HAL
archives-ouvertes.fr

Formal executable semantics for conformance in the MDE framework

Marina Egea, Vlad Rusu

▶ To cite this version:

Marina Egea, Vlad Rusu. Formal executable semantics for conformance in the MDE framework. Innovations in Systems and Software Engineering, Springer Verlag, 2010, 6, pp.73-81. 10.1007/s11334-009-0108-1 . inria-00527502

HAL Id: inria-00527502
https://hal.inria.fr/inria-00527502

Submitted on 21 Oct 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

Formal Executable Semantics for Conformance in the MDE Framework

Marina Egea - marinae@inf.ethz.ch - ETH Zurich (Switzerland)
Vlad Rusu - Vlad.Rusu@inria.fr - INRIA Rennes (France)

the date of receipt and acceptance should be inserted later

Abstract In the MDE framework, a metamodel is a language refering to some kind of metadata whose elements formalize concepts and relations providing a modeling language. An instance of this modeling language which adheres to its concepts and relations, is called a valid model, i.e., a model satisfying structural conformance to its metamodel. However, a metamodel frequently imposes additional constraints to its valid instances. These conditions are usually written in OCL and called well-formedness rules. In presence of these constraints, a valid model must adhere to the concepts and relations of its metamodel and fullfill its constraints, i.e., a valid model is a model satisfying semantical conformance to its metamodel. In this work, we provide a formal semantics to the notions of structural and semantical conformance between models and metamodels building on our previous work. Our definitions can be automatically checked using the ITP/OCL tool.

1 Introduction

Software systems are constantly growing in complexity, requiring software development teams to work at higher levels of abstraction in order to cope with this complexity. Modeling software is the key for software engineer teams to work at those abstract levels, to communicate their ideas, to detect design errors, and to be able to integrate their designs of different parts of a system. Model driven engineering (MDE) is a software development methodology [1] proposed by the Object Management Group (OMG) [2] which focuses on creating technology-independent models that can be refined to meet specific platforms. Their ultimate purpose is to serve as a basis to generate code automatically. The strength of this initiative are an increased productivity by maximizing compatibility between systems and enabling the communication between individuals working on a large system. Current practice has shown that indeed it is possible to automatically generate quite complete (and runnable) code from well-specified designs. Unfortunately, it has also shown that this automatic generation process is still far from being a routine one.

In this context, the term "metamodel" is used to refer to a model of some kind of metadata. Hence, we may consider a metamodel as an "abstract language" for describing different kinds of data, i.e., a metamodel is a modeling language without a concrete syntax or notation. We can argue that a metamodel defines a "model type" and at the same time provides the means to distinguish between valid and invalid models, that is, "structural conformance". Namely, the objects of a "conformant" model are necessarily instances of the classes of the associated metamodel (possibly) related by instances of associations between the metamodel's classes. Optionally, a metamodel may also define a set of validity conditions on the models. In this case, a valid model must also fullfill the set of imposed constraints. This is called "semantical conformance". The language most commonly used to add precision to the models is the Object Constraint Language (OCL) [3].

Such semantical aspects are crucial for ensuring model usability and for providing tool support. However, the details of the semantics of meta-models, models, and OCL have much been discussed in the literature, because their large specifications were not clear enough, were not totally consistent or lead to misunderstandings. To overcome these limitations, the use of formal specification languages have been proposed. Such languages yield precise descriptions of software systems

Address(es) of author(s) should be given

and are amenable to formal analysis. On the other hand, those languages require substantial expertise from developers, and they have been criticised for being unpractical, as substantial work is required to formally modeling and analysing systems. An effort to integrate both informal and formal approaches is needed. In this work we make a contribution to this effort by providing a formal semantics to the notion of conformance that can be automatically checked with existent tools. More concretely, we propose formal definitions for the notions of "structural conformance" and "semantical conformance" in order-sorted logic, building on our previous work [4] that defined an executable equational semantics fo OCL. Our definitions can be automatically checked using the ITP/OCL tool [5] written in the Maude formal specification language [6].

In Section 2, we provide some background on models, metamodels and conformance relations through examples. We also capture the essential concepts of model and metamodels, which we translate to order-sorted theories in Section 3. Also, in Section 3, we provide a formal definition of conformance as a theory interpretation. In Section 4 we show how model and metamodel theories are represented in Maude and how the conformance relation can be checked using the ITP/OCL tool. In Section 5, we provide some conclusions and discuss related and future work.

2 Metamodels, models and conformance

In this section we provide some background on metamodels, models, and conformance through examples. Also, we capture the essential modeling elements as tuple structures, setting up the language that we will use for our formal definitions afterwards. We will consider only MOF-compliant metamodels, i.e., only metamodels that can be described using MOF elements [7].

