A type-theoretic framework for certified model transformations

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

Documento de Investigación No. 9
Facultad de Ingeniería
Universidad ORT Uruguay
9 de julio de 2010

ISSN 1688-8707

Documento de Investigación

ISSN 1688-8707

A type-theoretic framework for certified model transformations

Daniel Calegari (Instituto de Computación, UDELAR)
Carlos Luna (Instituto de Computación, UDELAR; 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.9

Facultad de Ingeniería
Universidad ORT Uruguay

9 de julio de 2010

A Type-Theoretic Framework for Certified Model Transformations

Daniel Calegari[1], Carlos Luna[1,2] Nora Szasz[2], and Álvaro Tasistro[2]

[1] Instituto de Computación, Universidad de la República, Uruguay
{dcalegar,cluna}@fing.edu.uy
[2] Facultad de Ingeniería, Universidad ORT Uruguay
{luna,szasz,tasistro}@ort.edu.uy

Abstract. We present a framework based on the Calculus of Inductive Constructions (CIC) and its associated tool the Coq proof assistant to allow certification of model transformations in the context of Model-Driven Engineering (MDE). The approached is based on a semi-automatic translation process from metamodels, models and transformations of the MDE technical space into types, propositions and functions of the CIC technical space. We describe this translation and illustrate its use in a standard case study.

1 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 feasibility of the approach is based on the existence of a semi-automatic construction process driven by model transformations, starting from abstract models of the system and transforming them until an executable model is generated. In consequence, the quality of the whole process strongly depends on the quality of the model transformations. The highest level of quality is achieved by proving desired properties of the transformations. Although formal verification techniques may be expensive, they can be helpful in guaranteeing the correctness of critical applications where no other verification technique is acceptable. Since the MDE approach is intended to succeed in a broad spectrum, we think it is worth exploring how formal verification techniques could be applied within it.

As 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*. There are well known metamodeling languages like the MOF [2] and KM3 [3]. In some cases, there are conditions (called invariants) that cannot be captured by the structural rules of these languages, in which case modeling languages are supplemented with another logical language, e.g. the Object Constraint Language (OCL) [4]. There are different model transformation approaches, as described in [5,6]. In our case we select a model-to-model relational approach, which is based on specifying a transformation as a set of relations (rules) that must hold between source and target model elements. Languages within this approach are QVT [7] and ATL [8].

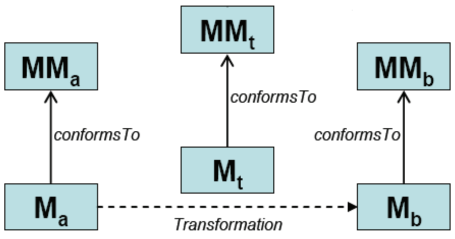

Fig. 1. An overview of model transformation

There are basically two levels at which the verification of a model transformation can be exercised: the model and the metamodel levels. Model-level verification works on specific source and target models related by a transformation. Verification techniques within this approach are mainly based on model-checking or simply testing. This approach is in many cases a practical and valuable aid but it cannot ensure the zero-fault level of quality since it checks a finite number of specific cases. Furthermore, there exist well-known limitations such as the state-explosion problem within model checking. An interesting work on the model-level verification of properties is [9] where the language Alloy is used for writing declarative model transformations and the Alloy Analyzer tool is used to conduct fully automated analysis of certain properties with the limitations mentioned above.

In contrast, metamodel-level verification ensures that a model transformation respects certain relations between model instances conforming to the source and target metamodels. This requires the use of formal verification techniques. Works within this approach are [10,11]. The first one is limited to the verification of model refinement transformations whereas the second one is used to prove only semantic equivalence of models.

We are concerned to *metamodel-level* verification of model transformations, considering *any kind of transformation and properties*. We expect the approach to be helpful whenever *zero-fault model transformations* are required.

We propose the construction of a type-theoretic framework for the certification of model transformations, representing the schema in Figure 1. In particular, we explore the idea of using the Calculus of Inductive Constructions (CIC) [12] as a technical space for dealing with provably correct model transformations. Within this framework, metamodels *MMa* and *MMb* above are represented as inductive types. On the other hand, each transformation rule of the model transformation *Mt* is represented as a logical formula (called *TRule*) of $\forall\exists$ form stating that for every model element satisfying a certain (pre-)condition, there exists a target model element which stands in the relation specified by the transformation rule with the source model element.

