Transforming Models with ATL[1]

Frédéric Jouault, Ivan Kurtev

ATLAS Group (INRIA & LINA, University of Nantes)
{frederic.jouault | ivan.kurtev}@univ-nantes.fr

Abstract. This paper presents ATL (ATLAS Transformation Language): a hybrid model transformation language that allows both declarative and imperative constructs to be used in transformation definitions. The paper describes the language syntax by using examples. Language semantics is described in pseudo-code and various optimizations of transformation executions are discussed. ATL is supported by a set of development tools such as an editor, a compiler, a virtual machine, and a debugger. A case study shows the applicability of the language constructs. Alternative ways for implementing the case study are outlined. ATL language features are classified according to a model that captures the commonalities and variabilities of the model transformations domain.

1 Introduction

Model transformations play an important role in Model Driven Engineering (MDE) approach. It is expected that writing model transformation definitions will become a common task in software development. Software engineers should be supported in performing this task by mature tools and techniques in the same way as they are supported now by IDEs, compilers, and debuggers in their everyday work.

One direction for providing such a support is to develop domain-specific languages designed to solve common model transformation tasks. Indeed, this is the approach that has been taken recently by the research community and software industry. As a result a number of transformation languages have been proposed. We observe that, even though the problem domain of these languages is fixed, they still differ in the employed programming paradigm. Current model transformation languages usually expose a synthesis of paradigms already developed for programming languages (declarative, functional, object-oriented, imperative, etc.). It is not clear if a single approach will prevail in the future. A deeper understanding and more experience based on real and non-trivial problems is still necessary. We believe that different approaches are suitable for different types of tasks. One class of problems may be easily solved by a declarative language, while another class is more amenable to an imperative approach.

In this paper we describe a transformation language and present how different programming styles allowed by this language may be applied to solve different types

[1] Work partially supported by ModelWare, IST European project 511731.

of problems. The language is named ATL (ATLAS Transformation Language) and is developed as a part of the AMMA (ATLAS Model Management Architecture) platform [3]. ATL is a hybrid language, i.e. it is a mix of declarative and imperative constructs.

We present the syntax and semantics of ATL by using a case study. Whenever necessary a more formal description is provided. For some aspects of the case study several solutions are discussed. This helps in identifying patterns and potential obstacles in defining ATL transformations. Based on this knowledge a software developer is aware of various possibilities for solving problems and their trade offs.

The paper is organized as follows. Section 2 gives an overview of the framework in which ATL is used. Section 3 presents the language constructs on the base of examples. Section 4 presents a case study that shows the applicability of ATL. Section 5 describes the tool support available for ATL: the ATL virtual machine, the ATL compiler, the IDE based on Eclipse, and the debugger. Section 6 presents an evaluation of ATL. Section 7 presents a brief comparison with other approaches for model transformations and outlines directions for future work. Section 8 gives conclusions.

2 General Overview of the ATL Transformation Approach

ATL is applied in a transformational pattern shown in Figure 1. In this pattern a source model *Ma* is transformed into a target model *Mb*. The transformation is driven by a transformation definition (or a transformation program) *mma2mmb.atl* written in the ATL language. The transformation definition is a model. The source and target models and the transformation definition conform to their metamodels *MMa*, *MMb*, and *ATL* respectively. The metamodels conform to the *MOF* metametamodel.

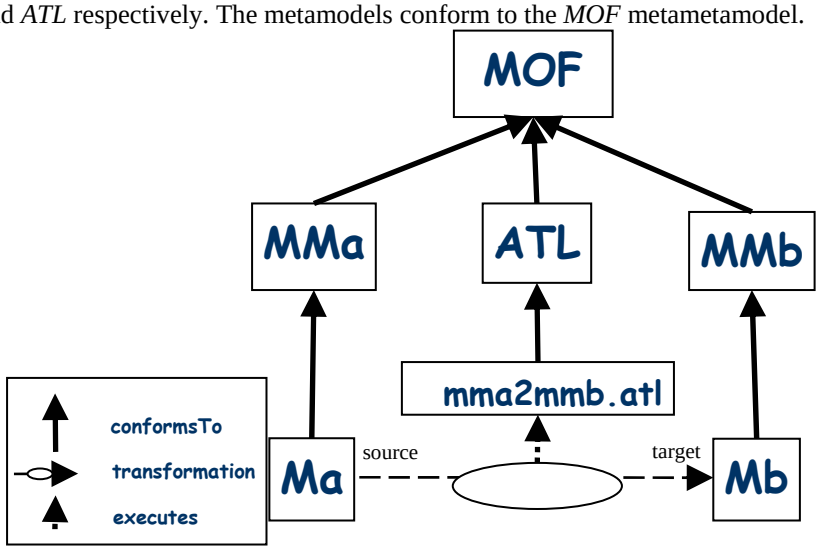

Figure 1 Overview of ATL transformational approach

ATL is a hybrid transformation language. It contains a mixture of declarative and imperative constructs. We encourage a declarative style of specifying transformations. However, it is sometimes difficult to provide a complete declarative solution for a given transformational problem. In that case developers may resort to the imperative features of the language.

ATL transformations are unidirectional, operating on read-only source models and producing a write-only target model. During the execution of a transformation the source model may be navigated but changes are not allowed. Target model cannot be navigated. A bidirectional transformation is implemented as a couple of transformations: one for each direction.

3 Presentation of ATL

In this section we present the features of ATL language. The syntax of the language is presented based on examples (sections 3.1-3.4). Then in section 3.5 we describe the execution semantics of ATL.

3.1 Overall Structure of Transformation Definition

Transformation definitions in ATL form *modules*. A module contains a mandatory *header* section, *import* section, and a number of *helpers* and *transformation rules*.

Header section gives the name of a transformation module and declares the source and target models. Below we give an example header section that will be used in the transformation definition for our case study.

```
module SimpleClass2SimpleRDBMS;
create OUT : SimpleRDBMS from IN : SimpleClass;
```

The header section starts with the keyword *module* followed by the name of the module. Then the source and target models are declared as variables typed by their metamodels. The keyword *create* indicates the target model. The keyword *from* indicates the source models. In our example the target model bound to the variable OUT is created from the source model IN. The source and target models conform to the metamodels *SimpleClass* and *SimpleRDBMS* respectively. In general more than one source and target models may be enumerated in the header section.

Helpers and transformation rules are the constructs used to specify the transformation functionality. They are explained in the next two sections.

3.2 Helpers

The term *helper* comes from the OCL specification ([12], section 7.4.4, p11), which defines two kinds of helpers: *operation* and *attribute* helpers.

In ATL, a helper can only be specified on an OCL type or a source type (coming from a source metamodel) since target models are not navigable. Operation helpers

can be used to specify operations in the context of a model element or in the context of a module. The main purpose of operation helpers is to perform navigation over the source models. Operation helpers can have input parameters and can use recursion. Operation helpers defined in the context of model elements allow polymorphic calls. Since navigation is only allowed on read-only source models, an operation always returns the same value for a given context and set of arguments.

Attribute helpers are used to associate read-only named values to source model elements. Similarly to operation helpers they have a name, a context, and a type. The difference is that they cannot have input parameters. Their values are specified by an OCL expression. Like operation helpers, attribute helpers can be recursively defined with the same constraints about termination and cycles.

Attribute helpers are almost like derived features of MOF 1.4 [11] or Ecore [5] but can be associated to a transformation and are not always attached to a metamodel. Whereas in EMF [5] and MDR [10] they are implemented in Java, in ATL they can be specified using OCL.

