Towards a Rewriting Logic Semantics for ATL (Extended version)

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 model transformation 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.

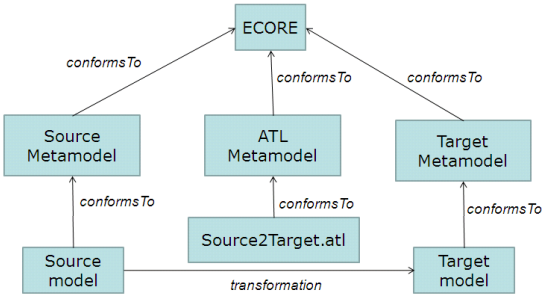


Fig. 1. ATL model transformation schema.

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

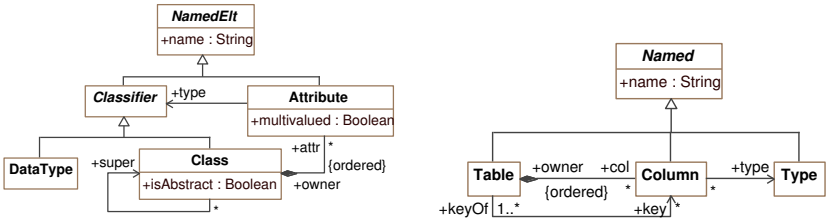
The structure of this document is as follows. After this introduction, sections 2 and 3 provide an introduction to ATL and Maude, respectively. Then, Sect. 4 presents how ATL language constructs can be encoded in Maude, and Sect. 5 describes the current tool support. Finally, Sect. 6 compares our work with other related proposals, and Sect. 7 draws some conclusions and outlines some future research activities.

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 (Fig. 1). 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 (see below). 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



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

Fig. 2. Transformation metamodels.

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 other ones 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.

The ATL Module data type also provides the **resolveTemp** operation for dealing with complex transformations. This specific operation makes it possible to point, from an ATL rule, to any of the target model elements (including non-default ones) that will be generated from a given source model element by an ATL matched rule. The operation resolveTemp has the following declaration: resolveTemp(var, target_pattern_name). The parameter var corresponds to an ATL variable that contains the source model element from which the searched target model element is produced. The parameter target_pattern_name is a string value that encodes the name of the target pattern element that maps the provided source model element (contained by var) into the searched target model element. This operation can be called from the target pattern and imperative sections of any matched or called rule.

In the rest of this section we will introduce the metamodels and models that will be used throughout the paper, as well as all the ATL transformations that will be later encoded in Maude.

2.1 Metamodels and models

In order to illustrate our proposal we present the metamodels that will be used for all our transformations. They are the ones used in 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. 2.

The class metamodel (Fig. 2(a)) consists of classes having a name which they inherit from the abstract class NamedElts. The principal class is the class Class, which contains a set of attributes of the type Attribute, and has the super references pointing to superclasses for modelling inheritance trees. The class DataType models primitive data types. Class and DataType inherit from Classifier, which serves to declare the type of Attributes. Attributes can be eather multivalued or not.

The relational metamodel (Fig. 2(b)), in turn, consists of classes having a name which they inherit from the abstract class Named. The principal class Table contains a set of Columns and has a reference to its keys. The class Column has the references owner and keyOf pointing to the Table it belongs to and of which it is part of the key (in case it is a key). Furthermore, Column has a reference to Type.

Our input model for all the transformations that will be presented contains two classes: Family and Person. The former class contains two attributes, one multivalued and one not; and the latter contains three attributes, one is multivalued and the other two are not. 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. Actually, it is the resulting model of the transformation shown in Sect. 2.2.

```
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;
}
```

2.2 Matched rules and helper

Here we present the transformation needed to get the target model presented in the previous subsection from the source model, as well as the ATL code of the transformation. It can be found in [4]. The header is:

```
module Class2Relation; -- Module Template
create OUT : RelationMM from IN : ClassMM;
```

This transformation is done by means of six matched rules, which involves declarative programming. The matched rules constitute the core of an ATL declarative transformation since they make it possible to specify 1) for which kinds of source elements target elements must be generated, and 2) the way the generated target elements have to be initialized. The transformations developed by each of these rules are explained one by one as follows.

– For each DataType instance, a Type instance has to be created.
  • Their names have to correspond.

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

– For each single-valued Attribute instance of the type DataType, a Column instance has to be created.
  • Their names and their types have to correspond.

```
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
      )
}
```

– For each multi-valued Attribute instance of the type DataType, a Table instance has to be created.
  • The Table's name is the name of the Attribute's Class concatenated with an underscore and the name of the Attribute.
  • The col reference set has to reference the two Columns described in the following.
  • An identifier Column instance has to be created.
    ∗ Its name has to be set to the Attribute's class name concatenated with "Id"
    ∗ Its type reference has to reference a Type with the name Integer.
  • A Column instance has to be created to contain the values of the Attribute.
    ∗ Their names and their types have to correspond.

```
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
        )
}
```

– For each single-valued Attribute of the type Class, a new Column has to be created.
  - Its name has to be set to the attribute's name concatenated with "id".
  - Its type reference has to reference a Type with the name Integer.

```
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
        )
}
```

– For each multi-valued Attribute of the type Class, a new Table has to be created.
  - The Table's name is the name of the Attribute's Class concatenated with an underscore and the name of the Attribute.
  - The col reference set has to reference the two Columns described in the following.
  - An identifier Column instance has to be created.
    * Its name has to be set to the Attribute's class name concatenated with "Id".
    * Its type reference has to reference a Type with the name Integer.
  - A foreign key Column instance has to be created.
    * Its name has to be set to the Attribute's name concatenated with "Id".
    * Its type reference has to reference a Type with the name Integer.

```
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 + 'Id',
             type <- thisModule.objectIdType
        ),
        k : Relational!Column (
            name <- at.name + 'Id',
            type <- thisModule.objectIdType
        )
}
```

– For each Class instance, a Table instance has to be created.
  - Their names have to correspond.
  - The col reference set has to contain all Columns that have been created for singlevalued attributes and also the key described in the following.

- The key reference set has to contain a pointer to the key described in the following.
- An Attribute instance has to be created as key.
  * Its name has to be set to ”objectId”
  * Its type reference has to reference a Type with the name Integer.

```
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
      )
}
```

Apart from these rules, a helper is needed. It is called objectIdType and it selects the first instance of the Type class of the relational model which is being constructed as result of the transformation whose name is “Integer”.

```
helper def: objectIdType : Relational!Type =
     Class!DataType.allInstances() ->select(e | e.name = 'Integer')->first();
```

The result of the whole transformation is the Relational model presented in Sect. 2.1, which is shown in Section 5.1 as a Maude model.

2.3 Lazy rules

By means of the transformations presented in this subsection we show how lazy rules work. They are like matched rules, but are only applied when called by another rule. The header of the first transformation is:

```
module Class2RelationalLazyRule; -- Module Template
create OUT : RelationalMM from IN : ClassMM;
```

The transformation, which contains a matched rule and a lazy rule called twice by the former, carries out the following actions:

– For each Attribute, a new Column is created.
  - The Column’s name is “First_col_of_” concatenated with the name of the Attribute.
  - The Column’s owner, i.e., the Table which will contain the Column created, is created by means of a lazy rule whose source element is the Attribute.
  - For each Attribute, a new Table is created by the lazy rule.
    * The Table’s name is “Table_of_” concatenated with the name of the Attribute.
    * The Table’s key is a new Column whose name is “Key_of_” concatenated with the name of the Attribute.
– For each Attribute, another new Column is created.

- The Column's name is "Second_col_of_" concatenated with the name of the Attribute.
- The Column's owner, i.e., the Table which will contain the Column created, is created by means of the same lazy rule that created the previous Table.
- For each Attribute, another new Table is created by the lazy rule.
  - The Table's name is "Table_of_" concatenated with the name of the Attribute.
  - The Table's key is a new Column whose name is "Key_of_" concatenated with the name of the Attribute.

```
rule CreateColumnsFromAttribute{
  from
      at : ClassMM!Attribute
  to
      co : RelationalMM!Column(
           name <- 'First_col_of_' + at.name,
           owner <- thisModule.createTable(at)
      ),
      c : RelationalMM!Column(
          name <- 'Second_col_of_' + at.name,
          owner <- thisModule.createTable(at)
      )
}
lazy rule createTable{
  from
      at : ClassMM!Attribute
  to
      tb : RelationalMM!Table(
           name <- 'Table_of_' + at.name,
           key <- k
      ),
      k : RelationalMM!Column(
          name <- 'Key_of_' + at.name,
          owner <- tb
      )
}
```

Please notice that the lazy rule is executed twice for each Attribute. Consequently, two Table and Column instances are created with the same name for each Attribute, "Table_of_..." and "Key_of_..." respectively.

The relational model created by this transformation from the source model shown in Sect. 2.1 will be shown in Sect. 5.1 as a Maude model.

Collect. Here we explain a special use of lazy rules called with a collect. It is used for calling lazy rules multiple times. The header of this transformation is:

```
module Class2RelationalLazyRuleCollect; -- Module Template
create OUT : RelationalMM from IN : ClassMM;
```

In this transformation there are a matched rule, a lazy rule called in the ordinary way, and another lazy rule called with a collect. The transformation develops these actions:

– For each Class, a new Column is created.
  - Their names have to correspond.
  - The Column's Type is created with a lazy rule which has the Class as source element and creates a Type whose name is "Column_type_of" concatenated with the name of the class.
  - The Column's owner is a Table described in the following.

- A Table instance has to be created as owner.
  - Its name is "Table_of_" concatenated with the name of the Class.
  - The col set is created by means of a lazy rule called with a collect whose parameters are the attributes of the Class.
  - For each Attribute, a new Column is created.
    - The Column's name is "Column_of_" concatenated with the name of the attribute.
    - The Column's type is a new Type whose name is "Type_of_" concatenated with the name of the Attribute.

```
rule CreateTableFromClass{
  from
      c : ClassMM!Class
  to
      out : RelationalMM!Column(
            name <- c.name,
            type <- thisModule.GetType(c),
            owner <- table
      ),
      table : RelationalMM!Table(
              name <- 'Table_of_' + c.name,
              col <- c.att->collect(e | thisModule.getColumns(e))
      )
}
lazy rule GetType{
  from
      c : ClassMM!Class
  to
      out : RelationalMM!Type(
            name <- 'Column_type_of_' + c.name
      )
}
lazy rule getColumns{
  from
      a : ClassMM!Attribute
  to
      out : RelationalMM!Column(
            name <- 'Column_of_' + a.name,
            type <- tp
      ),
    tp : RelationalMM!Type(
           name <- 'Type_of_' + a.name
    )
}
```

The relational model that results from this transformation with the source model shown in Sect. 2.1 will be shown in Sect. 5.1 as a Maude model.

