Experiment with a type-theoretic approach to the verification of model transformations

Calegari, Daniel
Luna, Carlos
Szasz, Nora
Tasistro, Álvaro

Documento de Investigación No. 8
Facultad de Ingeniería
Universidad ORT Uruguay
Setiembre, 2009
ISSN 1688-8707

# Documento de Investigación

ISSN 1688-8707

# Experiment with a type-theoretic approach to the verification of model transformations

**Daniel Calegari** (Instituto de Computación,UDELAR, Uruguay)
**Carlos Luna** (Facultad de Ingeniería Universidad ORT Uruguay)
**Nora Szasz (**Facultad de Ingeniería Universidad ORT Uruguay**)**
**Alvaro Tasistro (**Facultad de Ingeniería Universidad ORT Uruguay**)**

**Documento de Investigación No. 8**
Facultad de Ingeniería
Universidad ORT Uruguay
Setiembre, 2009

# Experiment with a Type-Theoretic Approach to the Verification of Model Transformations

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*—This paper presents ongoing work on the application of constructive type theory for safe development in Model-Driven Engineering. In particular, we put into practice a certification framework based on the Calculus of Inductive Constructions (CIC) as follows: we first describe methods for representing metamodels and specifications of model transformations as types and then outline principles for representing given transformations, written in a certain transformation language, as functions in CIC. Then properties of these functions are proven, including their correctness with respect to sufficiently expressive specifications, thereby providing a semi-formal way to verify the original transformations. We carry out a concrete case taken from an on-line base of basic examples and patterns of use of the ATL transformation language, implementing it on machine using the Coq proof assistant.**

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

## I. Introduction

Model-Driven Engineering (MDE) [1] is a software engineering paradigm based on the specification of models of a system as the primary development activity. The construction of a software system is driven by model transformations, starting form abstract models of the system and transforming them until an executable model (i.e. code) is generated. MDE success strongly depends on the availability of appropriate languages and tools to describe models, to perform model transformations and to validate their correctness.

In particular, the development of model transformations should –as it happens with ordinary program development– start off with the statement of a specification, characterizing in this case the kinds of models accepted as input as well as the corresponding output models. In the long run, we are interested in the application of formal methods in MDE and therefore wish to investigate approaches and tools to develop certified model transformations that respect given functional specifications or alternatively verify given transformations with respect to semantic properties. At the same time we want to consider all kinds of models and model transformations, including behavioral ones. Very few proposals take the kind of verification mentioned into account – some of them are [2]–[4]. In the same way, most of the proposals focus on the transformation of structural models of a system. There are few ones that consider behavioral aspects, as in [5]–[7]. In this work we experiment with the use of constructive type theory for MDE in two ways: first, as to the representation of metamodels and their models, and secondly as to the use of the formalism for specifying and verifying given transformations, written in some model transformation language, in at least a semi-formal manner.

As described in [8] and summarized in Figure 1, a model transformation takes as input a model `Ma` conforming to a given source metamodel `MMa` and produces as output another model `Mb` conforming to a given target metamodel `MMb`. The model transformation can be defined as well as a model `Mt` which itself conforms to a model transformation metamodel `MMt`. This last metamodel, along with the `MMa` and `MMb` metamodels, has to conform to a metametamodel `MMM`. There are different languages that support this approach [9], [10], like the KM3 [11] anf MOF [12] metamodeling languages, and the ATL [8] and QVT [13] transformation languages.

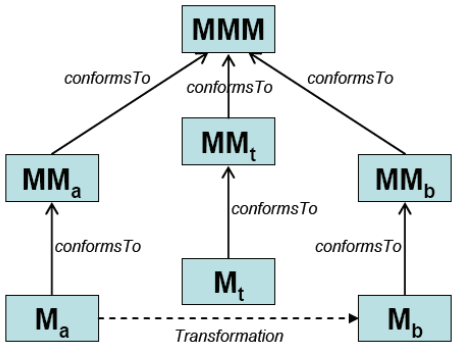


Figure 1. An overview of model transformation

We explore the idea of using the Calculus of Inductive Constructions (CIC) [14], [15] for representing and dealing with the schema above. The CIC is a version of type theory with dependent types which allows to express logical formulæ of higher order about ordinary objects of basic and inductive types as well as possibly infinite structures of co-inductive types. We choose this formalism since it is a