The correctness of the model transformation is stated as the following logical formula.

$$\forall\ Ma{:}MMa.\ (Pre(Ma) \rightarrow \forall\ Mb{:}MMb.\ (TRules(Ma,Mb) \rightarrow Post(Ma,Mb)))$$

where *Pre* is a translation of the source invariants, *TRules* is the conjunction of the transformation rules, and *Post* is a translation of the target invariants plus any other desired property to be proved. A proof of this formula ensures that the transformation rules satisfy the target invariants as well as the desired properties. We propose a semi-automatic translation process from the MDE technical space –used by developers– to the CIC technical space –used for formal verification.

The choice of the CIC is dictated by its very considerable expressive power as well as by the fact that it is supported by a tool of industrial strength, namely the Coq proof assistant [13]. As one example of its applicability, Coq has been used for the development and formal verification of a compiler of a large subset of the C programming language [14].

The idea of using type theory in the context of MDE has been formulated before by Poernomo in [15,16]. He formulates a type theory of his own –a variant of Martin-Löf's constructive type theory– and outlines a method for representing MOF models as types. Then, he follows the classical approach in type theory where pre- and post-conditions are represented as types, and a program (transformation) is derived as a function between those types. Our work differs in the representation of metamodels as will be explained later. Another important difference is that he performs program derivation to obtain a transformation whereas we translate a given transformation as a formula and verify this translation with respect to certain pre- and post-conditions. Finally, we base our proposal on an existent type theory with its corresponding supporting verification tool, which allows us to put into practice the ideas presented, unlike the works in [15,16].

As compared to previous work by the authors, the representation of model transformations described here differs substantially from the one presented in [17], as will be explained later.

The remainder of the paper is structured as follows. We first describe our framework in Section 2. In Section 3 we give some details about the formal representation of models and metamodels, and about model transformations in Section 4. Then, in Section 5 we explain how properties are verified. Finally, in Section 6 we present a short summary with concluding remarks and an outline of further work.

2 Outline of the Approach

We use the Calculus of Inductive Constructions (CIC) as a technical space for dealing with provably correct model transformations. In the following sections the CIC is introduced and our framework is outlined, using a running example.

2.1 The CIC as a Technical Space

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

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`.

We refer to [18,12] for further details on the CIC and Coq, respectively.

2.2 The Framework at a Glance

Although our approach is language independent, we are working with the ATL technical space. ATL (Atlas Transformation Language, [8]) is a hybrid of declarative and imperative transformation language. Since we are concerned with a model-to-model relational approach, we focused on the declarative part of ATL. In this context, an ATL transformation specification is composed of rules that define the correspondence between source and target model elements. In this technical space, source and target metamodels are specified using KM3 (Kernel MetaMetaModel, [3]) which provides a textual concrete syntax that eases the coding of metamodels.

During software construction a developer specifies the input and output metamodels and the transformation between them. We propose that at this point a separation of duties is implemented for performing formal verification. We conceive the participation of a (human, expert) verifier, who will carry out a semi-automatic translation process from the ATL to the CIC technical space, as outlined in Figure 2.

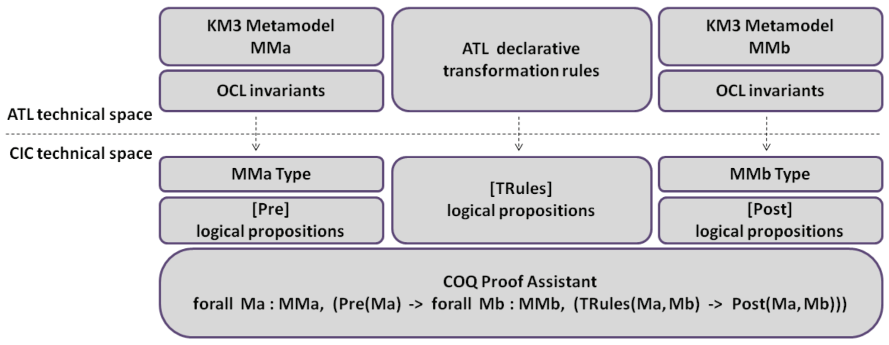
KM3 Metamodel MMa
OCL invariants
ATL declarative transformation rules
KM3 Metamodel MMb
OCL invariants
ATL technical space
CIC technical space
MMa Type
[Pre] logical propositions
[TRules] logical propositions
MMb Type
[Post] logical propositions
COQ Proof Assistant
forall Ma : MMa, (Pre(Ma) -> forall Mb : MMb, (TRules(Ma, Mb) -> Post(Ma, Mb)))

