Towards a Rewriting Logic Semantics for ATL

Javier Troya and Antonio Vallecillo

GISUM/Atenea Research Group, Universidad de Málaga, Spain
{javiertc,av}@lcc.uma.es

Abstract. As the complexity of model transformation (MT) grows, the need to count on formal semantics of MT languages also increases. Firstly, formal semantics provide precise specifications of the expected behavior of transformations, which are crucial for both MT users (to be able to understand them and to use them properly) and MT tool builders (to develop correct MT engines, optimizers, etc.). Secondly, we need to be able to reason about the MTs to prove their correctness. This is specially important in case of large and complex MTs (with, e.g., hundreds or thousands of rules) for which manual debugging is no longer possible. In this paper we present a formal semantics to the ATL model transformation language using rewriting logic and Maude, which allows addressing these issues. This formalization provides additional benefits, such as enabling the simulation of the specifications or giving access to the Maude toolkit to reason about them.

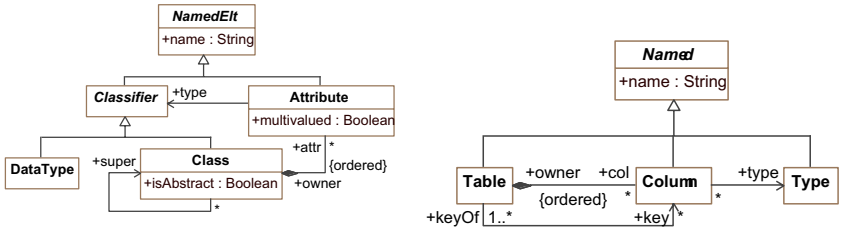
1 Introduction

Model transformations (MT) are at the heart of Model-Driven Engineering, and provide the essential mechanisms for manipulating and transforming models. As the complexity of model transformations grows, the need to count on formal semantics of MT languages also increases. Formal semantics provide precise specifications of the expected behavior of the transformations, which are crucial for all MT stakeholders: users need to be able to understand and use model transformations properly; tool builders need formal and precise specifications to develop correct model transformation engines, optimizers, debuggers, etc.; and MT programmers need to know the expected behavior of the rules and transformations they write, in order to reason about them and prove their correctness. This is specially important in case of large and complex MTs (with, e.g., hundreds or thousands of rules) for which manual debugging is no longer possible. For instance, in the case of rule-based model transformation languages, proving that the specifications are confluent and terminating is required. Also, looking for non-triggered rules may help detecting potential design problems in large MT systems.

ATL [1] is one of the most popular and widely used model transformation languages. The ATL language has been normally described in an intuitive and informal manner, by means of definitions of its main features in natural language. However, this lack of rigorous description can easily lead to imprecisions and misunderstandings that might hinder the proper usage and analysis of the language, and the development of correct and interoperable tools.

In this paper we investigate the use of rewriting logic [2] and its implementation in Maude [3], for giving semantics to ATL. The use of Maude as a target semantic domain

L. Tratt and M. Gogolla (Eds.): ICMT 2010, LNCS 6142, pp. 230–244, 2010.



(a) Source metamodel. (b) Target metamodel.

Fig. 1. Transformation metamodels

brings interesting benefits, because it enables the simulation of the ATL specifications and its formal analysis. In particular, we show how our specifications can make use of the Maude toolkit to reason about some properties of the ATL rules.

2 Transformations with ATL

ATL is a hybrid model transformation domain specific language containing a mixture of declarative and imperative constructs. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. During the execution of a transformation, source models may be navigated but changes are not allowed. Target models cannot be navigated.

ATL modules define the transformations. A module contains a mandatory header section, an import section, and a number of helpers and transformation rules. The header section provides the name of the transformation module and declares the source and target models (which are typed by their metamodels). Helpers and transformation rules are the constructs used to specify the transformation functionality.

Declarative ATL rules are called **matched rules** and **lazy rules**. Lazy rules are like matched rules, but are only applied when called by another rule. They both specify relations between source patterns and target patterns. The source pattern of a rule specifies a set of source types and an optional guard given as a Boolean expression in OCL. A source pattern is evaluated to a set of matches in source models. The target pattern is composed of a set of elements. Each of these elements specifies a target type from the target metamodel and a set of bindings. A *binding* refers to a feature of the type (i.e., an attribute, a reference or an association end) and specifies an expression whose value is used to initialize the feature. Lazy rules can be called several times using a collect construct. **Unique lazy rules** are a special kind of lazy rules that always return the same target element for a given source element. The target element is retrieved by navigating the internal traceability links, as in normal rules. Non-unique lazy rules do not navigate the traceability links but create new target elements in each execution.

In some cases, complex transformation algorithms may be required, and it may be difficult to specify them in a declarative way. For this reason ATL provides two imperative constructs: **called rules** and **action blocks**. A called rule is a rule called by others like a procedure. An action block is a sequence of imperative statements and can

be used instead of or in combination with a target pattern in matched or called rules. The imperative statements in ATL are the usual constructs for attribute assignment and control flow: conditions and loops.

ATL also provides the **resolveTemp** operation for dealing with complex transformations. This operation allows to refer to any of the target model elements generated from a given source model element: resolveTemp(srcObj,targetPatternName). The first argument is the source model element, and the second is a string with the name of the target pattern element. This operation can be called from the target pattern and imperative sections of any matched or called rule.