2.1 Metamodels

Metamodel descriptions define the structure and semantics of metadata. In a nutshell, the MOF modeling elements are: *classes* to hold metaobject information; *associations*, which model binary relationships; *inheritance* or *generalization* relationships to refine modeling elements; *operations*, which are "hooks" for accessing behavior associated with a class[1]; *attributes*, which define a value holder, typically in each instance of its class; *data types*, which model other data (e.g., primitive types); and *packages*, which are used to modularize the models, and to ease model imports, merging and extensions. From now on, to preserve the simplicity of the presentation, we will assume that we are working within just one package. The approach can be extended in a natural manner to consider several packages. The modeling concepts presented above adopt the shape of a MOF-compliant metamodel in Figure 1.

In general, metamodel constraints establish additional consistency rules on modeling elements. The standard language used for write these constraints is OCL, for instance, the well-formedness constraints for the UML and MOF metamodels are written in OCL. This language has an evaluation semantics and it has been shown useful both as a constraint and as a query language.

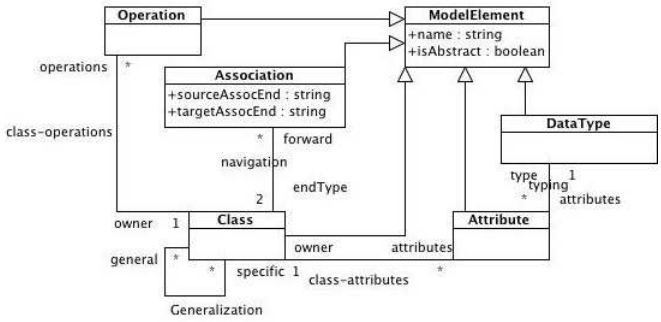

Fig. 1 Basic UML metamodel.

Some example constraints of application to our Basic UML metamodel are the following:

- Invariant 1: Neither direct nor indirect cycles are allowed in the generalization relationship, namely, self.allParents() → excludes(self), where
  context Class :: allParents() : Set(Class) =
  self.parents() → union(self.parents
  → collect(p|p.allParents)
  and
  context Class :: parents() : Set(Classifier) = self.general
- Invariant 2: (multiplicity) Each association is linked to two classes.
  context Association :
  self.endType → size() = 2

Notice that to properly define Invariant 1, we needed to define a recursive operation allParents(). This is typically the case in many metamodel invariants. Notice also that we consider multiplicity constraints as OCL invariants, like the example given in Invariant 2.

2.2 Models

Figure 3 shows a model of an automaton in UML class diagram notation, i.e., in concrete syntax. Its counter-

[1] Operations specify the names and type signatures by which the behavior is invoked, without specifying the behavior itself.

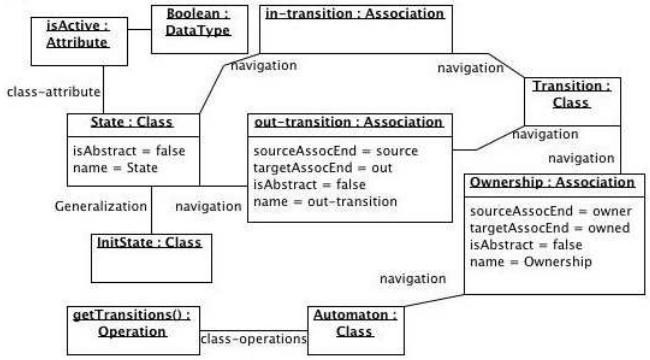

Fig. 2 Model of an automaton as a metamodel instance.

part in abstract syntax, i.e., as a metamodel instance, is shown in Figure 2. In this model, the boolean attribute *isActive* of the class *State* expresses the fact that control is/is not in a given state. The *InitState* subclass of *State* distinguishes initial states of automata. The class *Automaton* owns the *trace* attribute - a string of characters, generated by concatenating *labels* of *transitions* fired by the automaton. The *getTransitions* operation returns all the transitions of the automaton. Transitions own the *label* attribute to hold transition names. They are associated with their *source* and *target* states; their opposite roles from states are *in* and *out* transitions.

2.3 Conformance

The model in Figure 2 is intuitively *structurally conformant* to the metamodel depicted in Figure 1, i.e., it uses classes, associations, and attributes from the metamodel in the correct way. *Semantical conformance* requires, in addition to structural conformance, that the model also satisfies the set of OCL invariants of its metamodel. The model in Figure 2 is intuitively semantically conformant to the metamodel in Figure 1 since it obeys the invariants: it does not have cyclic generalizations and it fullfills multiplicity constraints.

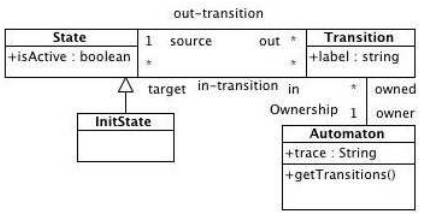

Fig. 3 Model of an automaton as a class diagram.