2.4 Unique lazy rules

They are a special kind of lazy rules. When a unique lazy rule is executed, it always returns the same target element for a given source element. The target element is retrieved by navigating the internal traceability links, in a way similar to standard rules. Non-unique lazy rules do not navigate the traceability links, and create new target elements in each execution. The following transformation is the same as the first shown in the previous subsection, but now the lazy rule is a unique lazy rule. Its header is:

```
module Class2RelationalUniqueLazyRule; -- Module Template
create OUT : RelationalMM from IN : ClassMM;
```

The transformation carries out the following actions:

- For each Attribute, a new Column is created.
  - The Column's name is "First_col_of_" concatenated with the name of the Attribute.
  - The Column's owner, i.e., the Table which will contain the Column created, is created by means of a unique lazy rule whose source element is the Attribute.
  - For each Attribute, a new Table is created by the lazy rule.
    - The Table's name is "Table_of_" concatenated with the name of the Attribute.
    - The Table's key is a new Column whose name is "Key_of_" concatenated with the name of the Attribute.
- For each Attribute, another new Column is created.
  - The Column's name is "Second_col_of_" concatenated with the name of the Attribute.
  - The Column's owner, i.e., the Table which will contain the Column created, is created by means of the same unique lazy rule that created the previous Table. Consequently, as the Table has been already created, it is not created again, but it is just referenced by this Column's owner.

```
rule CreateColumnsFromAttribute{
  from
      at : ClassMM!Attribute
  to
      co : RelationalMM!Column(
           name <- 'First_col_of_' + at.name,
           owner <- thisModule.createTable(at)
      ),
      c : RelationalMM!Column(
          name <- 'Second_col_of_' + at.name,
          owner <- thisModule.createTable(at)
      )
}
unique lazy rule createTable{
  from
      at : ClassMM!Attribute
  to
      tb : RelationalMM!Table(
           name <- 'Table_of_' + at.name,
           key <- k
      ),
      k : RelationalMM!Column(
          name <- 'Key_of_' + at.name,
          owner <- tb
      )
}
```

The result of the execution of this transformation will be shown in Sect 5.1 as a Maude model.

2.5 The imperative section

ATL enables developers to specify imperative code within dedicated blocks, either in matched or called rules. An imperative block is composed of a sequence of imperative statements. As in the Java C or C++ languages, each statement must be ended with

a semicolon character. The ATL implementation described in this paper provides four kinds of imperative statements: *assignments* (=), *conditional* branches (if), *loops* (for) and *called rules*. All of them will be treated in this paper.

The assignment statement The ATL assignment statement enables to assign values to either attributes that are defined in the context of the ATL module, or to target model element features. The syntax of the assignment statement conforms to the following scheme: target $<-$ exp;. Let us introduce an assignment statement in the DataType2Type matched rule presented in the example of Sect. 2.2:

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

The rule's imperative block concatenates the name of the Type instance created in the declarative section from the DataType instance with "_assignment". Note that in this example the Type's name is initialized in the declarative part and then it is modified in the imperative part.

The if statement The "if" statement enables to define alternative imperative treatments. Each "if" statement defines a condition. This condition must be an OCL expression that returns a boolean value. An "if" statement must also include a "then" statements section. This section, specified between curved brackets, contains the sequence of statements that is executed when the conditional expression is evaluated to true. An "if" statement may also include an optional "else" statements section. When specified, this section has to follow the "then" statements section. It is introduced by the keyword else, and must be also defined between curved brackets. This section contains the optional sequence of statements that has to be executed when the conditional expression is evaluated to false.

Here we add an imperative block to the MultiValuedClassAttribute2Column matched rule presented in the example of Sect. 2.2:

```
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 + 'Id',
           type <- thisModule.objectIdType
      ),
      k : Relational!Column (
          name <- at.name + 'Id',
          type <- thisModule.objectIdType
```

```
    )
  do{
      tb.name <- tb.name + '_Multi';
      if (tb.col->size() = 3){
          k.name <- 'key';
      }else{
          k.name <- 'key_else';
      }
  }
}
```

The imperative section contains an assignment statement followed by an "if" statement which also contains an "else" section. This imperative section concatenates the name of the Table instance created in the declarative part with "_Multi". The "if" section sets the name of the second Column to "key" if the Table contains three Column instances or to "key_else" if it does not contain three Columns. Since the Table contains two Columns as specified in the declarative part, the name of the second Column will be set to "key_else".

The for statement The "for" statement enables to define iterative imperative computations. The "for" statement defines an iteration variable (iterator) that will iterate over the different elements of the reference collection. For each of these elements, the sequence of statements contained by the "for" statement will be executed. Let us extend the imperative section of the rule MultiValuedClassAttribute2Column shown before to introduce a "for" statement which contains some imperative instructions:

```
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 + 'Id',
           type <- thisModule.objectIdType
      ),
      k : Relational!Column (
          name <- at.name + 'Id',
          type <- thisModule.objectIdType
      )
  do{
      tb.name <- tb.name + '_Multi';
      if (tb.col->size() = 3){
          k.name <- 'key';
      }else{
          k.name <- 'key_else';
      }
      for(c in out.col){
          c.name <- c.name + '_assign1';
          c.name <- c.name + '_assign2';
          if (c.name = 'key_else_assign1_assign2'){
            c.name <- c.name + '_ifYES';
          }else{
            c.name <- c.name + '_ifNO';
          }
          c.name <- c.name + '_afterIF';
      }
```

```
  }
}
```

The sequence of instructions added within the "for" statement are applied to every Column of the Table created by the rule. First of all, two assignments are made, where the name of the Column instances are concatenated with "_assign1" and "_assign2". After that, an "if" section is introduced. This section concatenates the name of each Column with "_ifYES" if the name of the Column was "key_else_assign1_assign2" or with "_ifNO" if it was not. Finally, another assignment is applied, where the name of the Columns are concatenated with "_afterIf".

Called Rules Called rules provide ATL developers with convenient imperative programming facilities. In some way, called rules can be seen as a particular type of helpers since they have to be explicitly called to be executed and they can accept parameters. However, as opposed to helpers, called rules can generate target model elements as matched rules do. A called rule has to be called from an imperative code section, either from a match rule or another called rule.

In the example below we add a called rule in the imperative section of the rule SingleValuedDataTypeAttribute2Column.

```
rule SingleValuedDataTypeAttribute2Column {
  from
      a : ClassMM!Attribute (
          a.type.oclIsKindOf(ClassMM!DataType) and not a.multivalued
      )
  to
      out : RelationMM!Column (
            name <- a.name,
            type <- a.type
      )
  do{
      thisModule.AddColumn('New_Column');
  }
}
--Called Rule
rule AddColumn(n : String){
  to
      out : RelationMM!Column(
            name <- n
      )
}
```

Now, the SingleValuedDataTypeAttribute2Column rule, apart from creating a new Column instance for each Attribute in the declarative part, executes a called rule, AddColumn, in the imperative part, with the string "New_Column" as argument. The called rule creates another new Column whose name is the string passed as parameter.

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 [6] and rewriting logic [2] specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. Because of its efficient rewriting engine, able to execute more than 3 million rewriting steps per

second on standard PCs, and because of its metalanguage capabilities, Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, or even programming languages. 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 [6], 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 => 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'.

For instance, the following Maude module, ACCOUNT, specifies a class Account with an attribute balance of sort integer (Int), and other class Withdraw (that models the action of withdrawing) with an object identifier (of sort Oid) and an integer as attributes, and a rule describing the behavior of the objects belonging to these classes. The rule debit specifies a local transition of the system when there is an object A of class Account and a Withdraw object requesting to withdraw an amount smaller or equal than the balance of A; as a result of the application of such a rule, the object representing the action is consumed, and the balance of the account is modified.

```
(omod ACCOUNT is
  protecting INT .
  class Account | balance : Int .
  class Withdraw | acc : Oid, amount : Int .
  vars A W : Oid .
  vars M Bal  : Int .
  crl [debit] :
    < W : Withdraw | acc : A, amount : M >
    < A : Account  | balance : Bal >
    => < A : Account | balance : Bal - M >
     if M <= Bal .
endom)
```

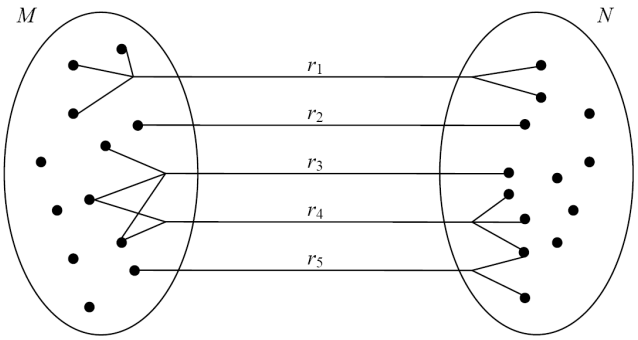


Fig. 3. Elements of a relation $R(M, N)$.

4 Encoding ATL into 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 (assignment statements, if statements, and called rules). 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). The interested reader could find all the examples that are shown in the following in [7], as well as a short description of which transformation is in each file.

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 from the source 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 (see Fig. 3).

Note that this approach works not only for providing structural semantics to ATL, but to any model transformation language. In fact, the set $R(M, N) = \{r_1, ..., r_k\}$ is nothing but the set of all *trace instances* of the transformation.

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.

The rest of this section describes how both the structural and behavioral aspects of ATL transformations can be encoded in Maude.

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 above. 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 that constitute the model. An object is a record-like structure of the form $< o : c \mid a_1 : v_1, ..., a_n : v_n >$ (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. The following Maude specification describes a fragment of the class metamodel shown in Fig. 2(a).

```
mod @CLASSMM@ is
  protecting @ECORE@ .
  op @ClassMm@ : -> @Metamodel .
  op ClassMm : -> @Package .

  sort DataType@ClassMm .
  subsort DataType@ClassMm < Classifier@ClassMm .
  op DataType@ClassMm : -> DataType@ClassMm .
  ...

  sort Class@ClassMm .
  subsort Class@ClassMm < Classifier@ClassMm .
  op Class@ClassMm : -> Class@ClassMm .
  op super@Class@ClassMm : -> @Reference .
  op att@Class@ClassMm : -> @Reference .
  op isAbstract@Class@ClassMm : -> @Attribute .
  ...
endm
```

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. These qualifications are omitted here to improve readability. See [9] for further details.