Attribute helpers can be considered as a means to decorate source models before transformation execution. A decoration of a model element may depend on the decoration of others. To illustrate the syntax of attribute helpers we consider an example.

```
1.  helper context SimpleClass!Class def :
2.        allAttributes : Sequence(SimpleClass!Attribute) =
3.        self.attrs->union(
4.              if not self.parent.oclIsUndefined() then
5.                    self.parent.allAttributes->select(attr |
6.                     not self.attrs->exists(at | at.name = attr.name)
7.                    )
8.              else Sequence {}
9.              endif
10.       )->flatten();
```

The attribute helper *allAtributes* is used to determine the set of all the attributes of a given class including the defined and the inherited attributes. It is associated to classes in the source model (indicated by the keyword *context* and the reference to the type in the source metamodel *SimpleClass!Class*) and its values are sequences of attributes (line 2). The OCL expression used to calculate value of the helper is given after the '=' symbol (lines 3-10).

This is an example of a recursive helper that uses the value of the same helper associated to the parent of current context class (line 5). If the context class does not have a parent then an empty sequence is used (line 8). This is the terminating case for the recursion.

Attribute helpers can also be used to establish links between source models: the type of an attribute helper can be a class in a metamodel different from its context metamodel. This corresponds to a basic form of model composition.

3.3 Transformation Rules

Transformation rule is the basic construct in ATL used to express the transformation logic. ATL rules may be specified either in a declarative style or in an imperative style. In this section we focus on declarative rules. Section 3.4 describes the imperative features of ATL.

Matched Rules

Declarative ATL rules are called *matched rules*. A matched rule is composed of a *source pattern* and of a *target pattern*.

Rule source pattern specifies a set of *source types* (coming from source metamodels and the set of collection types available in OCL) and a *guard* (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 association end) and specifies an expression whose value is used to initialize the feature.

The following snippet shows a simple matched rule in ATL.

```
1. rule PersistentClass2Table{
2.     from
3.             c : SimpleClass!Class (
4.                     c.is_persistent and c.parent.oclIsUndefined()
5.             )
6.     to
7.             t : SimpleRDBMS!Table (
8.                     name <- c.name
9.             )
10.}
```

The rule name *PersistentClass2Table* is given after the keyword *rule* (line 1). The rule source pattern specifies one variable of type *Class* (line 3). The guard (line 4) specifies that only persistent classes without superclasses will be matched.

The target pattern contains one element of type *Table* (lines 7-9). This element has one binding (line 8) that specifies an expression used to initialize the attribute *name* of the table. The symbol ‘<-‘ is used to delimit the feature to be initialized (left-hand side) from the initialization expression (right-hand side).

Execution Semantics of Matched Rules

Matched rules are executed over matches of their source pattern. Rule ordering is concerned with triggering of rules. This mechanism is described further in this section. Here we focus on the execution of a single rule over a single match.

For a given match the target elements of the specified types are created in the target model and are initialized using the bindings. Executing a rule on a match additionally creates a *traceability link* in the internal structures of the transformation engine. This link relates three components: the rule, the match (i.e. source elements)

and the newly created target elements. Traceability links may be considered as a model and serialized by an ATL engine as an additional product of the execution.

Actual feature initialization uses a specific value resolution algorithm, called ATL *resolve* algorithm. After the expression of a binding has been evaluated, the resulting value is first resolved before being assigned to the corresponding target feature. Resolution depends on the type of the value. If the type is primitive, then the value is simply assigned to the corresponding feature. If its type is a metamodel type there are two possibilities:

- when the value is a **target element** it is simply assigned to the feature;
- when the value is a **source element** it is first resolved into a target element using traceability links. The resolution results in an element from the target model, which is assigned to the feature. This algorithm uses traceability links to identify the target elements created from a given source element as a result of application of a transformation rule.

Thanks to this algorithm, target elements can be effectively linked together using source model navigation only. Finding the appropriate target elements is left to ATL execution engine.

Kinds of Matched Rules

There are several kinds of matched rules differing in the way how they are triggered.

- *Standard* rules are applied once for every match that can be found in source models;
- *Lazy* rules are triggered by other rules. They are applied on a match as many times as it is referred to by other rules. This means that a lazy rule may be applied multiple times on a single match, each time producing a new set of target elements;
- *Unique lazy* rules are also triggered by other rules. They are applied only once for a given match. If a unique lazy rule is triggered later on the same match the already created target elements are used.

Examples of standard and unique lazy rules are given in the case study. The ATL resolution algorithm describe above takes care of triggering lazy and unique lazy rules when a source element is referred to within an initialization expression.

Rule Inheritance

In ATL rule inheritance can be used as a code reuse mechanism and also as a mechanism for specifying polymorphic rules.

A rule (called subrule) may inherit from another rule (parent rule). A subrule matches a subset of what its parent rule matches. This implies a number of constraints on source patterns of subrules, which can be summarized in: “a subrule cannot change source pattern structure”. Source pattern types of the subrule must thus be either left unchanged or replaced by subtypes of those used in its parent. A subrule guard filters out elements that already match its parent guard. The actual guard of the subrule is the conjunction of both guards.

A subrule target pattern extends its parent target pattern using any combination of the following: by subtyping target types, by adding bindings, by replacing bindings, and by adding new target elements. Note that a binding cannot be simply extended, it

must be fully replaced. However, if the expression of a parent binding needs to be reused then it may be referred to using *super* keyword.

3.4 Imperative Features of ATL

The declarative style of transformation specification has a number of advantages. It is usually based on specifying relations between source and target patterns and thus tends to be closer to the way how the developers intuitively perceive a transformation. This style stresses on encoding these relations and hides the details related to selection of source elements, rule triggering and ordering, dealing with traceability, etc. Therefore, it can hide model transformation specific complex algorithms behind a simple syntax.

However, in some cases, other complex source-domain or target-domain specific algorithms may be required (e.g. matrix diagonalization) and it may be difficult to specify a pure declarative solution for them. There are several possible approaches to this issue:

- allow **native operation calls**. This solution has the drawback that it moves the control flow out of the transformation language semantics;
- offer an **imperative part** in the transformation language. In that way the control flow remains in the transformation language semantics but the developer must encode this control flow explicitly. There are potential problems with efficiency and optimization of such transformation specifications. Nonetheless, the developer can perform manual optimizations, even when an optimizing engine is not available;
- **extract data** that needs processing from the models to a domain-specific tool and then inject the result as models. A drawback of this approach is that it requires a heavyweight mechanism, but it offers large flexibility in the processing.

The third option is available in ATL since it is not a matter of model transformation. The first two options have one common drawback: what is executed out of the reach of the execution engine (either totally, from native code or partially, by imperative constructs) restricts declarative advantages. They seem, however, necessary as using them requires less work than using the third one. ATL therefore has an imperative part, based on two main constructs:

- **called rules**. A called rule is basically a procedure: it is invoked by its name and may take arguments. Its implementation can be native or specified in ATL (e.g. as a target pattern without source pattern since no match is needed).
- **action block**. An action block is a sequence of imperative statements and can be used instead of or in a combination with a target pattern in matched or called rules. The imperative statements available in ATL are the well known constructs for specifying control flow such as conditions, loops, assignments, etc. We do not give their syntax in this paper.

If either a called rule or an action block is used in an ATL program, this program is no longer fully declarative.

3.5 Execution of Transformation Definitions

In this section we discuss various aspects related to execution of ATL transformation definitions. First we present a pseudo code algorithm for execution of transformations. Then we discuss some optimization issues and provide information about the termination and determinism of ATL transformations.