In order to illustrate our proposal we will use the typical example of Class-to-Relational model transformation. This example and many others can be found in [4]. The two metamodels involved in this transformation are shown in Fig. 1. Our input model contains two classes: Family and Person. It is shown below, written in KM3 [5]:

```
datatype String;
datatype Integer;
class Family {
  attribute name : String;
  attribute members[*] : Person; }
class Person {
  attribute firstName : String;
  attribute closestFriend : Person;
  attribute emailAddresses[*] : String; }
```

An example of an output model is the following:

```
table Person {
  primary column objectId : Integer;
  column firstName : String;
  column closestFriendId : Integer; }
table Family_members {
  column familyId : Integer;
  column membersId : Integer; }
table Person_emailAddresses {
  column personId : Integer;
  column emailAddresses : String; }
table Family {
  primary column objectId : Integer;
  column name : String; }
```

An ATL transformation that takes the first model as input and produces the second is shown below. It has 6 rules, each one showing a particular ATL feature. The complete description of this example and its encoding in Maude can be found in [6]. Although the ATL rules are mostly self-explanatory, readers not fluent in ATL can consult [1,6] for all the details.

```
module Class2Relation; -- Module Template
create OUT : RelationMM from IN : ClassMM;
helper def: objectIdType : Relational!Type =
     Class!DataType.allInstances() ->select(e | e.name = 'Integer')->first();
rule DataType2Type {
  from dt : Class!DataType
  to    t : Relational!Type ( name <- dt.name ) }
rule SingleValuedDataTypeAttribute2Column {
  from at : Class!Attribute (
         at.type.oclIsKindOf(Class!DataType) and not at.multiValued )
  to   co : Relational!Column ( name <- at.name, type <- at.type ) }
rule MultiValuedDataTypeAttribute2Column {
  from at : Class!Attribute ( at.type.oclIsKindOf(Class!DataType)
                          and at.multiValued )
  to   tb : Relational!Table (name <- at.owner.name + '_' + at.name,
```

```
                              col  <- Sequence {co, coo}),
      co : Relational!Column (
         name <- at.owner.name + 'Id', type <- thisModule.objectIdType),
     coo : Relational!Column ( name <- at.name, type <- at.type ) }
rule SingleValuedClassAttribute2Column {
  from at : Class!Attribute (
         at.type.oclIsKindOf(Class!Class) and not at.multiValued )
  to   co : Relational!Column ( name <- at.name + 'Id',
                                type <- thisModule.objectIdType ) }
rule MultiValuedClassAttribute2Column {
  from at : Class!Attribute (
           at.type.oclIsKindOf(Class!Class) and at.multiValued )
  to   tb : Relational!Table ( name <- at.owner.name + '_' + at.name,
                               col <- Sequence {id, k} ),
       id : Relational!Column ( name <- at.owner.name.firstToLower() + 'Id',
                                type <- thisModule.objectIdType ),
       k : Relational!Column (
           name <- at.name + 'Id', type <- thisModule.objectIdType ) }
rule Class2Table {
  from c  : Class!Class
  to   tb : Relational!Table ( name <- c.name,
         col <- Sequence {k}->union(c.attr->select(iter | not iter.multiValued)),
         key <- Set {k} ),
       k : Relational!Column(name <- 'objectId', type <-thisModule.objectIdType)}
```

3 Rewriting Logic and Maude

Maude [3] is a high-level language and a high-performance interpreter in the OBJ algebraic specification family that supports membership equational logic [7] and rewriting logic [2] specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. We informally describe in this section those Maude's features necessary for understanding the paper; the interested reader is referred to [3] for more details.

Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. A distributed system is axiomatized in rewriting logic by a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory describing its set of *states* as the algebraic data type $T_{\Sigma/E}$ associated to the initial algebra (Σ, E), and R is a collection of rewrite rules. Maude's underlying equational logic is membership equational logic [7], a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort overloading of operators, and definition of partial functions with equationally defined domains.

Rewrite rules, which are written crl $[l] : t \Rightarrow t'$ if $Cond$, with l the rule label, t and t' terms, and $Cond$ a condition, describe the local, concurrent transitions that are possible in the system, i.e., when a part of the system state fits the pattern t, then it can be replaced by the corresponding instantiation of t'. The guard $Cond$ acts as a blocking precondition, in the sense that a conditional rule can be fired only if its condition holds.

The form of conditions is $EqCond_1$ /\ ... /\ $EqCond_n$ where each of the $EqCond_i$ is either an ordinary equation $t = t'$, a *matching equation* $t := t'$, a sort constraint $t : s$, or a term t of sort Bool, abbreviating the equation t = true. In the execution of a matching equation $t := t'$, the variables of the term t, which may not appear in the lefthand side

of the corresponding conditional equation, become instantiated by *matching* the term t against the canonical form of the bounded subject term t'.

4 Encoding ATL in Maude

To give a formal semantics to ATL using rewriting logic, we provide a representation of ATL constructs and behavior in Maude. We start by defining how the models and metamodels handled by ATL can be encoded in Maude, and then provide the semantics of matched rules, lazy rules, unique lazy rules, helpers and imperative sections. One of the benefits of such an encoding is that it is systematic and can be automated, something we are currently implementing using ATL transformations (between the ATL and Maude metamodels).