consolidated theory with powerful means of expression and tool support of industrial strength. Within this framework, metamodels like `MMa` and `MMb` above are represented as types, whilst model transformations are specified by logical formulæ of $\forall\exists$ form, i.e. assertions that for every source model conforming to a certain metamodel (type) 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. Proofs of such propositions in CIC are constructive and therefore allow to automatically extract from them (functional) programs computing the target model from the corresponding input, which are correct with respect to the given specification. This is thus a way in which it is possible to obtain certified zero-fault model transformations.

In this work we focus in the use of the approach for verifying pre-existing transformations, written in a certain given model transformation language. We outline principles that allow to represent the given transformation as a function in CIC. Then properties of this function can be proven, including its correctness with respect to sufficiently expressive specifications, thereby providing a semi-formal way to verify the originally given transformation. We put all these ideas into practice in a concrete case taken from an on-line base of basic examples and patterns of use of the ATL transformation language [19]. The whole work has been implemented on machine using the Coq proof assistant [15], [21].

The idea of using type theory in the context of MDE has been formulated before by Poernomo in [16], [17]. He formulates for that purpose a type theory of his own, which is a variant of Martin-Löf's constructive type theory, and outlines the methods for representing MOF [12] models as types and specifications of transformations as $\forall\exists$ formulæ in very much the same way that we employ here. However he does not put the ideas into practice in concrete examples. We, on the other hand, have the purpose to assess the concrete feasibility of the approach by carrying out a case study in its entirety, up to and including its implementation on machine. This has led us to choose both existing and already well tested theory and tool, namely the CIC with the Coq proof assistant. Also in contrast to Poernomo's work, we stress the question of translating transformations written in a given model transformation language into functions of CIC in order to give means for indirectly verifying the original transformations.

The rest of this paper is structured as follows. In Section II we give a brief outlook on the CIC. Section III presents the formalization of metamodels as types. In Section IV we show how to translate model transformations to functions in CIC. Section V shows how to (indirectly) verify the transformations by formally verifying their translation. From beginning to end we use a case study to illustrate the process. Finally, Section VI presents a short summary with concluding remarks and an outline of future work.

## 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.

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. Formalizing Metamodels

This section introduces the formal setting used to describe metamodels within CIC. Although this approach does not stick to some specific metamodel specification language, we use the Kernel MetaMetaModel (KM3) [11] to extract those metamodel constructions we need to represent. We start with a simple example, which will be extended in later sections. This example is illustrated using the Coq proof assistant [15]. The complete example can be found at [18].

### A. A Running Example: Tree2List

As a running example we use a basic transformation taken from [19], which describes how a tree is transformed into a list. This transformation is complete enough to illustrate those aspects discussed before.

The metamodel of Figure 2 represents a tree whose elements have a name as attribute. Nodes can be just internal nodes or leaves. Leaves can be of small, medium or big size. The root element should be a `Node`.

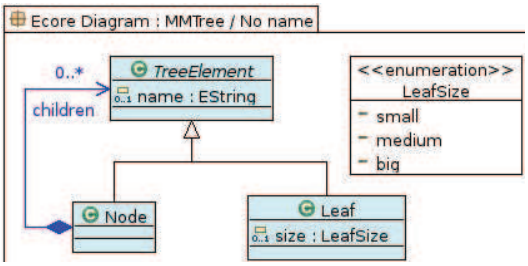


Figure 2. Tree Metamodel

This source metamodel is represented in KM3 as follows.

```
package MMTree {
  abstract class TreeElement {
    attribute name : String;
    reference parent[0-1] :
                    Node oppositeOf children;}

  class Node extends TreeElement {
    reference children[*] container :
                TreeElement oppositeOf parent;}

  class Leaf extends TreeElement {
    attribute size : LeafSize    }

  enumeration LeafSize {
    literal small;
    literal medium;
    literal big;       }
}
```

### B. Encoding Metamodels

In this section we show how we represent KM3 notions in CIC. We use Coq notation for the example, as described in Section II.