Algorithm for Executing ATL Transformations

The following algorithm presents the basics of the procedural semantics of ATL language. This algorithm is used by the ATL transformation engine. We assume that transformation definitions can contain both matched and called rules. The algorithm presents only the execution of rules. Handling of helpers is not included. Moreover, ATL resolve algorithm was already described in section 3.3 and is not detailed here.

```
execute called rule marked as entrypoint
-- This results in a traditional imperative control flow.

-- Match standard matched rules:
ForEach standard rule R {
     ForEach candidate pattern C of R {
     -- a candidate pattern is a set of elements matching the
     -- types of the source pattern of a rule

            evaluate the guard of R on C
            If guard is true Then
                   create target elements in target pattern of R
                   create TraceLink for R, C, and target elements
            Else
                   discard C
            EndIf
     }
}

-- Apply standard matched rules:
ForEach TraceLink T {
     R = the rule associated to T
     C = the matched source pattern of T
     P = the created target pattern of T

     -- Initialize elements in the target pattern:
     ForEach target element E of P {
            -- Initialize each feature of E:
            ForEach binding B declared for E {
                   expression =  initialization expression of B
                   value = evaluate expression in the context of C
                   featureValue = resolve value
                   set featureValue to corresponding feature of B
            }
     }
     execute action block of R in the context of C and T
     -- Imperative blocks can perform any navigation in C or T and
     -- any action on T. It is the programmer's responsibility
     -- to perform only valid operations.
}

execute called rule marked as endpoint
-- We have again an imperative control flow.
```

This algorithm starts with the execution of an optional called rule marked as *entry point*. This rule, in turn, may invoke other called rules. Then the algorithm proceeds with the execution of the standard matched rules in the transformation program (some of them may contain an action block). Note that rule matching and application are separated. This is not absolutely necessary. However, it is simpler to describe and implement a two stage algorithm because every *TraceLink* is available after the first stage. Therefore it is easy to resolve the required target elements in the second stage. Target elements could, however, be initialized as they are created. This would make the resolving algorithm more complex since some initializations may be delayed until the end of the transformation.

The presented algorithm does not suppose any order in rule matching, target elements creation for a match, target elements initialization for a *TraceLink*, and feature initialization of a target element. Action block (if present) must, however, be executed after having applied the declarative part of the rule. This eases the programmer's task since the target pattern is in a somewhat foreseeable state (imperative action blocks should only be used when declarative constructs are not enough).

In fact, order constraints can be made even less restrictive. For instance, all target element features can really be initialized in any order while the given algorithm enforces that all features of a given target element are initialized in block and that all target elements of a *TraceLink* are initialized in block. This is not absolutely necessary.

Some Possible Optimizations

The execution algorithm described in this paper is roughly what is implemented in the current ATL engine (compiler and virtual machine) although the imperative parts of the language are not fully supported yet. An engine may, however, use the flexibility of execution order constraints to perform some optimizations. For instance, a single iteration could be performed for source patterns with the same structure but with different guards.

Moreover, not all candidate patterns may need to be tested if some can be rejected by statically analyzing the guard. Consider a source model where elements of type *A* may contain elements of type *B* and a rule takes as source pattern every pair *(a : A, b : B)* with the constraints that *a* contains *b* (i.e. `a.bs->includes(b)` in OCL). Construction of the candidate patterns would look like the following code with two nested loops:

```
for every a of type A {
     for every b of type B {
            evaluate the guard
            do something
     }
}
```

If the guard is analyzed the code may be optimized in the following:

```
for every a of type A {
     for every b of type B contained in A {
            do something
     }
}
```

*B*s not contained in *a* are not tested.

Lazy rules can also be used for optimization. One of the expensive operations in model transformations is the pattern matching of the rule source patterns. Instead of specifying complex patterns and guards the software engineer may locate the required source elements using helpers or using the context of matched rules. Then these elements may be transformed by unique lazy rules. In that way elements are only transformed if they are referred to by a transformed element. Obviously this is a user-specified optimization rather than engine-inferred.

Calls to helpers may also be optimized. As we saw in section 3.2, operation and attribute helpers are side-effect free and operate on read-only models. Their result values can therefore be cached instead of computed each time they are required. Besides, although attribute helpers may be initialized in a pass performed before running the rest of the transformation, they may also be lazily evaluated when the helper value is read for the first time. Both alternatives produce equivalent results. However, the performance is different. Lazy evaluation leads to a better performance since only a subset of the attribute helpers values may be actually used during the execution.

Deterministic Execution and Termination

As long as lazy rules and called rules are not used, the execution algorithm terminates and is deterministic. Although the order of execution of rules is non-deterministic, different execution orders produce the same result for a given source model. As a matter of fact, source models are read-only: the execution of a rule cannot change the set of matches. In addition, target models are write-only: the initialization of a target element cannot impact the initialization of another. However, it is possible to have recursive helpers that do not terminate. In this case the transformation does not terminate either.

The problem with called rules is that within them we use a standard imperative language that cannot be proved to always terminate.

As for lazy rules, the problem is that recursive references can be written. For instance:

```
1.  lazy rule R1 {
2.      from
3.          s : Element
4.      to
5.          t : Element (
6.              value <- [R2.t]s
7.      )
8.  }
9.
10. lazy rule R2 {
11.     from
12.         s : Element
13.      to
14.         t : Element (
15.             value <- [R1.t]s
16.     )
17.  }
```

In this example there are two lazy rules *R1* and *R2* that refer to each other (lines 6 and 15). In that way it is possible to form infinite recursion.

Lines 6 and 15 also illustrate one explicit way for triggering a lazy rule. The rule name and the identifier of the required target element are separated by '.' and surrounded by square brackets followed by the source element on which the rule will operate.

4 Case Study: Transforming Class to Relational Models

In this section we present the solution written in ATL to the case study given in the call for papers of the workshop.

4.1 Source and Target Metamodels

In order to improve the clarity of the presentation we briefly repeat the case study already given in the call for papers. The case study requires transformation of simple class models to relational models. The class models conform to the source metamodel in Figure 2.

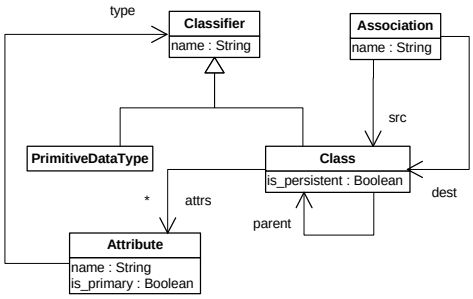

Figure 2 Source class metamodel

According to this metamodel classes have names and a number of attributes. Classes may be declared as persistent (attribute *is_persistent*). The type of an attribute is a classifier: either a primitive data type or a class. Attributes may be defined as primary (attribute *is_primary*). Classes may be related via associations. An additional constraint is imposed that every class have at least one attribute and at least one primary attribute (they may be inherited).

Relational models conform to the metamodel in Figure 3. Every model contains a number of tables. Each table has a number of columns, some of them are primary. A table may be associated to zero or more foreign keys. Each foreign key refers to a table and is associated with a number of columns that constitute the key.