4.1 Characterizing Model Transformations

In our view, a model transformation is just an algorithmic specification (let it be declarative or operational) associated to a relation $R \subseteq MM \times MN$ defined between two metamodels which allows to obtain a target model N conforming to MN from a source model M that conforms to metamodel MM [8].

The idea supporting our proposal considers that model transformations comprise two different aspects: *structure* and *behavior*. The former aspect defines the structural relation R that should hold between source and target models, whilst the latter describes how the specific source model elements are transformed into target model elements. This separation allows differentiating between the relation that the model transformation ensures from the algorithm it actually uses to compute the target model.

Thus, to represent the structural aspects of a transformation we will use three models: the source model M, the target model N that the transformation builds, and the relation $R(M, N)$ between the two. $R(M, N)$ is also called the *trace* model, that specifies how the elements of M and N are consistently related by R. Please note that each element r_i of $R(M, N) = \{r_1, ..., r_k\} \subseteq \mathbb{P}(M) \times \mathbb{P}(N)$ relates a set of elements of M with a set of elements of N.

The behavioral aspects of an ATL transformation (i.e., how the transformation progressively builds the target model elements from the source model, and the traces between them) is defined using the different kinds of rules (matched, lazy, unique lazy); their possible combinations and direct invocation from other rules, and the final imperative algorithms that can be invoked after each rule.

4.2 Encoding Models and Metamodels in Maude

We will follow the representation of models and metamodels introduced in [9], which is inspired in the Maude representation of object-oriented systems mentioned. We represent models in Maude as structures of sort @Model of the form $mm\{obj_1\ obj_2\ ...\ obj_N\}$, where mm is the name of the metamodel and obj_i are the objects of the model. An object is a record-like structure $< o : c \mid a_1 : v_1, ..., a_n : v_n >$ (of sort @Object), where o is the object identifier (of sort Oid), c is the class the object belongs to (of sort @Class), and $a_i : v_i$ are attribute-value pairs (of sort @StructuralFeatureInstance).

Given the appropriate definitions for all classes, attributes and references in its corresponding metamodel (as we shall see below), the following Maude term describes the input model shown in Sect. 2.

```
@ClassMm@ {
  < 'd1 : DataType@ClassMm | name@NamedElt@ClassMm : "Integer" >
  < 'd2 : DataType@ClassMm | name@NamedElt@ClassMm : "String" >
  < 'a1 : Attribute@ClassMm | multivalued@Attribute@ClassMm : false #
      name@NamedElt@ClassMm : "name" # type@Attribute@ClassMm : 'd2 #
      owner@Attribute@ClassMm : 'c1 >
  < 'a2 : Attribute@ClassMm | multivalued@Attribute@ClassMm : true #
      name@NamedElt@ClassMm : "members" # type@Attribute@ClassMm : 'c2 #
      owner@Attribute@ClassMm  : 'c1 >
  < 'a3 : Attribute@ClassMm | multivalued@Attribute@ClassMm : false #
      name@NamedElt@ClassMm : "firstName" # type@Attribute@ClassMm : 'd2 #
      owner@Attribute@ClassMm : 'c2 >
  < 'a4 : Attribute@ClassMm | multivalued@Attribute@ClassMm : false #
      name@NamedElt@ClassMm : "closestFriend" # type@Attribute@ClassMm : 'c2 #
      owner@Attribute@ClassMm : 'c2 >
  < 'a5 : Attribute@ClassMm | multivalued@Attribute@ClassMm : true #
      name@NamedElt@ClassMm : "emailAddresses" # type@Attribute@ClassMm : 'd2 #
      owner@Attribute@ClassMm  : 'c2 >
  < 'c1 : Class@ClassMm | isAbstract@Class@ClassMm : false #
      name@NamedElt@ClassMm : "Family" # att@Class@ClassMm : Sequence['a1 ; 'a2] #
      super@Class@ClassMm : null >
  < 'c2 : Class@ClassMm | isAbstract@Class@ClassMm : false #
      name@NamedElt@ClassMm : "Person" # att@Class@ClassMm :
      Sequence['a3 ; 'a4 ; 'a5]# super@Class@ClassMm : null >  }
```

Note that quoted identifiers are used as object identifiers; references are encoded as object attributes by means of object identifiers; and OCL collections (Set, OrderedSet, Sequence, and Bag) are supported by means of mOdCL [10].

Metamodels are encoded using a sort for every metamodel element: sort @Class for classes, sort @Attribute for attributes, sort @Reference for references, etc. Thus, a metamodel is represented by declaring a constant of the corresponding sort for each metamodel element. More precisely, each class is represented by a constant of a sort named after the class. This sort, which will be declared as subsort of sort @Class, is defined to support class inheritance through Maude's order-sorted type structure. Other properties of metamodel elements, such as whether a class is abstract or not, the opposite of a reference (to represent bidirectional associations), or attributes and reference types, are expressed by means of Maude equations defined over the constant that represents the corresponding metamodel element. Classes, attributes and references are qualified with their containers' names, so that classes with the same name belonging to different packages, as well as attributes and references of different classes, are distinguished. See [9] for further details.

