Model-Driven Engineering Based on Attribute Grammars

Daniel Calegari(✉) and Marcos Viera

Universidad de la República, Montevideo, Uruguay
{dcalegar,mviera}@fing.edu.uy

Abstract. The Model-Driven Engineering (MDE) paradigm proposes the construction of software based on an abstraction from its complexity by defining models, and on a (semi)automatic construction process driven by model transformations. In this paper we propose the use of attribute grammars for the specification of QVT-like (Query/View/Transformation) relational model transformations. We also present how the syntax and semantics of models can be represented, and we discuss the practical implications of this approach through the development of a case study.

Keywords: Model-Driven Engineering · Attribute grammars · QVT · Haskell

1 Introduction

The use of a model-centric approach for the specification of a system, and of automated mechanisms for its construction, improves efficiency on the whole process. The Model-Driven Engineering (MDE, [1]) paradigm is based on these practices. It envisions a software development life-cycle driven by models representing different views of the system to be constructed and model transformations providing a (semi)automatic construction process. Models are defined from metamodels, i.e. a model which introduces the syntax and semantics of certain domain-specific kind of models. The relation between a model and its metamodel is called conformance. A model transformation is basically the automatic generation of a target model from a source model, according to a set of rules that describe how certain elements in the source model can be transformed into certain others in the target model. The Object Management Group (OMG) has conducted a standardization process of languages and defined the MetaObject Facility (MOF, [2]) for metamodeling, and the Query/View/Transformation Relations (QVT-Relations, [3]) for declarative model transformations.

Modelware is the technical space [4] of MDE, i.e. a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. In contrast, Grammarware is the technical space of grammars and grammar-aware theories and software. Bridging of technical spaces is specially useful for adopting the benefits of the other technical space [5], e.g. the translation of

A. Pardo and S.D. Swierstra (Eds.): SBLP 2015, LNCS 9325, pp. 112–127, 2015.
DOI: 10.1007/978-3-319-24012-1_9

MDE elements (models, metamodels and transformations) into Grammarware elements should allow their integration into existing tools such as diff/merge, as well as the definition of declarative semantics associated to grammar productions. There are reasonable similarities between grammars and metamodels [5]. Since metamodels are language definitions, there is a relation between them and the concept of a grammar, as well as models conforming to a metamodel are like strings recognized by a grammar. Moreover, syntactical and semantical properties that must hold in a given model to be considered conformant to a metamodel, can be considered part of the semantics of a grammar. We also claim that model transformations can be considered part of this semantics.

In this paper we address the bridging of Modelware and Grammarware by representing MDE elements using Attribute Grammars (AGs, [6]). An AG is composed by an underlying context-free grammar, describing the structure of an Abstract Syntax Tree (AST), together with a set of attributes defined for each non-terminal which allows to compute and pass information downwards and upwards within the AST. In particular, we describe how metamodels can be represented as grammars, and their semantics, as well as QVT-like model transformations, as attributes of the grammar. AGs constitutes an executable method of specification, since it describes only a computation in terms of an AG and then automatically produces a program [7]. In this way we can derive a program for checking conformance and executing a model transformation. We also discuss the practical implications of this approach through the development of a case study[1] using the Utrecht University Attribute Grammar Compiler (`uuagc` [2], [8]); a preprocessor that generates Haskell code out of AG specifications.

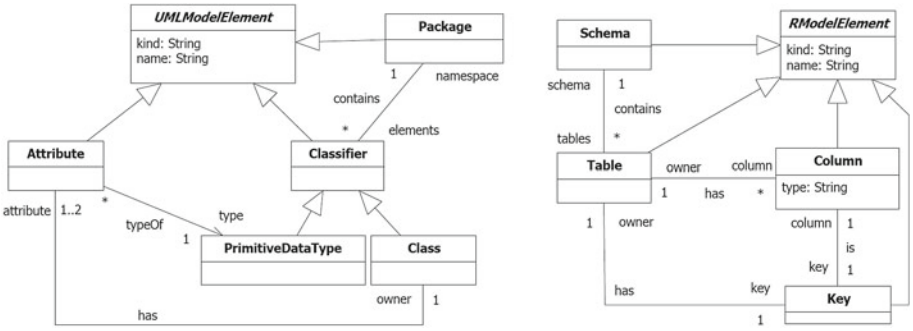
The remainder of the paper is structured as follows. In Sect. 2 we introduce the main concepts of MDE based on a running example. Then, in Sect. 3 we present how models and metamodels can be represented using AGs, such that is possible to verify conformance of a model with respect to its metamodel. In Sect. 4 we present the specification of QVT-like model transformation using AGs. Finally, in Sect. 5 we present related work and in Sect. 6 we present some conclusions and an outline of further work.