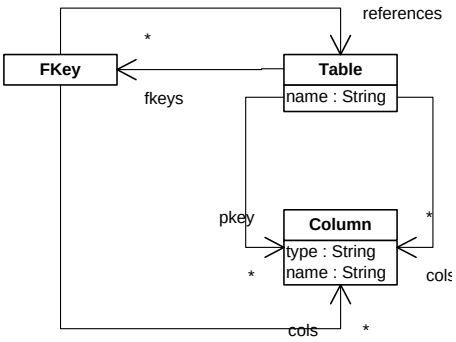

Figure 3 Target relational metamodel

Transformation rules are summarized below:

- Persistent classes that are roots of an inheritance hierarchy are transformed to tables;
- Table columns are derived from the attributes and associations of a class;
- Every attribute of a primitive type is transformed to a single column. If the attribute is primary it results to a primary column in the table;
- For every persistent root class a set of attributes and associations is derived by flattening the inheritance hierarchy. These attributes and associations are used to derive the columns of the result table;
- Attributes whose type is a non-persistent class and associations that point to such a class are transformed to a set of columns derived from the class. This rule is applied recursively until a set of primitive attributes is obtained. We assume that circularity in references to classes is not allowed. A class inherits the attributes and associations of its parent;
- Attributes whose type is a persistent class and associations that point to such a class are transformed to a foreign key and a set of columns contained by the key. The foreign key refers to the table derived from the persistent class. The columns are derived from the primary attributes of the persistent class. We assume that the same rule is also applied for the non-persistent classes whose top-most parent is a persistent class. We assume that the instances of these classes are kept in the table created from the root of the hierarchy. This is motivated by the flattening of the inheritance hierarchies[1].

We assume that primary attributes are always of primitive data types.

4.2 Transformation Specification SimpleClass2SimpleRDBMS

The following code presents the complete transformation program for the case study. It is written by using only the declarative features of ATL. In this section we explain the overall logic of the program.

[1] Another possibility is to treat all the non-persistent classes in a uniform way. We believe that the assumption we make does not simplify the transformation definition and does not remove any significant technical aspect from the transformation.

```
1.  module SimpleClass2SimpleRDBMS;
2.  create OUT : SimpleRDBMS from IN : SimpleClass;

3.  helper context SimpleClass!Class def :
4.     allAttributes : Sequence(SimpleClass!Attribute) =

5.     self.attrs->union(
6.            if not self.parent.oclIsUndefined() then
7.                   self.parent.allAttributes->select(attr |
8.                    not self.attrs->exists(at | at.name = attr.name)
9.                   )
10.           else Sequence {}
11.           endif
12.    )->flatten();

13. helper context SimpleClass!Class def :
14.    allAssociations : Sequence(SimpleClass!Association) =

15.    let defAssoc : SimpleClass!Association =
16.           SimpleClass!Association.allInstances()->select(assoc |
17.                  assoc.src = self) in
18.    defAssoc->union(
19.           if not self.parent.oclIsUndefined() then
20.              self.parent.allAssociations
21.           else Sequence {}
22.           endif
23.    )->flatten();

24. helper context SimpleClass!Class def :
25.    topParent : SimpleClass!Class =

26.    if self.parent.oclIsUndefined() then
27.           self
28.    else
29.           self.parent.topParent
30.    endif;

31. helper context SimpleClass!Class def :
32.    attributesOfSubclasses : Sequence(SimpleClass!Attribute) =

33.    let attrsInSubclasses : Sequence(SimpleClass!Attribute) =
34.           SimpleClass!Class.allInstances()->select(c |
35.                  c.parent=self
36.           )->collect(subclass |
37.                  subclass.attributesOfSubclasses
38.           )->flatten() in
39.    attrsInSubclasses->union(
40.           self.attrs->select(attr |
41.                  not attrsInSubclasses->exists(a |
42.                         a.name = attr.name)
43.    ))->flatten();

44. helper context SimpleClass!Class def :
45.    associationsOfSubclasses : Sequence(SimpleClass!Association) =
46.
47.    SimpleClass!Association.allInstances()->select(assoc |
48.                  assoc.src = self)->union(
49.                  SimpleClass!Class.allInstances()->select(c |
50.                         c.parent = self)->collect(subclass |
```

```
                subclass.associationsOfSubclasses)->flatten()
    )->flatten();

helper context SimpleClass!Class def :
    flattenedFeatures : Sequence(TupleType(
                trace : Sequence(OclAny),
                isPrimary : Boolean,
                isForeignKey : Boolean,
                ForeignKeyColumns : Sequence(Sequence(OclAny)))) =

    if self.topParent.is_persistent then
            self.topParent.attributesOfSubclasses->union(
                self.topParent.associationsOfSubclasses)
    else
            self.allAttributes->union(self.allAssociations)
    endif->collect(f |
            let feature : TupleType(type : SimpleClass!Classifier,
                                    isPrimary : Boolean) =
            if f.oclIsKindOf(SimpleClass!Attribute) then
                Tuple{type = f.type, isPrimary = f.is_primary}
            else
                Tuple{type = f.dest, isPrimary = false}
            endif in
    if feature.type.oclIsKindOf(SimpleClass!PrimitiveDataType) then
            Tuple {trace = Sequence {f},
                isPrimary = feature.isPrimary,
                isForeignKey = false,
                ForeignKeyColumns = OclUndefined
            }
    else if not feature.type.topParent.is_persistent then
            feature.type.flattenedFeatures->collect (t |
                Tuple {trace = t.trace->prepend(f),
                       isPrimary = t.isPrimary,
                       isForeignKey = t.isForeignKey,
                       ForeignKeyColumns = t.ForeignKeyColumns
                }
            )
    else let primaryFeatures : Sequence(OclAny) =
            feature.type.topParent.flattenedFeatures->select(t |
                t.isPrimary)->collect(pt |
                    Tuple{trace = pt->prepend(f),
                          isPrimary = pt.isPrimary,
                          isForeignKey = false,
                          ForeignKeyColumns = OclUndefined
                    }
                ) in
            primaryFeatures.prepend(Tuple{
                                trace = Sequence {f},
                                isPrimary = false,
                                isForeignKey = true,
                                ForeignKeyColumns = primaryFeatures
                            })
    endif endif
    )->flatten();

rule PersistentClass2Table{
    from
            c : SimpleClass!Class (
                c.is_persistent and c.parent.oclIsUndefined()
            )
```

```
108.   to
109.           t : SimpleRDBMS!Table (
110.                   name <- c.name,
111.                   cols <- c.flattenedFeatures->select(f |
112.                                   not f.isForeignKey
113.                           )->collect(ft | ft.trace),
114.                   pkey <- c.flattenedFeatures->select(f |
115.                                  f.isPrimary)->collect(ft | ft.trace),
116.                   fkeys <- c.flattenedFeatures->select(f |
117.                                    f.isForeignKey)
118.           )
119. }

120. unique lazy rule Feature2Column {
121.   from
122.           trace : Sequence(OclAny)
123.   to
124.           col : SimpleRDBMS!Column (
125.                   name <- trace->iterate(e; acc : String = '' |
126.                                   acc + if acc = ''
127.                                         then ''
128.                                         else '_' endif + f.name),
129.                   type <- trace->last().type
130.           )
131. }

132. unique lazy rule PersistentFeature2ForeignKey {
133.   from
134.           feature : TupleType(
135.                     trace : Sequence(OclAny),
136.                     isPrimary : Boolean,
137.                     isForeignKey : Boolean,
138.                     ForeignKeyColumns : Sequence(Sequence(OclAny)))
139.   using {
140.           last : OclAny = feature.trace->last();
141.           referencedClass : SimpleClass!Class =
142.                   if last.oclIsKindOf(SimpleClass!Attribute) then
143.                           last.type.topParent
144.                   else
145.                           last.dest.topParent
146.                   endif;
147.   }
148.   to
149.           fkey : SimpleRDBMS!FKey (
150.                   references <- referencedClass,
151.                   cols <- feature.ForeignKeyColumns
152.           )
153. }
```

The transformation specification may be split into two logical parts. The first part performs decoration of the source model and the second part contains the actual transformation rules. Since the transformation specification is declarative there is no explicit execution order among these parts.

