [\![\mathrm{MOF}]\!] \longrightarrow \overline{\mathit{SpecMEL}} : \mathscr{M} \mapsto \overline{\mathbb{A}(\mathscr{M})}$$

so that the term $\overline{\mathbb{A}(\mathscr{M})}$, for a specific metamodel $\mathscr{M}$, metarepresents the MEL theory $\mathbb{A}(\mathscr{M})$.

But since the universal theory $\mathcal{U}$ faithfully simulates all the deductive capabilities of an object level theory (Σ, E) by means of its meta-representation $\overline{(\Sigma, E)}$ as data [CMP07], this means that, given a metamodel $\mathscr{M}$ in MOF, anything we can do with its associated MEL theory $\mathbb{A}(\mathscr{M})$ we can likewise perform at the metalevel with its meta-representation as data $\overline{\mathbb{A}(\mathscr{M})}$. Furthermore, in an efficient implementation of the relevant deductive functionality of $\mathcal{U}$ such as the one provided in Maude's `META-LEVEL` module [CDE07], the difference in computation time between performing a deduction in $\mathbb{A}(\mathscr{M})$ or performing the analogous deduction at the metalevel with $\overline{\mathbb{A}(\mathscr{M})}$ is negligible in practice.

In this way, the term $\overline{\mathbb{A}(\mathscr{M})}$, that for a specific metamodel $\mathscr{M}$ meta-represents the MEL theory $\mathbb{A}(\mathscr{M})$, can then be used in different model-driven development scenarios, as illustrated in Section 5. The function $\mathbb{A}$ is completely defined in [Bor07] and is implemented as $\overline{\mathbb{A}}$ in MOMENT2. That the function $\overline{\mathbb{A}}$ is equationally definable follows from the general principle that $\mathbb{A}$ is a computable function and that all computable functions can be equationally defined by a finite set of confluent and terminating equations [BeT80]. In practical terms, this is just a matter of representing all the recursive definitions given for in $\mathbb{A}$ earlier in Section 4.4 as corresponding equational definitions at the metalevel.

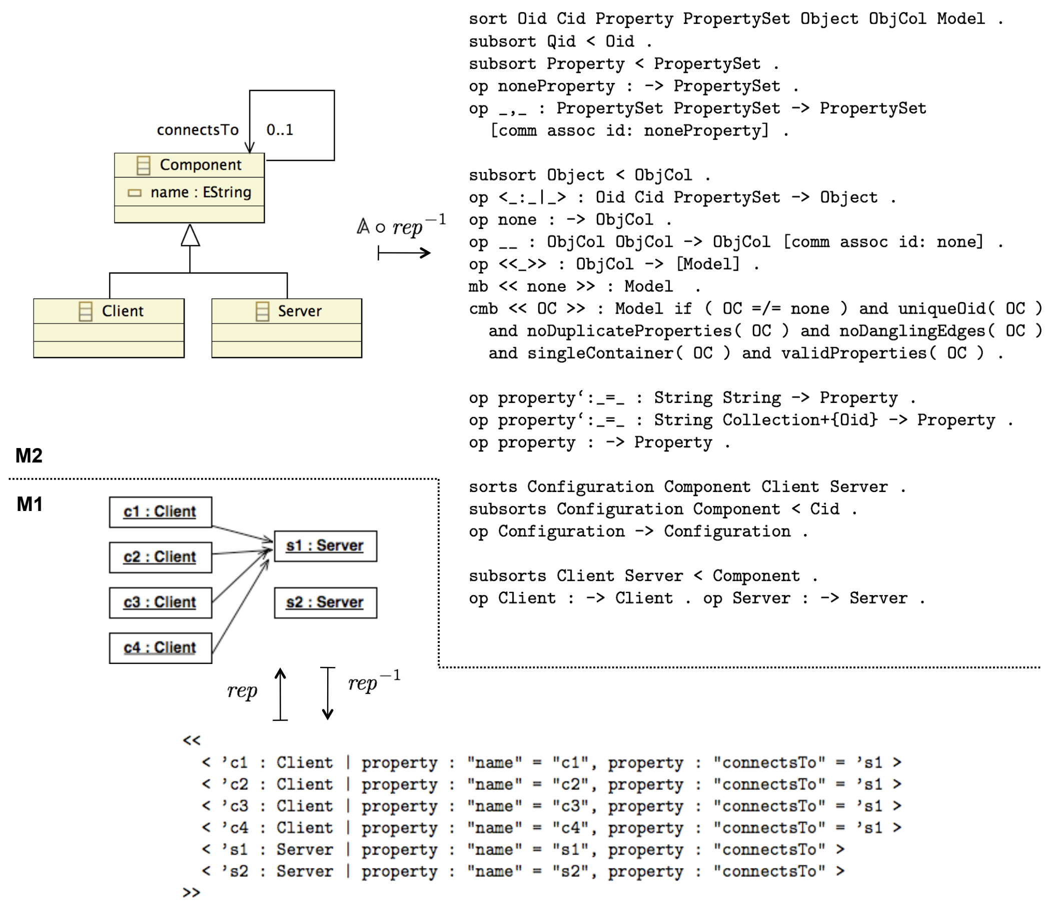
4.5. MOMENT2 and the Eclipse Modeling Framework

MOMENT2 is a suite of tools, built on top of the Eclipse Modeling Framework (EMF), that provides support for formal model-driven development. The mathematical foundations of this tool suite directly rely on the algebraic semantics $\mathbb{A}$ of MOF metamodels, presented in this work, which can be neatly exploited in the EMF, a widely used MOF-like metamodeling framework, by means of OMG standards, such as MOF, OCL and QVT. In this way, the formal analysis techniques that are presented in Section 5 can be applied to model-driven development practices in industrial environments, such as Rational Software Architect [RSA09] or OSATE AADL [SAE07]. In particular, MOMENT2 provides support for verifying metamodel conformance with OCL constraint satisfaction [BoM09], QVT-like model transformations and their verification based on Maude's reachability analysis and Maude's LTL model checker, as explained in [BHM09].

The integration of MOMENT2 into the EMF is based on a metamodel-independent, automatic bridge[6] between EMF and Maude. For each metamodel $\mathscr{M}$, the key components of this bridge are: *(i)* a model representation function $rep : [\![\mathscr{M}]\!] \longrightarrow [\![\mathscr{M}]\!]_{EMF}$, which represents each $\mathscr{M}$-conformant model $M \in [\![\mathscr{M}]\!]$ in the algebraic semantics as an EMF model instance[7] $rep(M) \in [\![\mathscr{M}]\!]_{EMF}$, where $[\![\mathscr{M}]\!]_{EMF}$ denotes the set of $\mathscr{M}$-conformant models in their EMF representation; and *(ii)* its inverse function $rep^{-1} : [\![\mathscr{M}]\!]_{EMF} \longrightarrow [\![\mathscr{M}]\!]$, which converts each EMF model instance to its corresponding algebraic representation $rep^{-1}(M_{EMF}) \in [\![\mathscr{M}]\!]$. Note that only the set $[\![\mathscr{M}]\!]$ has been given a formal representation. Therefore, the set $[\![\mathscr{M}]\!]_{EMF}$ and the functions rep and rep^{-1} are respectively realized by the EMF system and Java code in the MOMENT2 implementation. However, the bridge provided by the (rep, rep^{-1}) pair is quite robust, because the conformance relation is checked for $M_{EMF} \in [\![\mathscr{M}]\!]_{EMF}$ by EMF, and for $M \in [\![\mathscr{M}]\!]$ by Maude. Up to renaming of object identifiers, the functions rep and rep^{-1} are inverse of each other. Specifically, for each $M_{EMF} \in [\![\mathscr{M}]\!]_{EMF}$ we have $rep^{-1}(rep(M_{EMF})) = M_{EMF}$ and for each $M \in [\![\mathscr{M}]\!]$, $rep(rep^{-1}(M))$ differs from M only in a renaming of its object identifiers, due to the internal management of object identifiers as URIs in EMF.