2 Model-Driven Engineering

Every model *conforms* to a metamodel, which typically defines syntax and (static) semantics of modeling languages like UML. The MetaObject Facility (MOF, [2]) is a standard language for metamodeling. In few words, a metamodel defines classes which can belong to a hierarchical structure. Any class has properties which can be attributes (named elements with an associated type which can be a primitive type or another class) and associations (relations between classes in which each class plays a role within the relation). Every property has a multiplicity which constrains the number of elements that can be related through the property.

[1] Complete source code of our running example is available at https://www.fing.edu.uy/inco/grupos/coal/field.php/Research/ANII14.

[2] https://hackage.haskell.org/package/uuagc.



(a) UML metamodel (b) Relational metamodel

Fig. 1. Exampling metamodels

If there are conditions that cannot be captured by the structural rules of this language, the Object Constraint Language (OCL, [9]) is used to specify them. These considerations allow defining the conformance relation in terms of *structural* and *semantical* conformance. Structural conformance with respect to a MOF metamodel means that in a given model: every object and link is well-typed and the model also respects the multiplicity constraints. Semantical conformance means that a given model respects the invariants specified with the supplementary constraint language.

As an example, the metamodel in Fig. 1a defines UML class diagrams, where classifiers (classes and primitive types as string, boolean, integer, etc.) are contained in packages (association `contains`). Classes can contain attributes (association `has`) and may be declared as persistent (`kind = 'Persistent'`), whilst attributes have a type that is a primitive type (association `typeOf`). Notice that a class must contain only one or two attributes (multiplicity `1..2`), and also that the Classifier class is not abstract. We decided to handle these aspects differently from UML class diagrams in order to have a more complete example. The Relational diagrams metamodel in Fig. 1b defines schemas which contain a number of tables and each table has a number of columns. Each column has a name and a kind, and can be the primary key of the corresponding table.

A model transformation takes as input a model conforming to certain metamodel and produces as output another model conforming to another metamodel (possibly the same). Query/View/Transformation Relations (QVT-Relations, [3]) is a relational language which defines transformation rules as mathematical relations between source and target elements. A transformation is a set of interconnected relations: top-level relations that must hold in any transformation execution, and non-top-level relations that are required to hold only when they are referred from another relation. Every relation defines a set of variables, and source and target patterns which are used to find matching sub-graphs of elements in a model. Relations can also contain a `when` clause which specifies the conditions under which the relationship needs to hold, and a `where` clause which specifies the condition that must be satisfied by all model elements participating

```
transformation uml2rdbms  ( uml : UML , rdbms : RDBMS ) {
  key RDBMS::Schema {name};
  key RDBMS::Table  {name, schema};
  key RDBMS::Column {name, owner};
  key RDBMS::Key    {name, column};
  top relation PackageToSchema {
    pn : String;
    checkonly domain uml p:UML::Package { name = pn };
    enforce domain rdbms s:RDBMS::Schema { name = pn };
  }
  top relation ClassToTable {
    cn, prefix : String;
    checkonly domain uml c:UML::Class {
      namespace = p:UML::Package {}, kind = 'Persistent', name = cn
    };
    enforce domain rdbms t:RDBMS::Table {
      schema = s:RDBMS::Schema {}, name = cn,
      column = cl:RDBMS::Column { name = 'TID', typeT = 'NUMBER' },
      keyK = k:RDBMS::Key { name = 'PK', column = cl }
    };
    when { PackageToSchema(p, s); }
    where { AttributeToColumn(c, t) }
  }
  relation AttributeToColumn { ... }
}
```

Fig. 2. Class to relational transformation (excerpt)

in the relation. The `when` and `where` clauses, as well as the patterns may contain arbitrary boolean OCL expressions and can invoke other relations.

Consider the example of Fig. 2 which is a simplified version of the well-known Class to Relational transformation [3]. The transformation basically describes how persistent classes within a package are transformed into tables within a schema. The relation `PackageToSchema` states that any UML package is mapped into a relational schema. Moreover, the relation `ClassToTable` states that classes marked as persistent are mapped into tables with the same name, a primary key and an identifying column, such that the package to which the class belongs is in the relation with the schema to which the table belongs. The relation `AttributeToColumn` is called from the where clause of `ClassToTable` and maps primitive attributes of the persistent class to columns of the corresponding table. There are also keys, e.g. stating that the transformation must ensure that there cannot be two `Table`s with the same `name` within the same `Schema`.