2.4 Structures to hold metamodel and model information

Let Dt be a set of basic types, e.g., Booleans, Integers, and Strings. A MOF metamodel can be described using the following elements:

Definition 1 (Metamodel structure)

$$\mathcal{MM} = (C, Gen, At, Assoc, AsEnd)$$

where:

- $C = \{c \mid c \text{ is a metaclass}\}$ holds metamodel classes;
- $Gen \subseteq C \times C$ holds the generalization relationships;
- $At = \{At_{\langle c,v\rangle}\}_{\langle c,v\rangle \in C \times Dt}$ holds the class attibutes, where

  $$At_{\langle c,v\rangle} = \{at \mid at \text{ is an attribute of } c \text{ of type } Dt\}.$$

- $Assoc = \{Assoc_{\{c,c'\}}\}_{\{c,c'\} \subseteq C}$ holds metamodel associations, where

  $$Assoc_{\{c,c'\}} = \{as \mid as \text{ is an association between } c \text{ and } c'\}.$$

- $AsEnd = \{AsEnd_{\langle c,c'\rangle}\}_{\langle c,c'\rangle \in C \times C}$ holds association ends, where

  $$AsEnd_{\langle c,c'\rangle} = \{p \mid p \text{ is the role played by } c \text{ in an association } as \in Assoc_{\{c,c'\}}\}.$$

In the previous definition, we only consider those essential elements that are enough to build any MOF metamodel. Other elements like *cardinalities*, *aggregation* or *composition* relations are often considered since they are part of the UML metamodel. However, they do not provide more expressiveness to the structural part of metamodels as they can be equivalently expressed using OCL constraints on associations.

Next, we provide a structure to hold model information. We do not assume that a model always refers to a certain metamodel but we do assume that it provides instances, attributes, and links with type information. Otherwise, without this information, the model is just a drawing with a more or less intuitive meaning. Thus, we assume a set C of instance types, with a subset of basic types $Dt \subset C$, a set $At = \{At_{\langle c,v\rangle}\}_{\langle c,v\rangle \in C \times Dt}$ of attribute types, and a set of association ends $AsEnd = \{AsEnd_{\langle c,c'\rangle}\}_{\langle c,c'\rangle \in C \times C}$, much like in Definition 1.

Definition 2 (Model structure)

$$\mathcal{O}_{\mathcal{M}} = (O_{\mathcal{M}}, OAt_{\mathcal{M}}, OAsEnd_{\mathcal{M}}),$$

where:

- $O_{\mathcal{M}} = \{O_c\}_{c \in C}$ holds instances of type C, hence

  $O_c = \{o \mid o \text{ is an instance of type } c \in C\}$.

- $OAt_{\mathcal{M}} = \{OAt_{<c,v>}\}_{<c,v> \in C \times Dt}$ holds attribute values of type v provided in the instances of type c, where

  $OAt_{<c,v>} = \{at : O_c \to O_v \mid at \in At_{<c,v>}\}$.

- $OAsEnd_{\mathcal{M}} = \{OAsEnd_{<c,c'>}\}_{<c,c'> \in C \times C}$ holds the roles played by the instances of type c when they are linked to a set of instances of type c'. Hence,

$$OAsEnd_{<c,c'>} = \{p : O_{c'} \to \mathcal{P}(O_c) \mid p \in AsEnd_{<c,c'>}\}.$$

In the previous definition we build only on model elements that are essential to describe the structural part of a system, i.e., on those modeling elements that are enough to build what in UML is called a class diagram.

3 Metamodels and Models as Theories, Conformance as a Theory Interpretation

Membership equational logic (MEL) is an expressive version of equational logic. A full account of its syntax and semantics is given in [8]. MEL is implemented in the Maude system [6]. A MEL *specification* consists of

- a set of *sorts* (types);
- a partial order on sorts called the *subsorting* relation, which expresses the fact that some sorts can be subsorts of others;
- a set of *operations*, which are functions between the sorts. The number of input arguments of a function is called its *arity*. Constants can be seen as 0-ary functions.
- A set of *axioms* defining the operations. Axioms are possibly conditional *equations* between terms or *memberships* of terms into sorts.

A *term* is either a constant, a variable, or the application of a n-ary function to n terms of appropriate sorts. A *ground term* is a term without variables.

The MEL specification `STATE-LIST`, depicted in Figure 4 in Maude-like syntax, is an abstract language to describe lists of elements of the sort `State`. The sorts `State` and `StateList` are declared, and the sort `NonEmptyStateList` is declared a subsort of `StateList`. The `nil` constant and the `cons` function are the *constructors* of the sort `StateList`. The `_excludes_` function in infix notation is intended to check the absence of a class in a class list. Finally, notice that the specification is in the so-called *Order-Sorted Logic* fragment of MEL, since it does not contain any membership axioms.

```
spec STATE-LIST is
sorts State StateList NonEmptyStateList
subsort NonEmptyStateList < StateList
op nil : -> StateList
cons : State StateList → NonEmptyStateList
_excludes_ : StateList State → Bool
```

Fig. 4 Order sorted specification `STATE-LIST`.