Fig. 3 shows a summary of the formalization of the metamodel $\mathscr{M}$ of the example and a configuration of components as a model M in the formal framework, i.e., as a term of sort *Model*. In this way, MOMENT2 enables the prototyping and experimentation of formal model-driven development techniques and the formal analysis of model-based languages and models in Maude, as shown in Section 5.

[6] This bridge uses the API of the Maude Development Tools [Mau06] for integrating Maude into Eclipse.
[7] In EMF terminology, a metamodel at M2 is an EMF model and a model at M1 is an EMF model instance.



```
sort Oid Cid Property PropertySet Object ObjCol Model .
subsort Qid < Oid .
subsort Property < PropertySet .
op noneProperty : -> PropertySet .
op _,_ : PropertySet PropertySet -> PropertySet
  [comm assoc id: noneProperty] .

subsort Object < ObjCol .
op <_:_|_> : Oid Cid PropertySet -> Object .
op none : -> ObjCol .
op __ : ObjCol ObjCol -> ObjCol [comm assoc id: none] .
op <<_>> : ObjCol -> [Model] .
mb << none >> : Model  .
cmb << OC >> : Model if ( OC =/= none ) and uniqueOid( OC )
  and noDuplicateProperties( OC ) and noDanglingEdges( OC )
  and singleContainer( OC ) and validProperties( OC ) .

op property`:_=_ : String String -> Property .
op property`:_=_ : String Collection+{Oid} -> Property .
op property : -> Property .

sorts Configuration Component Client Server .
subsorts Configuration Component < Cid .
op Configuration -> Configuration .

subsorts Client Server < Component .
op Client : -> Client . op Server : -> Server .
```

```
<<
  < 'c1 : Client | property : "name" = "c1", property : "connectsTo" = 's1 >
  < 'c2 : Client | property : "name" = "c2", property : "connectsTo" = 's1 >
  < 'c3 : Client | property : "name" = "c3", property : "connectsTo" = 's1 >
  < 'c4 : Client | property : "name" = "c4", property : "connectsTo" = 's1 >
  < 's1 : Server | property : "name" = "s1", property : "connectsTo" >
  < 's2 : Server | property : "name" = "s2", property : "connectsTo" >
>>
```

Fig. 3. EMF-Maude mappings in MOMENT2.

4.6. A Metamodeling Framework with n Layers.

Although the number of metamodeling layers that are usually taken into account is up to four, the MOF 2.0 core specification allows the use of n layers [OMG06]. Our algebraic semantics for layers M3-M2[8] can be naturally extended to support n layers for arbitrary $n \geq 2$. Furthermore, it is not necessary to require that the corresponding semantic framework is MEL, which typically supports a more static semantics. To support dynamic semantics of models in an n-layer MOF framework, we may use rewriting logic and exploit the logic containment MEL $\subseteq$ RL, which induces a containment of specifications $\mathit{SpecMEL} \subseteq \mathit{SpecRL}$. Our present semantics, which is a semantics $\mathbb{A} = \mathbb{A}_1$, can be extended as follows. The first step of the extension chooses a model $Q_1 \in [\![\mathrm{MOF}]\!]$ and defines a new formal semantics for layer 2 as a function

$$\mathbb{A}_2 : [\![Q_1]\!] \longrightarrow \mathit{SpecRL}.$$

We again define the semantics of a model $Q_2 \in [\![Q_1]\!]$ using initial model semantics by using a sort *Model* in $\mathbb{A}_2(Q_2)$ and defining $[\![Q_2]\!] = T_{\mathbb{A}(Q_2),\mathit{Model}}$ where if $\mathbb{A}_2(Q_2)$ is the rewrite theory $(\Sigma, E \cup A, R)$, then $T_{\mathbb{A}_2(Q_2),\mathit{Model}}$ is, by definition, the carrier of sort *Model* in the initial algebra $T_{(\Sigma,E\cup A)}$.

Following inductively this way, if we have already provided a semantics up to level n using models $Q_0 \in [\![\mathrm{MOF}]\!]$ (*where* $Q_0 = \mathrm{MOF}$), $Q_1 \in [\![Q_0]\!]$, $Q_2 \in [\![Q_1]\!], \ldots, Q_{n-1} \in [\![Q_{n-2}]\!]$, then we can add layer $n+1$

[8] Recall from Section 3 that layer 0 is the meta-metamodel layer M3 of MOF, layer 1 is the layer M2 of metamodels $\mathscr{M} \in [\![\mathrm{MOF}]\!]$, and layer 2 is the layer M1 of models $M \in [\![\mathscr{M}]\!]$.

$$\begin{tikzcd}
0 & \llbracket Q_0 \rrbracket \ni Q_0 \arrow[r, mapsto, "\mathbb{A}_0"] & \mathbb{A}_0(Q_0) & & \\
1 & & \llbracket Q_0 \rrbracket \ni Q_1 \arrow[r, mapsto, "\mathbb{A}_1"] & \mathbb{A}_1(Q_1) & \\
\ldots & & & \ldots & \\
n+1 & & & \llbracket Q_n \rrbracket \ni Q_{n+1} \arrow[r, mapsto, "\mathbb{A}_{n+1}"] & \mathbb{A}_{n+1}(Q_{n+1})
\end{tikzcd}$$

Fig. 4. N-layer metamodeling framework for MOF.

in the same way by choosing a model $Q_n \in \llbracket Q_{n-1} \rrbracket$ and by defining a function

$$\mathbb{A}_{n+1} : \llbracket Q_n \rrbracket \longrightarrow \mathit{SpecRL}$$

again, defining for each $Q_{n+1} \in \llbracket Q_n \rrbracket$ its semantics by the identity

$$\llbracket Q_{n+1} \rrbracket = T_{\mathbb{A}_{n+1}(Q_{n+1}),\mathit{Model}}.$$

This process of iterated extension is schematically depicted in Figure 4, where $\mathbb{A}_0$ coincides with $\mathbb{A}$ obtaining the MOF theory $\mathbb{A}_0(Q_0) = \mathbb{A}(\mathrm{MOF})$, explained in Section 4.3.

Note that the first extension step $\mathbb{A}_2$ to our semantics $\mathbb{A}_1$ has already been considered, in a more general way, in the paper [BKM09], where for each $Q_1 \in \llbracket \mathrm{MOF} \rrbracket$ an *institution* [GoB92] $\mathcal{I}_{Q_1}$ is chosen and then an $\mathcal{I}_{Q_1}$-based semantics is given as a mapping $\mathbb{A}_2 : \llbracket Q_1 \rrbracket \longrightarrow \mathit{Spec}_{\mathcal{I}_{Q_1}}$.