4.3 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 contains the three models that capture the state of the transformation (see 4.1): the source, the trace and the target models. 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.

In the following we will present how to translate the rules presented in Sect. 2.2 in Maude. We will do it rule by rule. We also use some auxiliary functions that we will show as well. We start by showing the header of the Maude file, which is:

```
mod @CLASS2RELATION@ is

  protecting @CLASSMM@ .
  protecting @TRACEMMN@ .
  protecting @RELATIONALMM@ .
  protecting @FUNCTIONS4ATL@ .

  op ITER : -> Vid .
  vars DT@ T@ TC@ AT@ C@ CO@ COO@ TR@ CNT@ TI@ TB@ ID@ K@ : Oid .
  var LO : ListOrd .
  var SFS : Set{@StructuralFeatureInstance} .
  var OBJSET@ OBJSETT@ OBJSETTT@ : Set{@Object} .
  var SEQ : Sequence .
  var VALUE@CNT@ : Int .
  var CLASSMODEL@ : @Model .
```

Here, @CLASSMM@ is the source metamodel (class metamodel) of the transformation, @RELATIONALMM@ is the target metamodel (relational metamodel), @TRACEMM@ is the aditional metamodel for keeping all the traces of the transformation, and FUNCTIONS4ATL contains the helper and some additional functions that will be shown later. The rest are variables that will be used throughout the whole transformation.

The first rule, DataType2Type, is specified 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@}) ;
 (@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 should be applied over DataType instances. This is specified by requiring that in the lefthand side of the rule there must be an instance of type DataType@ClassMm. The trace model has to contain an element of type Counter and it does not matter what the relational model contains in the lefthand side.

The class model does not change in the righthand side of the rule, but the other two models need to change. This way, a new instance of the class Type has to be added to the relational model. The name of the new instance is the same as the name of the DataType instance. We allow the evaluation of OCL expressions using mOdCL [10] by enclosing them in double angle brackets (<< ... >>). Thus, the sentence << DT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> gets the name of the DataType instance, since CLASSMODEL@ represents the class model as specified in the first condition. A new trace element is added to the trace model having a sequence with the DataType instance as source element and a sequence with the Type instance as target element. The value of the counter is increased in the righthand side with the number of elements created, since it needs to be used in the following rule that will be applied. This counter is used in the conditions to give new identifiers to the new elements created (the trace and Type instances in this case) with the function newId. It is found in FUNCTIONS4ATL:

```
op newId : Int -> Qid .
  eq newId(I:Int) = oid(I:Int) .
```

Here, Qid is the type that represents the identifiers of objects. This function uses another one called oid which is shown here:

```
op oid : Int -> Qid .
  eq oid(I:Int) = qid(string(I:Int,10)) .
```

The Qid created by this function is a string with contains the caracter ' concatenated with the Integer passed as argument.

The last condition of the rule, that uses the function alreadyExecuted, ensures that this rule will not be applied again with the DataType instance of this rule. This function is also found in FUNCTIONS4ATL and is the next:

```
op alreadyExecuted : Sequence String @Model -> Bool .
  eq alreadyExecuted(SEQ, NAME, @TraceMm@ { < TR@ : Trace@TraceMm |
    srcEl@TraceMm : SEQ # rlName@TraceMm : NAME # SFS > OBJSET }) = true .
  eq alreadyExecuted(SEQ, NAME, @TraceMm@ { OBJSET }) = false [owise] .
```

The function returns true if there exists a trace in the trace model whose source element is the sequence passed as argument and the name passed is the name of the rule. Otherwise, it returns false.

The rest of the rules are not very different to this one. Therefore, only the new elements that appear in them will be explained.

The rule SingleValuedDataTypeAttribute2Column is:

```
crl[SingleValuedDataTypeAttribute2Column] :
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ }) ;
  (@RelationalMm@ {OBJSETTT@})
  ]
  =>
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 2 >
  < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ AT@ ]# trgEl@TraceMm :
        Sequence[ CO@ ]# rlName@TraceMm : "SingleValuedDataTypeAttribute2Column"
        # srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" >
  OBJSETT@ }) ;
  (@RelationalMm@ {
  < CO@ : Column@RelationalMm | name@Named@RelationalMm :
        << AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> #
        type@Column@RelationalMm : getTarget(<< AT@ .
        type@Attribute@ClassMm ; CLASSMODEL@ >>, @TraceMm@{ OBJSETT@ } ) >
  OBJSETTT@ })
  ]
  if
  CLASSMODEL@ := @ClassMm@ {
    < AT@ : Attribute@ClassMm | SFS >
    OBJSET@ } /\
  TR@ := newId(VALUE@CNT@) /\ CO@ := newId(VALUE@CNT@ + 1) /\
  not << AT@ . multivalued@Attribute@ClassMm ; CLASSMODEL@ >> /\
  not (getTarget(<< AT@ . type@Attribute@ClassMm ; CLASSMODEL@ >>,
        @TraceMm@{OBJSETT@}) == mt-ord) /\
  << AT@ . type@Attribute@ClassMm . oclIsKindOf(DataType@ClassMm) ;
        CLASSMODEL@ >> /\
  not alreadyExecuted(Sequence[ AT@ ], "SingleValuedDataTypeAttribute2Column",
        @TraceMm@ { OBJSETT@ } )
.
```

This rule is applied over Attribute instances of type DataType which are not multivalued. The first restriction is applied thanks to the condition not << AT@ . multivalued@Attribute@ClassMm ; CLASSMODEL@ >>. The second one is achieved with the condition << AT@ . type@Attribute@ClassMm . oclIsKindOf(DataType@ClassMm) ; CLASSMODEL@ >>, which applies the OCL function oclIsKindOf over the Attribute instance. When this rule is applied over an Attribute instance, a new Column instance is added to the relational model with the name of the Attribute. The Type of the Column created has to be the same as the Type created from the DataType of the Attribute. There-

fore, this Type has to be retrieved from the trace model, as the Type has to have been created by a previous rule, otherwise this rule is not applied. This way, the Type instance is retrieved with the function *getTarget*: getTarget(<< AT@ . type@Attribute@ClassMm ; CLASSMODEL@ >>, @TraceMm@ OBJSETT@). This function can be found in FUNCTIONS4ATL:

```
op getTarget : Sequence @Model -> Sequence .
  eq getTarget(Sequence[TR@ ; LO], @TraceMm@{OBJSET}) = << Sequence[ getTargEl(
  TR@,@TraceMm@{OBJSET}) ] -> union(getTarget(Sequence[LO],@TraceMm@{OBJSET}))>> .
  eq getTarget(TR@, @TraceMm@ {OBJSET} ) = getTargEl(TR@, @TraceMm@{OBJSET}) .
  eq getTarget(Sequence[mt-ord], @TraceMm@ { OBJSET }) = Sequence[mt-ord] .
```

This function receives a sequence and a model (the trace model) and returns a sequence. It takes the first element of the sequence, applies the function getTargEl over it and concatenates the result with the sequence that results of applying the function getTarget over the rest of elements of the sequence. If the sequence only contains an element, it applies the function getTargEl over it. This function is:

```
op getTargEl : OCL-Exp @Model -> OCL-Exp .
  eq getTargEl(SRC@, @TraceMm@ { < TR@ : Trace@TraceMm | srcEl@TraceMm :
    Sequence[SRC@;LO] # trgEl@TraceMm : Sequence[TRG@ ; LO]# SFS> OBJSET}) = TRG@.
  eq getTargEl(SRC@, @TraceMm@ { OBJSET }) = mt-ord [owise] .
```

It receives an OCL-Exp (which is an instance of any class) and the trace model, and it returns the target element. The function returns the element created from the element passed as argument by a rule previously executed by looking for the element in the trace model. If it exists a trace whose first element of the sequence that contains the source elements is the element passed as argument, it returns the first element of the sequence that contains the target elements. Otherwise, it returns nothing.

Since the rule must not be applied if the function getTarget does not return an element, the following condition is required: not (getTarget(<< AT@ . type@Attribute@ClassMm ; CLASSMODEL@ >>, @TraceMm@OBJSETT@) == mt-ord). This way, the rule that creates the Type instance from the DataType of the Attribute and adds it to the trace model has to be executed before this one.

The following is the Maude representation of the rule MultivaluedDataTypeAttribute2Column:

```
crl[MultivaluedDataTypeAttribute2Column] :
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
    OBJSET@ } ) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ } ) ;
  (@RelationalMm@ { OBJSETTT@ } )
  ]
  =>
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
    OBJSET@ } ) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 4 >
  < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ AT@ ]# trgEl@TraceMm :
      Sequence[ TB@ ; CO@ ; COO@ ]# rlName@TraceMm :
      "MultivaluedDataTypeAttribute2Column" # srcMdl@TraceMm : "Class"
      # trgMdl@TraceMm : "Relation" >
  OBJSETT@ } ) ;
  (@RelationalMm@ {
```

```
  < TB@ : Table@RelationalMm | name@Named@RelationalMm : << AT@ .
      owner@Attribute@ClassMm . name@NamedElt@ClassMm + "_" + AT@ .
      name@NamedElt@ClassMm ; CLASSMODEL@ >> #
    col@Table@RelationalMm : Sequence[ CO@ ; COO@ ] >
  < CO@ : Column@RelationalMm | name@Named@RelationalMm : << AT@ .
      owner@Attribute@ClassMm . name@NamedElt@ClassMm + "Id" ; CLASSMODEL@ >> #
    type@Column@RelationalMm : objectIdType(CLASSMODEL@, @TraceMm@{OBJSETT@}) >
  < COO@ : Column@RelationalMm | name@Named@RelationalMm : << AT@ .
      name@NamedElt@ClassMm ; CLASSMODEL@ >> # type@Column@RelationalMm :
      getTarget(<< AT@ . type@Attribute@ClassMm ; CLASSMODEL@ >>,
      @TraceMm@{OBJSETT@}) >
  OBJSETTT@ } )
  ]
  if
  CLASSMODEL@ := @ClassMm@ {
    < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ } /\
  TR@ := newId(VALUE@CNT@) /\ TB@ := newId(VALUE@CNT@ + 1) /\
    CO@ := newId(VALUE@CNT@ + 2) /\ COO@ := newId(VALUE@CNT@ + 3) /\
  << AT@ . multivalued@Attribute@ClassMm ; CLASSMODEL@ >> /\
  not (getTarget(<< AT@ . type@Attribute@ClassMm ; CLASSMODEL@ >>,
    @TraceMm@{OBJSETT@}) == mt-ord) /\
  << AT@ . type@Attribute@ClassMm . oclIsKindOf(DataType@ClassMm) ;
    CLASSMODEL@ >> /\
  not alreadyExecuted(Sequence[ AT@ ], "MultivaluedDataTypeAttribute2Column",
    @TraceMm@ { OBJSETT@ } )
.
```

This rule is similar to the previous one. However, it introduces two novelties. The first one is that there are now three new instances created in the relational model. This way, a new Table instance is created, and it contains two new Column instances. Consequently, these three instances have to be stored in a trace. Specifically, they are stored in the sequence that contains the elements created (trgEl@TraceMm). The second one is that it uses the helper to get the Type of a Column. This helper is shown in the next subsection.