As an example of a membership axiom, notice that the subsort declaration `NonEmptyStateList < StateList`, can be equivalently written as the conditional membership `z:StateList if z:NonEmptyStateList`, meaning that every not empty state list (i.e., every element of `NonEmptyStateList`) is also a state list (i.e., an element of `StateList`).

MEL specifications can be related by *theory interpretations*. We shall say that a theory T_1 *interprets* a theory T_2 if the sorts and subsorting relation of T_2 are exactly those of T_1, and the operations and axioms of T_2 include those of T_1. For instance, the specification given in figure 5, `interpretStateList`, interprets, in the defined sense, the specification `STATE-LIST`.

```
spec interpretStateList is
sorts State StateList NonEmptyStateList
subsort NonEmptyStateList < StateList
op nil : -> StateList
cons : State StateList → NonEmptyStateList
_excludes_ : StateList State → Bool
ops initState endState state3 : → State
cons(initState, cons(endState, nil) excludes state 3 = true
```

Fig. 5 An interpretation of `STATE-LIST`.

Finally, an order-sorted specification is *confluent and terminating* if for any ground term, by applying the equations (oriented from left to right as rewrite rules), a unique *canonical form* (i.e., a ground term that cannot be rewritten any further) is obtained after finitely many rewrites. In such specifications the equality between ground terms is decidable: two ground terms are equal iff their canonical forms are identical.

3.1 Metamodels and Models as Order-Sorted Theories

In this section we show how the metamodel and model structures provided in Section 2 are translated to order-sorted theories. We first describe the translation and then provide the formal definition.

The structural part of a metamodel $\mathcal{MM}$ is translated to an order sorted specification as follows:

- We always include the sorts: *Class*, *ClassCol* and *Value* with the constructors *nil* and *col*. *Integer*, *String* and *Boolean* are also included as subsorts of $Value$. These sorts allow us to define collection constructors just once, to deal with partiality, and to encode basic types;

- for each class $c \in C_{\mathcal{MM}}$ in the metamodel, a sort c, and a sort $Col[c]$ for lists of terms of sort c are created. Also, the following subsorting relations are included: $c \prec Class$ and $Col[c] \prec ClassCol$;
- for each pair $\langle c_1, c_2 \rangle \in Gen_{\mathcal{MM}}$ in the metamodel structure, we add two subsort declarations: $c_1 \prec c_2$ and $Col[c_1] \prec Col[c_2]$, i.e., we represent inheritance by subsorting which provides us with a partial ordering on classes and collection of classes;
- for each attribute $at \in At_{<c,v>}$ with $c \in C$ and $v \in Dt$ in the metamodel $\mathcal{MM}$, a function $at : c \to v$ is declared, i.e., an attribute becomes a function from the sort of the class to the sort of the attribute type;
- for each association $as \in Assoc_{\{c,c'\}}$ and association ends $r_1 \in AsEnd_{<c,c'>}$ and $r_2 \in AsEnd_{<c',c>}$ of as in the metamodel $\mathcal{MM}$, two function declarations are created: $r_1 : c \to c'Col$ and $r_2 : c' \to cCol$;
- The set of axioms $\Gamma_{\mathcal{MM}}$ is empty.

Definition 3 Metamodel Theories. A metamodel

$$\mathcal{MM} = (C, Gen, At, Assoc, AsEnd)$$

is specified by a theory $\mathcal{MM} = (\Omega, \Gamma)$ in order sorted logic, where $\Gamma = \emptyset$ and

- $\Omega = (S, \prec, \Sigma)$, where
  - $S = \{Class, ClassCol, Value\} \cup S_{Class} \cup S_{ClassCol} \cup S_{Value}$, with
    - $S_{Class} = \{c \mid c \in C\}$,
    - $S_{ClassCol} = \{Col[c] \mid c \in C\}$,
    - $S_{Value} = \{Boolean, Integer, String\}$,
  - $\prec \subseteq C \times C$ equals $Gen \cup (S_{Class} \times \{Class\}) \cup (S_{ClassCol} \times \{ClassCol\}) \cup (S_{Value} \times \{Value\})$,
  - $\Sigma = \{\Sigma_{q,k}\}_{q \in S^*, k \in S}$, where
    - $\Sigma_{c,Value} = \bigcup_{\langle c,v \rangle \in C \times (Dt \cup C)} \{at \mid at \in At_{\langle c,v \rangle}\}$,
    - $\Sigma_{c,Col[c']} = \bigcup_{\langle c,c' \rangle \in C \times C} \{p \mid p \in AsEnd_{\langle c,c' \rangle}\}$,
    - $\Sigma_{\lambda, ClassCol} = \{nil\}$,
    - $\Sigma_{Class.ClassCol, ClassCol} = \{col\}$,
    - $\Sigma_{q,k} = \emptyset$, otherwise.