4.3 Modeling ATL Rules

Matched rules. Each ATL matched rule is represented by a Maude rewrite rule that describes how the target model elements are created from the source model elements identified in the left-hand side of the rule (that represents the "to" pattern of the ATL rule). The general form of such rewrite rules is as follows:

```
crl [rulename] :
  Sequence[
    (@SourceMm@ { ... OBJSET@ }) ;
    (@TraceMm@ { ... OBJSETT@ }) ;
```

```
    (@TargetMm@ { OBJSETTT@ }) ]
  => Sequence[
       (@SourceMm@ { ... OBJSET@ }) ;
       (@TraceMm@ { ... OBJSETT@}) ;
       (@TargetMm@ { ... OBJSETTT@ }) ]
  if ...
     /\ not alreadyExecuted(..., "rulename", @TraceMm@ { OBJSETT@ }) .
```

The two sides of the Maude rule contain the three models that capture the state of the transformation (see 4.1): the source, the trace and the target models.[1] The rule specifies how the state of the ATL model transformation changes as result of such rule.

The triggering of Maude and ATL rules is similar: a rule is triggered if the pattern specified by the rule is found, and the guard condition holds. In addition to the specific rule conditions, in the Maude representation we also check (alreadyExecuted) that the same ATL rule has not been triggered with the same elements.

An additional Maude rule, called Init, starts the transformation. It creates the initial state of the model transformation, and initializes the target and trace models:

```
rl [Init] :
  Sequence[ (@ClassMm@ { OBJSET@ }) ]
  => Sequence[
    (@ClassMm@ { OBJSET@ }) ;
    (@TraceMm@ { < 'CNT : Counter@CounterMm | value@Counter@CounterMm : 1 > }) ;
    (@RelationalMm@ { none }) ] .
```

The traces stored in the trace model are also objects, of class Trace@TraceMm, whose attributes are: two sequences (srcEl@TraceMm and trgEl@TraceMm) with the sets of identifiers of the elements of the source and target models related by the trace; the rule name (rlName@TraceMm); and a reference to the source and target metamodels: srcMdl@TraceMm and trgMdl@TraceMm.

The trace model also contains a special object, of class Counter@CounterMm, whose integer attribute is used as a counter for assigning fresh identifiers to the newly created elements and traces. As an example, consider the first of the rules of Class2Relation transformation described in Sect. 2:

```
rule DataType2Type {
  from dt : Class!DataType
  to     t : Relational!Type ( name <- dt.name ) }
```

The encoding in Maude of such a rule is as follows:

```
crl[DataType2Type] :
  Sequence[
    (@ClassMm@ { < DT@ : DataType@ClassMm | SFS > OBJSET@ }) ;
    (@TraceMm@ {
      < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
      OBJSETT@ }) ;
    (@RelationalMm@ { OBJSETTT@ }) ]
  => Sequence[
    (@ClassMm@ { < DT@ : DataType@ClassMm | SFS > OBJSET@ }) ;
    (@TraceMm@ {
      < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 2 >
      < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ DT@ ] #
        trgEl@TraceMm : Sequence[ T@ ] # rlName@TraceMm : "DataType2Type" #
        srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" > OBJSETT@}) ;
```

[1] For simplicity, in this paper we will show examples where the transformation deals with only one input model. ATL can handle more than one, but the treatment in Maude is analogous—it is just a matter of including more models in the specification of the relation.

```
   (@RelationalMm@ { < T@ : Type@RelationalMm |
     name@Named@RelationalMm : << DT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> >
     OBJSETTT@ }) ]
 if CLASSMODEL@ := @ClassMm@ { < DT@ : DataType@ClassMm | SFS > OBJSET@ }
  /\ TR@ := newId(VALUE@CNT@) /\ T@ := newId(VALUE@CNT@ + 1)
  /\ not alreadyExecuted(Sequence[DT@],"DataType2Type",@TraceMm@ { OBJSETT@ }) .
```

This rule is applied over instances of class DataType, as specified in the left hand side of the Maude rule. The rule guard guarantees that the rule has not been already applied over the same elements. The guard is also used to define some variables used by the rule (CLASSMODEL@, TR@ and T@).

After the application of the rule, the state of the system is changed: the source model is left unmodified (ATL does not allow modifying the source models); a new trace (TR@) is added to the trace model; the value of the counter object is updated; and a new element (T@) is created in the target model. We allow the evaluation of OCL expressions using mOdCL [10] by enclosing them in double angle brackets (<< ... >>).

The encoding of the rest of the ATL matched rules in the example of Sect. 2 follow the same schema and can be found in [6].

Lazy rules. While matched rules are executed in non-deterministic order (as soon as their "to:" patterns are matched in the source model), lazy rules are executed only when they are explicitly called by other rules. Thus, we have modeled lazy rules as Maude operations, whose arguments are the parameters of the corresponding rule, and return the set of elements that have changed or need to be created. In this way the operations can model the calling of ATL rules in a natural way.

Maude operations representing ATL lazy rules do not modify the trace model, this is the responsibility of the Maude calling rule. For every invoked lazy rule a trace is created. The name of the ATL rule recorded in the trace is not the name of the lazy rule, but the name of the matched rule concatenated with "_" and with the name of the lazy rule. We represent them in this way because a lazy rule can be invoked by different calling rules, and in this way we know which matched rule called it.

