PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 10-01

Representation of metamodels using inductive types in a Type-Theoretic Framework for MDE

Daniel Calegari Carlos Luna Nora Szasz Alvaro Tasistro

2010

Representation of metamodels using inductive types in a Type-Theoretic Framework for MDE
Calegari, Daniel; Luna, Carlos; Szasz, Nora; Tasistro, Alvaro
ISSN 0797-6410
Reporte Técnico RT 10-01
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, 2010

Representation of Metamodels using Inductive Types in a Type-Theoretic Framework for MDE

Daniel Calegari*, Carlos Luna†, Nora Szasz†, Álvaro Tasistro†
*Instituto de Computación, Universidad de la República, Uruguay
Email: dcalegar@fing.edu.uy
†Facultad de Ingeniería, Universidad ORT Uruguay
Email: {luna,szasz,tasistro}@ort.edu.uy

***Abstract*—We present discussions on how to apply a type-theoretic framework –composed out by the Calculus of Inductive Constructions and its associated tool the Coq proof assistant– to the formal treatment of model transformations in the context of Model-Driven Engineering. We start by studying how to represent models and metamodels in the mentioned theory, which leads us to a formalization in which a metamodel is a collection of mutually defined inductive types representing its various classes and associations. This representation has been put into use for carrying out and verifying on machine the well-known case study of the Class to Relational model transformation. We finally end up discussing ways in which the framework can be used to obtain provably correct model transformations.**

***Keywords*-model-driven engineering, model transformations, correctness, constructive type theory**

I. Introduction

This paper reports ongoing work on the application of a type-theoretic framework to the formal treatment of model transformations in the context of Model-Driven Engineering (MDE) [1]. Model transformations take as input models conforming to a given source metamodel and produce output models conforming to a given target metamodel. That is, metamodels specify collections of models and can thus be equated to model languages. In turn, they conform to metametamodels, which can be understood as specifying model languages and usually conform just to themselves.

We explore the idea of using the Calculus of Inductive Constructions (CIC) [2], [3] as a technical space for representing and dealing with the schema above and obtaining provably correct model transformations. The choice of the CIC is dictated by its very considerable expressive power as well as by the fact that it is supported by tools of industrial strength, e.g. the Coq proof assistant [3], [4]. More specifically, the CIC is a type theory with dependent types which allows to write logical formulæ of higher order about objects of inductive and functional types as well as about potentially infinite structures of co-inductive types. Within this framework, metamodels are represented as types, whilst model transformation can be specified for example by means of logical formulæ of $\forall\exists$ form, i.e. assertions that for every input model conforming to a given metamodel (type) and satisfying a certain (pre-)condition, an output model exists which conforms to a given metamodel and which stands in a certain relation with the input model. Proofs of such propositions in the CIC are constructive and therefore it is possible to automatically extract from each of them a provably correct (functional) program computing the output model from any appropriate input. Alternatively, one can specify the transformation by asserting and proving directly the existence of such a function. These are thus two ways in which it is possible to obtain certified zero-fault transformations.

The aim of this paper is to discuss how to represent models and metamodels in the CIC. This leads to the consideration of entities (e.g. classes) that mutually reference each other (e.g. through associations) and their possible representation by means of co-inductive or, alternatively, mutually inductive types, which is the approach that we eventually favor. This is done in section III below, after having provided a brief outlook on the CIC in Section II. We have carried out our method on the well-known Class to Relational model transformation [5], which we have fully developed in Coq. The corresponding code is available at [6]. Finally, in section IV, besides exposing some conclusions, we discuss several ways in which the type-theoretic framework can be used either for the development of provably correct transformations or for the verification of pre-existing ones, which leads to the formulation of further work.

The idea of using type theory in the context of MDE has been formulated before by Poernomo in [7], [8]. Poernomo formulates a type theory of his own –a variant of Martin-Löf's constructive type theory– and outlines the methods for representing MOF [10] models as types as well as specifications of transformations as $\forall\exists$ formulæ. However, he requires the use of co-inductive types to treat cyclic references within metamodels, which we here avoid. Another difference from Poernomo's work is that we have fully implemented a significant case study on machine, thanks to the choice of the CIC and the Coq tool.