The above extension scheme is the case where $\mathcal{I}_{Q_1} = \textsc{rl}$. Furthermore, our scheme, by using initial model semantics at all layers can be extended to an arbitrary number of levels, whereas it is not clear how the more general institutional semantics of [BKM09] could be further extended to layer 3 for an arbitrary institution.

An interesting example of a useful extension of our semantics $\mathbb{A}_1$ to a rewriting logic semantics $\mathbb{A}_2$ is provided by the semantics (presented in [OBM09]) of (a fragment of) AADL models in rewriting logic as a mapping $\mathbb{A}_2 : \llbracket \mathrm{AADL} \rrbracket \longrightarrow \mathit{SpecRL}$, where an AADL model Q_2, including behavior information defined with AADL's behavioral annex, has an associated rewrite theory $\mathbb{A}_2(Q_2)$ that can be used in conjuction with the Real-Time Maude tool [OlM07] to both simulate and formally analyze real-time properties of the AADL model Q_2.

5. Applications in Model-Driven Development Scenarios

In this Section we present some model-driven scenarios in which the algebraic semantics $\mathbb{A}$ of MOF metamodels plays an important role. We illustrate the application of $\mathbb{A}$ by using Maude and its support for formal reasoning in the running example of model-based software architecture reconfigurations. These techniques are internally used in MOMENT2, so that they can be applied in EMF-based environments through OMG standards and without explicit contact with the underlying MEL and Maude.

Section 5.1 presents some scenarios where $\mathbb{A}$ provides the formal foundations for techniques and practices widely used in the model-driven community: metamodel conformance, semantics of MOF domain-specific languages, model transformations and model management. Section 5.2 focuses on the formal implications and advantages of our approach where formal analysis is made available within modeling environments: static analysis based on structural constraint satisfaction, and dynamic analysis based on reachability analysis and LTL model checking.

5.1. Model-Driven Development Scenarios

Model-driven development techniques can be formalized in our approach as follows.

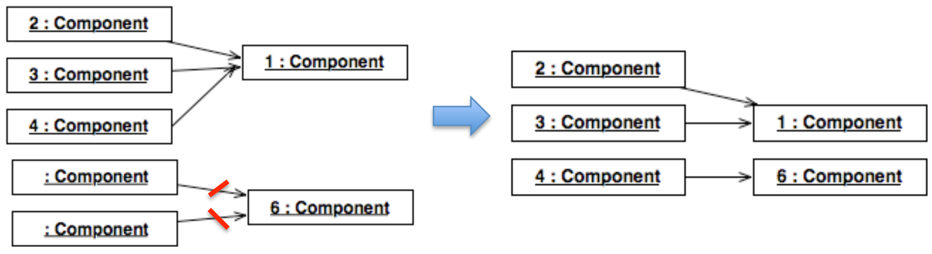


Fig. 5. Reconfiguration rule.

5.1.1. *Model conformance.*

MOMENT2 provides an automated, deductive mechanism to check the structural conformance relation $M : \mathscr{M}$ by relying on Maude's implementation of MEL. Consider the metamodel $\mathscr{M}$ and the model `conf1` in Fig. 3, given as a constant `conf1`. We can automatically check whether the model `conf1` conforms to the metamodel $\mathscr{M}$, formalized as the model type $[\![\mathscr{M}]\!]$ and syntactically represented by the sort *Model* in the theory $\mathbb{A}(\mathscr{M})$, by evaluating the following membership in Maude:

```
red conf1 :: Model .
result Bool: true
```

5.1.2. *Rewriting logic semantics of domain-specific languages.*

A MOF-based domain-specific language (DSL) provides modeling primitives that can be used to define concepts in a given domain. For example, real-time embedded systems can be specified with AADL [SAE07], web service workflows can be defined with BPEL [Wsb07], or ontologies can be defined with OWL [Lac05]. The abstract syntax of these languages is provided as MOF metamodels, their concrete syntax can be either textual or graphical, and their dynamic semantics can be defined either with approaches for defining the semantics of programming languages, with frameworks that provide library support for manipulating models, such as Kermeta [Tri08], or with graph transformation systems [Roz97, EMK99, EEP06]. The last one constitutes an interesting candidate for defining model-based DSLs due to the graph-based nature of models and to the formalization of the approach.

A MOF metamodel $\mathscr{M}$ is used to define concepts in a particular domain, such as components in our running example, where software architectures are configurations of components that may be connected to each other through the `connectsTo` association end. In this section, we enrich the algebraic semantics $\mathbb{A}(\mathscr{M})$ with a rewrite rule to define a dynamic semantics for component configurations. In particular, we add a dynamic connection load balancing strategy, so that a server component should not have more than two connections at a time, i.e., when a component has more than two connections, the spare connections are forwarded to other components with less than two incoming connections. We depict the reconfiguration as a graph transformation rule in Fig. 5, where a rule is defined with a left-hand side (LHS) pattern and a right-hand side (RHS) pattern. Each pattern consists of nodes that represent `Component` objects in a model (graph) M and edges representing `connectsTo` references between them. A reconfiguration can be applied whenever the LHS pattern of the rule can be matched against a specific configuration M of components, and then the edges are manipulated as follows: an edge in the LHS and not in the RHS is removed from the configuration, an edge not in the LHS but in the RHS is added, the rest of edges remain unmodified. Marked edges indicate that the edges must not exist in order to apply the rule; this notion is known as a *negative application condition* in the graph transformation community.

The graph-theoretic nature of models is axiomatized in our algebraic semantics as a set of objects modulo the associativity, commutativity, and identity axioms of set union. The semantics of a reconfiguration can then be naturally expressed as a *rewrite theory* [Mes92] extending the algebraic semantics $\mathbb{A}(\mathscr{M})$ of our metamodel specification with *rewrite rules* that are applied *modulo* the equational axioms. In this way, the above graph-transformation rule can be summarized at a high level as follows:

```
op free-reconfiguration : Model -> Model .
crl free-reconfiguration(M) =>
```

```
  free-reconfiguration(<< < O1 : C1 | PS1 >
    < O2 : C2 | property : "connectsTo" = O1, PS2 >
    < O3 : C3 | property : "connectsTo" = O1, PS3 >
    < O4 : C4 | property : "connectsTo" = O5, PS4 >
    < O5 : C5 | PS5 > OC >>)
if << < O1 : C1 | PS1 >
  < O2 : C2 | property : "connectsTo" = O1, PS2 >
  < O3 : C3 | property : "connectsTo" = O1, PS3 >
  < O4 : C4 | property : "connectsTo" = O1, PS4 >
  < O5 : C5 | PS5 > OC >> := M ∧ nac(O5, M) .
```

where the expression `P := M` matches the pattern `P` to the model variable `M`, `O1, O2, O3, O4, O5 : Oid`, `C1, C2, C3, C4, C5 : Component`,
`PS1, PS2, PS3, PS4, PS5 : PropertySet`, `OC : ObjCol`, `M : Model`, the LHS of the graph transformation rule in Fig. 5 corresponds to the pattern in the `P := M` expression, its RHS corresponds to the RHS of the rewrite (after the `=>` symbol), and the `nac(O5,M)` condition corresponds to the negative application condition that enables the application of the rule:

```
op nac : Oid Model -> Bool .
eq [counterexampleNAC] : nac(O1,
  << < O1 : C1 | PS1 >
    < O2 : C2 | property : "connectsTo" = O1, PS2 >
    < O3 : C3 | property : "connectsTo" = O1, PS3 > OC
  >>
) = false .
eq [satisfiedNAC] : nac(O1, M) = true [owise] .
```

The formalization of model transformations in rewriting logic and their verification in Maude presented in [BHM09] is also available in the MOMENT2 framework, extending the formalization of the MOF framework that is presented in this paper, relying on it, and providing a user-friendly EMF-based programming environment generated with openArchitectureWare[9]. The most recent version of MOMENT2 provides a QVT-like syntax to define transformation rules that are then automatically translated into rewrite rules and executed by Maude. In [BHM09], the authors present a more detailed treatment of the graph transformation notions that are explained in this section for formal verification purposes.