*Data Types and Enumerations:* The primitive data types supported by KM3 are String, Boolean, Integer and Double. This types are available in the Coq system via libraries equipped with many useful functions.

In KM3 it is also possible to define enumeration types and use them to define class attributes. Enumeration types are directly represented in Coq as simple inductive definitions. In the example, the enumeration `LeafSize` is represented as an inductive type with constructors `small`, `medium` and `large`:

```
Inductive LeafSize : Set :=
    | small : LeafSize
    | medium : LeafSize
    | big : LeafSize.
```

*Classes, Attributes and References:* In KM3 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). It may have an opposite reference and can be declared as being a container.

We represent associated classes together using mutually coinductive 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. If an opposite reference exists, there will be components in the source class that are elements of the target class and viceversa. This provokes circularity in the formation of the objects of the classes, i.e. they are in general not well-founded, which is why we need coinductive types. In the example, the classes can be defined as follows.

```
CoInductive TreeElement : Set :=
  | Build_TreeElement (name : string)
                      (parent : option Node)
  with Node : Set :=
         | Build_Node
                (children : list TreeElement)
  with Leaf : Set :=
         | Build_Leaf (size : LeafSize).
```

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 one of these definitions.

```
Definition TreeElement_name:
                      TreeElement -> string :=
  (fun t:TreeElement =>
    match t with
      | (Build_TreeElement n _) => n
  end).
```

*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, with 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 class definition for the example above we already show the following attribute definitions.

```
[1-1] -- name : string
[0-1] -- parent : option Node
[0-*] -- children : list TreeElement
```

*Generalization and Abstract Classes:* Classes may be abstract, i.e. have no objects. This can be represented by adding a fixed boolean component to the corresponding type indicating this fact. It is also possible to define generalization relations between classes. These relations are handled in the same way as references between the supertype and the subtypes. Each subtype has a reference to the supertype, while the supertype has one optional reference to each of its subtypes.

This impacts over the definition of the coinductive types defined before. We add to the constructors of each type a fixed parameter `abstract`, and parameters to represent the generalization relation, as follows.

```
CoInductive TreeElement : Set :=
  | Build_TreeElement (name : string)
                      (parent : option Node)
                      (abstract : bool := true)
                      (subNode : option Node)
                      (subLeaf : option Leaf)
  with Node : Set :=
         | Build_Node ...
                      (abstract : bool := false)
                      (super : TreeElement)
  ...
```

To complete this section we present the encoding of the target metamodel of the example. The metamodel is shown in Figure 3.

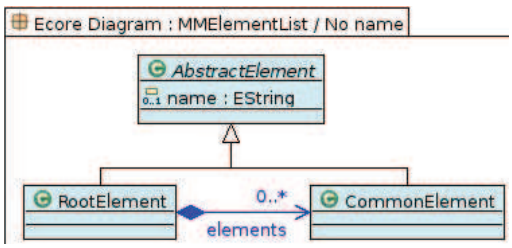


Figure 3. List Metamodel

It represents an ordered list of elements. The root element should be a `RootElement`. Our Coq representation of this metamodel is similar to the one done before, as shown next.

```
CoInductive AbstractElement : Set :=
  | Build_AbstractElement
    (name : string)
    (abstract : bool := true)
    (subCommonElement : option CommonElement)
    (subRootElement : option RootElement)
  with
    CommonElement : Set :=
    | Build_CommonElement
      (abstract : bool := false)
      (super : AbstractElement)
  with
    RootElement : Set :=
    | Build_RootElement
      (elements : list CommonElement)
      (abstract : bool := false)
      (super : AbstractElement).
```

### C. 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 defined in the metamodel.

Additional constraints can be defined as Coq predicates that must be satisfied by any model conforming to the metamodel. To this effect we attach the condition as another field of the record. In the example, a non-structural constraints is that a `TreeElement` must be either a `Node` or a `Leaf` but not both. This constraint can be expressed in Coq as follows:

```
Definition constraintMMTree (l : list Node) :
  Prop := forall n:Node, (In n l) ->
            forall te:TreeElement, (
              In te (getAllChildren n)) ->
              Xor ((isTypeOfNode te) = true)
                  ((isTypeOfLeaf te) = true).
```

where the function `getAllChildren` (defined later) returns the children of a node.