3 AG-based Structural and Semantical Conformance

As discussed in [5], describing a mapping from metamodels to grammars is in many ways more demanding than the opposite, since metamodels inherently contain more information than grammars, as for example the notion of inheritance between metamodel elements and properties. Moreover, any metamodel can be considered a graph of elements whereas grammars forms a tree. In what follows

we introduce how metamodels can be mapped into AGs in such a way that a model conforming with a metamodel is represented as a string recognized by the corresponding AG. We also describe how AGs allow us to address structural and semantical conformance checking as in the MDE world. Throughout this section we also introduce the main concepts of AGs related to our proposal.

Since we are focusing on model transformations, we do not consider some MOF constructs. In particular, we do not consider aggregation, uniqueness and ordering properties within a property end, operations on classes, and packages. Aggregation and operations are not used within transformations, whereas packages are just used for organizing metamodel elements (they can be considered syntactic sugar). Although uniqueness and ordering properties are neither commonly used, they can be considered within semantical conformance checking.

MOF elements can be translated to AGs as follows.

Classes and Hierarchies. Each class is translated to a non-terminal with a production rule resulting from the translation of their properties. If the class does not have a superclass, then its production rule includes a terminal *oid* of type *Int* representing an unique identifier of any instance of such class. Moreover, if the class has subclasses, the production rule defines a non-terminal *child* of type *ClassCh*, with *Class* the name of the class. This non-terminal defines one production rule for each subclass, such that each one defines only one non-terminal of the type of the corresponding subclass. If the class is not abstract, then the child is wrapped with a *Maybe*.

In Fig. 3 we show the grammar resulting from the translation of the UML class diagrams metamodel of Fig. 1a. In `uuagc` grammars are defined in **data** declarations, which are very similar to Haskell **data** declarations with named fields. Thus, for example, the declarations of *Classifier*, *MaybeClassifierCh* and *ClassifierCh* result from the translation of the class `Classifer`.

Datatypes and Enumerations. Our AGs are Haskell-based specifications. Thus, primitive types as string, boolean and integer are mapped to their corresponding Haskell types[3]. In the case of user defined datatypes, we translate them in the same way we do with classes. An enumeration is translated to a non-terminal with a choice of terminals corresponding to their values.

Properties and Multiplicities. Properties are defined by a name, an associated type which can be a primitive type or another class and a multiplicity constraining the number of elements that can be related through the property. Within the context of the production rule corresponding to the class who owns a property, we translate a property typed with a primitive type as a terminal of the translated type. Moreover, if the property is typed with a non-primitive type, we translate the property as a terminal of type *Int*, representing the identifier of the element that must be related through the property. If the multiplicity of the property accepts many elements, the type of the terminal is a list of the corresponding type. Finally, we use maybe if the multiplicity is 0..1. More narrow

[3] In `uuagc` everything that is in between brackets is considered as Haskell code.

```
data UML | UML model :: ListUMLModelElement
type ListUMLModelElement = [UMLModelElement]
data UMLModelElement    | UMLModelElement oid    :: {Int}
                                           kind   :: {String}
                                           name   :: {String}
                                           child  :: UMLModelElementCh
data UMLModelElementCh | UMLMECAtt        att    :: Attribute
                       | UMLMECPck        pck    :: Package
                       | UMLMECCla        cla    :: Classifier
data Package  | Package  elements :: {[Int]}      -- Classifiers
data Attribute | Attribute typ      :: {Int}      -- PrimitiveDatatype
                           owner    :: {Int}      -- Class
data Classifier    | Classifier namespace :: {Int}  -- Package
                              child       :: MaybeClassifierCh
type MaybeClassifierCh = maybe ClassifierCh
data ClassifierCh | ClassifierChPri pri   :: PrimDataType
                  | ClassifierChCla cla   :: Class
data PrimDataType | PrimDataType
data Class          | Class atts :: {[Int]}        -- Attribute
```

Fig. 3. Grammar for UML class diagrams metamodel

multiplicities are defined as attributes since they are considered as part of the structural conformance checking.

Metamodel. At the top of the grammar we need a root element with a production rule generating every other metamodel element on top of a hierarchy (isolated classes and datatypes are considered hierarchies of one element). Then, metamodels are represented as a list of such root elements. In our example, the root model element is *UMLModelElement*.