Special care should be taken when lazy rules are called from a collect construct. When lazy rules are not called from a collect, it is only necessary to write, in the target model, the identifier of the first object created by the lazy rule when we want to reference the objects it creates. But with lazy rules called from a collect we need to reference the sequence of objects created by the lazy rule. To do this, we use an auxiliary function, getOidsCollect, whose arguments are the ones of the lazy rule, the identifier of the first element created by the lazy rule, and the number of objects created in each iteration by the lazy rule. It returns a sequence with the identifiers of the objects created by the lazy rule, in the same order.

Unique lazy rules. This kind of rules deserve a special encoding in Maude, because their behavior is quite different from normal lazy rules. Now we need to check if the element created by the lazy rule is already there, or if it has to be created. If it was already there, we need to get its identifier. We also have to be careful with the traces, since only one trace has to be added for the elements created by a unique lazy rule.

Helpers. Helpers are side-effect free functions that can be used by the transformation rules for realizing the functionality. Helpers are normally described in OCL. Thus, their representation is direct as Maude operations that make use of mOdCL for evaluating the OCL expression of their body. For instance, the following Maude operation represents the objectIdType helper shown in the ATL example in Sect. 2:

```
op objectIdType : @Model @Model -> Oid .
  eq objectIdType(@ClassMm@{OBJSET}, @TraceMm@{OBJSETT})
    = getTarget(<< DataType@ClassMm . allInstances -> select( ITER | ITER .
      name@NamedElt@ClassMm .=. "Integer" ) -> asSequence() -> first() ;
      @ClassMm@{OBJSET} >> , @TraceMm@{OBJSETT}) .
```

This helper receives the class and trace models as arguments. It returns the first Type whose name is Integer by looking for it in the trace model with the getTarget operation. OCL expressions allInstances, select, asSequence and first are encoded as such.

The imperative section. We represent the imperative section of rules using a data type called Instr that we have defined for representing the different *instructions* that are possible within a do block. Currently we implement three types of instructions: *assignments* (=), *conditional* branches (if) and *called rules*. In the following piece of code we show how type Instr and the sequence of instructions (instrSeq) are defined:

```
sort Instr instrSeq .
subsort Instr < instrSeq .
op none : -> instrSeq [ctor] .
op _^_ : Instr instrSeq -> instrSeq [ctor id: none] .
op Assign : Oid @StructuralFeature OCL-Exp -> Instr [ctor] .
op If : Bool instrSeq instrSeq -> Instr [ctor] .
---Instruction for our called rule
op AddColumn : Int String -> Instr [ctor] .
```

Thus, the same instruction is used for assignments and conditional instructions. A new instruction is needed for each called rule (AddColumn in this case).

The ATL imperative section, which is within a do block, is encapsulated in Maude by a function called do which receives as arguments the set of objects created by the declarative part of the rule, and the sequence of instructions to be applied over those objects. It returns the sequence of objects resulting from applying the instructions:

```
op do : Set{@Object} InstrSequ -> Set{@Object} .
  eq do(OBJSET@, none) = OBJSET@ .
  eq do(OBJSET@, Assign(O@, SF@, EXP@) ^ INSTR@) =
     do(doAssign(OBJSET@, O@, SF@, EXP@), INSTR@) .
  eq do(OBJSET@, If(COND@, INSTR1@, INSTR2@) ^ INSTR@) =
     if COND@ then do(OBJSET@, INSTR1@ ^ INSTR@)
     else do(OBJSET@, INSTR2@ ^ INSTR@)
     fi .
  ---For each called rule, AddColumn in this case
  eq do(OBJSET@, AddColumn(VALUE@CNT@, NAME) ^ INSTR@) =
     do(doAddColumn(OBJSET@, VALUE@CNT@, NAME), INSTR@) .
```

We see that the function is recursive, so it applies the instructions one by one, in the same order as they appear in the ATL do block. When the function finds an Assign instruction, it applies the doAssign operation. When it finds an If instruction, it checks wether the condition is satisfied or not, applying a different sequence of instructions in each case. With regard to called rule instructions, the Maude do operation applies them as they appear. The two operations mentioned are the following:

```
op doAssign : Set{@Object} Oid @StructuralFeature OCL-Exp -> Set{@Object} .
  eq doAssign(< O@ : CL@ | SF@ : TYPE@ # SFS > OBJSET@, O@, SF@, EXP@) =
     < O@ : CL@ | SF@ : EXP@ # SFS > OBJSET@ .

op doAddColumn : Set{@Object} Int String -> Set{@Object} .
  eq doAddColumn(OBJSET@, VALUE@CNT@, NAME) =
     < newId(VALUE@CNT@) : Column@RelationalMm | name@Named@RelationalMm : NAME >
     OBJSET@ .
```

Function doAssign assigns an OCL expression to an attribute of an object. It receives the set of objects created in the declarative part, the identifier of the object and its attribute, and the OCL expression that will be assigned to the attribute of the object. The function replaces the old value of the attribute with the result of the evaluation of the OCL expression. Function doAddColumn creates a new Column instance. It receives the set of objects created by the declarative part of the rule, the counter for giving an identifier to the new object, and the String that will be the name of the Column.