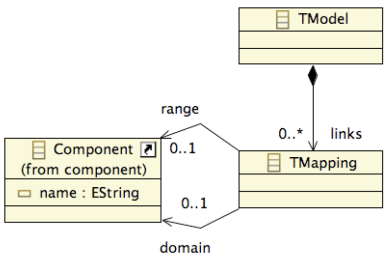
5.1.3. Model transformations and operators for model management.

Due to the executability of MEL specifications in Maude, the realization of MOF metamodels as MEL theories enhances the formalization and prototyping of model-driven development processes, such as:

- Model transformations [SeK03], where translations of models between different modeling languages can be performed.
- Model-driven roundtrip engineering [BGS08, PMD05], where a model that constitutes the abstract syntax tree of a program can be translated into a model that specifies the software at a higher level of abstraction. Code may be generated again from recovered models in an automated way. An important issue is to keep both the abstract description and the code synchronized to deal with changes in a consistent way.
- Model traceability [ANR06], where traceability is important to keep track of the changes applied to software artifacts in a software development process for maintenance and evolution purposes.
- Model management [Ber03], where models can be manipulated by means of generic operators that rely on mappings between models. These operators permit, for example, merging models, generating mappings between models, and computing differences between models; they can be used to solve complex scenarios such as the roundtrip problem.

We illustrate the application of MOMENT2 to trace component reconfigurations, which can be applied to adapt a configuration to a given criteria, as the load balance strategy discussed above. We consider a traceability metamodel $\mathscr{T}$ to define mappings between components as models T, such that $T : \mathscr{T}$. An equationally defined operator is presented to generate such models.

[9] An updated version with the new XText framework is ongoing work.



$\xmapsto{\mathbb{A}}$

```
sorts TModel TMapping .
subsorts TModel TMapping < Cid .

op TModel : -> TModel .
op TMapping : -> TMapping .

op property`:_=_ : String String -> Property .
op property`:_=_ : String Collection+{Oid} -> Property .
op property : -> Property .
```

Fig. 6. Traceability metamodel $\mathscr{T}$ and its algebraic realization $\mathbb{A}(\mathscr{T})$.

The metamodel $\mathscr{T}$ in Fig. 6 permits defining mappings between components that may belong to different configurations as *TMapping* instances. A *TModel* instance constitutes the container for all the mappings between two models. Given configurations `conf1` and `conf2`, a trace model can be defined between them in order to map the elements that represent the same component in both configurations. Such mappings can be used to keep track of the changes that have been applied to the initial configuration or to identify structurally-equivalent elements in a model management scenario as presented in [BCR05, BCR06].

A *model transformation*, defined as an equationally defined function, can be used to automatically generate a trace model between two configurations. The following *match* operator takes two component configurations (the models `conf1` and `conf2` in Fig. 1) as inputs and generates a trace model with mappings between the components of both configurations. The *match* operator uses an auxiliar operator *$match* that, in addition, takes a traceability model that is initialized with a *TModel* instance, and a natural number that is used to create object identifiers. The *$match* operator adds a new link whenever two components with the same name are found. More complex structural criteria can be used to match objects by simply traversing the objects that constitute the model by means of matching modulo associativity and commutativity axioms.

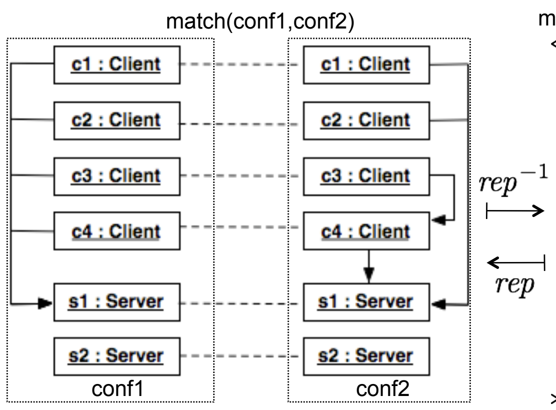
```
op match : Model Model -> ModelTrac .
eq match(M1,M2) = $match(M1,M2,init-trac,0) .

op $match : Model Model ModelTrac Nat -> ModelTrac .
eq $match( << < O1 : C1 | property : "name" = Name, PS1 > OC1 >>,
  << < O2 : C2 | property : "name" = Name, PS2 > OC2 >>,
  << < O3 : TModel | property : "links" = Set, PS3 > OC3 >>, N
) =
  $match( << OC1 >>, << OC2 >>,
    << < O3 : TModel |
         property : "links" = Set -> including( getNewOid(N) ), PS3 >
      < getNewOid(N) : TMapping | domain : O1, range : O2 > OC3 >>,
    N + 1 ) .
eq $match(M1,M2,TM, N) = TM [owise] .
```

where `Model{Trac}` is the sort of the model type $[\![\mathscr{T}]\!]$ in the theory $\mathbb{A}(\mathscr{T})$, `init-trac` is a constant that represents a trace model with a single *TModel* instance, the `getNewOid` function obtains a new object identifier from a natural number, and the `including` operator adds an element to a set, as in OCL. The trace model betweeen the configurations `conf1` and `conf2` can be generated by means of the command `red match(conf1, conf2)`. The output model is illustrated in Fig. 7.

5.2. Formal Analysis Scenarios

In this subsection, we present some applications of the algebraic semantics $\mathbb{A}$ for MOF metamodels to formal reasoning of model-driven development practices. In particular, static and dynamic analysis of model-based domain-specific languages (DSLs).



```
match(conf1,conf2)
 <<
   < '0 : TModel |
          property : "links" = Set{ '1, '2, '3, '4, '5, '6 } >
   < '1 : TMapping | property : "domain" = 'c1,
                     property : "range" = 'c1 >
   < '2 : TMapping | property : "domain" = 'c2,
                     property : "range" = 'c2 >
   < '3 : TMapping | property : "domain" = 'c3,
                     property : "range" = 'c3 >
   < '4 : TMapping | property : "domain" = 'c4,
                     property : "range" = 'c4 >
   < '5 : TMapping | property : "domain" = 's1,
                     property : "range" = 's1 >
   < '6 : TMapping | property : "domain" = 's2,
                     property : "range" = 's2 >
 >>
```

Fig. 7. Trace model and its term representation.

5.2.1. Static analysis.

Once a metamodel $\mathscr{M}$ is realized as a theory $\mathbb{A}(\mathscr{M})$, more axioms can be added to $\mathbb{A}(\mathscr{M})$ in order to enrich the semantics of the metamodel $\mathscr{M}$ or to add semantics to the corresponding DSL. These axioms may correspond to OCL constraints as shown in [BoM09] and can be used to verify that a specific model satisfies certain semantic properties.