The following rule is SingleValuedClassAttribute2Column:

```
crl[SingleValuedClassAttribute2Column] :
  Sequence
  [
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ });
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@});
  (@RelationalMm@ {OBJSETTT@ })
  ]
  =>
  Sequence
  [
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ });
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 2 >
  < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[AT@]# trgEl@TraceMm :
      Sequence[CO@]# rlName@TraceMm : "ClassAttribute2Column" # srcMdl@TraceMm :
      "Class" # trgMdl@TraceMm : "Relation" >
  OBJSETT@ });
  (@RelationalMm@ {
  < CO@ : Column@RelationalMm | name@Named@RelationalMm : << AT@ .
      name@NamedElt@ClassMm + "Id" ; CLASSMODEL@ >> # type@Column@RelationalMm
      : objectIdType(CLASSMODEL@, @TraceMm@{OBJSETT@}) >
  OBJSETTT@ })
```

```
  ]
  if CLASSMODEL@ := @ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ } /\
  TR@ := newId(VALUE@CNT@) /\ CO@ := newId(VALUE@CNT@ + 1) /\
  << AT@ . type@Attribute@ClassMm . oclIsKindOf(Class@ClassMm) ; CLASSMODEL@ >> /\
  not << AT@ . multivalued@Attribute@ClassMm ; CLASSMODEL@ >> /\
  not alreadyExecuted(Sequence[AT@], "ClassAttribute2Column", @TraceMm@{OBJSETT@})
  .
```

This rule will not be explained as it is analogous to the previous ones.

The next rule is MultiValuedClassAttribute2Column:

```
crl[MultiValuedClassAttribute2Column] :
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ });
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ });
  (@RelationalMm@ { OBJSETTT@ })
  ]
  =>
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ });
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 4 >
  < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[AT@]# trgEl@TraceMm :
      Sequence[TB@ ; ID@ ; K@]# rlName@TraceMm: "MultiValuedClassAttribute2Column"
      # srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" >
  OBJSETT@ });
  (@RelationalMm@ {
  < 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@}) >
  OBJSETTT@ })
  ]
  if CLASSMODEL@ := @ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ } /\
  TR@ := newId(VALUE@CNT@) /\ TB@ := newId(VALUE@CNT@ + 1) /\
  ID@ := newId(VALUE@CNT@ + 2) /\ K@ := newId(VALUE@CNT@ + 3) /\
  << AT@ . type@Attribute@ClassMm . oclIsKindOf(Class@ClassMm) ; CLASSMODEL@ >> /\
  << AT@ . multivalued@Attribute@ClassMm ; CLASSMODEL@ >> /\
  not alreadyExecuted(Sequence[AT@], "MultiValuedClassAttribute2Column", @TraceMm@
  { OBJSETT@ })
.
```

The last rule, Class2Table, is more complex:

```
crl [Class2Table] :
  Sequence[
  (@ClassMm@ {
  < C@ : Class@ClassMm | SFS >
  OBJSET@ } ) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ } ) ;
```

```
 (@RelationalMm@ { OBJSETTT@ } )
 ]
 =>
 Sequence[
 (@ClassMm@ {
 < C@ : Class@ClassMm | SFS >
 OBJSET@ } ) ;
 (@TraceMm@ {
 < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 3 >
 < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ C@ ]# trgEl@TraceMm :
     Sequence[ TB@ ; K@ ]# rlName@TraceMm : "Class2Table" # srcMdl@TraceMm :
     "Class" # trgMdl@TraceMm : "Relation" >
 OBJSETT@ } ) ;
 (@RelationalMm@ {
 < TB@ : Table@RelationalMm | name@Named@RelationalMm : << C@ .
     name@NamedElt@ClassMm ; CLASSMODEL@ >> #  col@Table@RelationalMm : <<
     Sequence[K@] -> union(getTarget(<< C@ . att@Class@ClassMm -> asSequence()
   -> select(ITER | not ITER . multivalued@Attribute@ClassMm) ; CLASSMODEL@ >> ,
     @TraceMm@{OBJSETT@})) ; CLASSMODEL@ >> #  key@Table@RelationalMm : Set{K@} >
 < K@ : Column@RelationalMm | name@Named@RelationalMm : "objectId" #
     type@Column@RelationalMm : objectIdType(CLASSMODEL@, @TraceMm@{OBJSETT@}) >
 OBJSETTT@ } )
 ]
 if CLASSMODEL@ := @ClassMm@ {
 < C@ : Class@ClassMm | SFS >
 OBJSET@ } /\
 TR@ := newId(VALUE@CNT@) /\ TB@ := newId(VALUE@CNT@ + 1) /\
 K@ := newId(VALUE@CNT@ + 2) /\
 not alreadyExecuted(Sequence[ C@ ], "Class2Table", @TraceMm@ { OBJSETT@ } ) /\
 << getTarget(<< C@ . att@Class@ClassMm->asSequence() -> select(ITER | not ITER .
   multivalued@Attribute@ClassMm) ; CLASSMODEL@ >>,@TraceMm@{OBJSETT@}) -> size()
   ; CLASSMODEL@ >> == << C@ . att@Class@ClassMm -> asSequence() -> select(ITER |
   not ITER . multivalued@Attribute@ClassMm) -> size() ; CLASSMODEL@ >>
.
```

This rule has to retrieve the Column instances that have been created from the Attribute instances of the Class which is in the lefthand side. It passes a sequence to the getTarget function, while the rules shown previously always passed only an element. The rule uses the select command to pass only the Attribute instances which are not multivalued. So, for each non-multivalued attribute, the function has to retrieve a column. It means that the rules that create these columns have to have been executed before. To ensure it, there is a condition which checks that the size of the sequence passed as argument to the function getTarget is the same as the size of the sequence returned by the function.

4.4 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.

4.5 Lazy rules

While matched rules are executed in non-deterministic order (as soon as their "to:" patterns are matched in the 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 lazy 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.

The first example shown in Sect. 2.3 is encoded in Maude as follows:

```
mod @Class2RelationalLazyRule@ is
  protecting @CLASSMM@ .
  ...
  vars T@ TC@ AT@ C@ CO@ TR@ CNT@ : Oid .
  var SFS : Set{@StructuralFeatureInstance} .
  var OBJSET@ OBJSETT@ OBJSETTT@ : Set{@Object} .
  var VALUE@CNT@ : Int .
  var CLASSMODEL@ : @Model .

---------------------- Lazy rule createTable ------------------------------
  op createTable : Oid @Model Int -> Set{@Object} .
  eq createTable(AT@, CLASSMODEL@, VALUE@CNT@) =  < newId(VALUE@CNT@) :
      Table@RelationalMm | name@Named@RelationalMm : << "Table_of_" + AT@ .
      name@NamedElt@ClassMm ; CLASSMODEL@ >> # key@Table@RelationalMm :
      newId(VALUE@CNT@ + 1) >
    < newId(VALUE@CNT@ + 1) : Column@RelationalMm | name@Named@RelationalMm :
      << "Key_of_" + AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> #
      owner@Column@RelationalMm : newId(VALUE@CNT@) >
  .
-------------------------------------------------------------------------
rl [Init] :
  ...

crl [CreateColumnsFromAttribute] :
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ }) ;
  (@RelationalMm@ {OBJSETTT@})
  ]
  =>
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 9 >
    --- Trace for the matched rule
  < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ AT@ ]# trgEl@TraceMm :
      Sequence[ CO@ ; C@]# rlName@TraceMm : "CreateColumnsFromAttribute" #
      srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" >
    --- Traces for the lazy rule. There are two of them because the lazy
    --- rule is called twice
  < T@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ AT@ ]# trgEl@TraceMm :
      Sequence [ newId(VALUE@CNT@) ; newId(VALUE@CNT@+1) ]# rlName@TraceMm :
```

```
      "CreateColumnFromAttribute_CreateTable" # srcMdl@TraceMm : "Class"
      # trgMdl@TraceMm : "Relation" >
  < TC@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ AT@ ]# trgEl@TraceMm :
      Sequence [ newId(VALUE@CNT@+2);newId(VALUE@CNT@+3) ]# rlName@TraceMm :
      "CreateColumnFromAttribute_CreateTable" # srcMdl@TraceMm : "Class" #
      trgMdl@TraceMm : "Relation" >
  OBJSETT@ }) ;
  (@RelationalMm@ {
  < CO@ : Column@RelationalMm | name@Named@RelationalMm : << "First_col_of_"
      + AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> # owner@Column@RelationalMm :
      newId(VALUE@CNT@) >
  createTable(AT@, CLASSMODEL@, VALUE@CNT@)
  < C@ : Column@RelationalMm | name@Named@RelationalMm : << "Second_col_of_" +
      AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> # owner@Column@RelationalMm :
      newId(VALUE@CNT@ + 2) >
  createTable(AT@, CLASSMODEL@, VALUE@CNT@ + 2)
  OBJSETTT@ })
  ]
  if
  CLASSMODEL@ := @ClassMm@ {
    < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ } /\
  CO@ := newId(VALUE@CNT@ + 4) /\ C@ := newId(VALUE@CNT@ + 5) /\
  TR@ := newId(VALUE@CNT@ + 6) /\ T@ := newId(VALUE@CNT@ + 7) /\
  TC@ := newId(VALUE@CNT@ + 8) /\
  not alreadyExecuted(Sequence[ AT@ ], "CreateColumnsFromAttribute",
    @TraceMm@ { OBJSETT@ } )
.
endm
```

The lazy rule is a function which receives as arguments the Attribute, the class model and the counter to create identifiers for the objects created by the lazy rule. The identifier of the first object created by the lazy rule is created with the counter passed as argument. The identifiers of the rest of the objects are created by adding units to the counter.

The call to the lazy rule is implemented quite different than in ATL. Thus, the first Column instance, created by the matched rule, is created by adding an object to the relational model, as we did in the example before. Regarding its owner, a Table which is created by the lazy rule, it references the identifier that will represent the Table. The same happens for the second Column. Regarding the lazy rule, the objects it creates have to be added to the relational model. To do that, it is only necessary to call the lazy rule as many times as it is called in the matched rule. The call to the lazy rule is written in the relational model, so that the result of the lazy rule is automatically added in this model. The rule is called twice. The difference between both calls is the value of the counter, since each created object has to have a different identifier.