For example, consider the following ATL rule, that contains an imperative part to modify the elements that have been created by the rule:

```
rule MultiValuedClassAttribute2Column {
  from at : Class!Attribute(at.type.oclIsKindOf(Class!Class) and at.multiValued)
  to   tb : Relational!Table (
            name <- at.owner.name + '_' + at.name,
            col <- Sequence {id, k} ),
       id : Relational!Column (
            name <- at.owner.name.firstToLower() + 'Id',
            type <- thisModule.objectIdType ),
       k :  Relational!Column (name <- at.name + 'Id',
            type <- thisModule.objectIdType )
  do {  tb.name <- tb.name + '_Multi';
        if (tb.col->size() = 2) { k.name <- 'key'; }
        else { k.name <- 'key_else'; }
        thisModule.AddColumn( New_Column ) ; }
}
```

The corresponding encoding in Maude is as follows:

```
crl[MultiValuedClassAttribute2Column] :
  Sequence[...]
  => Sequence[...
    (@RelationalMm@ { do(
       < TB@ : Table@RelationalMm | name@Named@RelationalMm : << AT@ .
         owner@Attribute@ClassMm . name@NamedElt@ClassMm + "_" + AT@ .
         name@NamedElt@ClassMm ; CLASSMODEL@ >> #
         col@Table@RelationalMm : Sequence[ID@ ; K@] #
         key@Table@RelationalMm : Set{K@} >
       < ID@ : Column@RelationalMm | name@Named@RelationalMm : << AT@ .
         owner@Attribute@ClassMm . name@NamedElt@ClassMm + "Id" ; CLASSMODEL@ >> #
         type@Column@RelationalMm:objectIdType(CLASSMODEL@,@TraceMm@{OBJSETT@})>
       < K@ : Column@RelationalMm | name@Named@RelationalMm : << AT@ .
         name@NamedElt@ClassMm + "Id" ; CLASSMODEL@ >> #
         type@Column@RelationalMm:objectIdType(CLASSMODEL@,@TraceMm@{OBJSETT@})>,
       Assign(TB@, name@Named@RelationalMm, << AT@ . owner@Attribute@ClassMm .
       name@NamedElt@ClassMm + "_" + AT@ . name@NamedElt@ClassMm ;
       CLASSMODEL@ >> + "_Multi") ^
       If(<< Sequence[ID@ ; K@] -> size() ; CLASSMODEL@ >> == 3,
       Assign(K@,name@Named@RelationalMm, "key"),
       Assign(K@, name@Named@RelationalMm, "key_else")) ^
       AddColumn(VALUE@CNT@, "New_Column") ) OBJSETTT@ })  ]
  if ...
```

The first argument of the function do is the set of objects created in the declarative part of the rule. Consequently, we make the declarative part of the rule to be executed

before the imperative part. This reproduces the way in which ATL works. The second argument is a sequence of instructions which contains, in this case, three instructions. The first instruction executed is the Assign, then the If, and, finally, the instruction that represents the called rule, AddColumn, is executed.

ResolveTemp. The function looks for the trace which contains the source element passed as first argument and returns the identifier of the element from the sequence of elements created from the source element. It can be implemented in Maude as follows:

```
op resolveTemp : Oid Nat @Model @Model -> Oid .
 eq resolveTemp(O@ , N@ , @TraceMm@{ < TR@ : Trace@TraceMm | srcEl@TraceMm :
    Sequence[O@] # trgEl@TraceMm : SEQ # SFS > OBJSET} , CLASSMODEL@ ) =
    if (<< SEQ -> size ( ) < N@ ; CLASSMODEL@ >>) then null
    else << SEQ -> at(N@) ; CLASSMODEL@ >>
    fi .
```

It has four arguments: the identifier of the source model element from which the searched target model element is produced; the position of the target object identifier in the sequence trgEl@TraceMm; and the trace and class models, respectively. It returns the identifier of the element to be retrieved. The major difference with the ATL function is that here we receive as second argument the position that the searched target model element has among the ones created by the corresponding rule. In ATL, instead, the argument received is the name of the variable that was given to the target model element when it was created. This difference is not important since it is easy to retrieve the position that the element has among the elements created by the ATL rule.

5 Simulation and Formal Analysis

Once the ATL model transformation specifications are encoded in Maude, what we get is a rewriting logic specification for it. Maude offers tool support for interesting possibilities such as model simulation, reachability analysis and model checking [3].

5.1 Simulating the Transformations

Because the rewriting logic specifications produced are executable, this specification can be used as a prototype of the transformation, which allows us to simulate it. Maude offers different possibilities for realizing the simulation, including step-by-step execution, several execution strategies, etc. In particular, Maude provides two different rewrite commands, namely rewrite and frewrite, which implement two different execution strategies, a top-down rule-fair strategy, and a depth-first position-fair strategy, respectively [3]. The result of the process is the final configuration of objects reached after the rewriting steps, which is nothing but a model.