Example 1 Basic UML metamodel as a Theory.
$BasicUML = (\Omega_{BasicUML}, \Gamma_{BasicUML})$, where

- $\Omega_{\mathbf{BasicUML}} = (S_{\mathbf{BasicUML}}, \prec_{\mathbf{BasicUML}}, \Sigma_{\mathbf{BasicUML}})$:
  - $S_{\mathbf{BasicUML}}$ is the union of the following sets:
    - $\{Class, ClassCol, Value\}$,
    - $S_{Class} = \{MClass, Association, Operation, Attribute, ModelElement, DataType\}$,
    - $S_{ClassCol} = \{Col[MClass], Col[Association], Col[Operation], Col[Attribute], Col[ModelElement], Col[DataType]\}$, and
    - $S_{Value} = \{Integer, Boolean, String\}$,
  - $\prec_{\mathbf{BasicUML}}$ contains, among others, the pairs $(MClass, Class)$, $(Association, Class)$,
  - $\Sigma_{\mathbf{BasicUML}}$ is the union of the following sets:
    - $\Sigma_{ModelElement,String} = \{name\}$,
    - $\Sigma_{ModelElement,Boolean} = \{isAbstract\}$,
    - $\Sigma_{MClass,String} = \{name\}$,
    - $\Sigma_{MClass,Boolean} = \{isAbstract\}$,
    - ...
    - $\Sigma_{Association,String} = \{name, sourceAssocEnd, targetAssocEnd\}$,
    - $\Sigma_{MClass,AssociationCol} = \{forward\}$,
    - $\Sigma_{MClass,OperationCol} = \{operations\}$,
    - $\Sigma_{Operation,MClassCol} = \{owner\}$,
    - $\Sigma_{MClass,AttributeCol} = \{classattributes\}$,
    - $\Sigma_{Attribute,MClassCol} = \{owner\}$,
    - $\Sigma_{Attribute,DataTypeCol} = \{type\}$,
    - $\Sigma_{DataType,AttributeCol} = \{attributes\}$,
    - $\Sigma_{Association,MClassCol} = \{endType\}$,
    - $\Sigma_{MClass,MClassCol} = \{specific, general\}$,
    - $\Sigma_{\lambda,ClassCol} = \{nil\}$,
    - $\Sigma_{ClassCol,ClassCol} = \{col\}$,
- $\Gamma_{\mathbf{BasicUML}} = \emptyset$.

Next, we describe our translation of models to order-sorted theories. The structure $\mathcal{O}_{\mathcal{M}}$ of a model $\mathcal{M}$ is translated to an order-sorted specification as follows:

- For each type c of an object in the model, we declare a sort c;
- for each element o of type c we declare a constant (as a 0-ary function) of the sort c, i.e., $o :\to c$.
- for each attribute value v of the attribute at in the instance o, we include an equation $at(o) = v$;
- for each association end $r_1 \in OAsEnd_{<c,c'>}$ linking an instance o and a collection of instances $o'_1, \ldots, o'_n$, we include an equation $r_1(o) = col(o'_1, col(o'_2, \ldots, nil))$.

Next, we provide the formal definition. Remember that a model $\mathcal{O}_{\mathcal{M}}$ assumes a set C of instance types, with a subset of basic types $Dt \subset C$, a set of attribute types $At = \{At_{\langle c,v \rangle}\}_{\langle c,v \rangle \in C \times Dt}$ and a set of association ends $AsEnd = \{AsEnd_{\langle c,c' \rangle}\}_{\langle c,c' \rangle \in C \times C}$.

Definition 4 Model Theories.

A model $\mathcal{O}_{\mathcal{M}} = (O_{\mathcal{M}}, Oat_{\mathcal{M}}, OAsEnd_{\mathcal{M}})$ is specified by an order-sorted theory $\mathcal{O}_{\mathcal{M}} = (\Omega_{\mathcal{M}}, \Gamma_{\mathcal{M}})$, where

