Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

A Rewriting Logic Semantics for ATL

Javier Troya[a] Antonio Vallecillo[a]

a. GISUM/Atenea Research Group. Universidad de Málaga, Spain.

Abstract As the complexity of model transformation (MT) grows, the need to rely on formal semantics of MT languages becomes a critical issue. Formal semantics provide precise specifications of the expected behavior of transformations, allowing users to understand them and to use them properly, and MT tool builders to develop correct MT engines, compilers, etc. In addition, formal semantics allow modelers to reason about the MTs and to prove their correctness, something 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 give a formal semantics of the ATL 3.0 model transformation language using rewriting logic and Maude, which allows addressing these issues. Such formalization provides additional benefits, such as enabling the simulation of the specifications or giving access to the Maude toolkit to reason about them.

Keywords ATL; Maude; Model Transformation; semantics.

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

ATL [JABK08] is one of the most popular and widely used model transformation languages. As usual in the community, the ATL language has been described in an intuitive and informal manner, by means of definitions of its main features in natural

Javier Troya, Antonio Vallecillo. *A Rewriting Logic Semantics for ATL*. In Journal of Object Technology, vol. 10, 2011, pages 5:1–29. doi:10.5381/jot.2011.10.1.a5

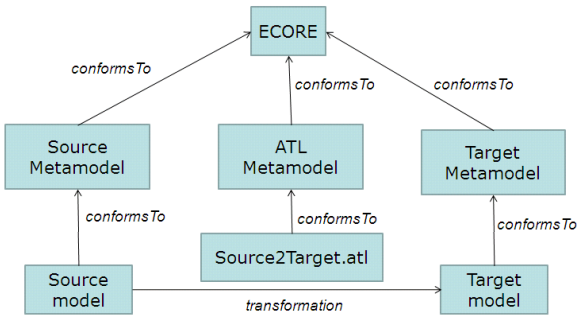


Figure 1 – ATL model transformation schema.

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. The other reference implementation of ATL is available as metamodels for the language and its virtual machine, and as a compiler from the language to the virtual machine and an interpreter for the virtual machine. The problem of this kind of implementation is that it is not abstract enough to provide meaningful semantics, and in an implementation-independent manner.

In this paper we investigate the use of rewriting logic [Mes92], and its implementation in Maude [CDE+07], 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 this sense, we provide a more abstract encoding than the ATL current implementation, together with an alternative specification of the transformations that can be simulated and analyzed for correctness.

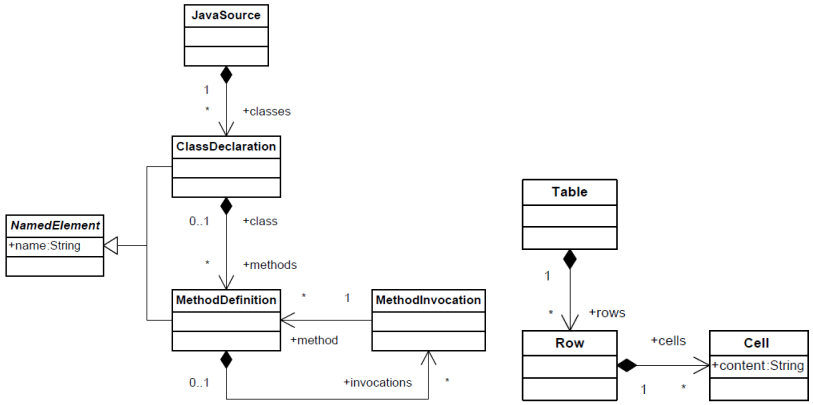
This paper is an extension of the one presented in the ICMT'10 conference [TV10a]. Here we deal with all new features of ATL version 3.0, and in particular we formalize the ATL refining mode — in addition to the ATL default execution semantics. Furthermore, we discuss some improvements in the Maude representation of the ATL rules to obtain better performance when simulating the ATL specifications. New ATL examples are also shown in this paper.

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

2 Transformations with ATL

ATL is a hybrid model transformation 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



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

Figure 2 – Metamodels used in the example transformation.

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

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

In some cases, complex transformation algorithms may be required, and it may be difficult to specify them in a declarative way. For this reason ATL provides two imperative constructs: **called rules** and **action blocks**. A called rule is a rule called by others like a procedure. An action block is a sequence of imperative statements and can be used instead of or in combination with a target pattern in matched or called rules. The imperative statements in ATL are the usual constructs for attribute assignment and control flow: conditions and loops.

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

ATL has two execution modes, the normal (default) execution mode and the refin-

ing one. In the former, the ATL developer has to specify, either by matched or called rules, the way to generate each of the expected target model elements. This execution mode suits to most ATL transformations where source and target metamodels are different. Using the refining mode, ATL developers can define transformations that modify the source model to obtain the target model, since both models conform to the same metamodel. This mode is further explained in section 2.1.

In order to illustrate our proposal for the ATL default execution mode we will use the example of JavaSource2Table model transformation [Ecl10], whose code is shown below. It has two matched rules, two lazy rules and two helpers. The complete description of this example and its encoding in Maude can be found in the technical report [TBV10]. Although the ATL rules are mostly self-explanatory, readers not fluent in ATL can also consult [JABK08, TV10b] for more details and examples.

```
module JavaSource2Table;
  create OUT : Table from IN : JavaSource;

helper def: allMethodDefs : Sequence(JavaSource!MethodDefinition) =
  JavaSource!MethodDefinition.allInstances()
  -> sortedBy(e | e.class.name + '_' + e.name)
  -> asSequence() ;
helper context JavaSource!MethodDefinition
  def : computeContent(col : JavaSource!MethodDefinition) : String =
    self.invocations -> select(i | i.method.name = col.name and
                    i.method.class.name = col.class.name) ->size();
rule Main {
  from s : JavaSource!JavaSource
  to t : Table!Table (
      rows <- Sequence{first_row, thisModule.allMethodDefs->
                collect(e | thisModule.resolveTemp(e, 'row')) }
      ),
      first_row : Table!Row (cells <-
         Sequence{first_col, thisModule.allMethodDefs
              -> collect (e | thisModule.getContentFirstRow(e))} ),
      first_col : Table!Cell (content <- '')
}
rule MethodDefinition {
  from m : JavaSource!MethodDefinition
  to row : Table!Row (cells <- Sequence{title_cel, thisModule.allMethodDefs ->
                collect (e | thisModule.getComputeContent(m, e))} ),
   title_cel : Table!Cell (content <- m.class.name + '.' + m.name
}
lazy rule getContentFirstRow {
  from m : JavaSource!MethodDefinition
  to c : Table!Cell (content <- m.class.name + '.' + m.name)
}
lazy rule getComputeContent{
  from m1 : JavaSource!MethodDefinition,
     m2 : JavaSource!MethodDefinition
  to c : Table!Cell (content <- m1.computeContent(m2).toString() )
}
```

The input model we have used in our transformations examples contains a JavaSource with two ClassDeclarations. It is shown in Fig. 3. Its corresponding target model when the JavaSource2Table transformation is applied over it is the Table model

FirstClass.java

```
public class FirstClass {
   public void fc_m1(){
   }
   public void fc_m2(){
      this.fc_m1();
      this.fc_m1();
   }
}
```

SecondClasss.java

```
public class SecondClass {
   public void sc_m1(){
      FirstClass a = new FirstClass();
      a.fc_m1();
   }
   public void sc_m2(){
      this.sc_m1();
   }
}
```

Figure 3 – JavaSource input model.