Other works that propose type systems of their own for object models are [11], [12], but they are oriented towards defining executable transformation languages and

not towards reasoning about models and transformations. In contrast, [13], [14] explore the use of Maude (an executable rewriting logic language) as a formal notation for describing models and metamodels, and proving certain properties about them. In these works references are represented by means of object identifiers, thus introducing a level of indirection. Our representation, which consists of the mutual definition of inductive types, allows for the use of induction and recursion for the purpose of navigation. Also, an important difference with these works is that our framework is oriented towards the specification and formal verification of model transformations, whereas this is not considered in the former works.

II. The Calculus of Inductive Constructions

The Calculus of Inductive Constructions is a type theory, i.e. in brief, a higher order logic in which the individuals are classified into a hierarchy of types. The types work very much as in strongly typed functional programming languages which means that, to begin with, there are basic elementary types, recursive types defined by induction like lists and trees (called inductive types) and function types. An example of inductive type is given by the following definition of the lists of elements of (parametric) type `A`, which we give in Coq notation (data types are called "Sets" in CIC):

```
Inductive list : Set :=
  | nil : list
  | cons : A -> list -> list.
```

The type is defined by its constructors, in this case `nil: list A` and `cons : A -> list A -> list A` and it is understood that its elements are obtained as finite combinations of the constructors. Well-founded recursion for these types is available via the `Fixpoint` operator.

A (dependent) record type is a non-recursive inductive type with a single constructor and projection functions for each field of the type.

When the requirement of finiteness is removed we obtain the possibility of defining infinite structures, as in

```
CoInductive Stream : Set :=
    Cons : A -> Stream -> Stream.
```

The possibly infinite elements of coinductive types are evaluated on demand, i.e. employing lazy evaluation. This means that only those finite parts of the structure that are required by functions operating on it are effectively computed. In this case the recursion needs not be well-founded, and is available with the `CoFixpoint` operator.

On top of this, a higher-order logic is available which serves to predicate on the various data types. The interpretation of the propositions is constructive, i.e. a proposition is defined by specifying what a proof of it is and a proposition is true if and only if a proof of it has been constructed. As a consequence, elementary predicates are also defined as inductive types, by giving the corresponding proof constructors. The type of propositions is called `Prop`. The logical constants are defined accordingly, which means in particular that each universally quantified proposition $\forall x{:}A.P(x)$ is a type of functions taking objects a of type A into proofs of $P(a)$, whereas existentially quantified propositions $\exists x{:}A.P(x)$ are types of pairs formed by objects a of type A and proofs of $P(a)$. In view of the latter, a formula of $\forall\exists$ form, i.e. $\forall x{:}A|P(x)\ \exists y{:}B|Q(x,y)$ is proven by a function that maps input objects of type A satisfying precondition P to output of type B which stands in the relation Q with the input, together with the proof that this latter condition holds. Then such formulæ constitute a natural way of expressing program specifications. From each proof of one such formula it is possible to mechanically extract a function in the sense of ordinary functional programming languages which operates only on the input and output data and leaves the proof of the condition Q aside. The resulting program, say f, remains provably correct with respect to the original specification, i.e. for all $a{:}A$ satisfying P, $Q(a, f(a))$ holds.

III. Formalization of Metamodels and Models

This section introduces the method used to describe metamodels and models within the CIC. Although the approach does not tie itself to a specific metamodel specification language, we use, for the sake of presentation, a simplified version of the MOF [10] equivalent to the Kernel MetaMetaModel (KM3) [15]. As to the CIC, we use the syntax of the Coq proof assistant [4].

We use the well-known Class to Relational model transformation [5] as an example, actually in the simplified version found in the QVT Specification [16]. The transformation describes how persistent classes of a simple UML class diagram are mapped to tables of a RDBMS model. Here we shall focus on the representation of metamodels and their models rather than on the transformation itself. Nevertheless, one version of the complete transformation developed following our approach can be found at [6].

In Figure 1 we show a metamodel of UML class diagrams. Classes can contain one or more attributes and can belong to a class hierarchy. Each attribute has a type that can be another class or a primitive datatype (string, boolean, integer, etc.). Associations are defined between classes with a direction from source to destination. We shall refer to this metamodel as "the example" from here on.

In what follows we try to achieve the greatest clarity in a minimum of space, given the restrictions. As a consequence our explanations will often be essentially intuitive, although clearly formalizable.