- $\Omega_{\mathcal{M}} = (S_{\mathcal{M}}, \prec_{\mathcal{M}}, \Sigma_{\mathcal{M}})$, where
  - $S_{\mathcal{M}} = \{c \mid c \in C\}$,
  - $\prec_{\mathcal{M}} = \emptyset$,
  - $\Sigma_{\mathcal{M}} = \{\Sigma_{q,k}\}_{q \in S^*, k \in S}$ whose only nonempty components are $\Sigma_{\lambda,c} = \{o :\to c | o \in O_c\}$, for $c \in C$,
- $\Gamma_{\mathcal{M}} = \Gamma_{OAt_{\mathcal{M}}} \cup \Gamma_{OAsEnd_{\mathcal{M}}}$, where:
  - $\Gamma_{OAt_{\mathcal{M}}} = \cup_{<c,v> \in C \times Dt} \{at(o) = at^{Oat_{<c,v>}}(o) \mid o \in O_c, at \in At_{<c,v>}\}$.
  - $\Gamma_{OAsEnd_{\mathcal{M}}} = \cup_{<c,c'> \in C \times C} \{p(o) = \lfloor p^{OAsEnd_{<c,c'>}}(o) \rfloor \mid o \in O_{c'}, p \in As_{<c,c'>}\}$,

where $(\lfloor \_ \rfloor)$ is a function that represents elements in $\mathcal{P}(\mathcal{O}_c)$. i.e., sets of objects in the type c, as terms of sort $Col[c]$. The function $(\lfloor \_ \rfloor)$ builds a *list* (using the list-constructors *col* and *nil*) with the objects in the given set, sorted by their names and without repetitions.

Example 2 The Automaton Model.

- $S_{\mathbf{Automaton}} = \{MClass, Association, Operation, Attribute, DataType\}$
- $\Sigma_{\mathbf{Automaton}}$ contains the following set:
  - $\Sigma_{\lambda,MClass} = \{State, InitState, Transition, Automaton\}$;
  - $\Sigma_{\lambda,Association} = \{out\text{-}transition, in\text{-}transition, Ownership\}$;
  - $\Sigma_{\lambda,Attribute} = \{label, isActive, trace\}$;
  - $\Sigma_{\lambda,Operation} = \{getTransitions\}$;
  - $\Sigma_{\lambda,DataType} = \{String, Boolean\}$.
- $\Gamma_{\mathbf{Automaton}} = \Gamma_{Oat_{\mathbf{Automaton}}} \cup \Gamma_{OAsEnd_{\mathbf{Automaton}}}$, where
  - $\Gamma_{Oat_{\mathbf{Automaton}}}$ contains the axioms:
    - $name(State) = \text{“}State\text{”}$,
    - $name(InitState) = \text{“}InitState\text{”}$,
    - ...
    - $isAbstract(ModelElement) = \text{“}true\text{”}$,
    - $isAbstract(State) = \text{“}false\text{”}$, ...
  - $\Gamma_{OAsEnd_{\mathbf{Automaton}}}$ contains the axioms:
    - $endType(out\text{-}transition) = col(Transition, col(State, nil))$,
    - $forward(State) = col(out\text{-}transition, nil)$,
    - $forward(Transition) = col(out\text{-}transition, nil)$, ...
    - $forward(Automaton) = col(Ownership, nil)$,
    - $forward(Transition) = col(Ownership, nil)$,
    - $owner(isActive) = col(State, nil)$,
    - $attributes(State) = col(isActive, nil)$
    - ...
    - $owner(getTransitions) = col(Automaton, nil)$,
    - $operations(Automaton) = col(getTransitions, nil)$
    - $type(isActive) = col(Boolean, nil)$,
    - $attributes(Boolean) = col(isActive, nil)$,
    - ...

3.2 Conformance as a theory interpretation

Now, we are ready to capture the notion of structural and semantical conformance of a model to a metamodel. These definitions capture the intuitive idea that models are essentially interpretations of metamodels.

Definition 5 Structural conformance Given a model M and a metamodel MM, we say that the model M has structural conformance to MM if and only if the theory $\mathcal{O}_{\mathcal{M}}$ is an interpretation of the theory $\mathcal{MM}$.

Intuitively, this definition says that a model is conformant to a metamodel iff it preserves the structure of the metamodel and provides an interpretation for the sorts and the function symbols that are present in the metamodel. As expected, according to our definition, the model depicted in Figure 2 is conformant to the metamodel depicted in Figure 1 since the specification given in Example 2 is an interpretation of the specification given in Example 1.

Next, we consider semantical conformance. In [4], there exists a proposal of a formal executable equational semantics for OCL that extends the order-sorted specifications of metamodels and models that we have shown above. This equational semantics is defined for a substantial subset of OCL, including many operations on primitive types, collection operations, iterator operations (except the most general one, i.e., *iterate* and, quantifiers. It also considers how to interpret (possibly) recursive user-defined operations. The standard library of OCL specified in this semantics (without user-defined operations) is proved to be convergent.