The decoration part consists of a set of attribute helpers (lines 3-102). They are additional attributes of source model elements assigned during the transformation. Attribute helpers are used for the following tasks:

- To determine the attributes and associations for every class including those inherited by the superclass. This is achieved by *allAttributes* (lines 3-12) and *allAssociations* (lines 13-23) attribute helpers;
- To determine the top-most parent of a given class (*topParent* attribute helper, lines 24-30). This helper is used when an attribute/association has as a type a non-persistent class whose top-level parent class is persistent;
- To flatten inheritance hierarchies with root a persistent class. In this case all the attributes and associations of the direct and indirect children classes are united and associated with the root persistent lass. Attribute helpers *attributesOfSublasses* (lines 31-43) and *associationsOfSubclasses* (lines 44-52) are used to perform this flattening;
- To flatten the features (either attributes or outgoing associations) of classes by performing the drill-down algorithm for handling the attributes of non-primitive types. This is done by *flattenedFeatures* (lines 53-102) attribute helper;

The last helper is the most complex. We will explain it in bigger details.

We treat attributes and outgoing associations of a class in a uniform way by referring to them as *features*[1]. For every feature we apply the rules for deriving table columns. If the feature is of primitive type a single column will be created. If the feature is of a non-persistent class then it results in a set of primitive features with recursive accumulation of names. If the feature is of persistent class (or the top parent is a persistent class) then we obtain the primary features of this class.

The purpose of the helper is to associate a set of primitive features to every class from which columns are directly derived. Since some of these features are result of a flattening they are in fact a sequence of features derived according to the drill-down algorithm. We call such a sequence *trace*. Therefore, we associate a set of traces to every class and create a column from every trace of a persistent class. The names of the columns are formed as a concatenation of the names of the features in the traces. To illustrate better the idea we give an example shown in Figure 4. The example shows a simple source model where classes are decorated with traces.

In Figure 4 traces are shown next to each class. Classes that inherit from a persistent class do not generate traces (*Student* and *Employee*). Primary features are underlined.

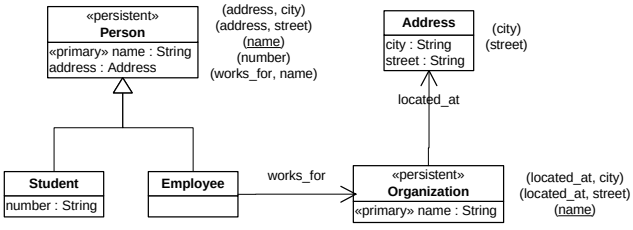

Figure 4 An example source model decorated with traces

[1] Unfortunately the source metamodel does not contain a common parent class for Attribute and Association.

The idea behind *flattenedFeatures* helper is to generate traces of features plus some additional information about the traces. The structure that holds this information is a tuple type with the following fields (lines 55-58):

- *trace* – contains the trace as a sequence of features;
- *isPrimary* – indicates if the trace generates a primary column;
- *isForeignKey* – indicates if the trace ends with a feature of a persistent type;
- *ForeignKeyColumns* – if the field *isForeignKey* is true then this field contains a sequence of traces that will be used to produce the columns of the foreign key. These traces are copies of the primary column traces of the persistent class for which the foreign key is generated;

Figure 4 shows only traces. The entire structure explained above is not shown for simplicity.

The logic in *flattenedFeatures* helper is to start with an initial list of features for a class and to build a sequence of tuples conforming to the described structure by applying flattening recursively. If the class is a persistent class then the initial list of features is the union of the values of *attributesOfSublasses* and *associationsOfSubclasses* attribute helpers (lines 59-61). If the class is a non-persistent class that does not inherit from a persistent class then the initial list is the union of values of *allAttributes* and *allAssociations* attribute helpers (line 63).

Transformation rules use the result of the decoration part to create the elements in the target model. The main work related to flattening is done by the helpers.

Rule *PersistentClass2Table* transforms persistent root classes to tables. The interesting part of this rule is the initialization of the features of the created tables.

The code in lines 111-113 initializes the *cols* slot of the table. The value of this slot is a collection of all the columns of the table. Columns are created from traces that do not represent foreign keys (this is encoded in the selection criteria in line 112). The value of the initialization expression is a sequence of traces, that is, a sequence of sequences of source model elements. Therefore this value must be resolved according to the ATL resolution algorithm. The resolution requires finding a rule that transforms the value of the expression into target model elements. Thus, in this feature initialization we have an implicit invocation of a transformation rule. The only suitable rule is *Feature2Column* unique lazy rule. This rule transforms traces to columns. It is triggered on demand. In our example it will be executed on every trace in the sequence generated by the initialization expression.

Furthermore the slot *pkey* contains all the primary columns of the table. Primary columns are a subset of the set of all the columns of the table. Primary columns are also generated by the rule *Feature2Column*. Similarly to the previous slot, in lines 114 and 115 we have an implicit invocation of *Feature2Column* rule. In other words the same rule may be triggered multiple times over the same source. This is a unique rule and this guarantees that invocations after the first time will return the same result. If the rule was not unique two different copies of the primary columns would be created.

Foreign keys are created by the *PersistentFeature2ForeignKey* unique lazy rule. It is implicitly triggered in line 116.

4.3 Discussion

The presented solution is just one among several potential solutions for the case study. In this section we discuss some implementation alternatives.

First of all, this solution relies on features of ATL that are not implemented yet in the current compiler. Current compiler does not fully support lazy rules and rules with multiple source elements and source elements that are of OCL types (e.g. sequences). The reason for giving this solution is to illustrate the full set of declarative constructs that will be available in ATL. We implemented a second solution that runs on the current version of the ATL virtual machine and may be compiled with the current ATL compiler. This solution will be made available on the Eclipse GMT project [8]. It shows that even with the current incomplete version, ATL is capable to handle completely the required case study.

A significant part of the presented transformation definition is implemented as attributes helpers. The values of attribute helpers may be cached thus increasing the performance (see sections 3.5 and 5.2). The current version of ATL engine allows enabling and disabling the caching mechanism. We performed a measure of performance increase resulting from caching attribute helper results on a version of the case study working on the current ATL engine. 40853 bytecodes were executed with cache disabled and 15543 with cache enabled. This corresponds to a 2.6 decrease of the number of executed bytecodes. Although this optimization is rather simple, it should be noted that no change has to be done by the developer in the program: cache needs just to be activated.

Furthermore the functionality of the helpers may be implemented in transformation rules. This would lead to recursive rules. It is not always easy to judge which functionality to be implemented as helpers and which one as transformation rules. In our case, the flattening functionality was related to the decoration of the source model and required only navigation over the source model. We decided to implement all the navigation functionality as helpers and to keep the transformation rules free from complex navigation expressions. This fits to the basic intention of these constructs: rules are used for creating target model elements and helpers are used for source model navigation.

Another dimension of alternatives is using implicit rule invocation (through the ATL resolution algorithm) versus explicit rule calls. In our case study we used implicit rule invocation. We believe that this leads to more adaptable transformation definitions and to loosely coupled rules. However, explicit rule calls may be useful in case of ambiguity in determining the applicable transformation rules for a given input. When the resolution algorithm tries to resolve the default traceability link and there are more than one rule that produces such a default link then a conflict among rules arises.