In the example a model conforming to the source metamodel `MMTree` would be a record of the following type.

```
Record MTree : Set :=
 mkMTree
   {MTree_treeElementAllInstances :
                              list TreeElement;
    MTree_leafAllInstances : list Leaf;
    MTree_nodeAllInstances : list Node;
    MTree_constraintMMTree :
    constraintMMTree MTree_nodeAllInstances}.
```

Finally, a model conforming to the target metamodel `MMTree` will be a record of the following type.

```
Record MElementList : Set :=
 mkMElementList
  {MElementList_abstractAllInstances :
                         list AbstractElement;
   MElementList_commonAllInstances :
                           list CommonElement;
   MElementList_rootAllInstances :
                            list RootElement}.
```

## IV. Representation of Transformation Language Constructs

There are different model transformation approaches, as described in [9], [10]. In our case we select a model-to-model relational approach. This is a declarative approach which is based on specifying relations among source and target model elements.

As with the formalization of metamodels, our approach does not stick to some specific model transformation language. However, we take the declarative part of the Atlas Transformation Language (ATL) [8] for a test of concepts, and show how it can be represented in Coq. Not every ATL construct is considered (e.g. modules and execution modes) since they are not relevant for our study.

We illustrate this proposal with the example we have been developing in the former sections. We specify an ATL transformation from `MMTree` to `MMElementList` that takes a tree as an input and produces a list with the following conditions [19]:

- the tree root (of type `MMTree!Node`) has to be transformed into the element list "root" (of type `MMElementList!RootElement`)
- every `MMTree!Leaf` element should be transformed into a `MMElementList!CommonElement` element
- the root element's `elements` reference should contain an ordered set composed of the "transformed equivalent" of the leaves. The order is given by its size (big ones comes at first, small ones comes at the end).

*Data Types:* ATL is based on the OMG Object Constraint Language (OCL) [20] for both its data types and its declarative expressions. It defines primitive types (boolean, integer, real, string), tuples, enumerates, collections (set, ordered set, bag, sequence), among others. All these types can be represented in Coq as described in section III.

There are special operations for data type manipulation. We show how we represent some interesting ones which are used in the example:

`oclIsUndefined()` tests whether the value of an expression is undefined. This is useful when applied on an attribute with a multiplicity 0..1 for example. By using the `option` type in the model representation in these cases we can check this condition.

`allInstances()` returns a set containing all the currently existing instances of some class. Our representation of models contains all the instances of each class.

`refImmediateComposite()` is a reflective operation that returns the immediate composite of an object. This can be represented as an opposite reference on a type and a function which retrieves that reference.

`oclIsTypeOf(t:oclType)`, `oclAsType(t:oclType)` The first one returns a boolean value stating whether an object is an instance of type `t` while the other casts an object to an instance of type `t`. We know the type of each variable at definition time since we do not represent polymorphic types. The only case these operations are useful is when representing generalization and we want to navigate from the supertype to one of it subtypes. These operations can be defined as boolean functions over the supertype returning an `option` type of one of it subtypes.

In the example we ask if a `TreeElement` is a `Leaf` with the function `isTypeOfLeaf` and then perform a downcast with the function `asTypeLeaf`. Note the use of the option type to indicate whether the cast can be performed.

```
Definition isTypeOfLeaf
            (te : TreeElement) : bool :=
  match (TreeElement_subNode te) with
    | None => match (TreeElement_subLeaf te)
                with
                | None => false
                | Some a => true
              end
    | Some a => false
  end.

Definition asTypeLeaf (te : TreeElement) :
      option Leaf := TreeElement_subLeaf te.
```

*OCL Declarative Expressions:* ATL uses additional OCL declarative expressions in order to structure the code. ATL `If-Then-Else`, `Let` (which enables the definition of variables) and constant expressions (constant values of any supported data type) are natively supported in Coq. Helper/attribute call expressions (call of an helper/attribute) as well as operation call expressions (call of a standard operation defined for a supported data type) are supported in Coq as we explain in the next subsection. Finally, the collection iterative expressions are supported in Coq with recursion operators on lists [21].