Recall the OCL invariant “Invariant 1” provided at the begining of this work whose expression was context Class invariant1 : self.allParents()− > excludes(self) in the context of the on the BasicUML metamodel. This invariant is first translated to one that is equivalent but more convenient for the translation: Class.allInstances− > forAll(c| c.allParents− > excludes(c)). Parsing and type checking metamodel invariants is done in the theory that extends the metamodel theory with OCL basic operations plus the operations defined to interpret iterator operations following the different operation bodies provided by the user. We call this theory $\mathcal{MM}_{\mathcal{OCL}}$. To evaluate the invariants, we join to this theory the interpretation provided by the model (Automaton in our example), we call it $\mathcal{M}_{\mathcal{OCL}}$. Invariant1 is translated (automatically) to the term $forAll1(\ allInstances(Class))$ whose canonical form is obtained by rewriting in $\mathcal{M}_{\mathcal{OCL}}$[2]. The possibility of translating user defined operations to this semantics provides it with much flexibility, on the contrary, its lack would have impeded defining many invariants included in MDE standards whose definitions involve user-defined recursive functions in OCL.

Example 3 OCL executable equational specification. Excerpt.

[2] Recall that this formal semantics has been proved to be confluent and terminating.

$\ldots$
$allInstances(Class) = col(State, col(InitState,$
$col(Transition, col(Automaton, nil))))$
$general(State) = nil$
$general(InitState) = nil$
$general(Automaton) = nil$
$general(Transition) = nil$
$collect(nil) = nil$
$collect(col(x, xs)) = union(allParents(x), collect(xs))$
$allParents(x) = union(parents(x), collect(parents(x))$
$parents(x) = general(x)$
$forAll(col(x, xs)) = ifThenElse(excludes($
$allParents(x), x),$
$forAll(xs), true)$
$forAll(nil) = true$
$\ldots$

It is obvious (taking into account also the metamodel and model theories) that the user-defined functions terminate, so we will obtain a normal form for $forAll(allInstances(Class))$ that in our case, is $true$.

Remark 1 Let MM be a metamodel and let Inv be a set of invariants written in OCL that parse and typecheck correctly using the types and vocabulary of MM. We call Φ the set that represents all the invariants inv in the theory $\mathcal{M}_{\mathcal{OCL}}$.

Definition 6 Semantical conformance Given a metamodel MM, and a set of OCL invariants Φ which parse and typecheck correctly in MM, we say that the model M is semantically conformant to MM if and only if i) $\mathcal{O}_{\mathcal{M}}$ is an interpretation of $\mathcal{MM}$ and, ii) the normal form of every invariant in Φ is $true$ in $\mathcal{M}_{\mathcal{OCL}}$.

4 Representation in Maude

In this section we show how to automatically check the definitions proposed in Section 2. We gain automatic tool support because of Maude reflective capabilities and because the system is able to actually execute the equational specifications. The tool ITP/OCL [5] is able to automatically generate the metamodel and model theories from command lines inserted by a user. The modules created by ITP/OCL in Maude notation are shown in Figures 4 and 7. Notice, that they follow Definitions 3 and 4. Through these commands the user declare a metamodel and according to these information, the tool requires that the models inserted afterwards are indeed structurally conformant to the metamodel already provided. Also, the user can insert invariants for his/her metamodel that may make use of user defined recursive operations (in this case, the tool cannot guarantee termination). Then the tool is able to automatically check whether all these invariants or only some of them invariants are fullfilled by the model, i.e., the tool is able to automatically check semantical conformance of a model to a metamodel by rewriting the terms corresponding to the invariants to their normal form in the appropriate $\mathcal{M}_{\mathcal{OCL}}$ theory. In Figure 4, we show an excerpt of this theory.

```
fmod BasicUML is
sorts s Class ClassCol Value .
sorts MClass Attribute Association ModelElement .
sorts Operation DataType .
subsort Operation MClass Attribute
        Association DataType < ModelElement .
sorts MClassCol AttributeCol AssociationCol
      ModelElementCol DataTypeCol .
subsort OperationCol ClassCol AttributeCol
        AssociationCol DataTypeCol <
ModelElementCol .
sorts Integer Boolean String .
op name : ModelElement -> Value .
...
ops reverse forward : MClass -> AssociationCol .
op classatrributes : MClass -> AttributeCol .
op owner : AttributeCol -> MClassCol .
ops specific general : MClass -> MClassCol .
op nil : -> ClassCol .
op col : Class ClassCol -> ClassCol .
endfmod
```

Fig. 6 The metamodel BasicUML as a Maude module. Excerpt.

```
fmod Automaton is
including BasicUML .
ops State InitState Transition Automaton : -> MClass .
ops out-transition in-transition
                  Ownership : -> Association .
ops label isActive trace : -> Attribute .
ops string boolean : -> DataType .

eq name(State) = "State" .
...
eq target(out-transition) = col(Transition, nil) .
eq reverse(out-transition) = col(State, nil) .
eq source(out-transition) = col(State, nil) .
eq forward(out-transition) = col(Transition, nil) .
...
eq owner(isActive) = col(State, nil) .
eq classattributes(State) = col(isActive, nil) .
eq type(isActive) = col(boolean, nil) .
eq attributes(boolean) = col(isActive, nil) .
...
```

Fig. 7 The model Automaton as a Maude module. Excerpt.