	FirstClass.fc_m1	FirstClass.fc_m2	SecondClass.sc_m1	SecondClass.sc_m2
FirstClass.fc_m1	0	0	0	0
FirstClass.fc_m2	2	0	0	0
SecondClass.sc_m1	1	0	0	0
SecondClass.sc_m2	0	0	1	0

Figure 4 – Table target model.

shown in Fig. 4. A visual explanation of how the transformation works is shown in Fig. 5.

2.1 ATL Refining Mode

ATL also defines another execution mode, in case the transformation modifies the source model. This is the ATL *refining mode*, in which the transformation defines the elements that should be changed and how. The rest of the elements in the model are implicitly copied by the ATL engine without modifications.

In the 2004 version of ATL, the copying was performed implicitly *only* for contained elements of copied elements, and it was mandatory to specify all bindings. The effort of copying some elements of a transformation, while modifying others, was reduced in the next version of the ATL language in 2006, which introduced some changes to the refining mode. In this new version every element stays unchanged if it is not explicitly matched by any of the transformation rules. However, the 2006 version did not provide support for the deletion of elements. The new ATL 2010 compiler implements a full in-place strategy, where elements can be deleted by the rules, and also reverse bindings for the first output pattern element are supported. Transformations in this mode are performed in two steps. In the first step, the transformation engine executes the rules which, as a result, produce a set of changes that is temporarily stored. In the second step, this set of changes is applied directly on the source model. Deletion is possible using the "drop" keyword in the to pattern.

Let us illustrate this execution mode using the well-known example of the Public2Private transformation, which makes all public attributes of a UML model private. Getters and setters are also created appropriately, as shown below.

```
module Public2Private;
  create OUT : UML refining IN : UML;
helper context String def : toU1Case : String =
  self.substring(1,1).toUpper() + self.substring(2,self.size());
rule Property {
  from publicAttribute :
```

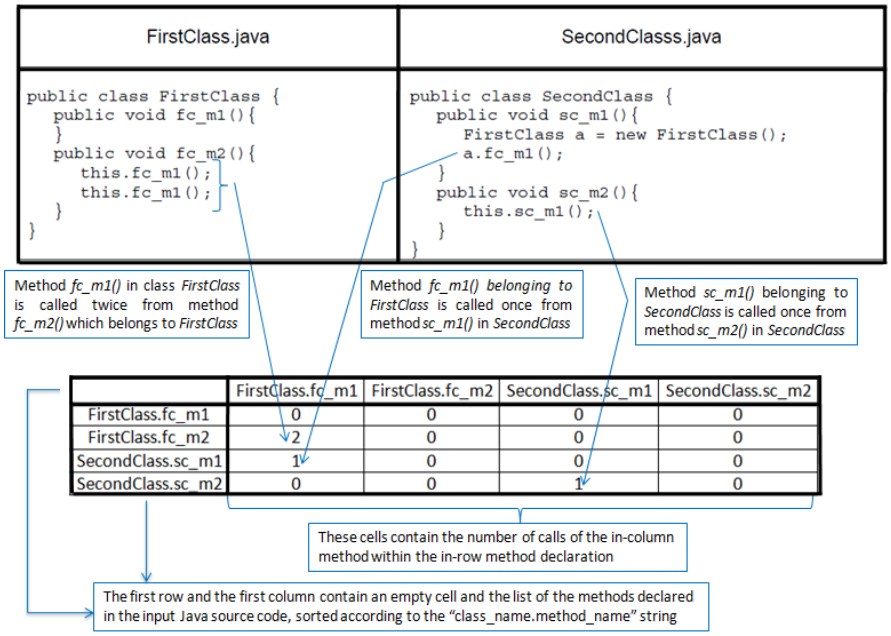


Figure 5 – Visual explanation of the transformation.

```
      UML!Property (publicAttribute.visibility = #public)
  to privateAttribute :
      UML!Property (visibility <- #private),
      getter : UML!Operation (name <- 'get'+publicAttribute.name.toU1Case,
        class <- publicAttribute.refImmediateComposite(),
        type <- publicAttribute.type),
      setter : UML!Operation (name <- 'set'+publicAttribute.name.toU1Case,
        class <- publicAttribute.refImmediateComposite(),
        ownedParameter <- setterParam),
      setterParam : UML!Parameter (name <- publicAttribute.name,
        type <- publicAttribute.type) }
```

Detailed information about this transformation can be found in [Ecl10]. The reference metamodel is a simplification of the UML metamodel with only the relevant information for this example, and is shown in Fig. 6.

3 Rewriting Logic and Maude

Maude [CDE+07] is a high-level language and a high-performance interpreter in the OBJ algebraic specification family that supports membership equational logic [BJM00] and rewriting logic [Mes92] specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. We informally describe in this section those Maude's features necessary for understanding the paper; the interested reader is referred to [CDE+07] for more details.

Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. A system is axiomatized in rewrit-

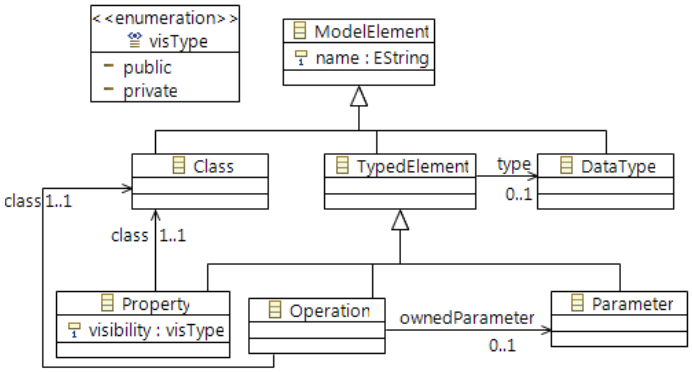


Figure 6 – Metamodel used in the Public2Private transformation.

ing 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 [BJM00], 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.

A condition is written $EqCond_1$ /\ ... /\ $EqCond_n$ 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 left hand side of the corresponding conditional equation, become instantiated by *matching* the term t against the canonical form of the bounded subject term t'.

4 Encoding ATL in Maude

To give a formal semantics to ATL using rewriting logic, we provide a representation of the ATL constructs and of their behavior in Maude. We start by defining how the models and metamodels handled by ATL can be encoded in Maude, and then we provide the semantics of matched rules, lazy rules, unique lazy rules, helpers, imperative sections, the resolveTemp function and the refining execution mode. 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).

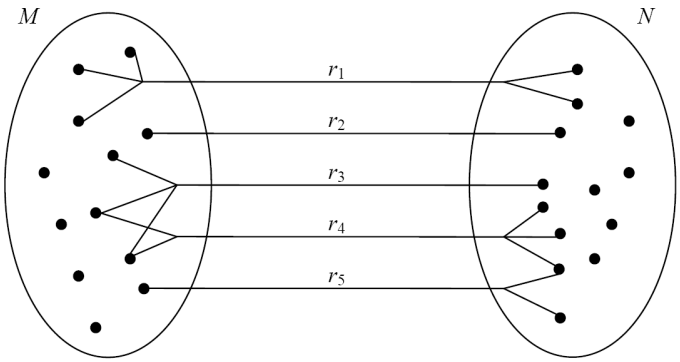


Figure 7 – Elements of a relation $R(M, N)$.

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 $R \subseteq MM_S \times MM_T$ defined between two metamodels which allows to obtain a target model M_T conforming to MM_T from a source model M_S that conforms to metamodel MM_S [Ste07]. In the most general case, a model transformation can be defined between multiple source and target metamodels. In this case MM_S and MM_T represent sets of metamodels.