For example, we can define a boolean predicate over a model that specifies that client components do not receive connections from other components in a specific configuration. This predicate is given as an equationally defined function that negates the satisfaction of the predicate when the property is not satisfied and that asserts its satisfaction in any other case:

```
op checkClientServer : Model -> Bool .
eq checkClientServer( << < O1 : C1 | property : "connectsTo" = O2, PS1 >
  < O2 : Client | PS2 > OC >> ) = false .
eq checkClientServer( M:Model ) = true [owise] .
```

where `O1, O2: Oid`, `C1 : Component`, `PS1, PS2: PropertySet`, and `OC : ObjCol`, and where the last equation uses Maude's `owise` feature, so that it is applied if and only if the other equations for `checkClientServer` fail. As explained in [CDE07], `owise` equations are semantically equivalent to ordinary but more verbose conditional equations. The models `conf1` and `conf2` in Fig. 1 can then be verified as follows:

```
red checkClientServer(conf1) .
result Bool: true

red checkClientServer(conf2) .
result Bool: false
```

where `conf2` does not satisfy the constraint.

5.2.2. Formal verification of behavioral specifications.

The rewriting logic semantics of DSLs, defined by means of $\mathbb{A}$ and additional rewrite rules, can also be used for formal dynamic analysis. Given a specific initial configuration M of components, we can use Maude's *search* command to model check whether or not all possible reconfigurations of an initial configuration preserve the constraint `checkClientServer`, introduced above.

The metamodel realization $\mathbb{A}(\mathscr{M})$, corresponding to Fig. 1, and the rewriting rule `free-reconfiguration` presented above define a state transition system, where states represent configurations M and M' of components and transitions $M \longrightarrow M'$ are given by one application of the reconfiguration rule. However, a reconfiguration of this kind could conceivably produce configurations M' of components that do not satisfy the constraint `checkClientServer`.

In Maude, the *search* command allows one to exhaustively explore (following a breadth-first strategy) the reachable state space defined by a state transition system as the one above, checking if an invariant is violated. We can use the *search* command to find out if the reconfiguration `free-reconfiguration` produces such an illegal configuration as follows[10]:

```
search [1] free-reconfiguration(conf1) =>+
  free-reconfiguration(<< < O1 : C1 | property : "connectsTo" = O2, PS1 >
    < O2 : Client | PS2 > OC >>) .
```

where `conf1` is a constant that represents the initial configuration M in Fig. 1. This command finds a counterexample, `conf2` in Fig. 1, where a client component is connected to another client component. An alternative reconfiguration rule can be defined to avoid this problem as shown below, by indicating that the node `O5` in the graph patterns of the rule in Fig. 5 is of type `Server`. No counterexamples are found when running again the search command with the new reconfiguration rule.

```
op safe-reconfiguration : Model -> Model .
crl safe-reconfiguration(M) =>
  safe-reconfiguration(<< < O1 : C1 | PS1 >
    < O2 : C2 | property : "connectsTo" = O1, PS2 >
    < O3 : C3 | property : "connectsTo" = O1, PS3 >
    < O4 : C4 | property : "connectsTo" = O5, PS4 >
    < O5 : Server | PS5 > OC >>)
if << < O1 : C1 | PS1 >
  < O2 : C2 | property : "connectsTo" = O1, PS2 >
  < O3 : C3 | property : "connectsTo" = O1, PS3 >
  < O4 : C4 | property : "connectsTo" = O1, PS4 >
  < O5 : Server | PS5 > OC >> := M / nac(O5, M) .
```

5.2.3. Linear temporal logic model checking.

Maude also provides a model checker where properties can be given as linear temporal logic (LTL) formulae [CDE07]. Taking into account the `safe-reconfiguration` rule that has been added to the theory $\mathbb{A}(\mathscr{M})$, we can also model check liveness properties, such as the fact that the server *s1* will always have a balanced load L eventually. This property can be formulated in LTL as $\Box \Diamond$ *balanced*(*"s1"*, L), where *balanced("s1", L)* is a state predicate that is satisfied when the server component with name *"s1"* has L or less connections. Following the guidelines provided in [CDE07], we defined a subtheory inclusion to use Maude's model checker, defined in the theory `MODEL-CHECKER`, into the theory $\mathbb{A}(\mathscr{M})$ that is extended with the rule `safe-reconfiguration`. The sort `Model` is defined as subsort of the sort `State`, used by the model checker to represent system states. The predicate symbol *balanced* is then defined as a parametric predicate `_|=_` between system states and state predicates can be equationally defined for the case in which the state predicate is satisfied:

```
subsort Model < State .

op balanced : String Nat -> Prop .
eq free-reconfiguration( << < O1 : Server |
  property : "name" = Name, PS1 > OC >> )
  |= balanced(Name, L)
= (countBalance(O1, << < O1 : Server |
  property : "name" = Name, PS1 > OC >>) <= L) .
```

where `countBalance` is an equationally defined function that counts the number of connections to a given component with identifier `O1` in the model `M`:

```
op countBalance : Oid Model -> Nat .
eq countBalance( O1, << < O2 : C2 |
  property : "connectsTo" = O1, PS2 > OC >> ) =
  1 + countBalance(O1, << OC >>) .
eq countBalance( O1, M ) = 0 [owise] .
```

[10] We have removed the types of the variables in the RHS of the command (after the `=>+` symbol) for the sake of simplicity.

The model predicate `balanced("s1",2)` can be used in a LTL formula to model check that the server `"s1"` will end up with 2 or less connections as follows:

```
red modelCheck(safe-reconfiguration(conf1), [] <> balanced("s1", 2)) .
```

6. Related Work

The meaning of the *metamodel* notion has been widely discussed in the literature, see for example [Lud04, Sei03, Kuh06, Ren04]. 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 $\mathscr{M}$, model type $[\![\mathscr{M}]\!]$, and metamodel realization $\mathbb{A}(\mathscr{M})$, 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 metamodels as Java programs is provided in the Java Metadata Interface (JMI) specification [JCP02], 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 $\mathbb{A}$ function gives us an executable formal specification of the algebraic semantics of any EMOF metamodel $\mathscr{M}$.

Although EMOF metamodels can be viewed as simplified UML class diagrams, formal approaches for metamodeling need a reflective mechanism, such as $\mathbb{A}$ in our approach, to provide the semantics of modeling languages. This mechanism is not needed in UML, where the modeling language is fixed. We focus on several approaches for metamodeling that rely on different formalisms.

The Meta-Modeling Language is a meta-circular language based on the MML calculus [CEK01], which provides an operational semantics for both UML modeling constructs and OCL operators. Modeling languages can be precisely defined in MML by explicitly specifying its abstract syntax, its *semantic domain* and a mapping between the concepts involved in both [CEK02]. This mapping can be viewed as the application of the function $\mathbb{A}$ to a specific metamodel $\mathscr{M}$.