5 Conclusion, Related, and Future Work

In this paper, we have proposed a formal definition of the concepts of model to metamodel structural and semantical conformance. Our approach formally captures the intuitive idea that models are essentially interpretations of metamodels. Our definitions extend in a natural way our previous work where we provided formal definitions for the notions of models and metamodels with

```
fmod OCLonAutomatonFromBasicUML is
including Automaton .
...
eq allInstances(Class) = col(State,col(InitState ,
col(Transition,col(Automaton,nil)))) .
eq general(State) = nil .
eq general(InitState) = nil .
eq general(Automaton) = nil .
eq general(Transition) = nil .
eq collect2(nil) = nil .
eq collect2(col(x,xs)) = union(allParents(x),collect1(xs)) .
eq allParents(x)=union(parents(x),collect2(parents(x)) .
eq parents(x)=general(x) .
eq forAll1(col(x,xs))=ifThenElse(includes(allParents(x),x),
true,forAll1(xs)) .
...
```

Fig. 8 The Automaton-OCL theory. Excerpt.

the aim of providing a formal semantics for the OCL language. We also show how conformance can be automatically checked using the ITP/OCL tool and discuss on related and future work. In a nutshell,

- metamodels, possibly enriched with OCL invariants, are represented as MEL specifications;
- models are represented as MEL specifications as well;
- structural conformance between a model and a metamodel means that the model theory provides an actual interpretation of the MEL specification denoting the metamodel;
- semantical conformance between a model and a metamodel requires, in addition to structural conformance, that all the invariants imposed on the metamodel become true in its instance model.

Probably, the closest work to ours is in [9,10] where they provide an algebraic definition for the notions of conformance and model transformation in the MOMENT2 framework. Perhaps, the major limitation of this approach concerning its definition of conformance is that they have not shown how to deal with user defined (possibly recursive) operations which although we understand that is not due to technical limitations, it prevents their system of being used with real specifications which usually make use of these kind of operations. On the other hand, considering model transformations is a matter of future work for us. Although both proposals share the same target formalism (equational logic) to define a semantics for OCL, and the same system (Maude) to develop tools based on these semantics, these are actually the unique coincidences between the two approaches. Concerning the used formalism, although both works employ equational logic (notice, however that [10] uses membership equational logic and we use order-sorted equational logic), the transformations from UML diagrams with OCL expressions to equational logic are completely different. They translate UML diagrams to terms and the semantics for the OCL expressions is given, basically, by an evaluating function taking as an argument the term representing the evaluation context; for each OCL expression, the definition of this function is provided by two (meta-) functions. Also, the equations generated by these functions are said to be always executable but this affirmation is not proven. In our work, however, UML models and OCL constraints are transformed into theories, which directly define i.e., without requiring the help of an evaluating function, the semantics of OCL expressions. Furthermore, we have formally proven that this semantics for OCL is indeed convergent.

There are also others academic and commercial tools offering support to metamodeling tasks that allow or grant some kind of conformance checking (only structural in the case of commercial tools except for Together CC, whose OCL support is limited). The paper [11] provides a good description of other approaches less related to ours. That work also presents how to do conformance checking using the PVS theorem prover in an interactive manner. The conformance checking supported by the Coq theorem prover in [12] is similar in features and methodology to the one supported by PVS.

References

1. Model-Driven Architecture. Available at `http://www.omg.org/mda/specs.htm`.
2. The Object Management Group. `http://www.omg.org`.
3. Object Constraint Language (OCL). Available at `http://www.omg.org/spec/OCL/2.0/`.
4. M. Egea. *An executable formal semantics for OCL with aoplications to model analysis and validation*. PhD thesis, Universidad Complutense de Madrid, 2008.
5. M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for UML + OCL static class diagrams. In Michael Johnson and Varmo Vene, editors, *AMAST*, volume 4019 of *Lecture Notes in Computer Science*, pages 368–373. Springer, 2006.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
7. Meta Object Facility (MOF) Core Specification. Available at `http://www.omg.org/spec/MOF/2.0/`.
8. A. Bouhoula, J.P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 1-2(236):35–132, 2000.
9. A. Boronat and J. Meseguer. An algebraic semantics for MOF. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2008.
10. A. Boronat. *Moment: a formal framework for model management*. PhD thesis, Universitat politecnica de Valencia, 2007.
11. Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 16(3), 2007.
12. Iman Poernomo. The meta-object facility typed. In Hisham Haddad, editor, *SAC*, pages 1845–1849. ACM, 2006.