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

Thus, to represent the structural aspects of a transformation we will use three models: the source model M_S, the target model M_T that the transformation builds, and the relation $R(M_S, M_T)$ between the two. $R(M_S, M_T)$ is also called the *trace* model, that specifies how the elements of M_S and M_T are consistently related by R. Please note that each element r_i of $R(M_S, M_T) = \{r_1, ..., r_k\} \subseteq \mathbb{P}(M_S) \times \mathbb{P}(M_T)$ relates a set of elements of M_S with a set of elements of M_T (see Fig. 7).

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

4.2 Encoding Models and Metamodels in Maude

We will follow the representation of models and metamodels introduced in [RVD09], which is inspired by the Maude representation of object-oriented systems. We represent models in Maude as structures of sort `@Model` of the form $mm\{obj_1\ obj_2\ ...\ obj_N\}$, where mm is the name of the metamodel and obj_i are the objects of the model. An object is a record-like structure $< o : c \mid a_1 : v_1, ..., a_n : v_n >$ (of sort `@Object`), where o is the object identifier (of sort `Oid`), c is the class the object belongs to (of sort

@Class), and $a_i : v_i$ are attribute-value pairs (of sort @StructuralFeatureInstance).

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

```
@javasourcemm@ {
  < 's : JavaSource@javasourcemm |
      classes@JavaSource@javasourcemm : Sequence[ 'c1 ; 'c2 ] >
  < 'c1 : ClassDeclaration@javasourcemm |
      name@NamedElement@javasourcemm : "FirstClass" #
      methods@ClassDeclaration@javasourcemm : Sequence[ 'm1 ; 'm2 ] >
  < 'm1 : MethodDefinition@javasourcemm |
      name@NamedElement@javasourcemm : "fc_m1" #
      invocations@MethodDefinition@javasourcemm : null #
      class@MethodDefinition@javasourcemm : 'c1 >
  < 'm2 : MethodDefinition@javasourcemm |
      name@NamedElement@javasourcemm : "fc_m2" #
      invocations@MethodDefinition@javasourcemm : Sequence [ 'i1 ; 'i1 ] #
      class@MethodDefinition@javasourcemm : 'c1 >
  < 'i1 : MethodInvocation@javasourcemm |
      method@MethodInvocation@javasourcemm : 'm1 >
  < 'c2 : ClassDeclaration@javasourcemm |
      name@NamedElement@javasourcemm : "SecondClass" #
      methods@ClassDeclaration@javasourcemm : Sequence [ 'm3 ; 'm4 ] >
  < 'm3 : MethodDefinition@javasourcemm |
      name@NamedElement@javasourcemm : "sc_m1" #
      invocations@MethodDefinition@javasourcemm : 'i2 #
      class@MethodDefinition@javasourcemm : 'c2 >
  < 'i2 : MethodInvocation@javasourcemm |
      method@MethodInvocation@javasourcemm : 'm1 >
  < 'm4 : MethodDefinition@javasourcemm |
      name@NamedElement@javasourcemm : "sc_m2" #
      invocations@MethodDefinition@javasourcemm : 'i3 #
      class@MethodDefinition@javasourcemm : 'c2 >
  < 'i3 : MethodInvocation@javasourcemm |
      method@MethodInvocation@javasourcemm : 'm3 >
}
```

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 [RD08].

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. Thus, each class is represented by a constant of a sort named after the class. This sort, which will be declared as subsort of sort @Class, is defined to support class inheritance through Maude's order-sorted type structure. Other properties of metamodel elements, such as whether a class is abstract or not, the opposite of a reference (to represent bidirectional associations), or attributes and reference types, are expressed by means of Maude equations. Classes, attributes and references are qualified with their containers' names, so that classes with the same name belonging to different packages, as well as attributes and references of different classes, are distinguished. See [RVD09] for further details.

4.3 Modeling ATL default execution mode

In this subsection all the features of the ATL default execution mode are described. For a more detailed explanation of all the examples shown in this subsection please refer to the extended technical report [TBV10].

4.3.1 Matched rules.

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

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

The two sides of the Maude rule contain the three models that capture the state of the transformation (see 4.1): the source, the trace and the target models.[1] It 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 (using the alreadyExecuted operation) that the same ATL rule has not been triggered with the same elements.

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

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

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

The trace model also contains a special object, of class Counter@CounterMm, whose integer attribute is used as a counter for assigning fresh identifiers to the newly created elements and traces. As an example, consider the MethodDefinition rule shown in section 2. An excerpt of its Maude implementation is shown below. From this excerpt, everything related to the call of the lazy rule has been removed for clarification

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

purposes. The whole implementation is shown and fully explained in the technical report [TBV10].

```
crl[MethodDefinition] :
  Sequence[
    (@JavaSourcemm@ { < M@ : MethodDefinition@javasourcemm | SFS > OBJSET@ }) ;
    (@TraceMm@ {
      < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ >
      OBJSETT@ }) ;
    (@Tablemm@ { OBJSETTT@ }) ]
  => Sequence[
    (@JavaSourcemm@ { < M@ : MethodDefinition@javasourcemm | SFS > OBJSET@ }) ;
    (@TraceMm@ {
      < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 3 >
      < TR@ : Trace@TraceMm | srcEl@TraceMm : Sequence[ M@ ] #
        trgEl@TraceMm : Sequence[ R@ ; TC@ ] # rlName@TraceMm : "MethodDefinition" #
        srcMdl@TraceMm : "JavaSource" # trgMdl@TraceMm : "Table" > OBJSETT@}) ;
    (@Tablemm@ { < R@ : Row@tablemm | cells@Row@tablemm :
                  << Sequence [ TC@ ] -> union ... >> >
      < TC@ : Cell@tablemm | content@Cell@tablemm : << M@ .
        class@MethodDefinition@javasourcemm . name@NamedElement@javasourcemm +
        "." + M@ . name@NamedElement@javasourcemm ; JAVASOURCEMODEL@ >> >
      OBJSETTT@ }) ]
  if JAVASOURCEMODEL@ := @JavaSourcemm@ { < M@ : MethodDefinition@javasourcemm |
                                          SFS > OBJSET@ }
    /\ TR@ := newId(VALUE@CNT@) /\ R@ := newId(VALUE@CNT@ + 1)
    /\ TC@ := newId(VALUE@CNT@ + 2) /\
    /\ not alreadyExecuted(Sequence[M@],"MethodDefinition",@TraceMm@ { OBJSETT@ }).
```

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

After the application of the rule, the state of the system is changed: the source model is left unmodified (ATL does not allow modifying the source models); a new trace (TR@) is added to the trace model; the value of the counter object is updated; and two new elements (R@ and TC@) are created in the target model. We allow the evaluation of OCL expressions using mOdCL [RD08] by enclosing them in double angle brackets (<< ... >>).