Finally, the presented solution is implemented in a declarative style. It is possible to implement it with the intended imperative features of ATL. However, in this paper we focus more on the declarative part of ATL.

4.4 Second Case Study: Converting Roman Numbers to Arabic Numbers

An important part of model transformations is extracting data from strings and conversion of strings. In many cases these tasks are not trivial. In this section we consider a small example of string conversions: from Roman numbers to Arabic numbers.

The solution for this problem is not a model transformation definition. It is implemented as an ATL operation helper and relies on the capabilities of OCL. The following code shows the solution to this problem. This helper may be integrated and used in a more complex program.

```
helper context String def:
     toIntegerFromRoman() : Integer =
     let rd : Sequence(String) = Sequence {'I', 'V', 'X', 'L', 'C',
                                           'D', 'M'} in
     let rv : Sequence(Integer) = Sequence {1, 5, 10, 50,100, 500,
                                            1000} in
      let r : TupleType(ret : Integer, prev : Integer) =

     self.toSequence()->iterate(e; acc : TupleType(ret : Integer,
                                       prev :
             let val : Integer = rv.at(rd->indexOf(e)) in
             if acc.prev = -1 then
                    Tuple {ret = 0, prev = val}
             else if acc.prev < val then
                    Tuple {ret = acc.ret - acc.prev, prev = val}
             else
                    Tuple {ret = acc.ret + acc.prev, prev = val}
             endif endif
       ) in r.ret + r.prev;
```

5 ATL Tools

The practical application of a computer language requires a set of supporting tools: compiler/interpreter, development environment, debugger, profiler, etc. In this section we present the available ATL tools that include the ATL transformation engine, the ATL integrated development environment (IDE) based on Eclipse, and the ATL debugger.

5.1 Requirements for ATL Tools

There is no unique implementation architecture for the execution semantics described in section 3. We can identify some alternatives:

- *interpretation* vs. *compilation* or a combination of both (a la Java);
- fully *sequential* vs. partially *parallel* execution. The flexibility of ordering constraints on the execution path makes the parallel execution doable: matching rules in parallel, testing the guard of a single rule over several candidate patterns in parallel, applying initializations in parallel, etc.;

Parallelizing the execution is more interesting when the underlying hardware is parallel. We chose to implement a sequential approach, because it is easier to implement and is sufficient for the present time. Dealing with very large models in the future may benefit from parallelizing the execution. Identification of parts in transformation programs suitable for parallel execution seems an interesting direction for future research.

The following requirements were formulated for the ATL tools:

- it should be easy to implement new language features;
- it should be easy to replace an execution engine with a more efficient one;
- it should be possible to debug the transformation program and the execution engine itself (e.g. a compiler, an interpreter, a virtual machine);
- the execution engine should be portable to several model handlers (e.g. MDR [10], EMF [5], etc.);

The implemented solution includes an ATL compiler, a virtual machine (VM), an IDE, and a debugger. These tools are described in the following two sections.

5.2 ATL Execution Engine

The architecture of the ATL execution environment is shown in Figure 5. It contains the following components arranged across several layers:

- *ATL Compiler*. ATL compiler transforms ATL programs into programs written in byte-code;
- *ATL Virtual Machine*. ATL VM executes the byte-code generated by the compiler. The virtual machine is specialized in handling models and provides a set of instructions for model manipulation;
- *Model Handler Abstraction Layer*. The virtual machine may run on top of various model management systems. To isolate the machine from their specifics an intermediate level is introduced called Model Handler Abstraction Layer. This layer translates the instructions of the VM for model manipulation to the instructions of a specific model handler;
- *Model Handlers*. These are components that provide programming interface for model manipulation. Some examples are Eclipse Modeling Framework (EMF) and MDR;
- *Model Repository*. Model repository provides storage facilities for models. As Figure 5 shows the simplest form of a model repository is the file system that stores models as XML files serialized according to the XMI standard;

Because of this layered architecture we achieve the requirements for flexibility of the execution engine. Additions of new language features affect mainly the ATL compiler. A more efficient execution requires changes in the implementation of the virtual machine and the compiler (some static optimizations may be performed by the compiler). Existing programs will run on top of a new VM provided that it conforms to the same set of instructions. A specification of ATL VM is provided on the GMT website [8]. Thanks to the Model Handler Abstraction Layer the virtual machine is relatively easy to port to a new model handler.

In section 3.5, we discussed possible optimizations of ATL programs execution. In the current implementation of ATL engine, we implemented some of them: attribute helpers values are lazily evaluated and their results are cached. For instance, considering the attribute helper specified in section 3.2: it may be rewritten as an operation helper without parameters. However, this will lead to a worse performance with the current ATL engine since the results of operation helpers are not cached. The operation helper for a given class will be executed every time when the attributes of a direct or indirect subclass are determined. In the implementation based on attribute helpers the value is calculated only once and reused afterwards. Additionally, we gave actual figures of achieved optimization in section 4.3.

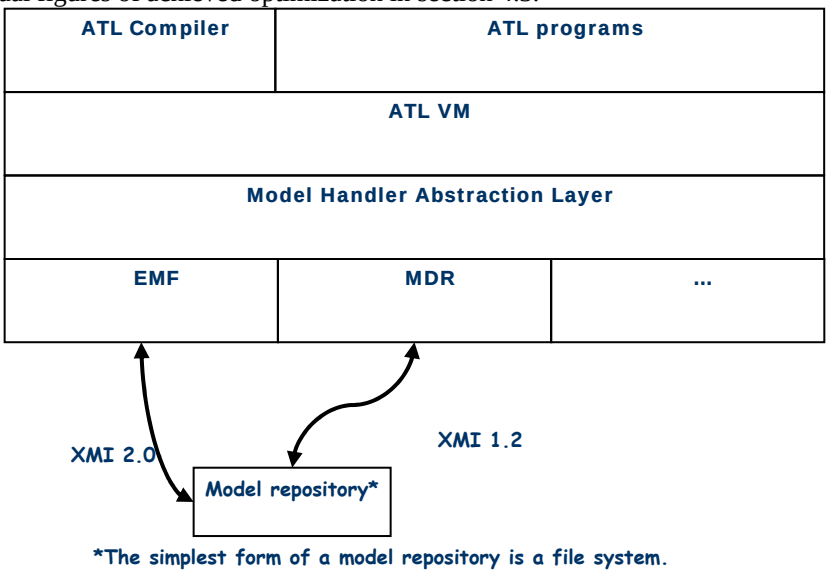

Figure 5 The architecture of the ATL execution engine

5.3 ATL IDE

The current ATL IDE is built on top of Eclipse. It includes an editor that provides view of the text with syntax highlighting, outline (view of the model corresponding to the text), and error reporting. The IDE uses the Eclipse interface to the ATL debugger.

Figure 6 shows a screenshot of the ATL IDE.