Fig. 2. An outline of our approach

The first step of the process is the formalization of the KM3 source and target metamodels as inductive types (`MMa` and `MMb`). Then, every ATL transformation rule is transformed into a logical proposition involving both the source and target metamodels (`TRules`). The third step is the translation of every OCL invariant into a logical proposition. Source invariants are taken as pre-conditions (`Pre`) of the model transformation. Target invariants and the desired properties of the transformation are taken as post-conditions (`Post`), which will be our proof goals. Finally, the verification is interactively performed in Coq by proving the post-conditions assuming that both the pre-conditions and the transformation rules hold. Additionally, the verifier can use the full expressiveness of Coq in order to include post-conditions that cannot be, or are not suitable to be, expressed in OCL.

2.3 A Running Example

We will illustrate our proposal by using an example based on a simplified version of the well-known Class to Relational model transformation [19]. Figure 3 shows both metamodels of this transformation. An UML class diagram consists of classes which contain one or more attributes. Each attribute has a type that is a primitive datatype. Every class, attribute and primitive data type is generalized into an abstract UML model element which contains a name and a kind (persistent or not persistent). On the other side, a RDBMS model consists of tables which contains one or more columns. Each column has a type, and every RDBMS model element is generalized into an abstract RDBMS model element.

The transformation describes how persistent classes of a simple UML class diagram are mapped to tables of a RDBMS model with the same name and kind. Attributes of the persistent class map to columns of the table. The type of the column is a string representation of the primitive data type associated to the attribute.

This example is clearly not a critical application where our approach is particularly helpful. However, it is complete and simple enough to exemplify out

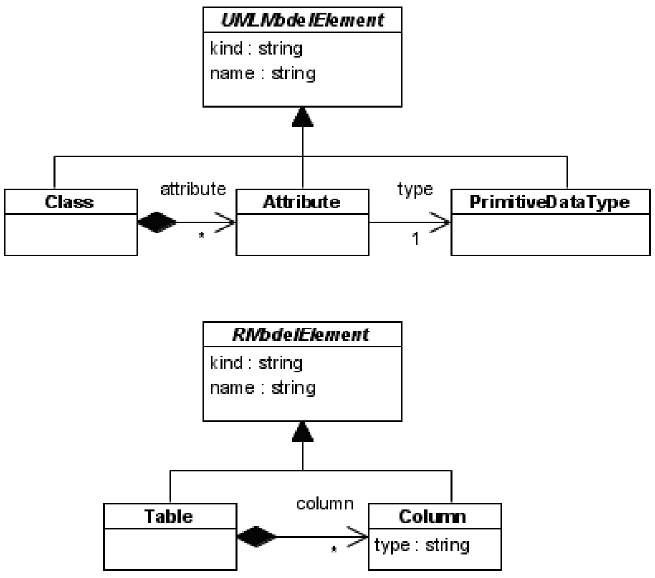

Fig. 3. UML metamodel and RDBMS metamodel

approach, and has been used as a standard test case for various transformation languages. For this reason, it will be used in the following sections in order to exemplify the verification process steps.

3 Formalization of Metamodels and Models

In this section we show how to represent metamodels and models in the CIC. For metamodels we show how every KM3 construction is translated into Coq notation, using the UML metamodel of the example mentioned in Figure 3 above. The whole translation has been defined and implemented as an ATL transformation. For space reasons we cannot include it here but it can be found in [20].

Data Types and Enumerations Coq supports primitive data types like strings, booleans and natural numbers, among others, via libraries equipped with many useful functions. This is enough to represent ATL primitive types. In ATL 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 with one constructor for each enumeration literal.

Classes and Attributes A class has attributes. An attribute has a name, a multiplicity and a type. We represent classes using inductive types. For each class its attributes are represented as components of the corresponding type by means of a constructor which has its attributes as parameters. A class can be abstract, meaning that there are no direct instances of it. This impacts on the representation of models as explained below, but not on the inductive representation of the class. In the example above, the class `UMLModelElement` is defined in Coq as follows.

```
Inductive UMLModelElement : Set :=
  | Build_UMLModelElement (oid : nat) (name : string) (kind : string)
```

Notice the presence of the component named `oid` in the representation of the `UMLModelElement`. 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 the CIC.