The `uuagc` preprocessor generates Haskell data types out of the grammar declarations. The following Haskell value, with type *UML*, is an example of a model that conforms to the metamodel represented by the grammar of Fig. 3.

```
umlModel = UML [ UMLModelElement 1 "" "Package" (UMLMECPck (Package [2,3,4]))
               UMLModelElement 2 "Persistent" "ID"
                 (UMLMECCla (Classifier 1 (Just (ClassifierChCla (Class [4])))))
               UMLModelElement 3 ""           "String"
                 (UMLMECCla (Classifier 1 (Just (ClassifierChPri PrimDataType))))
               UMLModelElement 4 "" "value"   (UMLMECAtt (Attribute 3 2))]
```

Referential and Inherited Properties. Properties are defined in their owning classes, and within a hierarchy they must be inherited by subclasses.

```
set EveryUMLModelElement = UMLModelElementCh Package Attribute Classifier
                           MaybeClassifierCh ClassifierCh Class PrimDataType
attr EveryUMLModelElement inh oid  :: {Int}
                          inh kind :: {Sting}
                          inh name :: {Sting}
sem UMLModelElement | UMLModelElement child.oid  = @oid
                                      child.kind = @kind
                                      child.name = @name
```

Fig. 4. Attributes defining *UMLModelElement* properties

In AGs, *inherited* attributes are used to pass information downward a tree. We define inherited attributes such that, for a given property, these attributes are copied to every subclass of the property owner. In our example we have that *UMLModelElement* defines three properties (*oid*, *kind* and *name*), thus we define inherited attributes, whose semantics is given by the original terminals of its production rule, and which are copied to their *child* elements. This is depicted in Fig. 4, where three inherited attributes (**inh**) are defined for every descendant of *UMLModelElement*. Semantic rules, starting with the keywork **sem**, define how the value of an attribute is computed. In the case of inherited attributes, is the parent who computes the values for its children. In the example, we define that the values of the attributes *oid*, *kind* and *name* of the child *child* of *UMLModelElement* are the values of the fields *oid*, *kind* and *name*, respectively. Semantic rules have to be defined for every production of all the non-terminal which has the attribute. However, if a rule for an inherited attribute is missing, the `uuagc` system derives a *copy-rule*, which just copies the value of the parent to its children. Thus, the declarations of Fig. 4 express that, for example, the value of the field *oid* is copied unchanged in the attribute *oid* to all the descendants of *UMLModelElement*.

Some properties are references to other non-primitive elements. In this case, we define a pair of *lookup* attributes for accessing these elements:

```
attr ListUMLModelElement UMLModelElement
                syn elemLookup_s :: {Int → Maybe UMLModelElement}
attr EveryInter inh elemLookup_i :: {Int → Maybe UMLModelElement}
```

$elemLookup_s$ is a *synthesized* attribute; i.e. an attribute that collects information in a bottom-up way. In this case, we construct a function which allows to lookup to an element into the list of model elements using its identifier. Then this function is distributed through the model (*EveryInter* means all the non-terminals but *UML*) using the inherited attribute $elemLookup_i$.

For each class with a production rule defining a non-terminal as a reference to other element, we define a higher-order attribute [10], i.e. a local attribute that acts as if it is an additional child of the production (also with attributes).

```
sem Attribute | Attribute inst.typ_    :: UMLModelElement
                        inst.typ_    = fromJust ( @lhs.elemLookup_i @typ)
                        inst.owner_  :: UMLModelElement
                        inst.owner_  = fromJust ( @lhs.elemLookup_i @owner)
```

Fig. 5. References (excerpt)

```
{
data Type = TPackage | TAttribute | TClassifier | TPrimitiveDataType | TClass
}
attr EveryInter syn types use {++} {[]} :: {[Type]}
sem Package          | Package          lhs.types = [TPackage]
sem Attribute        | Attribute        lhs.types = [TAttribute]
sem Classifier       | Classifier       lhs.types = TClassifier : @child.types
sem PrimDataType     | PrimDataType     lhs.types = [TPrimitiveDataType]
sem Class            | Class            lhs.types = [TClass]
```

Fig. 6. Collecting the types of an element

In the example depicted in Fig. 5 we represent the referential properties for the *Attribute* class. The keyword **inst** specifies that we are defining a higher-order attribute, while with **lhs** we refer to attributes coming from the left hand side (i.e. the parent). Except for the symbols starting with @, that refer to attribute values, the expressions on the right hand side of the =–signs of the semantic rules are plain Haskell code. Since *Attribute* defines *typ* and *owner* as referential properties (to a *PrimitiveDatatype* and a *Class*, respectively), we define higher-order attributes *typ_* and *owner_*, such that their values are defined by looking up the corresponding elements in the list of top elements; i.e. we dinamically copy the corresponding branches of the three as new children. We ensure that these elements exist by addressing structural conformance as explained next. Notice that we are generating an infinite structure, due to the cyclic references of the model. We make use of Haskell's lazy evaluation to avoid infinite computations, and unfold the structure only as much as needed.