*Helpers and Attributes:* ATL helpers factorize code that can be called from different points of an ATL transformation [8]. An ATL helper is defined by the following elements: a name, a context type, a return value type, an ATL expression that represents the code of the ATL helper, and an optional set of parameters, in which a parameter is identified by a couple (parameter name, parameter type). From a functional point of view an attribute is a helper that accepts no parameters. Both helpers and attributes are represented as functions in the richly-typed functional programming language provided by Coq. Functions range from simple to complex recursive and corecursive (possibly infinite) ones.

The example declares `isTreeNodeRoot()`, a helper function that checks whether the current tree element is the root of the tree.

```
helper context MMTree!Node def :
  isTreeNodeRoot() : Boolean =
    self.refImmediateComposite().
                              oclIsUndefined();
```

To represent this we define a function that receives a `Node` and checks if its supertype has a reference to a parent

type. Note the need of mutual references to allow navigation between supertypes and subtypes.

```
Definition isTreeNodeRoot (n: Node) : bool :=
  match (TreeElement_parent
                        (Node_super n)) with
    | None => true
    | Some _ => false
  end.
```

Another helper used in the example is a function which sorts the leaves retrieved via a Depth-First Search (DFS) according to their size and to their position in the tree.

```
helper context MMTree!Node def :
getLeavesInOrder():OrderedSet (MMTree!Leaf) =
  let leavesList : OrderedSet (MMTree!Leaf) =
   self.getAllChildren()->select(currChild
       | currChild.oclIsTypeOf(MMTree!Leaf))
 in
  leavesList->select(leaf | leaf.size = #big)
    ->union(leavesList->select(leaf
                      | leaf.size = #medium))
    ->union(leavesList->select(leaf
                      | leaf.size = #small));
```

The translation of this function to Coq is straightforward since `select` and `union` functions can be implemented using filtering and appending operations on lists.

```
Definition getLeavesInOrder (n : Node) :
  list Leaf :=
    app (filter (compareLeafSize big)
                (leavesList n))
        (app (filter (compareLeafSize medium)
                     (leavesList n))
             (filter (compareLeafSize small)
                     (leavesList n))).
```

The function `compareLeafSize` determines whether the size of a leaf has a given size. It is trivially defined using pattern matching. The function `leavesList` iterates over the node's children obtained with the function `getAllChildren` and selects those elements that are leaves. We determine whether a `TreeElement` is a `Leaf` with the function `isTypeOfLeaf` and perform a downcast with the function `asTypeLeaf` shown above.

```
Definition leavesList (n : Node) :
  list Leaf := fold_right selectCastLeaf nil
                        (getAllChildren n).

Definition selectCastLeaf (te : TreeElement)
  (ll : list Leaf) : list Leaf :=
  match (isTypeOfLeaf te) with
    | true => match (asTypeLeaf te) with
                | None => ll
                | Some a => cons a ll
              end
    | false => ll
  end.
```

Finally, the helper `getAllChildren` retrieves all tree elements via a DFS starting from the root node.

```
helper context MMTree!Node def :
 getAllChildren () :
  OrderedSet(MMTree!TreeElement) =
   self.children->iterate(child;
   elements:OrderedSet(MMTree!TreeElement) =
            OrderedSet{} |
    if child.oclIsTypeOf(MMTree!Node) then
      elements.union(child.getAllChildren())
    else
      elements.append(child)
    endif);
```

The corresponding Coq function is the following.

```
CoFixpoint getAllChildren (n : Node) :
                          list TreeElement :=
  fold_right appChild nil (Node_children n)
  with
    appChild (te : TreeElement)
             (l : list TreeElement) :
                          list TreeElement :=
    match (isTypeOfLeaf te) with
      | true => cons te l
      | false => match (asTypeNode te) with
                   | None => nil
                   | Some nc =>
                    app (getAllChildren nc) l
                 end
    end.
```

*Rules:* As defined in [8], the matched rules constitute the core of an ATL declarative transformation since they make it possible to specify the kind of source elements for which target elements must be generated and the way the generated target elements have to be initialized. A matched rule is introduced by the following construction.

```
rule rule_name {
  from in_var : in_type [(condition)]
       [using { var1 : var_type1 = init_exp1;
        ...  }]
  to out_var1 : out_type1 (bindings1),
     ... }
```

The source pattern is defined after the keyword `from`. It enables to specify a model element variable that corresponds to the type of source elements the rule has to match. When defined, the local variable section is introduced by the keyword `using`. The target pattern of a matched rule is introduced by the keyword `to`. It serves to specify the elements to be generated when the source pattern of the rule is matched, and how these generated elements are initialized (bindings). An optional condition (expressed as an ATL expression) within the rule source pattern is used to select the subset of the source elements that conform to the matching type.