Alloy [Jac06] is a declarative language based on first-order relational logic in which systems with constraints can be modeled. The Alloy analyzer [All09] provides an automated mechanism for constraint satisfaction with two main functionalities: *simulation* for producing valid instances of an Alloy specification and *assertion* for verifying constraints. [ABG07] provides an encoding of EMOF metamodels with OCL constraints into Alloy so that the Alloy analyzer [All09] is used to generate models that conform to a metamodel (automated test case generation) and to verify that OCL constraints can be satisfied. Counterexamples and logical inconsistencies are found when the constraints are not satisfied. In [ABK08], the Alloy analyzer is used to verify relational model transformations in order to ensure that a model transformation cannot produce invalid models. However, Alloy has a simple type system where only integers can be used in attribute values.

In [Poe06], constructive type theory is used for defining a typed metamodeling framework, where models, which are defined as terms, can also be represented as types by means of a reflection mechanism such as $\mathbb{A}$. 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 model belongs to the corresponding type by means of the Curry-Howard isomorphism.

The Diagram Predicate Framework, which combines notions from category theory and first-order logic extending the theory of generalized sketches [Mak97], has been used to formalize MOF metamodels and their OCL constraints diagrammatically [RRW09]. In this work, the authors illustrate how their approach can be used to formalize software artifacts in several layers of the MOF framework, where the conformance of a model to its metamodel is encoded as a graph morphism.

In the graph transformations field [Roz97, EMK99, EEP06], metamodels are defined as type graphs with node inheritance, and models are defined as attributed typed graphs. The main difference between type graphs and metamodels rely on the use of composition associations in EMOF metamodels, which can be used for defining hierarchies of composite objects in models. A notion of *graph with containments* is introduced in [BET08], where the authors show how graph transformations can be used as a formal backend for model transformations. In this way, the theory on graph transformations and related tools can be used to perform formal analysis of model transformations. In particular, the authors show how to analyse termination and confluence of model transformations that are encoded as graph transformations in the algebraic graph transformation environment AGG [Agg09].

There are a number of metamodeling approaches based on Maude. Maude already provides support for object-oriented programming [Mes93], where objects, the *isInstanceOf* relation and the *class specialization* relation are supported. The dynamics of object-oriented systems can be provided by means of term rewriting.

The static semantics of the UML metamodel (version 1.3) has been previously formalized as an algebraic specification in MEL [FeT01]. In this approach, the authors already took the MOF approach into account, although the MOF standard was in its early stages. In [FeT00, Fer02], the authors provide a formal four-layered framework where: (i) some parts of the MOF meta-metamodel are formalized in a MEL theory at M3 level (called MOF layer); (ii) the UML class diagram and the object diagram metamodels are provided as MEL theories, called *syntactic specification* and *semantic specification* respectively, at M2 level (called UML metamodel layer); (iii) UML class diagrams are defined as terms in the syntactic specification theory at M1 level (called *domain model* layer); and (iv) object diagrams are defined as terms in the semantic specification theory at M0-level (named *user objects* layer). A novel feature in this approach relies on the reuse of the reflective facilities of MEL to provide support for the evolution of UML-based software artifacts [ToF00]. The authors focused on the verification of static properties by using Maude as an implementation of MEL and the language to define the constraints.

Another approach [RRD07] based on Maude uses the KM3 language [JoB06] for indirectly defining EMOF metamodels. In this work, the authors present how KM3 specifications of metamodels can be represented as object modules [CDE07] in Full-Maude and how models can be defined as Maude collections of objects. However, no automated support is provided for representing models as terms since KM3 only permits defining the textual concrete syntax of metamodels. That is, mappings like rep^{-1} and *rep* are not defined for models that conform to metamodels that are extracted from KM3 specifications. In this way, the user has to define the models in Maude notation directly. The authors provide a mechanism to represent KM3 specifications of metamodels as collections of objects at a syntactic level so that Maude is used to statically analyse KM3 specifications: to check when two metamodels describe model subtypes, to infer metamodels from models and to compute metrics. However, model types are not algebraically characterized. Since KM3 metamodels can be represented in EMF automatically, KM3 metamodels can also be formalized through MOMENT2.

Our algebraic semantics $\mathbb{A}$ for EMOF metamodels $\mathscr{M}$ in MEL formally defines the notions of *metamodel realization* $\mathbb{A}(\mathscr{M})$, *model type* $[\![\mathscr{M}]\!]$ and *model conformance* $M : \mathscr{M}$. This means that $\mathbb{A}$ enables reasoning with model types at an algebraic level and not just at a syntactical level. Due to the graph nature of models, the algebraic semantics $\mathbb{A}$ for MOF can also be used as an algebraic environment for graph transformations, where Maude's analysis capabilities, such as reachability analysis and LTL model checking, can be reused. The complete algebraic formalization of EMOF metamodels together with OCL can be found in [BoM09, Bor07], where composition associations are also taken into account in the formalization. The algebraic semantics $\mathbb{A}$ is implemented in MOMENT2 where EMF is used as implementation of the EMOF standard. Furthermore, the generic mappings *rep* and rep^{-1} allow representing EMF models as terms in MOMENT2 in a transparent way to the user. This is an essential feature in MOMENT2, where the goal is to apply Maude for formal model management tasks by using OMG standards, such as MOF, OCL and QVT.

7. 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: metamodel realization $\mathbb{A}(\mathscr{M})$, model type $[\![\mathscr{M}]\!]$, and metamodel conformance $M : \mathscr{M}$. Our work provides an algebraic executable formalization of the MOF standard that can be reused in standard-compliant frameworks.

This algebraic framework opens a wide spectrum of interesting applications for model-driven development. In particular, we have shown how it can be used for automatically checking metamodel conformance, defining domain-specific languages and specifying model transformations and model management operators. In addition, Maude's formal verification facilities can be used for static and dynamic analysis of domain-specific languages, such as checking constraints over models, reachability analysis and LTL model checking.

The algebraic semantics for MOF provides the foundational notions for a model management tool suite, MOMENT2 [Mom09], that supports both OCL [BoM09] and QVT-like model transformations [BHM09]. MOMENT2 uses MOF, OCL and a QVT-like model transformation language as interface, so that the techniques that have been illustrated in this work are internally used in a transparent way to the user. For the development of this framework, we have relied on the experience gained in previous prototypes that gave algebraic executable specifications for OCL [BOG06], QVT [BCR06] and model management operators

[BCR05, BCR06]. 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.

In future work, we plan to apply the algebraic MOF framework together with the aforementioned tool for model transformations and Maude-based formal reasoning techniques in model-driven development scenarios, where software systems that are developed contain critical properties that have to be verified. In particular, we are considering the formal analysis of real-time embedded systems in the avionics domain, by using model-based languages like the Architecture Analysis and Design Language (AADL) [SAE07].

Acknowledgments.

We thank the anonymous referees for their comments and suggestions that have helped us to improve the exposition. 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

[ABG07] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *LNCS*, pages 436–450. Springer, 2007.

[ABK08] K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In *MODEVVA*, volume 5002 of *LNCS*. Springer, 2008.

[Agg09] AGG Homepage, 2009. `http://tfs.cs.tu-berlin.de/agg/`.

[All09] Alloy Analyzer Homepage, 2009. `http://alloy.mit.edu/`.

[ANR06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Syst. J.*, 45(3):515–526, July 2006.

[Ber03] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[BeT80] J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.

[BET08] E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In K. Czarnecki, editor, *MoDELS'08*, volume 5301 of *LNCS*, pages 53–67. Springer, 2008.