Structural and Semantical Conformance. AGs also allows us to address structural and semantical conformance. Structural conformance requires that the model is well-typed and that also respects the multiplicity constraints. Additional checks are mandatory in the case of referential properties and narrow multiplicity constraints. In the case of semantical conformance, we need to specify supplementary constraints. Besides we do not have a direct translation from OCL to AGs,

```
attr Every syn errs use {++} {[]} :: {[String]}
sem Attribute
  | Attribute lhs.errs     = case @loc.owner_ of
                               Nothing → ["Type oid" ++ show @owner ++ "not found."
                               Just _  → if elem TClass @owner_.types
                                         then []
                                         else ["Type Error for oid" ++ show @owner
sem Class
  | Class     lhs.errs     = @loc.errMul ++ @loc.errDup
      -- multiplicity constraint
      loc.errMul = let  atts = length $ atts
                   in   if (1 > atts) ∨ (atts > 2)
                        then [  @lhs.name ++ ": Multiplicity Error: "
                                ++ show atts ++ " attributes."]
                        else  []
      -- semantical conformance checking
      loc.errDup = let  dup = [l | l ← group (sort @atts_.names), length l > 1]
                   in   if length dup > 0
                        then [  show @lhs.oid
                                ++ ": Duplicated Names: " ++ show dup]
                        else  []
```

Fig. 7. Structural and semantical conformance checking (excerpt)

devised as future work, the potential of AGs allows this kind of checking. Note that higher order AGs are Turing complete [10]. To address typing requirements we define a synthesized attribute *types* (Fig. 6) for collecting the types of an element (its own type and their inherited types within a hierarchy). For synthesized attributes we can define *use* rules for the cases where the semantic rules are not explicitly declared. For example, for *types* the information is collected by appending (++) the lists coming from the children. We also define a synthesized attribute *errs* for collecting errors when checking conformance. This attribute is defined for each non-terminal with respect to their own conformance needs. In Fig. 7 we show some examples of conformance checks. We can see the definition of the inherited attributes and some structural and semantical conformance chekings. In particular, within *Attribute* we check that its referential property *owner* exists and it is well-typed (must be of type *TClass*). Moreover, in the context of a *Class* we define that a class must have only 1 or 2 attributes (multiplicity constraint) and also that the name of an attribute must be unique within a class (semantical conformance). For this last check we use the higher-order attribute *atts_*, giving the list of *Attribute* of a class, we collect their names and check if there are duplicates in the resulting list.

4 AG-based Model Transformations

In this section we describe how model transformations specified using QVT-Relations can be mapped to AGs. As a running example we use the (fragment of) `uml2rdbms` transformation, defined in Fig. 2 of Sect. 2.

The AG specification of a transformation generates a Haskell function that takes as input a model that conforms to the source metamodel and returns a function from an initial model to a final model conforming to the target metamodel. The transformation of the example is expressed as follows[4]:

$$uml2rdbms :: UML \rightarrow RDBMS \rightarrow RDBMS$$

Transformations are performed with check-enforce semantics; that is, first we check if the initial target model complies with the relations specified by the transformation, and then, only in the cases of relations that does not hold, the model is incrementally updated. When executing the *uml2rdbms* transformation to the *umlModel* defined in Sect. 3 with an empty initial target model we get:

```
uml2rdbms umlModel (RDBMS [])
> RDBMS [RModelElement 5 "" "Package" (RMECSch (Schema [1]))
         RModelElement 1 "" "ID"      (RMECTab (Table 5 [4,2] 3))
         RModelElement 2 "" "TID"     (RMECCol (Column "NUMBER" 1))
         RModelElement 3 "" "PK"      (RMECKey (Key [2] 1))
         RModelElement 4 "" "value"   (RMECCol (Column "VARCHAR" 1))]
```

But, if for example, we use this resulting model as the initial one, then the same model is obtained. In case of models not completely complying with the transformation specification, only the needed elements are inserted, e.g. if only the last column (4) is missing, then the result is the initial model with this column added (and the table updated to refer to this column).