We implement a rule in Coq as a function from the type of the source element to the type of the target elements. Local variables are auxiliary functions. The ATL example transformation is composed by two matched rules. The first one transforms a `Leaf` into a `CommonElement`.

```
rule Leaf2CommonElement {
  from s : MMTree!Leaf
  to t : MMElementList!CommonElement(
                  name <- s.name) }
```

We represent this in Coq as a cofixpoint function that constructs a `CommonElement` and its supertype (`AbstractElement`) from a `Leaf`. As elements are mutually referenced, corecursion is crucial to define the rule.

```
CoFixpoint Leaf2CommonElement (l : Leaf) :
  CommonElement :=
  Build_CommonElement (Build_AbstractElement
       (TreeElement_name (Leaf_super l))
       (Some (Leaf2CommonElement l)) None).
```

The second matched rule transforms root nodes into `RootElement` of a list. Each `RootElement` has as `elements` the result of mapping the matched rule `Leaf2CommonElement` to the result of the function `getLeavesInOrder()` applied to the root node.

```
rule TreeNodeRoot2RootElement {
  from rt : MMTree!Node (rt.isTreeNodeRoot())
  to lstRt : MMElementList!RootElement (
          name <- rt.name,
          elements <- rt.getLeavesInOrder()) }
```

We represent this as another cofixpoint function that constructs a `RootElement` from a `Node` mapping every `Leaf` to a `CommonElement`. Leaves are sorted using the function `getLeavesInOrder`.

```
CoFixpoint TreeNodeRoot2RootElement (n : Node) :
  RootElement :=
  Build_RootElement (map Leaf2CommonElement
                    (getLeavesInOrder n))
                    (Build_AbstractElement
      (TreeElement_name (Node_super n))
       None
      (Some (TreeNodeRoot2RootElement n))).
```

In view of the representation of models as consisting of collections of elements belonging to the various types that represent the classes of the metamodel, we must map the corresponding rule onto each one of such collections, filtering those elements satisfying the condition. Conditions can be directly translated into Coq boolean functions. The representation of collections as lists allows us to use the well-known mapping and filtering operators on lists for representing the rules. In our example, the translation of the transformation is as follows.

```
Definition Tree2List (m: MTree) :
  MElementList :=
  mkMElementList (
    map TreeNodeRoot2RootElement
        (filter isTreeNodeRoot
                (MTree_nodeAllInstances m))).
```

For the sake of simplicity we we have considered smaller records for both models, containing only those fields that are relevant in this transformation.

## V. Certified Model Transformations

We have already shown how to represent metamodels as types in CIC, and models as elements of the corresponding types. We now wish to (indirectly) verify the original ATL transformation by formally verifying its translation. To this effect we will express specifications as explained in Section II, i.e. as $\forall\exists$ formulæ. In this case we want to prove properties of the specific function $F$ obtained by the translation. The specifications to consider, which will have the form $\forall ma{:}\mathit{MMa}|P(ma)\ \exists mb{:}\mathit{MMb}|Q(ma,mb)$ will be solved by applying the function $F$ to any given input $ma$ and proving that $Q(ma, F(ma))$ holds whenever $P(ma)$ holds.

Returning to the example, the transformation can be specified by a proposition stating that for every model `Model_MMTree` in which there is only one tree `Node` root, there exists a model `Model_MMElementList` that fulfills the conditions stated at the beginning of Section IV.