[Bez05] J. Bézivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[BCR05] A. Boronat, J. A. Carsí, and I. Ramos. Automatic Support for Traceability in a Generic Model Management Framework. In A. Hartman and D. Kreische, editors, *ECMDA-FA*, volume 3748 of *LNCS*, pages 316–330. Springer, 2005.

[BCR06] A. Boronat, J. A. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In L. Baresi and R. Heckel, editors, *FASE*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.

[BCR06] A. Boronat, J. A. Carsí, I. Ramos, and P. Letelier. Formal model merging applied to class diagram integration. *Electr. Notes Theor. Comput. Sci.*, 166:5–26, 2007.

[BGS08] M. Bork, L. Geiger, C. Schneider, and A. Zündorf. Towards roundtrip engineering - a template-based reverse engineering approach. In I. Schieferdecker and A. Hartman, editors, *ECMDA-FA*, volume 5095 of *LNCS*, pages 33–47. Springer, 2008.

[BHM09] A. Boronat, R. Heckel, and J. Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In *FASE*, volume 5503 of *LNCS*. Springer, 2009.

[BKM09] A. Boronat, A. Knapp, J. Meseguer, and M. Wirsing. What is a Multi-Modeling Language? In *19th International Workshop on Algebraic Development Techniques*, volume 5486 of *LNCS*, pages 71–87. Springer, 2009.

[BoM09] A. Boronat and J. Meseguer. Algebraic Semantics of OCL-constrained Metamodel Specifications. In R. F. Paige and B. Meyer, editors, *TOOLS (47)*, LNBIP. Springer, 2009.

[BOG06] A. Boronat, J. Oriente, A. Gómez, I. Ramos, and J. A. Carsí. An Algebraic Specification of Generic OCL Queries Within the Eclipse Modeling Framework. In A. Rensink and J. Warmer, editors, *ECMDA-FA*, volume 4066 of *LNCS*, pages 316–330. Springer, 2006.

[Bor07] A. Boronat. *MOMENT: a formal framework for MOdel manageMENT*. PhD in Computer Science, Universitat Politènica de València (UPV), Spain, 2007. `http://www.cs.le.ac.uk/people/aboronat/papers/2007_thesis_ArturBoronat.pdf`.

[BJM00] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[BrM06] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1-3):386–414, 2006.

[CDE07] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude – A High-Performance Logical Framework.* Springer LNCS Vol. 4350, 2007.

[CDE99] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing and J. Woodcock, editors, *FM'99 — Formal Methods*, volume 1709 of *LNCS*, pages 1684–1703. Springer-Verlag, 1999.

[CEK01] T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation Semantics for UML. In H. Hußmann, editor, *FASE*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[CEK02] T. Clark, A. Evans, and S. Kent. Engineering modelling languages: A precise meta-modelling approach. In *FASE 02*, volume 2306 of *LNCS*, pages 159–173, London, UK, 2002. Springer.

[CeS04] I. Cervesato and M.-O. Stehr. Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types. In P. Degano, editor, *Proc. Fifth International Workshop on Rewriting Logic and its Applications (WRLA'2004)*, volume 117. Elsevier ENTCS, 2004. Barcelona, Spain, March 27 - 28, 2004.

[CMP07] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373:70–91, 2007.

[EEP06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer, March 2006.

[Emf09] Eclipse Organization. The Eclipse Modeling Framework, 2009. `http://www.eclipse.org/emf/`.

[EMK99] H. Ehrig, U. Montanari, H.-J. Kreowski, G. Rozenberg, and H.-J. Kreowski. *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 3.* World Scientific Publishing Company, July 1999.

[Fer02] J. L. Fernández Alemán. *A formalization proposal of the UML four-layered architecture.* PhD thesis, Murcia University, April 2002. (In Spanish).

[FeT00] J. L. Fernández and A. Toval. Can intuition become rigorous? Foundations for UML model verification tools. In *International Symposium on Software Reliability Engineering (ISSRE 2000).* IEEE, October 2000. San Jose, California, USA.

[FeT01] J. L. Fernández and A. Toval. *Seamless Formalizing the UML Semantics through Metamodels*, pages 224–248. Unified Modeling Language: Systems Analysis, Design, and Development Issues. Idea Group Publishing, 2001.

[GoB92] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *J. ACM*, 39(1):95–146, 1992.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[JCP02] Java Community Process. The Java Metadata Interface (JMI) Specification (JSR 40), 2002. `http://www.jcp.org/en/jsr/detail?id=40`.

[JoB06] F. Jouault and J. Bézivin. Km3: A dsl for metamodel specification. In R. Gorrieri and H. Wehrheim, editors, *FMOODS*, volume 4037 of *LNCS*, pages 171–185. Springer, 2006.

[Kuh06] T. Kühne. Matters of (meta-) modeling. *Software and Systems Modeling (SoSyM)*, 5:369–385(17), December 2006.

[Lac05] L. W. Lacy. *Owl: Representing Information Using the Web Ontology Language.* Trafford Publishing, January 2005.

[Lud04] J. Ludewig. Models in software engineering - an introduction. *Inform., Forsch. Entwickl.*, 18(3-4):105–112, 2004.

[Mak97] M. Makkai. Generalized sketches as a framework for completeness theorems. *Journal of Pure and Applied Algebra*, 115(1):49–79, February 1997.

[Mau06] Maude Development Tools, 2006. `http://moment.dsic.upv.es`.

[MeR04] J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proc. Intl. Joint Conf. on Automated Reasoning IJCAR'04, Cork, Ireland, July 2004*, pages 1–44. Springer LNAI 3097, 2004.

[MeR07] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373:213–237, 2007.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes93] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[Mes98] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.

[Mom09] MOMENT2, 2009. `http://www.cs.le.ac.uk/~aboronat/tools/moment2`.

[OBM09] P. Ölveczky, A. Boronat and J. Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. To appear as technical report at UIUC.

[OlM07] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1–2):161–196, 2007.

[OMG06] OMG. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006.

[PMD05] E. V. Paesschen, W. D. Meuter, and M. D'Hondt. Selfsync: a dynamic round-trip engineering environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 146–147, New York, NY, USA, 2005. ACM.

[Poe06] I. Poernomo. The meta-object facility typed. In H. Haddad, editor, *SAC*, pages 1845–1849. ACM, 2006.

[Ren04] A. Rensink. Subjects, models, languages, transformations. In J. Bézivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings.* Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[RRD07] J. R. Romero, J. E. Rivera, F. Durán, and A. Vallecillo. Formal and Tool Support for Model Driven Engineering with Maude. *Journal of Object Technology*, 6(9):187–207, 2007.

[RSA09] IBM. Rational Software Architect for WebSphere Software, 2009. `http://www-01.ibm.com/software/awdtools/swarchitect/websphere/`.