Since the semantic function generated by the AG system should be a function that takes as input a *RDBMS* and results in a *RDBMS*, we define at the root of the grammar an inherited attribute *input* and a synthesized attribute *output*, both with type *RDBMS*.

$$\begin{aligned} \textbf{attr}\ UML\ \textbf{inh}\ input &:: \{RDBMS\} \\ \textbf{syn}\ output &:: \{RDBMS\} \end{aligned}$$

We define a rule as a function that, given a list of relational model elements returns an updated list of relational model elements. Top rules produced by the elements of the source UML grammar are collected (i.e. composed) bottom-up by a synthesized attribute *top*.

$$\textbf{attr}\ EveryInter\ \textbf{syn}\ top\ \textbf{use}\ \{(.)\}\ \{id\} :: \{[T.RModelElement] \rightarrow [T.RModelElement]\}$$

Thus, a transformation is defined as the application of the top rules to the input list of elements.

[4] *RDBMS* is the data type that represents the grammar corresponding to the metamodel of Fig. 1b.

```
{
type Relation = [T.RModelElement] → ([Int], [T.RModelElement])
}
attr Package UMLModelElementCh UMLModelElement syn p2S :: {Relation}
sem Package
| Package (lhs.counter, loc.s) = nextUnique @lhs.counter
          loc.p2S = case @lhs.name of
                      pn → addSchema (mkSchema @loc.s "" pn [])
          lhs.top = snd . @loc.p2S
{
mkSchema s k pn tl = (RModelElement s k pn (RMECSch (Schema tl)))
addSchema ns []                  = ([oid ns], [ns])
addSchema ns (r : rs) | ns ≡ r   = ([oid r],  r : rs)
                      | otherwise = let (s, rs') = addSchema ns rs in (s, r : rs')
}
```

Fig. 8. Implementation of the relation `PackageToSchema` (excerpt)

```
sem UML | UML lhs.output = let (RDBMS elems) = @lhs.input
                           in  RDBMS ( @model.top elems)
```

The rules are created from the relations specified in the transformation. For each relation, we define an attribute at the non-terminal representing the main element of the source domain pattern of the rule. For example, in Fig. 8, for the relation `PackageToSchema` we define an attribute *p2S* at *Package*.

A *Relation* takes an initial target model and returns a pair composed by the list of possibly introduced elements and the resulting target model. The patterns in QVT-Relations are traduced to pattern matching. We use a chained attribute *counter* to generate unique identifiers for the new elements. A chained attribute is a pair of attributes (synthesized and inherited) with the same name that are used to walk through the tree keeping a sort of state; in this case a number.

Thus, for a given *Package* with name *pn*, we create an empty *Schema* with name *pn* and identifier a new unique number. The function *addSchema* inserts this new schema only if an equal schema does not already belong to the list. Equality in model elements (≡) is defined in terms of the keys declared in the transformation. Thus, if two schemas have the same name we consider they as equals, even if they have different identifiers. If the new schema is not inserted, the returned identifier is the one of the existing schema (not a new one).

Since `PackageToSchema` is a top relation, we use *p2S* to define the *top* attribute by forgetting the identifiers of the inserted elements.

In Fig. 9 we show how the relation `PackageToSchema` is mapped to an attribute *c2T*. This rule only applies if the given class is of kind "`Persistent`"; otherwise the initial target model is returned unchanged. First we apply the

relations included in the `when` clause, in this case the *p2S* relation the *Class* inherited from the *namespace_* of the *Classifier*. Then, the model is sequentially (possibly) updated with a new table, column and key. The addition of elements that must be referred by other elements in the model, implies the need to update such other elements, adding their references. For example, to (possibly) add a *Table* we first use *addTable′* to possibly add the new table, in a similar way as we described in the case of *Schema*, and then if the table was added we use *updSchema* to update the schema *s*. After (possibly) adding the new elements to the model, we apply the `where` clause relations to the resulting model. In the example of Fig. 9 we apply the non-top `AttributeToColumn` relation (*a2C*), given a table *t* and model *r4*.