The values assigned to the model element features in the bindings in the declarative ATL are determined by the so-called *resolution algorithm*. If the value is a simple value or an element from the target model then the value is used as is. If the value is an element from the source model then the ATL engine tries to obtain a target element by using the traces. If it is not possible to find such a target element then the source element is used. This allows cross-model references and is an important feature of ATL. This resolution algorithm is represented in Maude by one operation called getTargetEl that, given an OCL expression which points to an (source or target) element and the trace model, returns the appropriate element.

```
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 }) = SRC@ [owise] .
```

Since the OCL expression may point to more than one element, the resolution algorithm has to take all of them into account. This is the reason of the following function:

```
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 applies the getTargEl function to every element in the sequence, and returns the sequence with all the elements retrieved by the getTargetEl function.

4.3.2 Lazy rules.

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

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

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

4.3.3 Unique lazy rules.

This kind of rules deserve a special representation in Maude, because their behavior is quite different from normal lazy rules. In this case, we need to check if the elements created by the lazy rule are already there, otherwise they have to be created. If they were already there, we need to get their identifiers. 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.

4.3.4 Helpers.

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

```
op allMethodDefs : @Model -> Sequence .
 eq allMethodDefs(JAVASOURCEMODEL@) =
  << MethodDefinition@javasourcemm . allInstances ->
  sortedBy( ITER | << ITER . class@MethodDefinition@JavaSourceMM .
  name@NamedElement@JavaSourceMM + "_" + ITER .
  name@NamedElement@JavaSourceMM ; JAVASOURCEMODEL@ >>) ; JAVASOURCEMODEL@ >> .
```

This helper receives the source model, JavaSource, as argument. It builds the sequence of all method definitions in all existing classes. The sequence it returns, of type MethodDefinition, is ordered according to their class name and method name.

4.3.5 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 four types of instructions: *assignments* (=), *conditional* branches (if), loops (for) and *called rules*. In the following piece of Maude code we show how Instr type and the sequence of instructions (InstrSequ) are defined:

```
sort Instr InstrSequ .
subsort Instr < InstrSequ .
op none : -> InstrSequ [ctor] .
op _^_ : Instr InstrSequ -> InstrSequ [ctor id: none] .
op Assign : Oid @StructuralFeature OCL-Exp -> Instr [ctor] .
op If : Bool InstrSequ InstrSequ -> 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 NewTable : Int String -> Instr [ctor] .
```

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

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

```
op do : Set{@Object} InstrSequ -> Set{@Object} .
  eq do(OBJSET@, none) = OBJSET@ .
  eq do(OBJSET@, Assign(O@, SF@, EXP@) ^ INSTR@) =
     do(doAssign(OBJSET@, O@, SF@, EXP@), INSTR@) .
  eq do(OBJSET@, If(COND@, INSTR1@, INSTR2@) ^ INSTR@) =
   if COND@ then do(OBJSET@, INSTR1@ ^ INSTR@)
   else do(OBJSET@, INSTR2@ ^ INSTR@)
   fi .
  ---For each called rule, AddColumn in this case
  eq do(OBJSET@, doNewTable(VALUE@CNT@, NAME) ^ INSTR@) =
     do(doNewTable(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 doAssign and doNewTable operations 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 doNewTable : Set{@Object} Int String -> Set{@Object} .
eq doNewTable(OBJSET@, VALUE@CNT@, NAME) =
  < newId(VALUE@CNT@) : Table@tablemm | rows@Table@tablemm : newId(VALUE@CNT@+1) >
  < newId(VALUE@CNT@ + 1) : Row@tablemm|cells@Row@tablemm : newId(VALUE@CNT@+2) >
  < newId(VALUE@CNT@ + 2) : Cell@tablemm | content@Cell@tablemm : NAME >
  OBJSET@ .
```

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

The following code shows how the do function works when the instruction is a For:

```
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@) .
```

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@, TARGETMODEL@, EXP@) ^
       INSTR1@) ^ INSTR@) = do(doAssign(OBJSET@, AT@, SF@, << AT@ . SF1@
            ; TARGETMODEL@ >> + 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 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@, TARGETMODEL@, INSTR1@,
   INSTR2@) ^ INSTR3@) ^ INSTR@) =
     if (ST@ == "==") then
```

```
      if (<< AT@ . SF@ ; TARGETMODEL@ >> == 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@ ; TARGETMODEL@ >> >= 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@ ; TARGETMODEL@ >> <= 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@ ; TARGETMODEL@ >> > 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@ ; TARGETMODEL@ >> < 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@ ; TARGETMODEL@ >> =/= 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, 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 feature 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 show the imperative constructs all together in one ATL rule, consider the following one, that contains an imperative part to modify the elements that have been created by the rule:

```
rule Main {
  from s : JavaSource!JavaSource
  to c : Table!Table(rows <- Sequence{row}),
     row : Table!Row(cells <- Sequence{cell1, cell2, cell3}),
     cell1 : Table!Cell(content <- 'FirstCell'),
```

```
    cell2 : Table!Cell(content <- 'SecondCell'),
    cell3 : Table!Cell(content <- 'ThirdCell')
  do{
    cell1.content <- cell1.content + '_assignment';
    if (row.cells -> size() = 3) {
      cell2.content <- 'Condition_satisfied';
    } else {
      cell2.content <- 'Condition_not_satisfied';
    }
    for (i in row.cells) {
      i.content <- i.content + '_assign_for';
      if (i.content = 'ThirdCell_assign_for'){
        i.content <- i.content + '_if_for_satisfied';
      } else {
        i.content <- i.content + '_if_for_not_satisfied';
      }
      i.content <- i.content + '_after_if_for';
    }
    thisModule.NewTable('NewTable');
  }
}
```

The corresponding encoding in Maude is as follows:

```
crl[Main] :
  Sequence[...] =>
  Sequence[...
    (@Tablemm@ { do (
      < T@ : Table@tablemm | rows@Table@tablemm : R@ >
      < R@ : Row@tablemm | cells@Row@tablemm : Sequence [ C1@ ; C2@ ; C3@ ] >
      < C1@ : Cell@tablemm | content@Cell@tablemm : "FirstCell" >
      < C2@ : Cell@tablemm | content@Cell@tablemm : "SecondCell" >
      < C3@ : Cell@tablemm | content@Cell@tablemm : "ThirdCell" >,
      Assign(C1@, content@Cell@tablemm,
        << C1@ . content@Cell@tablemm ; TABLEMODEL@ >> + "_assignment") ^
      If(<< Sequence[C1@ ; C2@ ; C3@] -> size() ; JAVASOURCEMODEL@ >> == 3,
        Assign(C2@, content@Cell@tablemm, "Condition_satisfied"),
        Assign(C2@, content@Cell@tablemm, "Condition_not_satisfied")) ^ --- endIf
      For(<< R@ . cells@Row@tablemm ; TABLEMODEL2@ >>,
        AssignAttFor(content@Cell@tablemm, content@Cell@tablemm, TABLEMODEL2@,
          "_assign_for") ^
        IfFor("==", content@Cell@tablemm, "ThirdCell_assign_for", TABLEMODEL3@,
          AssignAttFor(content@Cell@tablemm, content@Cell@tablemm, TABLEMODEL3@,
            "_if_for_satisfied"),
          AssignAttFor(content@Cell@tablemm, content@Cell@tablemm, TABLEMODEL3@,
            "_if_for_not_satisfied") ) ^ --- endIfFor
        AssignAttFor(content@Cell@tablemm, content@Cell@tablemm, TABLEMODEL4@,
          "_after_if_for") ) ^ --- endFor
      NewTable(VALUE@CNT@ + 6, "NewTable"))
      OBJSETTT@ }
  )
] if...
```

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

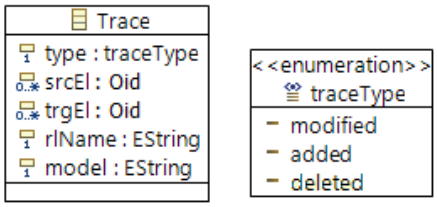


Figure 8 – Trace class.

executed before the imperative part. This is the way in which ATL works. The second argument is a sequence of instructions. It 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, a For instruction, containing three instructions (two assignments and a if block), is executed. Finally, the instruction that represents the called rule, NewTable, is executed.

4.3.6 ResolveTemp

The resolveTemp function looks for the trace that contains the source element passed as first argument, and returns the identifier of the element from the sequence of elements created from the source element. Its representation in Maude is as follows:

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

It has four arguments: the identifier of the source model element from which the searched target model element is produced; the position of the target object identifier in the sequence trgEl@TraceMm; and the trace and class models, respectively. It returns the identifier of the element to be retrieved. The major difference with the ATL function is that here we receive as second argument the position that the searched target model element has among the ones created by the corresponding rule. In ATL, instead, the argument received is the name of the variable that was given to the target model element when it was created. This deviation from ATL is merely due to technical reasons: we do not use variable names in this function because we do not store variable names in traces. A trace contains a sequence with the identifiers of the target elements that were created from the source elements. As it is a sequence, it is ordered and, consequently, every element identifier has a position within the sequence. Therefore, the difference of passing as argument the position of the element identifier in the sequence instead of the variable name is not significant since it is easy to retrieve the position of the element among those created by the ATL rule.

4.4 ATL refining mode in Maude

As explained in section 2.1, the ATL refining execution mode transforms the elements identified by the source patterns according to the behaviour defined in the rules. Those model elements that are not explicitly affected by the rules (either directly or indirectly) remain unchanged.

The semantics of this ATL execution mode can be specified in Maude in a similar way to the one used to specify the normal execution mode. However, the traces will be treated in a slightly different manner, because we do not need to specify and maintain traces between elements that have not been modified by the transformation. In fact, traces in this execution mode can be considered as *model differences* between the elements of both models: the old and new versions of the model being transformed. Thus, we have defined each trace as an instance of the class shown in Fig. 8, which follows the approach used in [RV08].

Traces of type modified represent the transformation of an object from the source model into another object in the target model where at least one of its attributes (or references) have been modified. Traces whose type is added, in turn, represent the addition of a new object (or more than one) in the target model.

In the 2010 implementation of the refining mode, ATL allows to remove objects. To represent this new feature we have traces of type deleted, whose source elements (srcEl) are the deleted objects, and the set of target elements (trgEl) is empty. A Maude rule that represents an ATL rule where objects are deleted simply contains these objects in its left hand side, but not in its right hand side, and it creates a trace of type deleted as mentioned above.

In the encoding of a transformation in refining mode in Maude, the Init rule is also different, since now the source model is copied into the target one. Thus, in case of the Public2Private example introduced in section 2.1, this rule is as follows:

```
rl [Init] :
  Sequence[(@UMLSimpMm@ { OBJSET@ })]
  =>
  Sequence[(@UMLSimpMm@ { OBJSET@ }) ;
    (@TraceMm@ {< 'CNT : Counter@CounterMm | value@Counter@CounterMm : 1 >}) ;
    (@UMLSimpMm@ { OBJSET@ })] .
```

After the application of this rule, both the source and target models contain the same elements. Then, the target model is modified as the ATL matched rules are executed, "navigating" the source model.

Models navigability and in-place transformations

The ATL documentation [Gro06] states that, both in normal and in refining execution modes, source models are read-only and target models are write-only. This means that only source models can be navigated and, therefore, the state of the target model does not affect the behavior of the transformation.

This is an important detail that significantly affects the way in which ATL works in refining mode. In fact, it is a common mistake to confuse the behavior of the ATL refining mode with the typical behavior of the in-place transformations used by most rewriting systems, including graph grammars or even Maude rules. In these rewriting systems, a set of rules modifies the state of a configuration of objects (i.e., a model) one by one. Thus, after the application of each rule the state of the system is changed, and subsequent rules will be applied on the system on this new state. In this way, the target model after the application of one rule becomes the source model in the next step. In other words, the transformation navigates the target model, which is continuously updated by every executed rule. However, this is not the way in which the ATL refining mode works. In ATL the rules always read (i.e., navigate) the state of the source model, which remains unchanged during all the transformation execution. This is the approach we have followed in our representation in Maude, too.

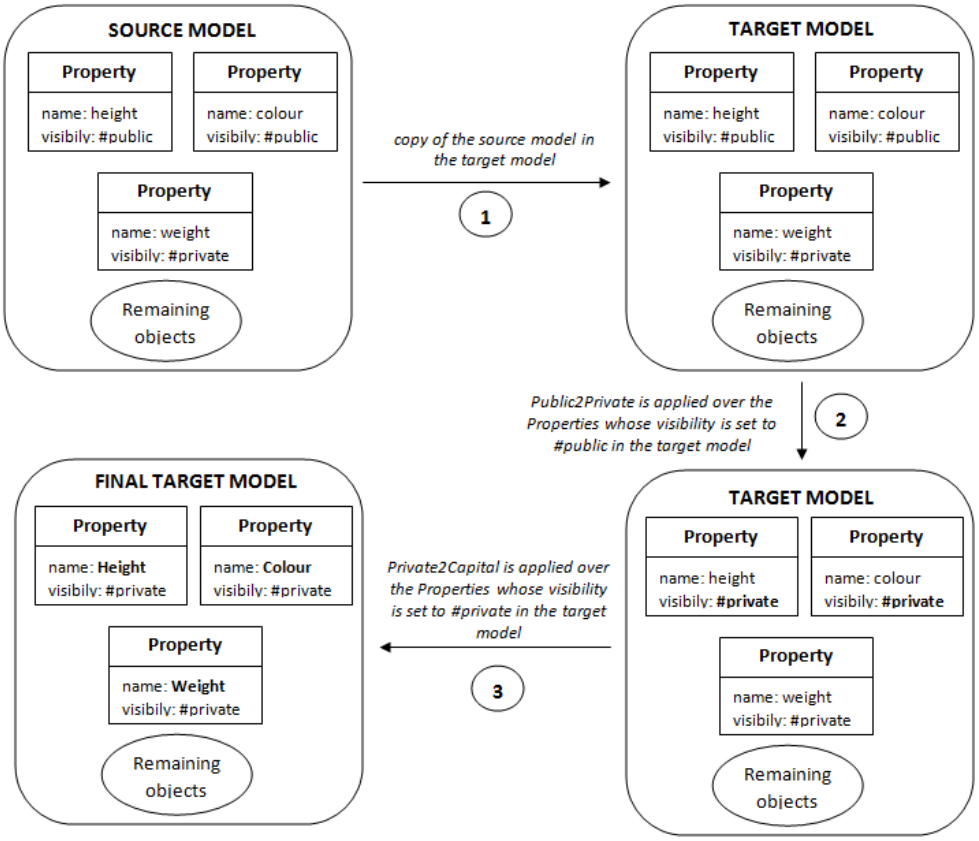


Figure 9 – In-place behavior: navigability on the target model.

In order to illustrate this difference, let us go back to the Public2Private transformation and imagine that we add another matched rule that changes all private Properties, capitalizing the first letter of their names. Let us call this rule Private2Capital.

If ATL worked in a pure in-place manner (i.e., navigating the target model), the transformation would change the names of all properties: all of them will end up being private and with the first letter of their names in capitals (see the example shown in Fig. 9). However, the ATL refining mode navigates the source model. This means that, at the end of the execution of the Public2Private transformation, only those properties that were originally private in the source model will have their names capitalized, while the original public properties in the model will be transformed into private properties but their names will not be changed (see Fig. 10).

5 Simulation and Formal Analysis

Once the ATL model transformation specifications are encoded in Maude, what we get is a rewriting logic specification for it. Maude offers tool support for interesting possibilities such as model simulation, reachability analysis and model checking [CDE+07].

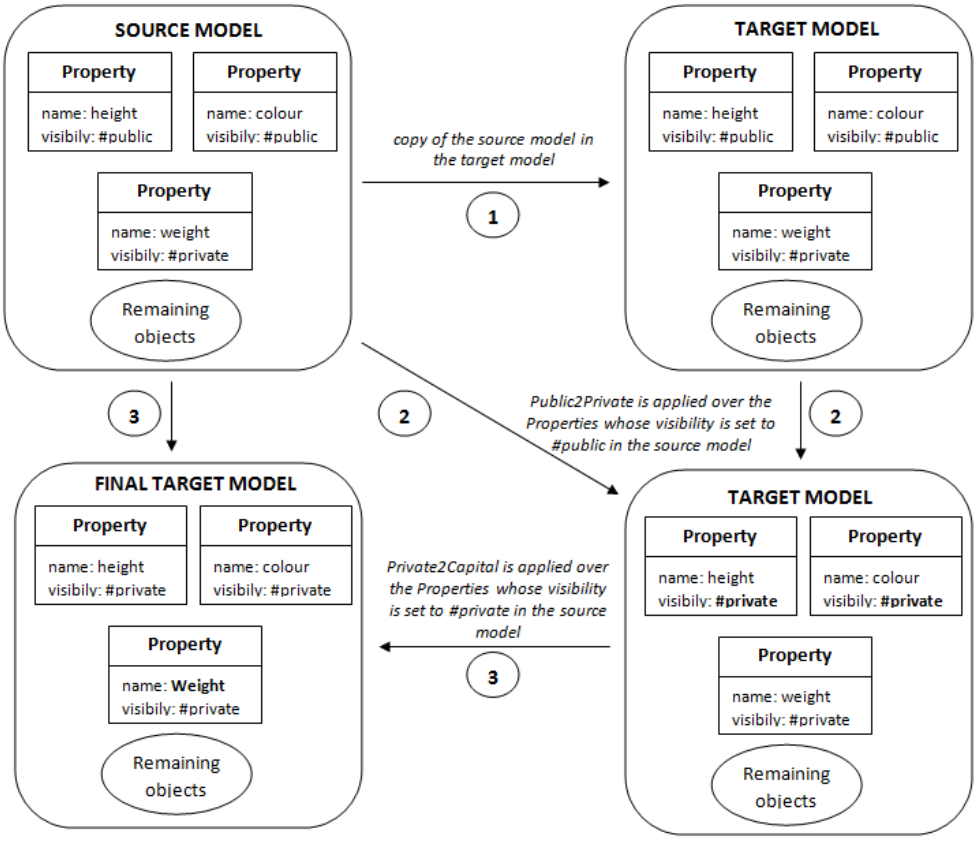


Figure 10 – ATL refining mode: navigability always on the source model.

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 performing 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 [CDE+07]. The result of the process is the final configuration of objects reached after the rewriting steps, which is nothing but a model.

For example, the JavaSource2Table ATL model transformation described in section 2, when executed in default mode over the JavaSource source model shown in Fig. 3, results in a sequence of three models: the source, the trace and the target model. The encoding in Maude of this last one, which conforms to the Table metamodel and is displayed in Fig. 4, is shown below.