[Roz97] G. Rozenberg. *Handbook of Grammars and Computing by Graph Transformation, Vol. 1.* World Scientific Publishing Company, January 1997.
[RRW09] Y. L. Adrian Rutle, Alessandro Rossini and U. Wolter. A Diagrammatic Formalisation of MOF-Based Modelling Languages . In R. F. Paige and B. Meyer, editors, *Objects, Components, Models and Patterns, 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29 - July 3, 2008. Proceedings*, LNBIP. Springer, 2009.
[SAE07] SAE. AADL, 2007. `http://www.aadl.info/`.
[Sei03] E. Seidewitz. What models mean. *Software, IEEE*, 20(5):26–32, 2003.
[SeK03] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.
[SRM09] T. Serbanuta, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 2007:305–340, 2009.
[StM04] M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework. In *Essays in Memory of Ole-Johan Dahl*, pages 334–375. Springer LNCS Vol. 2635, 2004.
[ToF00] A. Toval and J. L. Fernández. Formally modeling uml and its evolution: a holistic approach. *Kluwer Academic Publishers*, September 2000. FMOODS'00, Formal Methods for Open Object-Based Distributed Systems. Stanford, California, USA.
[Tri08] Triskell Team. Kermeta, 2008. `http://www.kermeta.org/`.
[Wsb07] OASIS. Web Services Business Process Execution Language Version 2.0, 2007. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`.

A. Semantics of the *Model* sort in a $\mathbb{A}(\mathscr{M})$ theory

In this section, we provide an algebraic definition, in Maude notation, of the boolean predicates that are used in the membership that defines the semantics of the sort `Model` in Section 4.3–4.4.

The `uniqueOid` predicate checks that there are no duplicate object identifiers in different objects.

```
op uniqueOid : ObjCol -> Bool .
eq uniqueOid( < O : C1 | PS1 > < O : C2 | PS2 > OC ) = false .
eq uniqueOid( OC ) = true [owise] .
```

The `noDuplicateProperties` predicate checks that there are no properties with the same name. The `getPropertyName` is a projection function that obtains the name of a property.

```
op noDuplicateProperties : ObjCol -> Bool .
ceq noDuplicateProperties( < O : C | P1, P2, PS > OC ) = false
if getPropertyName(P1) == getPropertyName(P2) .
eq noDuplicateProperties( OC ) = true [owise] .
```

The `noDanglingEdges` predicate checks that there are no references to objects that are not defined in the model. This is achieved by traversing all references in all objects of a model.

```
op noDanglingEdges : ObjCol -> Bool .
eq noDanglingEdges( OC ) = noDEInOC( OC, OC ) .

op noDEInOC : ObjCol ObjCol -> Bool .
eq noDEInOC( none, OC ) = true .
eq noDEInOC( < O : C | PS > OC1, OC ) =
  noDEInPS(PS, OC) and noDEInOC( OC1, OC ) .

op noDEInPS : PropertySet ObjCol -> Bool .
eq noDEInPS((property : PN = V:Collection+{Oid}, PS), OC) =
  noDEInCollection+( V:Collection+{Oid}, OC ) and-then
  noDEInPS( PS, OC ) .
eq noDEInPS( PS, OC ) = true [owise] .

op noDEInCollection+ : Collection+{Oid} ObjCol -> Bool .
eq noDEInCollection+( O, < O : C | PS > OC ) = true .


eq noDEInCollection+( empty-set#Oid, OC ) = true .
eq noDEInCollection+( Set{ O }, < O : C | PS > OC ) = true .
eq noDEInCollection+( Set{ O, Mgm }, < O : C | PS > OC ) =
noDEInCollection+( Set{ Mgm }, < O : C | PS > OC ) .
--- more equations consider the cases for ordered sets, bags and sequences

eq noDEInCollection+( V:Collection+{Oid}, < O : C | PS > OC ) =
  false [owise] .
```

The `singleContainer` predicate checks that an object is only contained in at most one container, guaranteeing that a model is a hierarchical graph. The operator `metaProp(C,PN)` returns the object that defines the property named `PN` in the class named `C` in the metamodel as an intance of the MOF *Property* and `getX` operators project a property value from the tuple that defines an object. For instance, the operator `op _.'getBool(_) : Object String -> Bool .` returns the value of a boolean property `PN` in an object `Obj` as `Obj . getBool(PN)`. The operators `get` and `getString` are likewise defined for objects and string values.

```
--- single container
op singleContainer : ObjCol -> Bool .
eq singleContainer(OC) = $singleContainer(OC, OC) .

op $singleContainer : ObjCol ObjCol -> Bool .
eq $singleContainer(none, OC2) = true .
eq $singleContainer(Obj OC1, OC2) =
  $singleContainer(OC1, OC2) and (#cont(Obj, OC2) <= 1) .

op #cont : Object ObjCol -> Nat .
ceq #cont(< O : C | PS >, < O2 : C2 | property : PN = OidCol , PS2 > OC ) =
  1 + #cont(< O : C | PS >, OC )
if metaProp(C,PN) . getBool("containment") and
   OidCol -> includes( O ) .

ceq #cont(< O : C | PS >, < O2 : C2 | property : PN = O , PS2 > OC ) =
  1 + #cont(< O : C | PS >, OC )
if metaProp(C,PN) . getBool("containment") .

eq #cont(Obj, OC) = 0 [owise] .
```

The `validProperties` predicate checks that the properties in objects are valid by ensuring that: *i)* the names used in properties are defined in the metamodel, *ii)* property values correspond to property types and *iii)* values satisfy the cardinality constraint of the corresponding property. The functions `IsOidProperty(P)` and `IsStringProperty(P)` check that the type of the corresponding property `P` are of type `Oid` or `String`, respectively.

```
op validProperties : ObjCol -> Bool .

eq validProperties( none ) = true .
eq validProperties( Obj OC ) =
  validPropertiesInObject(Obj) and-then validProperties(OC) .

op validPropertiesInObject : Object -> Bool .
eq validPropertiesInObject( Obj ) =  true [owise] .
eq validPropertiesInObject( < O : C | P, PS > ) =
  if (invalidName(C, P)  or not(propertyValidType(C, P))
    or (IsOidProperty(metaProp(C, getPropertyName(P)))
    and not(propertyValidCardinality(C, P))))
  then false
  else validProperties( < O : C | PS > ) fi .

--- i)
op invalidName : Cid Property -> Bool .
eq invalidName(C, P) = (classObject(C) . get("eAllStructuralFeatures")
  . getString("name") -> excludes( getPropertyName(P) )) [owise] .
```

```
--- ii)
op propertyValidType : Cid Property -> Bool .
eq propertyValidType( C, property : PN ) = true .
eq propertyValidType( C, property : PN = V:Collection+{String} ) =
  IsStringProperty(metaProp(C, PN)) .
--- more equations deal with other built-in types

--- iii)
op propertyValidCardinality : Cid Property -> Bool .
eq propertyValidCardinality( C, class : ClassName ) = true .

--- no value
eq propertyValidCardinality( C, property : PN ) =
 (metaProp(C, PN) . getInt( "lowerBound" ) == 0) .

--- only 1 value
eq propertyValidCardinality( C, property : PN = V:MetaOid) =
   ((metaProp(C, PN) . getInt( "lowerBound" )) <= 1)
  and-then
   ((metaProp(C, PN) . getInt( "upperBound" )) == 1) .

--- more than one value
eq propertyValidCardinality( C, property : PN = V:Collection{Oid}) =
  ((metaProp(C, PN) . getInt( "lowerBound" ))
    <= (V:Collection{Oid} -> size))
  and
  ( ((metaProp(C, PN) . getInt( "upperBound" )) == -1 )
    or
    ((metaProp(C, PN) . getInt( "upperBound" ))
      >= (V:Collection{Oid} -> size))
  ) .
```