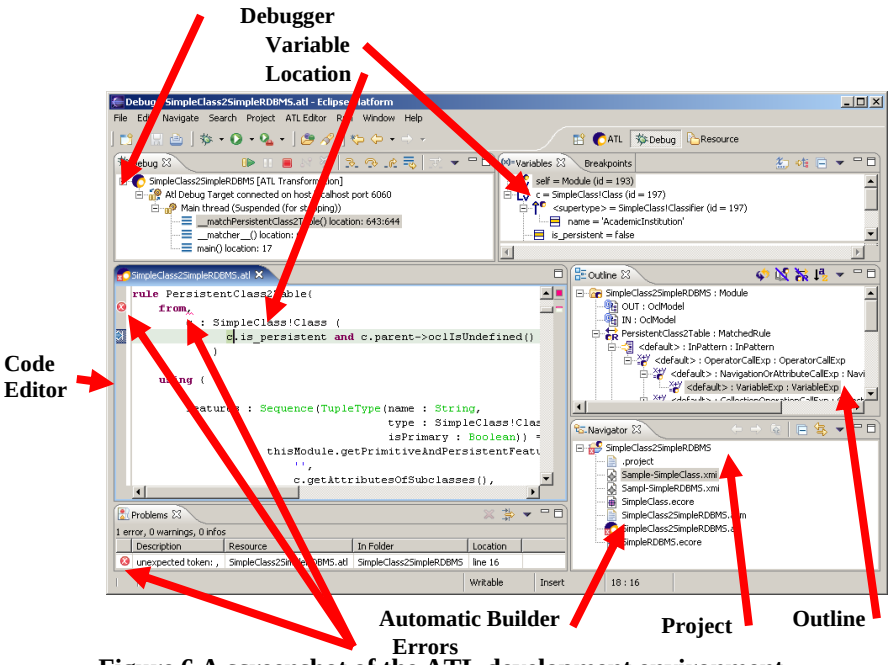

Figure 6 A screenshot of the ATL development environment

The figure indicates various components in the IDE: the code editor, error indications, the builder, the project view, the outline view, and the variables watch used during a debugging session.

6 Discussion

In this section we present an evaluation of ATL according to criteria derived from the design decisions found in other transformation languages. In addition, we provide an overview of the features of the language currently supported by the ATL compiler and the features that will be supported in a future release.

6.1 Classification Categories for Transformation Languages

Czarnecki and Helsen [6] present a domain analysis of existing model transformation approaches. The results of this analysis are summarized in a feature model that presents the commonalities and variabilities in the domain. Here we briefly describe the main categories of classification and their variation areas. Variations indicate the design choices made in existing model transformation approaches. The categories are presented in the subsequent sections.

Transformation Rules
Transformation rules are the basic constructs in transformation definitions. They have left-hand side and right-hand side, which may or may not be syntactically separated. The areas of variation found in transformation rules are:

- *Directionality*: rules may be executed in one or two directions;
- *Rule parameterization*: rules may receive additional input via parameters;
- *Intermediate structures*: some approaches allow intermediate model structures;

Source-Target Relationship
This classification category captures the relation between source and target models. The following variations are found:

- Source and target models are different;
- Source and target is the same model: this allows updates in the source model (*in-place update*);

Rule Application Strategy
Generally, a rule may match more than one element/tuple in the source model. Therefore a strategy for the rule application on the matches is required. Strategies may be:

- *Deterministic*: an algorithm governs the order of application of rules over matches;
- *Non-deterministic*: the order of application of rules may be different for different executions of the same transformation on the same source model;
- *Interactive*: the user specifies the strategy;

Rule Scheduling
Rule scheduling mechanism is responsible for the order in which the rules are applied. It may vary in four areas:

- *Form*: concerns the way the order is expressed. The form may be *implicit* and *explicit*. Implicit form of scheduling relies on implicit relations among the rules. Explicit form of scheduling uses dedicated constructs to control the order. Explicit scheduling may be *internal* and *external*. Internal scheduling uses control flow structures within rules and explicit rule invocation. External scheduling uses scheduling logic separated from the transformation rules;
- *Rule selection*: rule selection may rely on explicit *condition* on the source elements. Since many rules may be applicable on a single source element there may be a need of *rule conflict resolution* (e.g. via rule priority);
- *Rule iteration*: iteration may be based on recursion, looping, fixpoint iteration, and combination of them;
- *Phasing*: a transformation definition is separated into phases usually executed sequentially. Each phase uses certain set of rules;

Rule Organization
Rule organization concerns relations among transformation rules. Three variation areas are related to this category:

- *Modularity mechanisms*: these are mechanisms for grouping of rules into packaging constructs;
- *Reuse mechanisms*: allow rules to reuse existing rules in new rule definitions;
- *Organizational structure*: rules may be organized according to the source language, target language, or in other independent way;

Traceability Links
Traceability links record correspondences between source and target elements established during transformation execution. Generally, two approaches are followed for maintaining traceability links:

- *User-based*: the user is responsible to create links as ordinary model elements;
- *Dedicated support*: transformation language and transformation engine provide support for maintaining links. This support may be *automatic* and *manual;*

Directionality
Some languages allow transformation definitions only in one direction: from source to target model. These transformations are known as *unidirectional* transformations. Other languages allow definitions that may be executed in both directions. These transformations are known as *bidirectional* transformations.

6.2 Classifying ATL

This section classifies ATL according to the categories explained in Section 6.1. Results are summarized in Table 1.

Category		Classification of ATL
Transformation Rules	Structure	Syntactically separated left-hand side and right-hand side parts with variables and patterns.
	Directionality	Unidirectional rules from source to target elements.
	Rule parameters	Supported in called rules only.
	Intermediate Structures	Supported via attribute helpers and OCL tuples.
Source-Target Relationship		Separated source and target models without possibility for in-place update. See also the feature *Refining mode* in the next section 6.3
Rule Application Strategy	Deterministic	Supported for imperative called rules only.
	Non-deterministic	Rule application on all the matches of the rule source pattern in a non-deterministic order (see discussion on determinism in section 3.5).
	Interactive	Not supported.
Rule Scheduling	Form	Mix of implicit and explicit form of scheduling. Implicit scheduling supported through matched rules. Explicit form is supported through invocation of called rules and control flow constructs (internal explicit form).
	Rule selection	Based on a rule source guard. Conflicts among standard matched rules are not allowed when they lead to ambiguity in resolving the default traceability links.
	Rule iteration	Recursion is supported.
	Phasing	Possible by applying two distinct transformations in a sequence.
Rule Organization	Modularity mechanisms	Transformation modules and libraries.
	Reuse mechanisms	Rule inheritance and module inclusion.
	Organizational Structure	Source-driven.
Traceability Links		Dedicated automatic support. Storage of links is handled by the transformation engine.
Directionality of transformations		Unidirectional (from source model to target model).

Table 1. ATL features according to the classification of transformation approaches

6.3 Currently Supported ATL features

Table 2 presents a summary of the features of ATL currently supported by the compiler and some features that could be implemented as future extensions. Stars indicate the supported features. An explanation of some of the features is given in the numbered list after the table.