Three new trace elements have been added to the trace model. The purpose of the first one is to have a trace for the execution of the matched rule, where the source element is the Attribute instance and the target elements are the two Column instances created by the rule. The other two trace elements are added to keep track of the lazy rules executed in case these traces need to be used by other rules. In these two traces, the name of the rule executed is the name of the matched rule concatenated with "_" and the name of the lazy rule, since lazy rules can be executed by different rules and this way we know which rule called the lazy one. The source element of these traces is the Attribute instance and the target elements are the elements created by the lazy rule, i.e., the identifiers of these objects.

Collect. Here we present how the calls to lazy rules by a collect are implemented in Maude. The example shown in Sect. 2.3, which contains a matched rule, a lazy rule called in the ordinary way, and another lazy rule called with a collect, is encoded in Maude as follows:

```
mod @Class2RelationalLazyRuleCollect@ is
  protecting @CLASSMM@ .
  ...
  vars T@ TC@ C@ CO@ TR@ CNT@ TB@ : Oid .
  var SFS  : Set{@StructuralFeatureInstance} .
  var OBJSET@ OBJSETT@ OBJSETTT@ : Set{@Object} .
  vars VALUE@CNT@ OBJS@CREATED@ I@ : Int .
  var CLASSMODEL@ : @Model .
  var LO : ListOrd .

---------------------- Lazy rule getType -------------------------------------
  op getType : Oid @Model Int -> Set{@Object} .
  eq getType(C@, CLASSMODEL@, VALUE@CNT@) =
    < newId(VALUE@CNT@) : Type@RelationalMm | name@Named@RelationalMm :
        << "Column_type_of_" + C@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> > .
------------------------------------------------------------------------------
---------------------- Lazy rule getColumns ----------------------------------
  op getColumnsCollect : Sequence @Model Int -> Set{@Object} .
    eq getColumnsCollect(Sequence[AT@ ; LO], CLASSMODEL@, VALUE@CNT@) =
    < newId(VALUE@CNT@) : Column@RelationalMm | name@Named@RelationalMm :
        << "Column_of_" + AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> #
      type@Column@RelationalMm : newId(VALUE@CNT@ + 1) >
    < newId(VALUE@CNT@ + 1) : Type@RelationalMm | name@Named@RelationalMm :
        << "Type_of_" + AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> >
    getColumnsCollect(Sequence[LO], CLASSMODEL@, VALUE@CNT@ + 2) .
  eq getColumnsCollect(Sequence[mt-ord], CLASSMODEL@, VALUE@CNT@) = none [owise] .
------------------------------------------------------------------------------
------------ Function that gets the Oid of the objects created by a ----------
----------------------------lazy rule called by Collect  ----------------------
  op getOidsCollect : Sequence Int Int -> Sequence .
    eq getOidsCollect(Sequence[AT@ ; LO], VALUE@CNT@, OBJS@CREATED@) =
      << Sequence [newId(VALUE@CNT@)] -> union (getOidsCollect(Sequence[LO],
      VALUE@CNT@ + OBJS@CREATED@, OBJS@CREATED@)) >> .
    eq getOidsCollect(Sequence[mt-ord],VALUE@CNT@,OBJS@CREATED@)=Sequence[mt-ord].
------------------------------------------------------------------------------
rl [Init] :
  ...

crl [CreateColumnFromAttribute] :
  Sequence[
  (@ClassMm@ {
  < C@ : Class@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ }) ;
  (@RelationalMm@ {OBJSETTT@})
  ]
  =>
  Sequence[
  (@ClassMm@ {
  < C@ : Class@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + I@ + 6 >
  < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ C@ ]# trgEl@TraceMm :
      Sequence[ CO@ ]# rlName@TraceMm : "CreateColumnFromAttribute" #
      srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" >
  --- Traza aadida por parte del getType :
  < T@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ C@ ]# trgEl@TraceMm :
      Sequence [ newId(VALUE@CNT@) ]# rlName@TraceMm :
      "CreateColumnFromAttribute_GetType" # srcMdl@TraceMm : "Class" #
```

```
    trgMdl@TraceMm : "Relation" >
  --- Traza aadida por parte del getColumnsCollect
  < TC@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ C@ ]# trgEl@TraceMm :
    Sequence [ getOidsCollect(<< C@ . att@Class@ClassMm ; CLASSMODEL@ >>,
    VALUE@CNT@ + 1, 2) ]# rlName@TraceMm :
    "CreateColumnFromAttribute_GgetColumnsCollect" # srcMdl@TraceMm : "Class"
    # trgMdl@TraceMm : "Relation" >
  OBJSETT@ }) ;
  (@RelationalMm@ {
  < CO@ : Column@RelationalMm | name@Named@RelationalMm : << C@ .
    name@NamedElt@ClassMm ; CLASSMODEL@ >> # type@Column@RelationalMm :
    newId(VALUE@CNT@) # owner@Column@RelationalMm : TB@ >
  getType(C@, CLASSMODEL@, VALUE@CNT@)
  < TB@ : Table@RelationalMm | name@Named@RelationalMm : << "Table_of_" + C@ .
    name@NamedElt@ClassMm ; CLASSMODEL@ >> # col@Table@RelationalMm :
    getOidsCollect(<< C@ . att@Class@ClassMm ;CLASSMODEL@ >>, VALUE@CNT@+1, 2)>
  getColumnsCollect(<< C@ . att@Class@ClassMm ; CLASSMODEL@ >>, CLASSMODEL@,
    VALUE@CNT@ + 1)
  OBJSETTT@ })
  ]
  if
  CLASSMODEL@ := @ClassMm@ {
    < C@ : Class@ClassMm | SFS >
  OBJSET@ } /\
  I@ := 1 + (<< C@ . att@Class@ClassMm -> size() ; CLASSMODEL@ >> + << C@ .
    att@Class@ClassMm -> size() ; CLASSMODEL@ >>) /\
  TR@ := newId(VALUE@CNT@ + I@ + 1) /\ T@ := newId(VALUE@CNT@ + I@ + 2) /\
  TC@ := newId(VALUE@CNT@ + I@ + 3) /\
  CO@ := newId(VALUE@CNT@ + I@ + 4) /\  TB@ := newId(VALUE@CNT@ + I@ + 5) /\
  not alreadyExecuted(Sequence[ C@ ], "CreateColumnFromAttribute", @TraceMm@
    { OBJSETT@ } )
.
endm
```

The lazy rule getType is similar to the one of the example before, but now it only returns one element. Regarding the lazy rule called by a collect, getColumnsCollect, it is more complex. It receives a sequence with all the elements (Attribute instances in this case) to iterate, it also receives the class model and the value of the counter to give fresh identifiers to the elements created. As the previous lazy rule, it returns a set of objects that will be added to the relational model. This rule iterates over the elements of the sequence by creating a new Column instance and a new Type instance for the first element of the sequence. Then, it calls the function again but now without the first element of the sequence and with the value of the counter updated. The function keeps doing this until there are no more elements in the sequence. This way, when this function is called from the relational model, all the elements created are added to it.

As explained in the example above, in lazy rules that are not called by a collect, it is only necessary to write, in the relational model, the identifier of the first object created by the lazy rule when we want to reference the objects created by the rule, type@Column@RelationalMm : newId(VALUE@CNT@) in this case. But with lazy rules called by a collect it is different since we need to reference the sequence of objects created by the lazy rule. To do this, we use an auxiliary function, getOidsCollect. It receives as arguments the sequence of objects that is also received by the lazy rule(<< C@ . att@Class@ClassMm ; CLASSMODEL@ >>), 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. The function iterates over the elements of the sequence received as the first argument by taking the first element of the sequence and adding the identifier

that corresponds to the element created by the lazy rule from that element to the sequence that will be returned. This sequence is concatenated with the one that results from applying the function again to the sequence received as argument, but taking out the first element, and updating the value of the identifier by adding as many units to it as elements are created in each iteration by the lazy rule.