For space and readability reasons we just show a simplified version of the postcondition stating that there is only one `RootElement` in the target model. The proof of the full specification given in section IV is available at [18].

```
Lemma Cert_Tree2List:
   forall ma: MTree, OneRoot ma ->
      {mb: MElementList | OneElement ma mb}.

Definition OneRoot (ma:MTree) : Prop :=
  length (MTree_nodeAllInstances ma) = 1
    /\ forall n:Node,
         (In n (MTree_nodeAllInstances ma))
         -> (isTreeNodeRoot n) = true.

Definition OneElement (ma:MTree)
   (mb: MElementList) : Prop := length
      (MElementList_rootAllInstances mb) = 1.
```

In the proof of this lemma we use the function `Tree2List` to build the target model.

The interactive proof assistant helps building proofs using tactics that implement backward reasoning, from conclusions to premises. We refer to the Coq documentation [21] for further details. For those readers familiar with the Coq environment we show the proof of the lemma `Cert_Tree2List`.

```
Proof.
intros;exists(Tree2List m);unfold OneElement.
transitivity(length (filter isTreeNodeRoot
                (MTree_nodeAllInstances m))).
unfold Tree2List; apply map_length.
transitivity(length(MTree_nodeAllInstances m));
   [f_equal;apply immutableListLemma;
    apply H | apply H].
Qed.
```

`immutableListLemma` proves that if a filter is applied on a list and every element of the list satisfies the filtering predicate, the list does not change

## VI. Conclusions and Further Work

We have shown that type theory can be used in practice as a certification framework wherein to handle metamodels, models and transformations. To this effect, we have implemented on machine the representation of metamodels as types and models as elements of the corresponding types, outlined a method for translating a number of ATL constructs into the functional language of type theory, and shown how to represent a model-to-model relational approach to the specification of model transformations, with the corresponding correctness proofs. All this has allowed to perform an indirect verification of given model transformations.

The outcome of the experiment is encouraging as to the possibility of scaling up the approach to real size transformations, especially thanks to the use of an industrial-strength proof assistant as Coq.

A natural next step is to investigate fully formal verification of transformations, which is specially interesting for automating standard model refinements. To this end, we should develop a formal semantics and proof rules for transformation languages which can be done in particular by completing a formal translation to type theory along the lines outlined in this work.

## 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*, ser. Lecture Notes in Computer Science, M. J. Butler, L. Petre, and K. Sere, Eds., vol. 2335. Springer, 2002, pp. 286–298.

[2] D. Varró and A. Pataricza, "Automated Formal Verification of Model Transformations," in *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, ser. Technical Report, no. TUM-I0323. Technische Universität München, September 2003, pp. 63–78.

[3] A. Narayanan and G. Karsai, "Towards Verifying Model Transformations," *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 191–200, 2008.

[4] C. Pons and D. Garcia, "A Lightweight Approach for the Semantic Validation of Model Refinements," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 43–61, 2008.

[5] I. Raedts, M. Petkovic, Y. S. Usenko, J. M. E. M. van der Werf, J. F. Groote, and L. J. Somers, "Transformation of BPMN Models for Behaviour Analysis," in *MSVVEIS*, J. C. Augusto, J. Barjis, and U. Ultes-Nitsche, Eds. INSTICC PRESS, 2007, pp. 126–137.

[6] M. Garcia and R. Möller, "Certification of Transformations Algorithms in Model-driven Software Development," in *Software Engineering 2007*, ser. GI-Edition Lecture Notes in Informatics, W.-G. Bleek, J. Räsch, and H. Züllighoven, Eds., vol. 105, 2007, pp. 107–118.

[7] B. Graaf and A. van Deursen, "Model-Driven Consistency Checking of Behavioural Specifications," in *MOMPES '07: Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 115–126.

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

[9] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, March 2006.

[10] K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, 2006.

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

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

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

[14] 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.

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

[16] I. Poernomo, "A type theoretic framework for formal metamodelling," in *Architecting Systems with Trustworthy Components*, ser. Lecture Notes in Computer Science, R. H. Reussner, J. A. Stafford, and C. A. Szyperski, Eds., vol. 3938. Springer, 2004, pp. 262–298.

[17] 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.

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

[19] "ATL Basic Examples and Patterns." [Online]. Available: $www.eclipse.org/m2m/atl/basicExamples_patterns/$

[20] OMG, "UML 2.0 Object Constraint Language," Object Management Group, Formal Specification formal/06-05-01, 2006.

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