```
@JavaSourceMm@ {
  < 's : JavaSource@javasourcemm |
      classes@JavaSource@javasourcemm : Sequence[ 'c1 ; 'c2 ] >
  < 'c1 : ClassDeclaration@javasourcemm |
      name@NamedElement@javasourcemm : "FirstClass" #
```

```
    methods@ClassDeclaration@javasourcemm : Sequence[ 'm1 ; 'm2 ] >
< 'm1 : MethodDefinition@javasourcemm |
    name@NamedElement@javasourcemm : "fc_m1" #
    invocations@MethodDefinition@javasourcemm : null #
    class@MethodDefinition@javasourcemm : 'c1 >
< 'm2 : MethodDefinition@javasourcemm |
    name@NamedElement@javasourcemm : "fc_m2" #
    invocations@MethodDefinition@javasourcemm : Sequence [ 'i1 ; 'i1 ] #
    class@MethodDefinition@javasourcemm : 'c1 >
< 'i1 : MethodInvocation@javasourcemm |
    method@MethodInvocation@javasourcemm : 'm1 >
< 'c2 : ClassDeclaration@javasourcemm |
    name@NamedElement@javasourcemm : "SecondClass" #
    methods@ClassDeclaration@javasourcemm : Sequence [ 'm3 ; 'm4 ] >
< 'm3 : MethodDefinition@javasourcemm |
    name@NamedElement@javasourcemm : "sc_m1" #
    invocations@MethodDefinition@javasourcemm : 'i2 #
    class@MethodDefinition@javasourcemm : 'c2 >
< 'i2 : MethodInvocation@javasourcemm |
    method@MethodInvocation@javasourcemm : 'm1 >
< 'm4 : MethodDefinition@javasourcemm |
    name@NamedElement@javasourcemm : "sc_m2" #
    invocations@MethodDefinition@javasourcemm : 'i3 #
    class@MethodDefinition@javasourcemm : 'c2 >
< 'i3 : MethodInvocation@javasourcemm |
    method@MethodInvocation@javasourcemm : 'm3 > }
```

Although the Maude specifications can be used for running the ATL transformations, and not only for simulating it, the performance of the Maude specifications is not comparable with ATL (see Section 5.4). However, our proposal does not try to compete with ATL in this respect. It is not the goal of this work to use Maude for implementing model transformations, but for providing semantics to ATL. The fact that Maude specifications are executable gives us an implementation of the transformation, but such an implementation is not intended to be used as an alternative to ATL in practice—just for verification purposes.

5.2 Reachability analysis

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

Maude search command allows us to explore (following a breadthfirst strategy up to a specified bound) the reachable state space in different ways, looking for certain states of special interest. Other possibilities would include searching for any state

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

(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.

5.3 Checking other properties

After the simulation is completed, it is also possible to 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 to specify because of Maude's facilities for manipulating sets using order-sorted unification modulo associativity and commutativity:

```
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] .
```

We can also use a similar operation to traverse the trace model and check that every source element has been transformed by at most one ATL match rule. In fact, in ATL only one matched rule can be applied on a given model element (this ensures some kind of confluence of the ATL rules, too).

In general there are two ways of dealing with such constraints, depending on whether we want the Maude rules to enforce them during their execution or not. In the first case these constraints will be added to the Maude rules. In the second case the mapping to Maude will not consider them, so these situations will occur if they happen in the ATL code. But we will be able to check them once the transformation is done by exploring the trace model, as mentioned above.

For example, to check that no more than one ATL rule is applied over a single source element we have defined the singleApplicability operation:

```
op singleApplicability : OCL-Exp String @Model Int -> @Object .
  eq singleApplicability(SR@, NAME, @TraceMm@ { < TR@ : Trace@TraceMm |
    srcEl@TraceMm : Sequence[SR@ ; LO] # rlName@TraceMm : NAME' # SFS > OBJSET},
    VALUE@CNT@) =
    if NAME =/= NAME' then
      < newId(VALUE@CNT@) : TraceSA@TraceMm | rl1Name@TraceMm : NAME #
        rl2Name@TraceMm : NAME' # srcEl@TraceMm : SR@ # errMsg@TraceMm : "Rules " +
        NAME + " and " + NAME' + " are applied over the same source element: " +
        SR@ >
      else singleApplicability(SR@, NAME, @TraceMm@ { < TR@ : Trace@TraceMm |
        srcEl@TraceMm : Sequence[LO] # rlName@TraceMm : NAME' # SFS > OBJSET},
        VALUE@CNT@)
      fi .
  eq singleApplicability(SR@, NAME, TRACEMODEL@, VALUE@CNT@) = none [owise] .
```

This operation receives as arguments a source model element, a string with the name of the rule from which the function is called, the trace model and the counter

to create new identifiers. The function looks for a trace created by a different rule (we check that the names of the rules are different: NAME =/= NAME') where the element received as argument is present in the srcEl@TraceMm part.

The problem, as usual, is what to do when a problem is encountered during the execution of the rule. In this case our encoding generates a special kind of trace (TraceSA@TraceMm), which captures every error found. Such traces store the problematic rule names (those whose left hand side parts contain the same source model element), the source model element and an error message.

As an example of how this function is called, we add a call to it in the MethodDefinition matched rule that we presented above. We can see the call in the next listing:

```
crl[MethodDefinition] :
  Sequence[
    (@JavaSourcemm@ { < M@ : MethodDefinition@javasourcemm | SFS > OBJSET@ }) ;
    (@TraceMm@ { <CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@>
       OBJSETT@ }) ;
    (@Tablemm@ { OBJSETTT@ }) ]
  => Sequence[
    (@JavaSourcemm@ { < M@ : MethodDefinition@javasourcemm | SFS > OBJSET@ }) ;
    (@TraceMm@ {
      < CNT@ : Counter@CounterMm | value@Counter@CounterMm : VALUE@CNT@ + 4 >
      ...
      singleApplicability(M@, "MethodDefinition", @TraceMm@{OBJSETT@},VALUE@CNT@)
      OBJSETT@}) ;
    (@Tablemm@ { ... }) ]
  if JAVASOURCEMODEL@ := ...
    /\ TR@ := newId(VALUE@CNT@ + 1) /\ R@ := newId(VALUE@CNT@ + 2)
    /\ TC@ := newId(VALUE@CNT@ + 3) /\
    /\ not alreadyExecuted(Sequence[M@],"MethodDefinition",@TraceMm@ { OBJSETT@ }).
```

5.4 Questions of efficiency

Another improvement over the proposal presented in [TV10a] is the use of a more compact encoding of the Maude representation of the ATL rules. Maude is a very expressive language, which allows many different ways to represent the same concepts or the same behaviors. Each encoding, although functionally and semantically equivalent, may be different regarding other non-functional aspects such as performance, readability or understandability, among others.

In the previous sections we have shown the encoding that was also used in [TV10a]. This encoding is rather natural (to the Maude users) and convenient for representing the behavior of ATL constructs and rules. However, when it comes to simulating and analyzing the specifications, it may be significantly improved in several ways.

The aim of the modifications is to remove as many guards as possible from the Maude rules, so that the rewrite process does not need to evaluate conditions for triggering them. Thus, we have avoided the use of auxiliary variables that were declared as guards with the ":=" operator by replacing them with their corresponding expressions in the places where these auxiliary variables were used. We have also got rid of the AlreadyExecuted function, which had to navigate the trace model in each rule invocation, by introducing an auxiliary model in the Maude rules that contains the elements from the input model that have not yet been transformed by ATL rules.

	Original encoding	Optimized encoding	ATL
125 Classes, 500 Attributes	15”	4”	0.3”
250 Classes, 1000 Attributes	1’37”	15”	0.5”
375 Classes, 1500 Attributes	5’53”	40”	0.8”
500 Classes, 2000 Attributes	16’09”	1’37”	1.1”
750 Classes, 3000 Attributes	58’28”	4’02”	2”
1250 Classes, 5000 Attributes	3h16’49”	16’37”	3”
2000 Classes, 8000 Attributes	17h57’15”	1h04’19”	5”

Table 1 – Comparative performance figures.

This new model initially coincides with the input model of the transformation and, when a rule is executed on a set of elements, these elements are removed from the model. We have removed the evaluation of some OCL expressions in the conditions of the Maude rules by checking them in the input model. We have also tried to avoid the use of OCL expressions in the right hand side of Maude rules when initializing objects attributes in the target model by specifying variables for the values of the objects' attributes in the input model. Please refer to [TBV10] for a complete description of the performed modifications.

This alternative encoding provides significant improvements in efficiency and performance, as shown in Table 1 for the ATL Class2Relational transformation [TV10a]. Still, it is not comparable to the performance of the equivalent ATL transformation (shown in the last column).

The problem is that the new Maude encoding is much more verbose and less easy to read and understand. However, this new encoding can be automatically obtained from the previous one, hence allowing an automatic transformation from one to the other. This is why we have detailed here the original encoding, because it is functionally equivalent and much easier to read and understand.

6 Related Work

The definition of a formal semantics for ATL has received attention by different groups, using different approaches. For example, in [dRJK+06] 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 [BS06, ABK07] have proposed the use of Alloy to formalize and analyze graph transformation systems, and in particular ATL. These analysis 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 analysis 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 [ABK07]: it has a simple type system with only integers; models in Alloy are static, and thus the approach presented in [ABK07] 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 [Poe08] 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 relies on a simple encoding of MetaClasses as mixed inductive/co-inductive structured types which the current proof assistant does not handle well due to structural guard constraints for co-inductive definitions [PM10]. Alternative approaches encode models as graph covering trees and additional links [GS10, GSMP11] or as a classical mathematical graphs relying on nodes and relations between nodes [TCCG07]. These approaches 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 [Cou97] proposes a combination of graph grammars with second order monadic logic to study graph properties and their transformations. Schürr [SWZ99] 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 [BHM09] 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.

In this paper we have dealt with all new features of ATL version 3.0, and in particular we have formalized the ATL refining mode. Many works have been dedicated to the semantics of the default execution mode, but no one seems to be focused on the refining mode despite the importance this execution mode is gaining. For example, Tisi et al. propose in [TCJ10] the use of this execution mode to implement Higher-Order Transformations (HOTs). They are model transformations that analyze, produce or manipulate other model transformations [TJF+09]. Writing HOTs is generally considered a time-consuming and error-prone task, and often results in verbose code. Refining mode is used in [TCJ10] to facilitate the definition of HOTs in ATL, and they recommend the developers to consider in-place refining mode for every transformation modification and (de)composition.

Finally, Maude has been proposed as a formal notation and environment for specifying and effectively analyzing models and metamodels [RVD09, BM08]. Simulation, reachability and model-checking analysis are possible using the tools and techniques provided by Maude [RVD09]. We build on these works, making use of one of these formalizations to represent the models and metamodels that ATL handles.

7 Conclusions and Future Work

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

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

In addition to the analysis possibilities mentioned here, the use of rewriting logic and Maude opens up the way to using many other tools for ATL transformations in the Maude formal environment. In this respect, we are trying to make use of the Maude Termination Tool (MTT) [DLM08] and the Church-Rosser Checker (CRC) [DM10] for checking the termination and confluence of ATL specifications.

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

References

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M. Küster. Analysis of Model Transformations via Alloy. In Benoit Baudry, Alain Faivre, Sudipto Ghosh, and Alexander Pretschner, editors, *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[BHM09] Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting logic semantics and verification of model transformations. In *Proc. of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, pages 18–33. Springer-Verlag, 2009. doi:10.1007/978-3-642-00593-0_2.

[BJM00] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1):35–132, 2000. doi:10.1007/BFb0030589.

[BM08] Artur Boronat and José Meseguer. An algebraic semantics for MOF. In *Proc. of FASE'08*, volume 4961 of *LNCS*, pages 377–391. Springer, 2008. doi:10.1007/978-3-540-78743-3_28.

[BS06] Luciano Baresi and Paola Spoletini. On the use of Alloy to analyze graph transformation systems. In *Proc. of ICGT'06*, number 4178 in LNCS, pages 306–320. Springer, 2006. doi:10.1007/11841883_22.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, Heidelberg, Germany, 2007. doi:10.1007/978-3-540-71999-1.

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of graph grammars and computing by graph transformation. Vol. I: Foundations*, pages 313–400, 1997.

[DLM08] Francisco Durán, Salvador Lucas, and José Meseguer. MTT: The Maude Termination Tool (System Description). In *Proc. of the 4th international joint conference on Automated Reasoning (IJCAR'08)*, volume 5195 of *LNAI*, pages 313–319, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-71070-7_27.

[DM10] Francisco Durán and José Meseguer. A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In *Proc. of WRLA 2010*, volume 6381 of *LNCS*, pages 69–85. Springer, 2010. doi:10.1007/978-3-642-16310-4_6.

[dRJK+06] Davide di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio. Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique, Nantes, France, April 2006.

[Ecl10] Eclipse M2M Project. ATL, 2010. http://www.eclipse.org/m2m/atl/atlTransformations/.

[Gro06] Atlas Group. *ATL: Atlas Transformation Language, ATL User Manual*. LINA and INRIA, 2006. http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual%5Bv0.7%5D.pdf.

[GS10] Mathieu Giorgino and Martin Strecker. BDDs verified in a proof assistant (Preliminary report). In *Proc. of Theoretical and Applied Aspects of Program Systems Development (TAAPSD 2010)*, Univ. Taras Shevchenko, Kiev (Ukraine), October 2010. Presses universitaires de l'Universit Taras Shevchenko. http://www.irit.fr/~Martin.Strecker/Publications/taapsd10.html.

[GSMP11] Mathieu Giorgino, Martin Strecker, Ralph Matthes, and Marc Pantel. Verification of the Schorr-Waite algorithm – from trees to graphs. In *Logic-Based Program Synthesis and Transformation - 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers*, volume 6564 of *LNCS*, pages 67–83. Springer, 2011. doi:10.1007/978-3-642-20551-4_5.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. Available from: http://www.sciencedirect.com/science/article/B6V17-4SFXK3H-1/2/0fa2857e74bc5ed41ca54eea199d9c17, doi:DOI:10.1016/j.scico.2007.08.002.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. doi:10.1016/0304-3975(92)90182-F.

[PM10] Celia Picard and Ralph Matthes. Coinductive graph representation: the problem of embedded lists. In Rachid Echahed, Annegret Habel, and Mohamed Mosbah, editors, *Proc. of Graph Computation Models (GCM 2010)*, pages 133–148, Enschede, The Netherlands, October 2010. Available from: `http://gcm-events.org/gcm2010/pages/gcm2010-preproceedings.pdf`.

[Poe08] Iman Poernomo. Proofs-as-model-transformations. In *Proc. of ICMT'08*, volume 5063 of *LNCS*, pages 214–228, Zurich, Switzerland, 2008. Springer. `doi:10.1007/978-3-540-69927-9_15`.

[RD08] Manuel Roldán and Francisco Durán. Representing UML models in mOdCL. Technical Report. `http://maude.lcc.uma.es/mOdCL`, 2008.

[RV08] José E. Rivera and Antonio Vallecillo. Representing and operating with model differences. In *Proc. of TOOLS 2008*, volume 11 of *LNBIP*, pages 141–160, Zurich, Switzerland, June 2008. Springer. `doi:10.1007/978-3-540-69824-1_9`.

[RVD09] José E. Rivera, Antonio Vallecillo, and Francisco Durán. Formal specification and analysis of domain specific languages using Maude. *Simulation: Transactions of the Society for Modeling and Simulation International*, 85(11/12):778–792, 2009.

[Ste07] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proc. of MODELS 2007*, volume 4735 of *LNCS*, pages 1–15. Springer, October 2007. `doi:10.1007/978-3-540-75209-7_1`.

[SWZ99] Andy Schürr, Andreas J. Winter, and Albert Zündorf. The PROGRES approach: language and environment. In *Handbook of graph grammars and computing by graph transformation. Vol. II: Applications, languages, and tools*, pages 487–550, 1999.

[TBV10] Javier Troya, José M. Bautista, and Antonio Vallecillo. *A Rewriting Logic Semantics for ATL (Extended Version)*. Universidad de Málaga, November 2010. `http://atenea.lcc.uma.es/Descargas/ATLinMaudeJOT.pdf`.

[TCCG07] Xavier Thirioux, Benoît Combemale, Xavier Crégut, and Pierre-Loïc Garoche. A Framework to Formalise the MDE Foundations. In *International Workshop on Towers of Models (TOWERS 2007)*, pages 14–30, Zurich, June 2007.

[TCJ10] Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Improving higher-order transformations support in ATL. In *Proc. of ICMT 2010*, volume 6142 of *LNCS*, pages 215–229, Málaga, Spain, June 28-29 2010. Springer. `doi:10.1007/978-3-642-13688-7_15`.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *Proc. of MDA-FA 2009*, volume 5562 of *LNCS*, pages 18–33. Springer, 2009. `doi:10.1007/978-3-642-02674-4_3`.

[TV10a] Javier Troya and Antonio Vallecillo. Towards a rewriting logic semantics for ATL. In *Proc. of ICMT 2010*, volume 6142 of *LNCS*, pages

230–244, Málaga, Spain, June 28-29 2010. Springer. doi:10.1007/978-3-642-13688-7_16.

[TV10b] Javier Troya and Antonio Vallecillo. *Towards a Rewriting Logic Semantics for ATL (Extended Version)*. Universidad de Málaga, January 2010. Technical Report. http://atenea.lcc.uma.es/Descargas/ATLinMaude.pdf.

About the authors

Javier Troya is PhD student at the Department of Computper Science at the University of Málaga, where he received a MSc Degree in Computer Science in 2008. His research interests include Model-Driven Software Development and its industrial applications, as well as the formal semantics of model transformation languages. For further information, please visit http://www.lcc.uma.es/~jtc or contact him at javiertc@lcc.uma.es.

Antonio Vallecillo is Professor of Computer Science at the University of Málaga. His research interests include Open Distributed Processing, Model-Based Engineering, Componentware, Software Quality, and the industrial use of formal methods. For further information about his research projects and publications, please visit http://www.lcc.uma.es/~av or contact him at av@lcc.uma.es.

Acknowledgments The authors would like to thank Franciso Durán and José E. Rivera for their comments and suggestions on the paper, and to Jordi Cabot and Salvador Martínez for helping us understand the precise behaviour of the ATL refining mode. We would also like to thank the reviewers for their insightful and constructive comments and suggestions. Finally, we need to acknowledge José Bautista for developing the ATL transformations between ATL and Maude that implement the encoding. This work has been partially supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184.