A. Formalizing Metamodels

The representation of metamodels in the CIC requires to devise methods for formalizing the constructs characteristic of the object paradigm, e.g. object identity, model subtyping

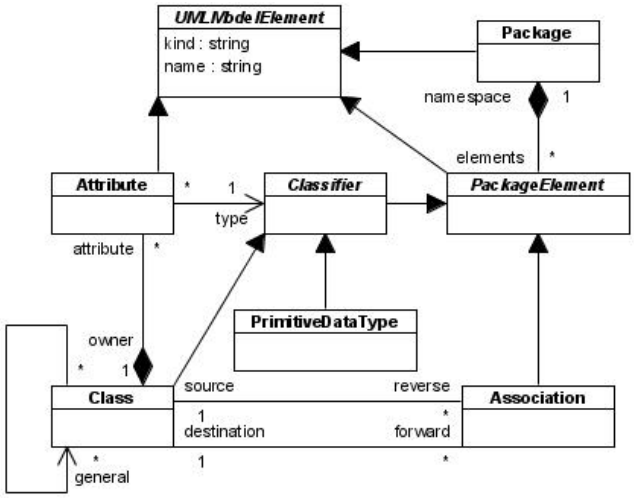

Figure 1. UML Class Metamodel

and model-type inference. As the reader will notice in the next subsections, the construction of the representation here described can in a large measure be automatized for a given metamodeling language.

Data Types and Enumerations: Coq supports primitive data types like strings, booleans and natural numbers, among others, via libraries equipped with many useful functions. It is also possible to define enumeration types and use them to define class attributes. Enumeration types are directly represented in Coq as inductive types. In the example, suppose we want to represent the kind of a `UMLModelElement` with an enumeration type `ModelType` instead of using a string as in the diagram. The enumeration type can be represented as an inductive type with constructors `Persistent` and `Non-Persistent`, as follows.

```
Inductive ModelType : Set :=
    | Persistent : ModelType
    | Non_Persistent : ModelType.
```

Classes, Attributes and References: A class consists of attributes and references. An attribute has a name, a multiplicity and a type. A reference has a name, a multiplicity and a type (of the element been referenced), and it may have an opposite reference (bidirectional references). We represent associated classes together using mutually inductive types. For each class its attributes and references are represented as components of the corresponding type by means of a constructor which has its attributes and references as parameters. In the example, the classes and their corresponding attributes as well as the associations can be defined as follows.

```
Inductive UMLModelElement : Set :=
  | Build_UMLModelElement (name : string)
                          (kind : string)
  with Package : Set :=
       | Build_Package (oid : nat)
        (elements : list PackageElement)
  with PackageElement : Set :=
       | Build_PackageElement
  with Classifier : Set :=
       | Build_Classifier
  with Class : Set :=
       | Build_Class (oid : nat)
                     (general : list Class)
                (attribute : list Attribute)
  with PrimitiveDataType : Set :=
       | Build_PrimitiveDataType (oid : nat)
  with Association : Set :=
       | Build_Association (oid : nat)
                           (source : Class)
                      (destination : Class)
  with Attribute : Set :=
       | Build_Attribute (oid : nat)
                      (type : Classifier).
```

In order to manipulate the components of the classes we define projections for each attribute and reference. They are trivially defined using pattern matching. As an example we show just the projection of the attribute `name` of the `UMLModelElement` class.

```
Definition UMLModelElement_name
           (o : UMLModelElement) : string :=
  match o with
    | (Build_UMLModelElement n _ _ _ _) => n
  end.
```

Notice the presence of the components named `oid` in the representation of each class. This provides a means for identifying the actual objects that are to be instances of the various classes, i.e. the `oid`s implement object identity, beyond the constructor identity provided by CIC. We can thus tell whether two objects are the same by using a boolean function associated to their class.

Now, if for given class and reference an opposite reference exists, the elements of the source class must form part of those of the target class and viceversa, i.e. the objects of both classes are not well-founded. In general, this situation might arise whenever several classes are mutually related through cycling associations and we can characterize it as the admissibility of circularity in the actual construction of (thereby infinite) objects. In these cases we seem to need co-inductive types, as pointed out in [7], [9]. However, taking such an approach forces us in the general case to introduce all the mutually connected classes as mutually defined co-inductive types. And then some disadvantages arise, concerning both the correctness of the representation and its ease of use.