Notice that, for clarity reasons, we are assuming that both the `when` and `where` clauses hold. In case any of them is not fulfilled (returning an empty list of added/checked elements), the pair $([\,], r)$ has to be returned. Moreover, we did not focus on how OCL expressions (in which QVT is strongly based) can be represented. This is part of future work.

```
sem Class
 | Class (loc.c1, loc.t) = nextUnique @lhs.counter
         (loc.c2, loc.c) = nextUnique @loc.c1
         (loc.c3, loc.k) = nextUnique @loc.c2
         loc.c2T = case ( @lhs.namespace, @lhs.kind, @lhs.name) of
                     (p, "Persistent", cn) → λr →
                       let ([s], r1) = @lhs.p2S_i r
                           ([t], r2) = addTable    (mkTable @loc.t "" cn s [] 0) r1
                           ([c], r3) = addColumn (mkColumn @loc.c "" "TID" "NUMBER" t) r2
                           ([k], r4) = addKey      (mkKey @loc.k "" "PK" [c] t) r3
                           (cs, r5)  = @loc.a2C t r4
                       in  (t : c : k : cs, r5)
                     _                       → λr → ([], r)
         lhs.top = snd. @loc.c2T
{
addTable nt rs = let (s, t)       = (schema nt, oid nt)
                     ([t′], rs′)  = addTable′ nt rs
                 in  ([t′], (if t′ ≡ t then updSchema s t else id) rs′)
addTable′ nt                      = ([oid t], [nt])
addTable′ nt (r : rs) | nt ≡ r    = ([oid r], r : rs)
                      | otherwise = let (t, rs′) = addTable′ nt rs in (t, r : rs′)
updSchema s t []                    = []
updSchema s t (r : rs) | s ≡ (oid r) = addTable2Schema t r : rs
                       | otherwise   = r : updSchema s t rs
}
```

Fig. 9. Implementation of the relation `ClassToTable` (excerpt)

5 Related Work

The representation of MDE elements in terms of a shallow embedding of the languages by providing a syntactic translation into Grammarware concepts has been proposed before [5,11–13]. The translations have some minimal differences between them with respect to the representation of hierarchical elements and properties within. In few words, some proposals model hierarchies as a flattening of elements, move properties from the topmost (or bottommost) element of a hierarchy to every bottommost (or topmost, respectively) element in order to have access to those inherited properties, or discard some intermediate elements within a hierarchy since they do not have any property of their own. Besides these translations generate more optimal grammars, they lose traceability with respect to the original metamodel. Thus it could be neither appropriate for the definition of a model transformation (as attributes related to the main element of the source domain) nor for the definition of the reversal translation from the AG to their corresponding metamodel. Moreover, properties are represented as an occurrence of a non terminal of the typing class, or by-name, depending on multiplicities and aggregations. We use a homogeneous representation by using identifiers referencing elements on top of a hierarchy. Higher-order attributes allow accessing every required property.

With respect to conformance, in [14] the authors propose a formal approach for the definition of metamodels (not based on MOF) using a meta-notation extending BNF and the specification of constraints on models in a formal logic language. Moreover, in [11] the authors define general rules to derive a context-free EBNF grammar from a MOF-compliant metamodel. They also use these mapping rules to generate a Java compiler in which parser actions are added to check semantical conformance. In our proposal, structural and semantical conformance is addressed using the same language of AGs. In [15] the authors use reference attribute grammars (RAGs, [16]) for the specification of metamodel semantics. They basically represent metamodels as in the other referred proposals, but they use reference attributes (the main difference between AGs and RAGs) in order to model non-containment properties. They also define several attributes for representing derived properties and operations (not supported by our proposal). RAGs allow to define a graph-like structure, more similar to the concepts behind a metamodel. However, we can get a similar representation by using a combination of IDs and higher-order attributes.

Up to our knowledge, with respect to model transformations there is only one work [7] defining how to represent a model transformation using AGs. The transformation is represented as attributes and the output is a text that corresponds to the target model in accordance with its grammar. However, this work only present general ideas, not using any transformation language as a reference (e.g. QVT as we do) and exemplifying the proposal using an extremely reduced version of a model transformation. Moreover, they do not ensure that the generated string indeed conforms to the target grammar, as we do by generating an instance conforming to the target grammar.

6 Conclusions and Future Work

We have explored the use of AGs for the representation of MDE elements (models, metamodels and model transformations). Any metamodel is represented with an AG, and models conforming to it are represented as strings recognized by the corresponding grammar. We exhaustively use attributes for handling references between metamodel elements, for structural and semantic conformance checking and for representing QVT-like model transformations. We also developed a case study using UUAGC which demonstrates the feasibility of this approach.

The representation of metamodels and models could be easily automated (as a model-to-text transformation) since there is a straightforward representation of the basic elements (as Haskell types) and the generated attributes directly depend on the structure of the metamodel. In this way, structural conformance can be automatically verified. Moreover, it could be possible to include the automated translation into a modeling environment, bridging the gap between model-driven and attribute grammar practitioners.

By focusing on QVT, we are trying to structure the way we define model transformations using AGs. The case study showed that there is some direct relation between QVT constructs and their AG representation. We still need to study if it is possible to automatically generate an AG from a QVT specification. Our AG-based approach can be classified as a direct manipulation approach, which offers little or no support or guidance in implementing transformations. In this sense, we can explore the definition of an embedded domain specification language (DSL) for model transformations. This DSL could be used for expressing model transformations within the Grammarware technical space, without depending on the Modelware technical space.