In order to manipulate the components of the classes we define projections for each attribute. 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.
```

References A reference represents an association between classes. It has a name, a multiplicity and a type (of the element been referenced), and it may have an opposite reference (bidirectional association). The natural choice for representing associations is by lists of pairs of related elements. In order to optimize navigability, the references of each class can be considered as components of the corresponding type, in the same way as attributes. This results in using mutually inductive types for representing related classes.

In the example below, the references from a `Class` to its `Attribute`s is represented as a component within the constructor of the `Class` type.

```
Inductive Class : Set :=
  | Build_Class ...
                (attribute : list Attribute)
```

Now, if for a 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 [15,17]. 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 use the first representation mentioned above, i.e. associations as list of pairs of related elements. This procedure has as a particular case that of the bi-directional associations.

Deciding which references are represented in either way in an optimal way depends on how the model elements are used in the transformation. This is a point in which, although possible, the automation of the representation of the metamodel might not be desirable. In the implemented translation of [20] the references of each class are represented as components of the corresponding type, and circularity must be manually "cut" by the human verifier.

Multiplicities Attributes and references have multiplicities. Each multiplicity has a lower and an upper value and multiplicity `1` is assumed if none is declared.

When representing references as components, multiplicity reflects itself in the type of the component. This is the same as in the case of attributes. 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. If the upper multiplicity value is greater than `1`, the multiplicity is represented with a (possibly ordered) list type. Multiplicity 1 corresponds just to the type of the component.

In the example we have the following multiplicities.

```
[1-1] -- name : string
[0-*] -- attribute : list Attribute
```

When representing associations as list of pairs of related elements, multiplicity is enforced by explicit constraints on the number of pairs allowed for each element of the participant classes.

Generalization and Abstract Classes In ATL it is also possible to define generalization relations between classes. The CIC does not have a notion of subtyping between types, unlike [15]. We represent this notion as references from the subtypes to it(s) supertype(s). With this representation we can easily navigate from a subclass to a property of it superclass. Notice that when a generalization exists, the `oid`s are located only at the topmost supertype.

In the example, the `PrimitiveDataType` class has a reference to its supertype `UMLModelElement`.

```
PrimitiveDataType : Set :=
  | Build_PrimitiveDataType (super : UMLModelElement)
```

The whole UML metamodel in Figure 3 is defined in Coq as follows.

```
Inductive UMLModelElement : Set :=
  | Build_UMLModelElement (oid : nat) (name : string) (kind : string).

Inductive Class : Set :=
  | Build_Class (super : UMLModelElement)
                (attribute : list Attribute)
  with
    Attribute : Set :=
      | Build_Attribute (super : UMLModelElement)
                        (type : PrimitiveDataType)
  with
    PrimitiveDataType : Set :=
      | Build_PrimitiveDataType (super : UMLModelElement).
```

Finally, a model that conforms to a metamodel is represented as a record containing the lists of instances of each non-abstract metamodel element, as suggested in [21]. In the example a model conforming to the UML metamodel would be a record of the following type.

```
Record SimpleUML : Set :=
 mkSimpleUML {classAllInstances : list Class;
              primitiveDataTypeAllInstances : list PrimitiveDataType;
              attributeAllInstances : list Attribute
             }.
```

In the example every element in the model is reachable from the `Class` instances, so the record can in fact be reduced to just a list of classes.

4 Translation of the Model Transformation

We show next how ATL constructs can be translated into Coq notation, using the example transformation in Section 2.3. Briefly, every ATL declarative transformation rule is transformed into a logical proposition, and helper (auxiliary) functions are translated into Coq functions. Not every ATL construct is considered since some of them (e.g. modules) are not relevant for our study. The whole transformation of the example can be found in [20].

Data Types ATL's data types are based on the OCL. They include 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 3.

OCL Declarative Expressions ATL uses additional OCL declarative expressions in order to structure the code. ATL's `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. Finally, the collection iterative expressions are supported in Coq with recursion operators on lists.

Helpers and Attributes Helper/attribute call expressions as well as operation call expressions are OCL-based expressions. ATL helpers factorize code that can be called from different points of an ATL transformation. 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 pair (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. The main issue in this translation is that Coq imposes the condition that every recursion be well-founded, which has to be proven in each case.

In the example there is the following helper function that transforms a `PrimitiveDataType` into a string which represents the type of a column in the database.