For example, the result of the ATL model transformation described in Section 2, when applied to the source Class model, is a sequence of three models: the source, the trace and the target Relational model. This last one is shown below.

```
@RelationalMm@{
  < '2 : Type@RelationalMm | name@Named@RelationalMm : "Integer" >
  < '10 : Type@RelationalMm | name@Named@RelationalMm : "String" >
  < '23 : Table@RelationalMm | name@Named@RelationalMm : "Person",
    col@Table@RelationalMm : Sequence ['24 ; '4 ; '21],
```

```
    key@Table@RelationalMm : Set {'24} >
  < '24 : Column@RelationalMm | name@Named@RelationalMm : "objectId",
    type@Column@RelationalMm : '2 >
  < '21 : Column@RelationalMm | name@Named@RelationalMm : "firstName",
    type@Column@RelationalMm : '10 >
  < '4 : Column@RelationalMm | name@Named@RelationalMm : "closestFriendId",
    type@Column@RelationalMm : '2 >
  < '6 : Table@RelationalMm | name@Named@RelationalMm : "Family_members",
    col@Table@RelationalMm : Sequence ['7 ;'8],key@Table@RelationalMm : Set {'8}>
  < '7 : Column@RelationalMm | name@Named@RelationalMm : "FamilyId",
    type@Column@RelationalMm : '2 >
  < '8 : Column@RelationalMm | name@Named@RelationalMm : "membersId",
    type@Column@RelationalMm : '2 >
  < '14 : Table@RelationalMm | name@Named@RelationalMm : "Person_emailAddresses",
    col@Table@RelationalMm : Sequence ['15 ; '16] >
  < '15 : Column@RelationalMm | name@Named@RelationalMm : "PersonId",
    type@Column@RelationalMm : '2 >
  < '16 : Column@RelationalMm | name@Named@RelationalMm : "emailAddresses",
    type@Column@RelationalMm : '10 >
  < '18 : Table@RelationalMm | name@Named@RelationalMm : "Family",
    col@Table@RelationalMm : Sequence ['19 ; '12],key@Table@RelationalMm :
    Set {'19} >
  < '19 : Column@RelationalMm | name@Named@RelationalMm : "objectId",
    type@Column@RelationalMm : '2 >
  < '12 : Column@RelationalMm | name@Named@RelationalMm : "name",
    type@Column@RelationalMm : '10 >
}
```

After the simulation is complete we can also analyze the trace model, looking for instance for rules that have not been executed, or for obtaining the traces (and source model elements) related to a particular target model element (or viceversa). Although this could also be done in any transformation language that makes the trace model explicit, the advantages of using our encoding in Maude is that these operations become easy because of Maude's facilities for manipulating sets:

```
op getSourceElements : @Model Oid -> Sequence .
  eq getSourceElements(@TraceMm@{< TR@ : Trace@TraceMm | srcEl@TraceMm :
     SEQ # trgEl@TraceMm : Sequence[O@ ; LO] # SFS > OBJSET}, O@) = SEQ .
  eq getSourceElements(@TraceMm@{< TR@ : Trace@TraceMm | srcEl@TraceMm :
     SEQ # trgEl@TraceMm : Sequence[T@ ; LO] # SFS > OBJSET}, O@)
     = getSourceElements(@TraceMm@{< TR@ : Trace@TraceMm | srcEl@TraceMm :
     SEQ # trgEl@TraceMm : Sequence[LO] # SFS > OBJSET}, O@) .
  eq getSourceElements(@TraceMm@{OBJSET} , O@) = Sequence[mt-ord] [owise] .
```

5.2 Reachability Analysis

Executing the system using the rewrite and frewrite commands means exploring just one possible behavior of the system. However, a rewrite system does not need to be Church-Rosser and terminating,[2] and there might be many different execution paths. Although these commands are enough in many practical situations where an execution path is sufficient for testing executability, the user might be interested in exploring all possible execution paths from the starting model, a subset of these, or a specific one.

Maude search command allows us to explore (following a breadthfirst strategy up to a specified bound) the reachable state space in different ways, looking for certain states

[2] For membership equational logic specifications, being Church-Rosser and terminating means not only confluence (a unique normal form will be reached) but also a sort decreasingness property, namely that the normal form will have the least possible sort among those of all other equivalent terms.

of special interest. Other possibilities would include searching for any state (given by a model) in the execution tree, let it be final or not. For example, we could be interested in knowing the partial order in which two ATL matched rules are executed, checking that one always occurs before the other. This can be proved by searching for states that contain the second one in the trace model, but not the first.

6 Related Work

The definition of a formal semantics for ATL has received attention by different groups, using different approaches. For example, in [11] the authors propose an extension of AMMA, the ATLAS Model Management Architecture, to specify the dynamic semantics of a wide range of Domain Specific Languages by means of Abstract State Machines (ASMs), and present a case study where the semantics of part of ATL (namely, matched rules) are formalized. Although ASMs are very expressive, the declarative nature of ATL does not help providing formal semantics to the complete ATL language in this formalism, hindering the complete formalization of the language—something that we were pursuing with our approach.

Other works [12,13] have proposed the use of Alloy to formalize and analyze graph transformation systems, and in particular ATL. The analyses include checking the reachability of given configurations of the host graph through a finite sequence of steps (invocations of rules), and verifying whether given sequences of rules can be applied on an initial graph. These analyses are also possible with our approach, and we also obtain significant gains in expressiveness and completeness. The problem is that Alloy expressiveness and analysis capabilities are quite limited [13]: it has a simple type system with only integers; models in Alloy are static, and thus the approach presented in [13] can only be used to reason about static properties of the transformations (for example it is not possible to reason whether applying a rule r_1 before a rule r_2 in a model will have the same effect as applying r_2 before r_1); only ATL declarative rules are considered, etc. In our approach we can deal with all the ATL language constructs without having to abstract away essential parts such as the imperative section, basic types, etc. More kinds of analysis are also possible with our approach.