ATL feature			Current version	Future extensions
OCL helpers	operations and attributes in the context of	metamodel types, OCL primitive and tuple types, transformation module (i.e. static)	*	
		OCL collection types		*
	references (i.e. with opposite) (1)			*
	other ways to specify values of attribute or reference helper (2)			*
Code reuse	helpers libraries		*	
	rule libraries (importable modules)			*
Matched rules	standard		*	
	lazy			*
	unique lazy			*
	rule inheritance			*
	multiple source elements			*
ATL resolve algorithm	standard		*	
	with rule inheritance			*
	with lazy rules			*
	more strongly typed explicit resolving (3)			*
Refining mode (4)			*(basic)	*(improved)
Traceability			internal	external
Imperative part	ATL called rules			*
	native called rules			*
	action blocks			*
OCL type checking			Dynamic	Static (following the specification)

Table 2 ATL features summary

(1) Such reference helpers could be used to optimize source model decoration: instead of explicitly linking *A* to *B* and *B* to *A*, only one direction would have to be initialized. For instance, instead of:

```
helper context A def: b : B =
    B.allInstances()->select(e |
                        e.name = self.name)->asSequence()->first();
helper context B def: a : A =
    A.allInstances()->select(e |
                        e.name = self.name)->asSequence()->first();
```

we would have:

```
helper context A def: b : B oppositeOf a =
    B.allInstances()->select(e |
                        e.name = self.name)->asSequence()->first();
```

(2) Currently, the only way to specify an attribute helper value is by specifying an OCL expression. Mechanisms making complex mappings simpler could be implemented; for instance: linking elements having the same name without manually implementing hashing. Instead of:

```
helper context A def: b : B oppositeOf a =
    B.allInstances()->select(e |
                       e.name = self.name)->asSequence()->first();
```

we would have:

```
helper context A def: b : B oppositeOf a linkWhen self.name = b.name;
```

(3) Corresponds to explicit rule polymorphism through inheritance from abstract rules. As a matter of fact, because of present lack of type checking in ATL engine, references to other rules cannot be checked for correctness. Therefore, although polymorphism currently works, it is not statically checked for sanity.

(4) In ATL, source models are read-only and target models are write-only; this prohibits in-place transformations. However, such transformations are quite common in certain domains. Therefore, ATL provides a mechanism to answer this need: refining mode. This mode can be used for transformations having the same source and target metamodel. Unmatched source elements are automatically copied into target model, as if a default copying rule was present. Note that refining mode may be implemented as in-place model transformation by an ATL engine provided the result is proved to be the same.

7 Related and Future Work

In the last couple of years we observed a number of proposals for model transformation languages. Some of them are a response to the QVT RFP issued by OMG [13]. As we explained in Section 2 ATL is applicable in QVT transformation scenarios where transformation definitions are specified on the base of MOF metamodels. However, ATL is designed to support other transformation scenarios going beyond QVT context where source and target models are artifacts created with various technologies such as databases, XML documents, etc. In that way ATL serves the purpose of the AMMA platform as a generic data management platform.

Another class of transformation approaches relies on graph transformations theory [1][14]. ATL is not directly based on the mathematical foundation of these approaches. An interesting direction for future research is to formalize the ATL

semantics in terms of graph transformation theory. The declarative part of ATL is especially suitable for this.

Some approaches stress on providing graphical syntax for transformation definitions [9][15]. Currently ATL does not have a graphical syntax. This is a second important direction for future research. It is related to the model weaving approach [7] (another integral part of the AMMA platform) in which relations among model elements are established by using a visual tool. These relations may be interpreted in various ways: as compositional operators, transformations, equivalence, etc.

ATL has already been used in various contexts. More than twenty examples are already published on GMT website some of them consisting of several ATL programs. In all examples XMI is used to represent models. However, in a large number of them, it is only used as intermediary representation between transformations. Source and target are indeed often not XMI models but other kind of XML documents, text files, spreadsheets, etc. This consideration is important because we think model transformation cannot be limited to XMI if real-life problems are to be solved [2].

In [4] we present another application of ATL by showing how it can be used to check models if they satisfy given constraints. A simple specific target metamodel is defined to represent diagnostics resulting from evaluation of these constraints as a set of problems (i.e. constraint violations). OCL constraints defined on a metamodel can then be translated into ATL rules generating such problems. Diagnostic models can subsequently be transformed into any convenient representation. We plan to extend this work and show how ATL can be used to compute any kind of metrics on models.

Although ATL does not permit bidirectional transformation specifications this becomes possible by coupling ATL with model weaving [7]. We showed an example of this technique at GTTSE'2005 (Summer School on Generative and Transformational Techniques in Software Engineering) hold in Braga, Portugal in July 2005.

Static type checking of OCL expressions used in ATL programs is not implemented in current compiler. It is however necessary to be closer to OCL 2.0 specification. Moreover, it will help transformation developers by reducing the number of runtime errors requiring recompilation and relaunching of programs. We believe an ATL type checker could be implemented in the form of an ATL transformation. This approach seems indeed more interesting than using ad-hoc means (e.g. Java code), but is more challenging since it relies on ATL itself. We think current ATL engine is almost mature enough for this purpose and we already have an initial prototype working on simple OCL expressions (i.e. using only primitive types).

8 Conclusions

In this paper we presented ATL: a hybrid model transformation language developed as a part of the ATLAS Model Management Architecture. ATL is supported by a set of development tools built on top of the Eclipse environment: a compiler, a virtual machine, an editor, and a debugger.

The current state of ATL tools already allows solving non-trivial problems. This is demonstrated by the increasing number of implemented examples and the interest shown by the ATL user community that provides a valuable feedback.

The applicability of ATL was demonstrated in a case study. We identified alternative ways for implementing the case study. Alternatives are based of different programming styles, e.g. declarative and imperative. ATL allows both styles to be used in transformation definitions depending on the problem at hand. We encourage a declarative approach for defining transformations whenever possible. We believe that this approach allows transformation developers to focus on the essential relations among the model elements and to leave the handling of complex execution algorithms and optimizations to the ATL compiler and virtual machine.

References

[1] Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A.The Design of a Simple Language for Graph Transformations, Journal in Software and System Modeling, in review, 2005

[2] Bézivin, J., Dupe, G., Jouault, F., Pitette, G., and Rougui, J. E. First experiments with the ATL model transformation language: Transforming XSLT into XQuery, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, Anaheim, CA, USA (2003)

[3] Bézivin, J., Jouault, F., and Touzet, D. An Introduction to the ATLAS Model Management Architecture. Research Report LINA, (05-01)

[4] Bézivin, J., Jouault, F. Using ATL for Checking Models. To appear in the proceedings of the GraMoT workshop of GPCE 2005 conference in Tallinn, Estonia

[5] Budinsky, F., Steinberg, D., Raymond Ellersick, R., Ed Merks, E., Brodsky, S. A., Grose, T. J. Eclipse Modeling Framework, Addison Wesley, 2003

[6] Czarnecki, K., Helsen, S. Classification of model transformation approaches. OOPSLA2003 Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA, 2003

[7] Didonet Del Fabro, M, Bézivin, J, Jouault, F, and Valduriez, P. Applying Generic Model Management to Data Mapping. To appear in the Proceedings of the Journées Bases de Données Avancées (BDA05), 2005

[8] Eclipse Foundation, Generative Model Transformer Project, http://www.eclipse.org/gmt/

[9] Kalnins, A., Barzdins, J., Celms, E. Model transformation language MOLA. In U. Asmann (Ed.), Proceedings of Model Driven Architecture: Foundations and Applications 2004. Linkoping, Sweden, 2004

[10] Netbeans Meta Data Repository (MDR). http://mdr.netbeans.org

[11] OMG. Meta Object Facility (MOF) Specification, version 1.4, OMG Document formal/2002-04-03

[12] OMG. Object Constraint Language (OCL). OMG Document ptc/03-10-14

[13] OMG. MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10, 2002

[14] Varró, D., Varró, G., Pataricza, A. Designing the automatic transformation of visual languages. Journal of Science of Computer Programming, vol. 44, pp. 205-227, Elsevier, 2002

[15] Willink, E. UMLX: A graphical transformation language for MDA. In A. Rensink (Ed.), Model Driven Architecture: Foundations and Applications 2003, CTIT Technical Report TR-CTIT-03-27, University of Twente, the Netherlands, 2003