4.6 Unique lazy rules

In this section we will present the same first example presented in the previous section, but, this time, the rule createTable is a unique lazy rule. Consequently, the result produced by this transformation will be different. The implementation in Maude is also quite different, since 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. We achieve these results with auxiliary functions that we can see in the implementation:

```
mod @Class2RelationalUniqueLazyRule@ is
  protecting @CLASSMM@ .
  ...
  vars T@ TC@ AT@ C@ CO@ TR@ CNT@ : Oid .
  var SFS : Set{@StructuralFeatureInstance} .
  var OBJSET@ OBJSETT@ OBJSETTT@ : Set{@Object} .
  var VALUE@CNT@ : Int .
  var CLASSMODEL@ TRACEMODEL@ : @Model .
  var NAME : String .
  var CL@ : @Class .

------------------- Unique lazy rule createTable ---------------------------
  op createTable : Oid String @Model @Model Int -> Set{@Object} .
  eq createTable(AT@, NAME, CLASSMODEL@, TRACEMODEL@, VALUE@CNT@) =
    if alreadyExecuted(Sequence[AT@], NAME, TRACEMODEL@) then none else
    < newId(VALUE@CNT@) : Table@RelationalMm | name@Named@RelationalMm :
        << "Table_of_" + AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> #
    key@Table@RelationalMm : newId(VALUE@CNT@ + 1) >
    < newId(VALUE@CNT@ + 1) : Column@RelationalMm | name@Named@RelationalMm
        : << "Key_of_" + AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> #
    owner@Column@RelationalMm : newId(VALUE@CNT@) >
    fi .
-----------------------------------------------------------------------------
--- Function that gets an Oid from the TRACEMODEL@ if it was already created,
---otherwise it calls the function getElem to get the Oid that is to be created
  op getOidUnique : String Oid @Model @Model Int -> Oid .
  eq getOidUnique(NAME, AT@, CLASSMODEL@, TRACEMODEL@, VALUE@CNT@) =
    if alreadyExecuted(Sequence[AT@], NAME, TRACEMODEL@)
    then getElem(NAME, AT@, TRACEMODEL@) else newId(VALUE@CNT@)
      fi .
-----------------------------------------------------------------------------
--- Function that gets and Oid which already exists--------------------------
  op getElem : String Oid @Model -> Oid .
  eq getElem(NAME, AT@, @TraceMm@ { < TR@ : Trace@TraceMm | srcEl@TraceMm
      : Sequence[ AT@ ] # trgEl@TraceMm : Sequence [ C@ ] # rlName@TraceMm :
      NAME # srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" >
      OBJSET@ } ) = C@ .
  eq getElem(NAME, AT@, @TraceMm@ { OBJSET@ } ) = mt-ord [owise] .
-----------------------------------------------------------------------------
------Function that creates a trace only if it did not exist already---------
  op createTrace : Oid String Int @Model @Model -> Set{@Object} .
  eq createTrace(AT@, NAME, VALUE@CNT@, @TraceMm@ { < T@ : Trace@TraceMm |
      srcEl@TraceMm : Sequence[ AT@ ] # rlName@TraceMm : NAME # SFS >
      OBJSET@ }, CLASSMODEL@ ) = none .
```

```
  eq createTrace(AT@, NAME, VALUE@CNT@, TRACEMODEL@, CLASSMODEL@) =
      < newId(VALUE@CNT@) : Trace@TraceMm | rlName@TraceMm : NAME #
      srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" # trgEl@TraceMm
      : Sequence[ getOidUnique(NAME, AT@, CLASSMODEL@, TRACEMODEL@,
      VALUE@CNT@)] # srcEl@TraceMm : Sequence[AT@] > [owise] .
------------------------------------------------------------------------------
rl [Init] :
  ...

crl [CreateColumnsFromAttribute] :
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ }) ;
  (@RelationalMm@ {OBJSETTT@})
  ]
  =>
  Sequence[
  (@ClassMm@ {
  < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ }) ;
  (@TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 9 >
    --- Trace for the matched rule
  < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ AT@ ]# trgEl@TraceMm :
      Sequence[ CO@ ; C@]# rlName@TraceMm : "CreateColumnsFromAttribute" #
      srcMdl@TraceMm : "Class" # trgMdl@TraceMm : "Relation" >
    --- The trace for the unique lazy rule must be created iff it did not
    --- exist already
  createTrace(AT@, "createTable", VALUE@CNT@ , TRACEMODEL@, CLASSMODEL@)
  OBJSETT@ }) ;
  (@RelationalMm@ {
  < CO@ : Column@RelationalMm | name@Named@RelationalMm : << "First_col_of_"
      + AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> # owner@Column@RelationalMm :
      getOidUnique("createTable", AT@, CLASSMODEL@, TRACEMODEL@, VALUE@CNT@ + 1) >
  createTable(AT@, "createTable", CLASSMODEL@, TRACEMODEL@, VALUE@CNT@ + 1)
  < C@ : Column@RelationalMm | name@Named@RelationalMm : << "Second_col_of_" +
      AT@ . name@NamedElt@ClassMm ; CLASSMODEL@ >> # owner@Column@RelationalMm :
      getOidUnique("createTable", AT@, CLASSMODEL@, TRACEMODEL@, VALUE@CNT@ + 1) >
  OBJSETTT@ })
  ]
  if
  CLASSMODEL@ := @ClassMm@ {
    < AT@ : Attribute@ClassMm | SFS >
  OBJSET@ } /\
  TRACEMODEL@ := @TraceMm@ {
  < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
  OBJSETT@ } /\
  CO@ := newId(VALUE@CNT@ + 3) /\ C@ := newId(VALUE@CNT@ + 4) /\
  TR@ := newId(VALUE@CNT@ + 5) /\
  not alreadyExecuted(Sequence[ AT@ ], "CreateColumnsFromAttribute",
    @TraceMm@ { OBJSETT@ } )
.
endm
```

Function createTable is different from the one in the previous example, that used non-unique lazy rules. Now, we have to check if the resulting objects of the rule have already been created for the Attribute instance passed as parameter. If they do exist, this function has to return nothing, since it means that the resulting objects were already created. Otherwise, the function returns the set of objects created by the rule. As in the previous example, this function is called from the relational model in the righthand

side, since the elements created by the rule have to be added to the ones contained in this model.

In the ATL code of this transformation, presented in Sect. 2.4, the unique lazy rule was called twice. The first call created the elements and got their identifiers. The second one only got the identifiers, since the objects had been already created by the previous call. In our Maude code, we distinguish between creating the elements and getting their identifiers. The former is done by the call to the function createTable, and the latter is achieved with the function getOidUnique. Since unique lazy rules create elements only the first time they are called (for the same arguments), we have only called the function createTable once. Had we called it twice, the result would have been the same, since the second call would not have produced anything.

As we have said before, for the references to the objects created by the unique lazy rule now we use an auxiliary function, getOidUnique. With non-unique lazy rules it was enough writing the identifier of the first element created by the lazy rule, but now it is not that simple since the object could have been created by a different rule. This function checks if the element already existed by using the function alreadyExecuted. If it did, then the function gets the identifier of the first element created by the unique lazy rule by calling the function getElem, which searches for this element in the trace model. Otherwise, if the element does not exist, it returns the identifier that will be applied to the element when the rule creates it.

Traces are created now in a different way, too. One trace is added for the matched rule. As for the unique lazy rule, an auxiliary function is used to create the trace, called createTrace. This function creates the trace for the unique lazy rule if it did not exist before. Therefore, it checks if a trace with the Attribute passed as parameter and the name createTable exists in the trace model. If it does, it adds nothing. Otherwise, it adds a new trace containing the Attribute's identifier, the name of the rule (createTable), and the identifier of the first element created by the unique lazy rule, using the function getOidUnique as mentioned before.

4.7 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. We implement the four types of instructions: *assignments* (=), *conditional* branches (if), loops (for) and *called rules*. In the following piece of code we show how the Instr type 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] .
---Instructions for loops
op For : Sequence InstrSequ -> Instr [ctor] .
op AssignAttFor : @StructuralFeature @StructuralFeature @Model OCL-Exp
                  -> Instr [ctor] .
op IfFor : String @StructuralFeature OCL-Exp @Model InstrSequ InstrSequ
           -> 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 call rule (AddColumn in this case) and three instructions are used for loops.

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} instrSeq -> 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 new 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.

Regarding how the do function works when the instruction is a For, we show the following piece of code:

```
eq do(OBJSET@, For(Sequence[AT@ ; LO], INSTR1@) ^ INSTR@) = do(OBJSET@,
       For(AT@, INSTR1@) ^ For(Sequence[LO], INSTR1@) ^ INSTR@) .
eq do(OBJSET@, For(Sequence[AT@ ; LO], INSTR1@ ^ INSTR2@) ^ INSTR@) =
       do(OBJSET@, For(AT@, INSTR1@) ^ For(Sequence[LO], INSTR1@) ^
       For(Sequence[AT@ ; LO], INSTR2@)  ^ INSTR@) .
eq do(OBJSET@, For(Sequence[mt-ord], INSTR1@) ^ INSTR@) = do(OBJSET@, INSTR@) .
eq do(OBJSET@, For(AT@, none) ^ INSTR@) = do(OBJSET@, INSTR@) .
```

Please note that we keep here the recursion mentioned before, so it will not be mentioned again. In the general case, the For function receives a sequence of objects and a sequence of instructions. It applies the sequence of instructions to every object of the sequence received in the first argument. When the next instruction to be applied is

an assignment (AssignAttFor instruction) over a single object (which is a sequence with only one element), the following piece of code is applied:

```
eq do(OBJSET@, For(AT@, AssignAttFor(SF@, SF1@, RELATIONALMODEL@, EXP@) ^
       INSTR1@) ^ INSTR@) = do(doAssign(OBJSET@, AT@, SF@, << AT@ . SF1@
            ; RELATIONALMODEL@ >> + EXP@), For(AT@, INSTR1@) ^ INSTR@) .
```

The AssignAttFor instruction receives two structural features (which is the type of the objects attributes in our Maude encoding), the model containing the objects that have been created by the rule so far (needed because they may be referenced) and an OCL expression. The aim of this instruction in this version is to assign to the value of the element attribute passed as first argument in the AssignAttFor the value of its attribute in the second argument plus the OCL expression received in the fourth argument.

When the instruction found inside the For is an IfFor, the function do works as follows:

```
eq do(OBJSET@, For(AT@, IfFor(ST@, SF@, EXP@, RELATIONALMODEL@, INSTR1@,
   INSTR2@) ^ INSTR3@) ^ INSTR@) =
     if (ST@ == "==")  then
       if (<< AT@ . SF@ ; RELATIONALMODEL@ >> == EXP@)
         then do(OBJSET@, For(AT@, INSTR1@ ^ INSTR3@) ^ INSTR@)
         else do(OBJSET@, For(AT@, INSTR2@ ^ INSTR3@) ^ INSTR@)
       fi
     else if (ST@ == ">=")  then
       if (<< AT@ . SF@ ; RELATIONALMODEL@ >> >= EXP@)
         then do(OBJSET@, For(AT@, INSTR1@ ^ INSTR3@) ^ INSTR@)
         else do(OBJSET@, For(AT@, INSTR2@ ^ INSTR3@) ^ INSTR@)
       fi
     else if (ST@ == "<=")  then
       if (<< AT@ . SF@ ; RELATIONALMODEL@ >> <= EXP@)
         then do(OBJSET@, For(AT@, INSTR1@ ^ INSTR3@) ^ INSTR@)
         else do(OBJSET@, For(AT@, INSTR2@ ^ INSTR3@) ^ INSTR@)
       fi
     else if (ST@ == ">")  then
       if (<< AT@ . SF@ ; RELATIONALMODEL@ >> > EXP@)
         then do(OBJSET@, For(AT@, INSTR1@ ^ INSTR3@) ^ INSTR@)
         else do(OBJSET@, For(AT@, INSTR2@ ^ INSTR3@) ^ INSTR@)
       fi
     else if (ST@ == "<")  then
       if (<< AT@ . SF@ ; RELATIONALMODEL@ >> < EXP@)
         then do(OBJSET@, For(AT@, INSTR1@ ^ INSTR3@) ^ INSTR@)
         else do(OBJSET@, For(AT@, INSTR2@ ^ INSTR3@) ^ INSTR@)
       fi
     else if (ST@ == "=/=")  then
       if (<< AT@ . SF@ ; RELATIONALMODEL@ >> =/= EXP@)
         then do(OBJSET@, For(AT@, INSTR1@ ^ INSTR3@) ^ INSTR@)
         else do(OBJSET@, For(AT@, INSTR2@ ^ INSTR3@) ^ INSTR@)
       fi
     else none
     fi fi fi fi fi fi .
```

Let us remind the reader that the IfFor function receives as arguments a string, a structural feature, an OCL expression, the model containing all the elements created by the rule so far, and two sequences of instructions. The If instruction shown before was much simpler because the condition of the *if* is written inside the imperative part of the rule (as we show see later), so it is passed to the function as the boolean result. Now, however, the condition needs to be created inside the function because it needs to be evaluated for each element of the sequence which is inside the For. Thus, the string received by the IfFor instruction contains the kind of comparisons that will be made in the condition (in this version, they are "==", ">=", "<=", ">", "<" and "= / ="). The structural fea-

ture contains the name of the attribute whose value will be compared in the condition with the OCL expression received in the third argument. If the condition is satisfied, the sequence of instructions received in the fifth argument are applied; otherwise, the instructions received in the sixth argument are applied.