Within the case study we addressed the inclusion of OCL expressions. However, further exploration is required in order to exhaustively represent OCL within AGs. This will provide a uniform way of expressing constraints on transformation rules, and on metamodels for semantical conformance checkings. Moreover, it will provide a way of addressing some OCL-based approaches for the verification of a model transformation [17].

Besides an AG describes a computation and then a program is automatically generated, we need to specify some aspects which are abstractly handled by the transformation engine when a declarative approach is used, e.g. when elements must be created or updated. Far from being a problem, this could be useful for the representation of other transformation aspects, e.g. rule scheduling (order or rule invocation), multi-directional transformations, tracing, multiple source and target domains in a transformation, etc. Furthermore, since attribute computations are expressed as Haskell expressions, the Haskell type system (and novel type-level programming techniques) can be exploited to provide partial proofs of properties of the models and model transformations. For example, generated grammars can be represented using the structure defined in [18] to represent correct-by-construction mutually dependent structures and manipulate them in a type-safe way. Further work is required in this sense.

Finally, we need to continue developing case studies in order to strengthen our results. Particularly, complex examples could allow the comparison between our proposal and other transformation engines with respect to execution times.

Acknowledgements. This work has been partially funded by the Agencia Nacional de Investigación e Innovación (ANII, Uruguay).

References

1. Kent, S.: Model driven engineering. In: Proceedings of Integrated Formal Methods, pp. 286–298 (2002)
2. OMG: Meta Object Facility (MOF) 2.0 Core Specification. Specification Version 2.0, Object Management Group (2003)
3. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation. Final Adopted Specification Version 1.1, Object Management Group (2009)
4. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: an initial appraisal. In: CoopIS, DOA 2002 Federated Conferences, Industrial Track (2002)
5. Paige, R.F., Kolovos, D.S., Polack, F.A.C.: A tutorial on metamodelling for grammar researchers. Sci. Comput. Program. **96**, 396–416 (2014)
6. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theor. **2**(2) , 127–145 (1968). Correction: Math. Syst. Theor. **5**(1), 95–96 (1971)
7. Dehayni, M., Féraud, L.: An approach of model transformation based on attribute grammars. In: Masood, A., Léonard, M., Pigneur, Y., Patel, S. (eds.) OOIS 2003. LNCS, vol. 2817, pp. 412–423. Springer, Heidelberg (2003)
8. Swierstra, S., Alcocer, P.A., Saraiva, J.: Designing and implementing combinator languages. In: Swierstra, S., Oliveira, J., Henriques, P. (eds.) Adv. Funct. Program. Lecture Notes in Computer Science, vol. 1608, pp. 150–206. Springer, Heidelberg (1999)
9. OMG: Object Constraint Language. Formal Specification Version 2.4, Object Management Group (2014)
10. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. SIGPLAN Not. **24**(7), 131–145 (1989)
11. Gargantini, A., Riccobene, E., Scandurra, P.: Deriving a textual notation from a metamodel. In: Proceedings of Workshop on Milestones, Models and Mappings for Model-Driven Architecture. Volume WP06-02, ISSN1574-0846 of CTITSeries. (2006)
12. Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Technical Report 606, Turku Centre for Computer Science (2003)
13. Grammes, R., Gotzhein, R.: Towards the harmonisation of UML and SDL. In: de Frutos-Escrig, D., Núñez, M., (eds.) Proceedings of Formal Techniques for Networked and Distributed Systems 2004, Madrid Spain, 27–30 September 2004, pp. 61–78. Springer (2004)
14. Zhu, H.: An institution theory of formal meta-modelling in graphically extended bnf. Front. Comput. Sci. **6**(1), 40–56 (2012)
15. Bürger, C., Karol, S., Wende, C., Aßmann, U.: Reference attribute grammars for metamodel semantics. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 22–41. Springer, Heidelberg (2011)

16. Magnusson, E., Hedin, G.: Circular reference attributed grammars - their evaluation and applications. Sci. Comput. Program. **68**(1), 21–37 (2007)
17. Calegari, D., Szasz, N.: Verification of model transformations: a survey of the state-of-the-art. Electr. Notes Theor. Comput. Sci. **292**, 5–25 (2013)
18. Baars, A.I., Swierstra, S.D., Viera, M.: Typed transformations of typed abstract syntax. In: TLDI 2009: Proceedings of the 4th International Workshop on Types in Language Design and Implementation, pp. 15–26. ACM, New York (2009)