```
helper context SimpleUML!PrimitiveDataType def :
  primitiveTypeToStringType : String =
    if (self.name = 'INTEGER')
    then 'NUMBER'
    else if (self.name = 'BOOLEAN')
         then 'BOOLEAN'
         else 'VARCHAR'
         endif
    endif;
```

The Coq function resulting from its translation is as follows. Notice that the comparison between strings (`=`) is performed by an auxiliary function `string_eq_bool` which returns a boolean value.

```
Definition primitiveTypeToStringType (primitiveType : string) : string :=
  match (string_eq_bool primitiveType "INTEGER") with
    | true => "NUMBER"
    | false => match (string_eq_bool primitiveType "BOOLEAN") with
                 | true => "BOOLEAN"
                 | false => "VARCHAR"
               end
  end.
```

Matched Rules 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 that 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.

Matched rules are generally translated into propositions of the form

$$\forall\ a{:}A.\ (a \in \mathit{InstA} \wedge \mathit{Cond}(a) \rightarrow \exists\ b{:}B.\ (b \in \mathit{InstB} \wedge \mathit{Rel}(a,b)))$$

expressing that for every object a of the type A (of the source model element in the matched rule) in the set *InstA* of all instances of type A which satisfies certain condition *Cond*, there exists an object b of the type B (of the target model element in the matched rule) in the set *InstB* of all instances of type B, for whom the relation *Rel* holds. The relation *Rel* is a conjunction of the bindings defined in the matched rule. If there are no other matched rules that define the existence of a target model element for whom the same relation *Rel* holds, then the proposition must state the unique existence of the target model element. There are also propositions describing the relation in the reverse direction (i.e. from the target to the source elements).

The formulæ thus obtained amount to (basic) specifications of the transformation rules at a propositional level. This stands in contrast to the approach in [17] where transformations were represented as functions about which the relevant properties had to be proven, leading to lengthy work that can now be avoided.

In the example we have the following matched rule that transforms an `Attribute` of a class diagram into a `Column` of the database. The name of the column will be the name of the attribute, and the type of the column will be the name of the `PrimitiveDataType` associated to the attribute (the helper already introduced is used in this case).

```
        rule AttributeToColumn{
            from a : SimpleUML!Attribute ()
            to c : SimpleRDBMS!Column (
                     name <- a.name,
                     type <- a.type.primitiveTypeToStringType
                   )
        }
```

We represent this matched rule as follows:

```
Definition AttributeToColumn (c : Class) (t : Table) : Prop :=
  (forall atr:Attribute, In atr (Class_attribute c) ->
    exists! col:Column, In col (Table_column t) /\
      RModelElement_name (Column_super col) = Attribute_name atr /\
      Column_type col = primitiveTypeToStringType
                     (PrimitiveDataType_name (Attribute_type atr)))
  /\
  (forall col:Column, In col (Table_column t) ->
    exists! atr:Attribute, In atr (Class_attribute c) /\
      RModelElement_name (Column_super col) = Attribute_name atr /\
      Column_type col = primitiveTypeToStringType
                       (PrimitiveDataType_name (Attribute_type atr))).
```

Notice that the relation is described in both directions, and also that there is only one source and target elements for which the proposition holds.

For the sake of completeness we present the other matched rule that transforms every persistent `Class` into a `Table` of the database. The name of the table must be the same as the class, and the columns of the table will be the transformation of the attributes of the class which is performed by the matched rule `AttributeToColumn`.

```
        rule ClassToTable{
            from c : SimpleUML!Class (c.kind = 'Persistent')
            to t : SimpleRDBMS!Table (
                       name <- c.name,
                       cols <- c.attribute
                   )
        }
```

This matched rule is represented in Coq as follows.