To use all the instructions presented in an example, let us show an ATL rule containing all these features:

```
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 + 'Id',
           type <- thisModule.objectIdType ),
      k :  Relational!Column (name <- at.name + 'Id',
           type <- thisModule.objectIdType )
do{
     tb.name <- tb.name + '_Multi';
     if (tb.col->size() = 3){
         k.name <- 'key';
     }else{
         k.name <- 'key_else';
     }
     thisModule.AddColumn('New_Column');
     for(c in out.col){
         c.name <- c.name + '_assign1';
         c.name <- c.name + '_assign2';
         if (c.name = 'key_else_assign1_assign2'){
           c.name <- c.name + '_ifYES';
         }else{
           c.name <- c.name + '_ifNO';
         }
         c.name <- c.name + '_afterIF';
     }
 }
}
```

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, << TB@ . name@Named@RelationalMm ;
       RELATIONALMODEL@ >> + "_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") ^
     For(<< TB@ . col@Table@RelationalMm ; RELATIONALMODEL2@ >>,
       AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
         RELATIONALMODEL2@, "_assign1") ^
       AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
```

```
        RELATIONALMODEL3@, "_assign2") ^
      IfFor("==", name@Named@RelationalMm, "key_else_assign1_assign2",
       RELATIONALMODEL4@,
        AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
        RELATIONALMODEL4@, "_ifYES"),
        AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
        RELATIONALMODEL4@, "_ifNO")
      ) ^
      AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
        RELATIONALMODEL5@, "_afterIF")
    )
)
OBJSETTT@ }) ]
if ... /\
  RELATIONALMODEL@ := @RelationalMm@ {
    < 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@}) >} /\
  RELATIONALMODEL2@ := @RelationalMm@ {do(objectsSet(RELATIONALMODEL@),
    Assign(TB@, name@Named@RelationalMm, << TB@ . name@Named@RelationalMm ;
      RELATIONALMODEL@ >> + "_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") ) } /\
  RELATIONALMODEL3@ := @RelationalMm@ {do(objectsSet(RELATIONALMODEL2@),
    For(<< TB@ . col@Table@RelationalMm ; RELATIONALMODEL2@ >>,
      AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
        RELATIONALMODEL2@, "_assign1")) ) } /\
  RELATIONALMODEL4@ := @RelationalMm@ {do(objectsSet(RELATIONALMODEL2@),
    For(<< TB@ . col@Table@RelationalMm ; RELATIONALMODEL2@ >>,
      AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
        RELATIONALMODEL2@, "_assign1") ^
      AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
        RELATIONALMODEL3@, "_assign2")) ) } /\
  RELATIONALMODEL5@ := @RelationalMm@ {do(objectsSet(RELATIONALMODEL2@),
    For(<< TB@ . col@Table@RelationalMm ; RELATIONALMODEL2@ >>,
      AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
        RELATIONALMODEL2@, "_assign1") ^
      AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
        RELATIONALMODEL3@, "_assign2")    ^
      IfFor("==", name@Named@RelationalMm, "key_else_assign1_assign2",
       RELATIONALMODEL4@,
        AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
          RELATIONALMODEL4@, "_ifYES"),
        AssignAttFor(name@Named@RelationalMm, name@Named@RelationalMm,
          RELATIONALMODEL4@, "_ifNO") ) ) ) } .
```

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 is the why in which ATL works. The second argument is a sequence of instructions which contains, in this case, four instructions. The first instruction executed is an Assign. Then an If block with two assignments inside is executed. After this, the instruction that represents the called rule, AddColumn, is executed. Finally, a For instruction, containing four instructions (three assignments and a if block), is executed.

ResolveTemp. Let us present here the encoding of the function resolveTemp, which is implemented by the following piece of code:

```
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 .
```

The function receives in the first argument the identifier of the source model element from which the searched target model element is produced. The second argument is a natural number containing the position that the identifier of the object which is wanted to be retrieved has in the sequence in the field trgEl@TraceMm of the object of the trace model which was created when the corresponding rule was executed. The third and fourth arguments contain the trace and class models, respectively. It returns the identifier of the element to be retrieved.

The function looks for the trace which contains the source element passed as first argument and returns the identifier of the element which is wanted to be returned by retrieving it from the sequence of elements created from the source element.

Please note that the biggest difference between this function and the one in ATL is that here we receive as second argument the position that the searched target model element has among the ones created in the corresponding rule. In ATL, instead, the argument received is the name of the variable that was given to the searched 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 in the ATL rule.

5 Simulation and Formal Analysis

Once the system specifications are encoded in Maude, what we get is a rewriting logic specification of the model transformation. 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.

This way, the results of the ATL model transformations described in Section 2, when applied to the source Class model, are a sequence of three models: the source, the trace and the target Relational model. This last will be shown in the following for every transformation.

Matched rules and helper. This is the Relational model resulting from the model transformation whose rules were presented in Section 4.3 and its helper in Section 4.4. It is equivalent to the resulting model presented in Section 2.1.

```
@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 >
}
```

Lazy rules. Here we show the resulting Relational models from applying the transformations shown in Sect. 4.5. For the first of them, called @Class2RelationalLazyRule@, the relational model obtained is:

```
@RelationalMm@{
  < '1 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_name" #
    key@Table@RelationalMm : '2 >
  < '2 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_name" #
    owner@Column@RelationalMm : '1 >
  < '5 : Column@RelationalMm | name@Named@RelationalMm : "First_col_of_name" #
    owner@Column@RelationalMm : '1 >
  < '3 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_name" #
    key@Table@RelationalMm : '4 >
  < '4 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_name" #
    owner@Column@RelationalMm : '3 >
  < '6 : Column@RelationalMm | name@Named@RelationalMm : "Second_col_of_name" #
    owner@Column@RelationalMm : '3 >
  < '10 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_members" #
    key@Table@RelationalMm : '11 >
  < '11 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_members" #
    owner@Column@RelationalMm : '10 >
  < '14 : Column@RelationalMm | name@Named@RelationalMm : "First_col_of_members" #
    owner@Column@RelationalMm : '10 >
  < '12 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_members" #
    key@Table@RelationalMm : '13 >
  < '13 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_members" #
    owner@Column@RelationalMm : '12 >
```

```
  < '15 : Column@RelationalMm | name@Named@RelationalMm :
    "Second_col_of_members" # owner@Column@RelationalMm : '12 >
  < '19 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_firstName" #
    key@Table@RelationalMm : '20 >
  < '20 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_firstName" #
    owner@Column@RelationalMm : '19 >
  < '23 : Column@RelationalMm | name@Named@RelationalMm :
    "First_col_of_firstName" # owner@Column@RelationalMm : '19 >
  < '21 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_firstName" #
    key@Table@RelationalMm : '22 >
  < '22 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_firstName" #
    owner@Column@RelationalMm : '21 >
  < '24 : Column@RelationalMm | name@Named@RelationalMm :
    "Second_col_of_firstName" # owner@Column@RelationalMm : '21 >
  < '28 : Table@RelationalMm | name@Named@RelationalMm :
    "Table_of_closestFriend" # key@Table@RelationalMm : '29 >
  < '29 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_closestFriend" #
    owner@Column@RelationalMm : '28 >
  < '32 : Column@RelationalMm | name@Named@RelationalMm :
    "First_col_of_closestFriend" # owner@Column@RelationalMm : '28 >
  < '30 : Table@RelationalMm | name@Named@RelationalMm :
    "Table_of_closestFriend" # key@Table@RelationalMm : '31 >
  < '31 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_closestFriend" #
    owner@Column@RelationalMm : '30 >
  < '33 : Column@RelationalMm | name@Named@RelationalMm :
    "Second_col_of_closestFriend" # owner@Column@RelationalMm : '30 >
  < '37 : Table@RelationalMm | name@Named@RelationalMm :
    "Table_of_emailAddresses" # key@Table@RelationalMm : '38 >
  < '38 : Column@RelationalMm | name@Named@RelationalMm :
    "Key_of_emailAddresses" # owner@Column@RelationalMm : '37 >
  < '41 : Column@RelationalMm | name@Named@RelationalMm :
    "First_col_of_emailAddresses" # owner@Column@RelationalMm : '37 >
  < '39 : Table@RelationalMm | name@Named@RelationalMm :
    "Table_of_emailAddresses" # key@Table@RelationalMm : '40 >
  < '40 : Column@RelationalMm | name@Named@RelationalMm :
    "Key_of_emailAddresses" # owner@Column@RelationalMm : '39 >
  < '42 : Column@RelationalMm | name@Named@RelationalMm :
    "Second_col_of_emailAddresses" # owner@Column@RelationalMm : '39 >
}
```

As it can be seen in the model, and as mentioned in Sect. 4.5, two Table and Column instances have been created for each Attribute with the same name, since lazy rules always create new elements when they are executed.

With regard to the second model transformation, which is the one containing a lazy rule called by a **collect**, the resulting Relational model is:

```
@RelationalMm@{
  < '11 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_Family" #
    col@Table@RelationalMm : Sequence ['2 ; '4] >
  < '10 : Column@RelationalMm | name@Named@RelationalMm : "Family" #
    owner@Column@RelationalMm : '11 # type@Column@RelationalMm : '1 >
  < '2 : Column@RelationalMm | name@Named@RelationalMm : "Column_of_name" #
    type@Column@RelationalMm : '3 >
  < '4 : Column@RelationalMm | name@Named@RelationalMm : "Column_of_members" #
    type@Column@RelationalMm : '5 >
  < '1 : Type@RelationalMm | name@Named@RelationalMm : "Column_type_of_Family" >
  < '3 : Type@RelationalMm | name@Named@RelationalMm : "Type_of_name" >
  < '5 : Type@RelationalMm | name@Named@RelationalMm : "Type_of_members">

  < '24 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_Person" #
    col@Table@RelationalMm :  Sequence ['13 ; '15 ; '17] >
  < '23 : Column@RelationalMm | name@Named@RelationalMm : "Person" #
    owner@Column@RelationalMm : '24 # type@Column@RelationalMm : '12 >
  < '13 : Column@RelationalMm | name@Named@RelationalMm : "Column_of_firstName" #
    type@Column@RelationalMm : '14 >
  < '15 : Column@RelationalMm | name@Named@RelationalMm :
```