Other works provide formal semantics to model transformation languages using types. For intance, Poernomo [14] uses Constructive Type Theory (CTT) for formalizing model transformation and proving their correctness with respect to a given pre- and post-condition specification. This approach can be considered as complementary to ours, each one focusing on different aspects.

There are also the early works in the graph grammar community with a logic-based definition and formalization of graph transformation systems. For example, Courcelle [15] proposes a combination of graph grammars with second order monadic logic to study graph properties and their transformations. Schürr [16] has also studied the formal specification of the semantics of the graph transformation language PROGRES by translating it into some sort of non-monotonic logics.

A different line of work proposed in [17] defines a QVT-like model transformation language reusing the main concepts of graph transformation systems. They formalize their model transformations as theories in rewriting logic, and in this way Maude's

reachability analysis and model checking features can be used to verify them. Only the reduced part of QVT relations that can be expressed with this language is covered. Our work is different: we formalize a complete existing transformation language by providing its representation in Maude, without proposing yet another MT language.

Finally, Maude has been proposed as a formal notation and environment for specifying and effectively analyzing models and metamodels [9,18]. Simulation, reachability and model-checking analysis are possible using the tools and techniques provided by Maude [9]. We build on these works, making use of one of these formalizations to represent the models and metamodels that ATL handles.

7 Conclusions and Future Work

In this paper we have proposed a formal semantics for ATL by means of the representation of its concepts and mechanisms in Maude. Apart for providing a precise meaning to ATL concepts and behavior (by its interpretation in rewriting logic), the fact that Maude specifications are executable allows users to simulate the ATL programs. Such an encoding has also enabled the use of Maude's toolkit to reason about the specifications.

In general, it is unrealistic to think that average system modelers will write these Maude specifications. One of the benefits of our encoding is that it is systematic, and therefore it can be automated. Thus we have defined a mapping between the ATL and the Maude metamodels (a *semantic mapping* between these two semantic domains) that realizes the automatic generation of the Maude code. Such a mapping has been defined by means of a set of ATL transformations, that we are currently implementing.

In addition to the analysis possibilities mentioned here, the use of rewriting logic and Maude opens up the way to using many other analysis tools for ATL transformations. In this respect, we are working on the use of the Maude Termination Tool (MTT) [19] and the Church-Rosser Checker (CRC) [20] for checking the termination and confluence of ATL specifications.

Finally, the formal analysis of the specifications needs to be done in Maude. At this moment we are also working on the integration of parts of the Maude toolkit within the ATL environment. This would allow ATL programmers to be able to conduct different kinds of analysis to the ATL model transformations they write, being unaware of the formal representation of their specifications in Maude.

Acknowledgements. The authors would like to thank Franciso Durán and José E. Rivera for their comments and suggestions on a previous version of the paper, and to the anonymous reviewers for their insightful comments and suggestions. This work has been partially supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184.

References

1. The AtlanMod Team: ATL (2010), `http://www.eclipse.org/m2m/atl/doc/`
2. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)

3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Eclipse (ATL), http://www.eclipse.org/m2m/atl/atlTransformations/
5. Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
6. Troya, J., Vallecillo, A.: Formal Semantics of ATL Using Rewriting Logic (Extended Version). Universidad de Málaga (2010), http://atenea.lcc.uma.es/Descargas/ATLinMaude.pdf
7. Bouhoula, A., Jouannaud, J.P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236(1), 35–132 (2000)
8. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
9. Rivera, J.E., Vallecillo, A., Durán, F.: Formal specification and analysis of domain specific languages using Maude. Simulation: Transactions of the Society for Modeling and Simulation International 85(11/12), 778–792 (2009)
10. Roldán, M., Durán, F.: Representing UML models in mOdCL. Manuscript (2008), http://maude.lcc.uma.es/mOdCL
11. di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique, Nantes, France (2006)
12. Baresi, L., Spoletini, P.: On the use of Alloy to analyze graph transformation systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 306–320. Springer, Heidelberg (2006)
13. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. In: Baudry, B., Faivre, A., Ghosh, S., Pretschner, A. (eds.) Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation, pp. 47–56 (2007)
14. Poernomo, I.: Proofs-as-model-transformations. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 214–228. Springer, Heidelberg (2008)
15. Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: Handbook of graph grammars and computing by graph transformation, foundations, vol. I, pp. 313–400 (1997)
16. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: language and environment. In: Handbook of graph grammars and computing by graph transformation: applications, languages, and tools, vol. 2, pp. 487–550 (1999)
17. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 18–33. Springer, Heidelberg (2009)
18. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
19. Durán, F., Lucas, S., Meseguer, J.: MTT: The Maude Termination Tool (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 313–319. Springer, Heidelberg (2008)
20. Durán, F., Meseguer, J.: A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In: Proc. of WRLA 2010. LNCS, Springer, Heidelberg (2010)