```
  Definition ClassToTable (ma : SimpleUML) (mb : SimpleRDBMS) : Prop :=
    (forall c:Class, In c (MClass_classAllInstances ma) /\
      Class_kind c = "Persistent" ->
        exists! t:Table, In t (MRelational_tableAllInstances mb) /\
          Class_name c = Table_name t /\
          AttributeToColumn c t)
    /\
    (forall t:Table, In t (MRelational_tableAllInstances mb) ->
      exists! c:Class, In c (MClass_classAllInstances ma) /\
        Class_kind c = "Persistent" /\
        Class_name c = Table_name t /\
        AttributeToColumn c t).
```

5 Verification of Properties

OCL invariants of both source and target metamodels are translated into propositions in the CIC. This, at a large measure, can be done automatically following the ideas presented in [22]. The desired properties of the transformation are specified in the CIC by the verifier using the full potential of the logic. These properties will in general establish relations between (any instances of) the source and target metamodel connected by the transformation.

A simple property of the example transformation is that the length of the `Columns` within a `Table` must be grater than zero. This can be written in OCL as follows.

```
context Table inv:
  self.column->length() > 0
```

In Coq, this property can be expressed as follows.

```
Definition TableAtLeastOneCol (model : SimpleRDBMS) : Prop :=
  forall t:Table, (In t (MRelational_tableAllInstances model)) ->
    length (Table_column t) > 0.
```

This property holds by the fact that every `Attribute` is transformed into a `Column` and that every `Class` has at least one `Attribute`. This information is given in the transformation rules and in the source invariants, respectively.

The invariants of the target metamodel and any other desired properties of the transformation (*Post*) are interactively verified in Coq by assuming that the invariants of the source metamodel (*Pre*), and the transformation rules (*TRules*) hold. In this way, the correctness proposition becomes:

$$\forall\ Ma{:}MMa.\ (Pre(Ma) \rightarrow \forall\ Mb{:}MMb.\ (TRules(Ma,Mb) \rightarrow Post(Ma,Mb)))$$

In the example, the Coq lemma to prove is as follows.

```
Definition Post (ma : SimpleUML) (mb : SimpleRDBMS) : Prop :=
  TableAtLeastOneCol mb.

Definition TRules (ma : SimpleUML) (mb : SimpleRDBMS) : Prop :=
  ClassToTable ma mb.

Lemma Cert_Class2Relational:
  forall ma:SimpleUML, Pre ma -> forall mb:SimpleRDBMS, TRules ma mb
    -> Post ma mb.
```

Notice that in this case the transformations rules are only the satisfaction of the `ClassToTable` rule, since every source model element involved in the transformation is reached from the class elements. The postcondition is the property `TableAtLeastOneCol`, defined above.

The Coq proof assistant helps building proofs using tactics (inference rules). We refer to the Coq documentation [13] for further details. The proof of this property can be found in [20].

There are other properties which can be proved for this transformation, for example the following OCL invariants.

```
context Table inv:
  Table.allInstances()->isUnique(name)

context Table inv:
  self.cols->isUnique(name)
```

The first one states that the name of a `Table` is unique. This holds because every `Class` is transformed into a `Table` and because the name of a `Class` is unique. The second property states that the name of a `Column` is unique within a `Table`, which holds because the name of an `Attribute` is unique within a `Class`.

The framework allows stating and proving more interesting properties which involve both the source and target metamodels. In these cases the properties cannot be expressed in OCL but can be expressed in Coq. For example, we proved that the number of tables is equal to the number of persistent classes, in Coq notation:

```
Definition ClassTableEqLen (ma : SimpleUML)
                           (mb : SimpleRDBMS) : Prop :=
  length (filter isPersistent (MClass_classAllInstances ma)) =
    length (MRelational_tableAllInstances mb).
```

The proof of this property is done by induction on both `MClass_classAllInstances` and `MRelational_tableAllInstances` which are the lists of all the instances of type `Class` and `Table`, respectively. This proof can also be found in [20]. In a similar way, we can prove that the number of `Column`s within any `Table` is equal to the number of `Attribute`s of the corresponding `Class`.

6 Conclusions and Further Work

We have described a type-theoretic framework that allows the full formal verification of model transformations, at a metamodel level, and considering any kind of transformations and properties. We have proposed a separation of duties between developers and verifiers, based on a semi-automatic translation process switching from the ATL to the CIC (Calculus of Inductive Constructions) technical space as implemented on Coq. Within this framework, source and target metamodels (*MMa* and *MMb*) are represented as inductive types, and each transformation rule of the model transformation *Mt* is represented as a logical formula of ∀∃ form stating that for every model element satisfying a certain (pre-)condition, there exists a target model element which stands in the relation specified by the transformation rule with the source model element (*TRule*). The correctness of the model transformation is stated by a formula.

$$\forall\ Ma{:}MMa.\ (Pre(Ma) \rightarrow \forall\ Mb{:}MMb.\ (TRules(Ma,Mb) \rightarrow Post(Ma,Mb)))$$

where *Pre* is a translation of the source invariants, *TRules* is the conjunction of the transformation rules, and *Post* is a translation of the target invariants plus any other desired property to be proved. A proof of this formula ensures that the transformation rules satisfy the target invariants as well as the desired properties.

The translation into Coq of the KM3 metamodels can be performed fully automatically. On the other hand, at the moment the verifiers have to deal with: references in the metamodels that must be "cut" to avoid circularity in an optimal way, the translation of the OCL invariants –which can at a large measure be done automatically– and the translation of the ATL transformation rules and helpers. Then he can proceed to perform the formal verification.

As a proof of concepts, we have applied our approach to a simplified version of the well-known Class to Relational model transformation broadly studied in the literature [19]. The resulting Coq code can be found in [20].

With this approach we lose full automation to gain in return strength of the achieved results. We think the approach could be particularly helpful in proving the existence of zero-fault model transformations within the development of critical systems.

So far the non-automatic parts of the process of translation involved in our proposal can in general be carried out directly enough to indeed provide increased confidence in the outcome.

We are currently working on setting up a semantics of transformation languages in type theory which will lead to a greater automatical capability at the level of the framework, particularly concerning the outlined method for translating transformations.

Our medium-term goals are the full development of the framework and its integration with ATL and Coq. In the long-term we will work on simplifying the proof process. In this direction we aim at generating auxiliary libraries with proofs of basic properties and also work on proof patterns detection in order to improve the facility of use of the proof assistant.

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 [20].

References

1. Kent, S.: Model-Driven Engineering. LNCS **2335**, Springer (2002) 286–298
2. OMG: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group, Specification Version 2.0, 2003.
3. ATLAS Group: Kernel MetaMetaModel. LINA & INRIA. Manual v0.3 (2005)
4. OMG: UML 2.0 Object Constraint Language. Object Management Group, Specification Version 2.0, 2006.
5. Mens, P., V. Gorp, P.: A Taxonomy of Model Transformation. ENTCS **152**, Springer (2006) 125–142.
6. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches IBM Systems Journal **45**-3 (2006) 621–645.
7. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation. Object Management Group, Specification Version 1.0, 2008.
8. ATLAS Group: Atlas Transformation Language. LINA & INRIA. User Guide. (2009)
9. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. 4^{th} Workshop on Model-Driven Engineering, Verification and Validation. (2007) 47–56
10. Pons, C., García, D.: A Lightweight Approach for the Semantic Validation of Model Refinements. ENTCS **220**, Springer (2008) 43–61
11. Giese, H., et al.: Towards Verified Model Transformations. 3^{rd} International Workshop on Model Development, Validation and Verification. (2006) 78–93
12. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer-Verlag (2004)
13. The Coq Development Team: The Coq Proof Assistant: Reference Manual. (2009).
14. Leroy, X.: Formal Verification of a Realistic Compiler. Commun. ACM **52** (2009) 107–115
15. Poernomo, I.: A Type Theoretic Framework for Formal Metamodelling LNCS **3938**, Springer (2004) 262–298
16. Poernomo, I.: Proofs-as-Model Transformations. 1^{st} International Conference on Theory and Practice of Model Transformations. (2008) 214–228
17. Calegari, D., Luna, C., Szasz, N., Tasistro, A.: Experiment with a Type-Theoretic Approach to the Verification of Model Transformations. 2^{nd} Chilean Workshop on Formal Methods. (2009)
18. Coquand, T., Paulin, C.: Inductively Defined Types Proc. Intl. Conf. on Computer Logic, Springer (1990) 50–66.
19. Bézivin, J., Rumpe,B., Schürr, A., Tratt, L.: Model Transformations in Practice Workshop" in *MoDELS Satellite Events* LNCS **3844**, Springer (2005) 120–127
20. Verification of UML-Based Behavioral Model Transformations Project, `http://www.fing.edu.uy/inco/grupos/coal/field.php/Proyectos/ANII09`
21. Steel, J., Jézéquel, J.M.: On Model Typing. SoSyM **6**, Springer (2007) 401–413
22. Beckert, B., Keller, U., Schmitt, P.: Translating the Object Constraint Language into First-Order Predicate Logic. Workshop at Federated Logic Conferences. (2002)