```
    "Column_of_closestFriend" # type@Column@RelationalMm : '16 >
  < '17 : Column@RelationalMm | name@Named@RelationalMm :
    "Column_of_emailAddresses" # type@Column@RelationalMm : '18 >
  < '18 : Type@RelationalMm | name@Named@RelationalMm : "Type_of_emailAddresses" >
  < '12 : Type@RelationalMm | name@Named@RelationalMm : "␣Column_type_of_Person" >
  < '14 : Type@RelationalMm |name@Named@RelationalMm : "Type_of_firstName">
  < '16 : Type@RelationalMm | name@Named@RelationalMm : "Type_of_closestFriend" >
}
```

For each Class instance of the input model, this transformation creates a new Column and a new Table as the Column's owner in the matched rule, so these two instances are associated this way. The type of the Column is created by means of a lazy rule. Regarding the Table, its col reference points to a sequence of new Column instances created by means of a lazy rule called by a collect. In this lazy rule, one Column and one Type are created for each Attribute contained in the Class.

According to the input model presented in Section 2.1, we have two Class instances, one of them contains two Attribute instances and the other one contains three. This way, what we finally obtain with this transformation are two Table instances and two Column instances associated to these Tables. One Type instance is created by the first lazy rule for each of the Columns. With regard to the Tables, two new Column instances are created for the first one and three new instances are created for the second one by means of the lazy rule called by a collect. In the same lazy rule, a new Type instance is created for each of these Column instances.

Unique lazy rules. Here we show the results of the transformation shown in Sect. 4.6, @Class2RelationalUniqueLazyRule@, which is actually the same as the model transformation with the lazy rule executed before but with a unique lazy rule instead of the non-unique createTable. Consequently, the result produced by this transformation is different, since in the example before a new Table and Column instances were created for each Attribute instance. But with the unique lazy rule, the Table and Column instances are created only the first time the rule is called because the rest of the calls just retrieve the identifier of the Table:

```
@RelationalMm@{
  < '2 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_name"
    # key@Table@RelationalMm : '3 >
  < '3 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_name"
    # owner@Column@RelationalMm : '2 >
  < '4 : Column@RelationalMm | name@Named@RelationalMm : "First_col_of_name"
    # owner@Column@RelationalMm : '2 >
  < '5 : Column@RelationalMm | name@Named@RelationalMm : "Second_col_of_name"
    # owner@Column@RelationalMm : '2 >
  < '11 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_members"
    # key@Table@RelationalMm : '12 >
  < '12 : Column@RelationalMm |name@Named@RelationalMm : "Key_of_members"
    # owner@Column@RelationalMm : '11 >
  < '13 : Column@RelationalMm | name@Named@RelationalMm :"First_col_of_members"
    # owner@Column@RelationalMm : '11 >
  < '14 : Column@RelationalMm | name@Named@RelationalMm :
    "Second_col_of_members" # owner@Column@RelationalMm : '11 >
  < '20 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_firstName"
    # key@Table@RelationalMm : '21 >
  < '21 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_firstName"
    # owner@Column@RelationalMm : '20 >
  < '22 : Column@RelationalMm | name@Named@RelationalMm :
    "First_col_of_firstName" # owner@Column@RelationalMm : '20 >
  < '23 : Column@RelationalMm | name@Named@RelationalMm :
```

```
    "Second_col_of_firstName" # owner@Column@RelationalMm : '20 >
  < '29 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_closestFriend"
    # key@Table@RelationalMm : '30 >
  < '30 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_closestFriend"
    # owner@Column@RelationalMm : '29 >
  < '31 : Column@RelationalMm | name@Named@RelationalMm :
    "First_col_of_closestFriend" # owner@Column@RelationalMm : '29 >
  < '32 : Column@RelationalMm | name@Named@RelationalMm :
    "Second_col_of_closestFriend" # owner@Column@RelationalMm : '29 >
  < '38 : Table@RelationalMm | name@Named@RelationalMm : "Table_of_emailAddresses"
    # key@Table@RelationalMm : '39 >
  < '39 : Column@RelationalMm | name@Named@RelationalMm : "Key_of_emailAddresses"
    # owner@Column@RelationalMm : '38 >
  < '40 : Column@RelationalMm | name@Named@RelationalMm :
    "First_col_of_emailAddresses" # owner@Column@RelationalMm : '38 >
  < '41 : Column@RelationalMm | name@Named@RelationalMm :
    "Second_col_of_emailAddresses" # owner@Column@RelationalMm : '38 > }
```

In the previous example with the lazy rule, two Table and a Column (the Table's key) instances were created for each Attribute with the same name. Now, however, only one Table and one Column (the Table's key) are created for each Attribute by the unique lazy rule. This way, there are half the Table instances that were before and each of them has three Column instances associated: the key and the two Columns created by the matched rule ("First_col_.." and "Second_col_...").

The imperative section. In this subsection we show the Relational model resulting from applying the model transformation shown in Section 4.3 with the helper of Section 4.4 and with the imperative section added to the rule MultiValuedClassAttribute2Column shown in Section 4.7.

```
@RelationalMm@{
  < '10 : Type@RelationalMm | name@Named@RelationalMm : "String" >
  < '13 : Column@RelationalMm | name@Named@RelationalMm : "name" #
    type@Column@RelationalMm : '10 >
  < '15 : Table@RelationalMm | name@Named@RelationalMm : "Person_emailAddresses" #
    col@Table@RelationalMm : Sequence ['16 ; '17] >
  < '16 : Column@RelationalMm | name@Named@RelationalMm : "PersonId" #
    type@Column@RelationalMm : '2 >
  < '17 : Column@RelationalMm | name@Named@RelationalMm : "emailAddresses" #
    type@Column@RelationalMm : '10 >
  < '19 : Table@RelationalMm | name@Named@RelationalMm : "Family" #
    col@Table@RelationalMm : Sequence ['20 ; '13] #
    key@Table@RelationalMm : Set {'20} >
  < '2 : Type@RelationalMm | name@Named@RelationalMm : "Integer" >
  < '20 : Column@RelationalMm | name@Named@RelationalMm : "objectId" #
    type@Column@RelationalMm : '2 >
  < '23 : Column@RelationalMm | name@Named@RelationalMm : "firstName" #
    type@Column@RelationalMm : '10 >
  < '25 : Table@RelationalMm | name@Named@RelationalMm : "Person" #
    col@Table@RelationalMm : Sequence ['26 ; '23 ; '4] #
    key@Table@RelationalMm : Set {'26} >
  < '26 : Column@RelationalMm | name@Named@RelationalMm : "objectId" #
    type@Column@RelationalMm : '2 >
  < '4 : Column@RelationalMm | name@Named@RelationalMm : "closestFriendId" #
    type@Column@RelationalMm : '2 >
  < '6 : Table@RelationalMm | name@Named@RelationalMm : "Family_members_Multi" #
    col@Table@RelationalMm : Sequence ['7 ; '8] #
    key@Table@RelationalMm : Set {'8} >
  < '7 : Column@RelationalMm | name@Named@RelationalMm :
    "FamilyId_assign1_assign2_ifNO_afterIF" # type@Column@RelationalMm : '2 >
  < '8 : Column@RelationalMm | name@Named@RelationalMm :
    "key_else_assign1_assign2_ifYES_afterIF" # type@Column@RelationalMm : '2 >
  < '5 : Column@RelationalMm | name@Named@RelationalMm : "New_Column" >  }
```

This resulting model is very similar to the one that resulted from applying the matched rules without imperative parts, but with slight differences due to the imperative section. This way, the name of the Table whose identifier is '6 has been concatenated with "_Multi", since this Table has been created by the rule MultiValuedClassAttribute2Column. We see as well that the name of the Column which is the key of the Table mentioned before is now "key_else_assign1_assign2_ifYES_afterIF", due to the instructions contained in the imperative block. The name of the other column of the Table has been modified as well, and now it is "FamilyId_assign1_assign2_ifNO_afterIF". Finally, a new Column has been created with the name "New_Column" due to the called rule.

5.2 Analyzing the trace model

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] .
```

5.3 Reachability analysis

Executing the system using the rewrite and frewrite commands means exploring just one possible behavior of the system. However, a rewrite system do not need to be Church-Rosser and terminating,[1] 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 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.

[1] 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.

A complete reachability analysis, including the presented features and others, will be present in future works, providing results of different executions of the system.

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 the Maude toolkit to reason about the corresponding 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 (i.e., a *semantic mapping* between these two semantic domains) that realizes the automatic generation of the Maude specifications. 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 system modelers to be able to conduct different kinds of analysis to the ATL model transformations, being unaware of the formal representation of their specifications in Maude.

Acknowledgements. The authors would like to thank Franciso Durán and José E. Riverafor their comments and suggestions on the paper.

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) (1992) 73–155
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework. Volume 4350 of LNCS. Springer, Heidelberg, Germany (2007)

4. Eclipse: (ATL) http://www.eclipse.org/m2m/atl/atlTransformations/.
5. Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: Proc. of FMOODS'06. (2006) 171–185
6. Bouhoula, A., Jouannaud, J.P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science **236**(1) (2000) 35–132
7. Troya, J., Vallecillo, A.: Formal Semantics of ATL Using Rewriting Logic (Resources). Universidad de Málaga. (2010) http://atenea.lcc.uma.es/index.php/Main_Page/Resources/ATL-Maude.
8. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Proc. of MoDELS 2007. Volume 4735 of LNCS., Springer (2007) 1–15
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) (2009) 778–792
10. Roldán, M., Durán, F.: Representing UML models in mOdCL. Manuscript. http://maude.lcc.uma.es/mOdCL (2008)
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: Proc. of ICGT'06. Number 4178 in LNCS, Springer (2006) 306–320
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. (2007) 47–56
14. Poernomo, I.: Proofs-as-model-transformations. In: Proc. of ICMT'08. Number 5063 in LNCS, Zurich, Switzerland, Springer (2008) 214–228
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: volume I. foundations. (1997) 313–400
16. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: language and environment. In: Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools. (1999) 487–550
17. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: Proc. of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09), Springer-Verlag (2009) 18–33
18. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. In: Proc. of FASE'08. Volume 4961 of LNCS., Springer (2008) 377–391
19. Durán, F., Lucas, S., Meseguer, J.: MTT: The Maude Termination Tool (System Description). In: Proc. of the 4th international joint conference on Automated Reasoning (IJCAR'08). Volume 5195 of LNAI., Berlin, Heidelberg, Springer (2008) 313–319
20. Durán, F., Meseguer, J.: A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In: Proc. of WRLA 2010. LNCS, Springer (2010)