The main problem is that there will in general be cycles of references of classes of the model for which no actual cycle at the level of object formation is intended to occur, even when some other reference cycles in the same model allow the circularity of object formation. Hence, we have in general that some of the classes involved in the represented metamodel will not be intended to actually contain infinite structures, namely those participating in references in which

no actual cycle at the level of objects is admissible. But even in such cases, the definition of the classes as co-inductive allows them to contain infinite structures. This compromises the correctness of the representation in at least two respects: first, circularity at the level of objects cannot be prevented at syntax (type) level and secondly, the termination of functions on these types cannot be enforced. Although these restrictions could be imposed on the co-inductive definition of the model, that would lead to a representation too awkward to manage in practice.

We therefore decide to use only inductive types. This is enough for the cases in which no circularity at the level of objects is to be allowed, since the well-foundedness of the latter is imposed by construction. More precisely, we represent directly only unidirectional references, using mutually inductive types whenever circularity at the level of the objects is not allowed in the metamodel. If, on the contrary, we should have to allow for such circularity then we "cut" the association cycle at a convenient point and represent the now loose references directly as pairs of source and target instances in a separate structure. This procedure has as a particular case that of the bi-directional associations.

In the example, we assume that the attributes of a class cannot have that same class as type. Should this be not assumed (e.g. in the case of allowing for classes representing recursive types), circularity in the formation of the instances of `Class` would be allowed. In such case we could resolve this problem, by "cutting" the reference from `Attribute` to `Classifier` (i.e. erasing the second argument, named `type`, of the constructor of the type `Attribute`) and adding the corresponding information as explicit ordered pairs of the following type:

```
Inductive AttrClass : Set :=
  | Build_AttrClass (src : Attribute)
                    (dest : Classifier)
```

Deciding which references to cut may be in general non-trivial and depend on the transformation to be written. This is a point in which, although possible in principle, the automatization of the representation of the metamodel might not be desirable.

Multiplicities: Attributes and references have multiplicities. Each multiplicity has a lower and an upper value and multiplicity 1 is assumed if none is declared. Multiplicity 1 is represented with a given type. Multiplicity 0..1 is represented with the `option` type constructor, which has constructors `None` representing no element and `Some x` for elements `x` in the original type. Finally, if the upper multiplicity value is greater than 1, the multiplicity is represented with a (possibly ordered) list type. In the example we have the following attribute definitions.

```
[1-1] -- name : string
[0-1] -- subAttribute : option Attribute
[0-*] -- elements : list PackageElement
```

Generalization and Abstract Classes: Classes may be abstract, i.e. have no objects. If needed, this can be represented by adding a fixed boolean component to the corresponding type.

It is also possible to define generalization relations between classes as references between the supertype and the subtypes. Each supertype has an optional reference to each of its subtypes, but to avoid bidirectional references subtypes do not have a reference to the supertype. This impacts on the definition of the inductive types defined before. We add parameters to the constructors of each type to represent the generalization relation. For example, to represent the generalization between `Classifier` and `Class` and `PrimitiveDataType` we need to modify the definition of `Classifier` as follows.

```
...
Classifier : Set :=
  | Build_Classifier (subClass : option Class)
         (subPrimitiveDataType :
                     option PrimitiveDataType)
...
```

We can now define functions to infer the concrete type of an instance in a hierarchy and downcast that instance to the corresponding type.

```
Definition isTypeOfClass (c : Classifier)
  : bool := match (Classifier_subClass c) with
              | None => false
              | Some a => true
            end.

Definition asTypeClass (c : Classifier)
  : option Class := Classifier_subClass c.
```

Since we do not have bidirectionality, we cannot directly navigate from a subclass to a property of its superclass. We need in this case to search for the corresponding instance in the collection of all instances of the supertype. As a result we have partial functions, as the one presented next to upcast an `Attribute` as an instance of a `Classifier`. Partiality is achieved by use of the `option` type former in the result type of the function.

```
Fixpoint Attribute_super
  (l : list UMLModelElement)
  (a : Attribute) : option UMLModelElement :=
  match l with
   | nil => None
   | cons m l2 =>
      match (UMLModelElement_subAttribute m)
      with
       | None => Attribute_super l2 a
       | Some a2 =>
          match (beq_Attribute a a2) with
            | true => Some m
            | false => Attribute_super l2 a
          end
      end
 end.
```

B. Formalizing Models

We represent a model that conforms to a metamodel as a record that contains a collection (represented as a list) with all the instances of each class (represented as an inductive type) defined in the metamodel. In the example a model conforming to the source metamodel `SimpleUML` would be a record of the following type:

```
Record SimpleUML : Set :=
 mkSimpleUML {umlModelElementAllInstances
                : list UMLModelElement;
              packageAllInstances
                : list Package;
              packageElementAllInstances
                : list PackageElement;
              classifierAllInstances
                : list Classifier;
              classAllInstances
                : list Class;
              primitiveDataTypeAllInstances
                : list PrimitiveDataType;
              associationAllInstances
                : list Association;
              attributeAllInstances
                : list Attribute }.
```

In the case discussed above in which the attributes of a class were allowed to have that same class as type, giving rise to circularity in the formation of instances of `Class` and to the subsequent introduction of the type `AttrClass` to cut such circularity, we should add a field to the former record containing a list of pairs of that type.

Additional constraints can be defined as Coq predicates that must be satisfied by any model conforming to the metamodel. We must consider two kinds of constraints, namely the structural constraints and the invariants.

The structural constraints are those satisfied in the original formulation of the metamodel, due to the structural rules of the modeling language, but not in the CIC representation. In the example, a structural constraint is that any `Classifier` must be either a `Class` or a `PrimitiveDataType` but not both. This constraint can be expressed in Coq as follows:

```
Definition sc_SubClassifier (m : SimpleUML)
  : Prop :=
  forall c:Classifier,
       (In c (classifierAllInstances m)) ->
   Xor ((isTypeOfClass c) = true)
       ((isTypeOfPrimitiveDataType c) = true).
```

Invariants, on the other hand, are conditions that must be satisfied by every model but cannot be captured by the structural rules of the modeling language. In this case the modeling language must be supplemented by another language, e.g. OCL. Invariants can be non-structural constraints (like uniqueness and domain of attributes, referential integrity, etc) and business rules (general domain properties that must be satisfied). An invariant of the example is that the name of an attribute is unique within a `Class`. In OCL:

```
context Class inv:
  self.attribute->isUnique(name)
```

This constraint can be expressed in Coq as follows:

```
Definition nsc_AttributeUniqueness
  (m : SimpleUML) : Prop :=
  forall o:Class,
    (In o (classAllInstances m)) ->
    let names := map Attribute_name
                     (Class_attribute o)
    in
      forall name:string,
        (In name names) ->
        (unique string string_eq_bool
                names name) = true.
```

All the constraints, be they structural or invariants, are added as fields to the records representing the models.

IV. Conclusions and Further Work

We have explored methods for representing metamodels and the models conforming to them in a type-theoretic framework given by the Calculus of Inductive Constructions. We have thereby arrived at the conclusion that it is adequate and convenient to formalize metamodels as collections of mutually defined inductive types. Each of these types represents a class of the metamodel in question and its elements contain in general elements of other classes of the model, namely those to which the class being represented bears an association. We have shown how to deal within this framework with those cases in which circularity in the formation of objects is allowed, as for instance with bidirectional associations. In general, this is solved by breaking the problematic association cycles at some point and adding a representation of the thereby vanishing association explicitly as a type of pairs. Constraints of any kind can also be dealt with without difficulty.

The main benefit of representing models and metamodels within the CIC is that we use the same formalism to deal with both model elements and constraints (structural and invariants). In particular, to build a model that conforms to certain metamodel it is necessary to provide not only the set of model elements but also a proof that those elements satisfy the constraints. Therefore, in this formalism model conformance is equivalent to type checking. The Coq system is used as an editor for the CIC and provides assistance for the construction of well-typed elements in a unified environment. Besides, we can go further an prove general properties about metamodels and models. These will be expressed as propositions in the CIC and proved using the Coq proof assistant.

Within this context, model transformations can be naturally defined as functions from a source to a target metamodel. They can be specified by logical formulæ of $\forall\exists$ form, i.e. assertions that for every source model conforming to a certain metamodel and satisfying a certain (pre-)condition,

another target model exists which conforms to a certain metamodel and which stands in a certain relation with the source model. A correct transformation is then a function from the source metamodel to the target metamodel that satisfies the specification, i.e. that can be used to obtain an output that satisfies the condition under $\exists$ for any input that conforms to the source metamodel and satisfies the pre-condition.

One possible approach to build such a transformation is to use the Coq extraction mechanism. Since proofs of propositions in the CIC are constructive, it is possible to automatically extract functional programs from proofs of $\forall\exists$ formulæ. These functions are provably correct with respect to the given specification [17]. The extraction mechanism in Coq allows generation of such functions in several functional languages, like Haskell, Scheme and OCaml.

Another approach is to directly write the function that transforms the source model into the target model and prove it correct. This case can be adapted for providing verification of pre-existing transformations: take namely a transformation specified in some transformation language (QVT or ATL [18], among others), encode it into Coq's functional language and then prove the result correct. This is actually the approach explored in [9] where we experiment on how to encode ATL constructs, and also the one taken for the development of the Class to Relational QVT model transformation case study [6]. The fact that we have actually completed these cases shows the feasibility of the general approach. We have omitted in this paper details of this aspect of our work for reasons of space.

Of course the fully formal way to accomplish the latter approach requires to employ formally founded (and desirably automatic) translation from the source transformation language into the CIC. This amounts to developing a denotational semantics of at least a significant fragment of the source language in type theory. One first step in this direction is to automatize a translation into Coq of metamodels written in a given metamodeling language. This must be subject of further work. We hope also to apply either of the approaches just considered to still more significant case studies, i.e. of transformations and properties of greater generality.

ACKNOWLEDGEMENT

This work has been partially funded by the National Research and Innovation Agency (ANII) of Uruguay through the "Verification of UML Based Behavioral Model Transformations" project.

REFERENCES

[1] S. Kent, "Model-Driven Engineering" in *IFM*, LNCS 2335. Springer, 2002, pp. 286–298.

[2] T. Coquand and C. Paulin, "Inductively Defined Types" in *COLOG-88: Proceedings of the international conference on Computer logic*. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 50–66.

[3] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. SpringerVerlag, 2004.

[4] The Coq Development Team, *The Coq Proof Assistant: Reference Manual*, 2009, version 8.2. [Online]. Available: http://coq.inria.fr

[5] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt, "Model Transformations in Practice Workshop" in *MoDELS Satellite Events*, LNCS 3844. Springer, 2005, pp. 120–127.

[6] "Verification of UML-Based Behavioral Model Transformations Project". [Online]. Available: www.fing.edu.uy/inco/grupos/coal/field.php/Proyectos/ANII09

[7] I. Poernomo, "A Type Theoretic Framework For Formal Metamodelling" in *Architecting Systems with Trustworthy Components*, LNCS 3938. Springer, 2004, pp. 262–298.

[8] I. Poernomo, "Proofs-as-Model Transformations" in *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 214–228.

[9] D. Calegari, C. Luna, N. Szasz, and A. Tasistro, "Experiment with a Type-Theoretic Approach to the Verification of Model Transformations". Universidad ORT Uruguay, Facultad de Ingeniera. Documento de Investigacin no.8, 2009. [Online]. Available: www.ort.edu.uy/fi/pdf/documento8fi.pdf

[10] OMG, "Meta Object Facility (MOF) 2.0 Core Specification". Object Management Group, Specification Version 2.0, 2003.

[11] J. Steel and J. M. Jézéquel, "On model typing". *Software and Systems Modeling*, vol. 6, no. 4, pp. 401–413, December 2007.

[12] J. Edwards, D. Jackson, and E. Torlak, "A type system for object models". *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 189–199, 2004.

[13] J. Romero, J. Rivera, F. Durán, and A. Vallecillo, "Formal and Tool Support for Model Driven Engineering with Maude". *Journal of Object Technology*, vol. 6, no. 9, 2007.

[14] A. Boronat, J. Meseguer, and I. Ramos, "An Algebraic Semantics for MOF" in Fundamental Approaches to Software Engineering. pp. 377–391, 2008.

[15] ATLAS Group, *KM3: Kernel MetaMetaModel*, Manual v0.3 ed., LINA & INRIA, 2005.

[16] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation". Object Management Group, Specification Version 1.0, 2008.

[17] C. Paulin-Mohring, "Extracting F(omega)'s Programs from Proofs in the Calculus of Constructions" in *POPL*, 1989, pp. 89–104.

[18] ATLAS Group, *ATL: Atlas Transformation Language*, User Manual v0.7 ed., LINA & INRIA, 2006.