Transforming Models with ATL*

Frédéric Jouault and 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 and semantics by using examples. 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. In addition to the current features, the planned future ATL features are briefly discussed.

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 common, 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 of problems. The language is named ATL (ATLAS Transformation Language) and is developed as a part of the AMMA (ATLAS Model Management Architecture) platform [2]. ATL is a hybrid language, i.e. it is a mix of declarative and imperative constructs.

* Work partially supported by ModelWare, IST European project 511731.

J.-M. Bruel (Ed.): MoDELS 2005 Workshops, LNCS 3844, pp. 128 – 138, 2006.

We present the syntax and semantics of ATL informally by using examples. Space limit does not allow presenting the full ATL grammar and a detailed description of its semantics. A simple case study illustrates the usage of the language.

The paper is organized as follows. Section 2 gives an overview of the context 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 a brief comparison with other approaches for model transformations and outlines directions for future work. Section 7 gives conclusions.

2 General Overview of the ATL Transformation Approach

ATL is applied in a transformational pattern shown in Fig. 1. In this pattern a source model *Ma* is transformed into a target model *Mb* according to a transformation definition *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 [8].

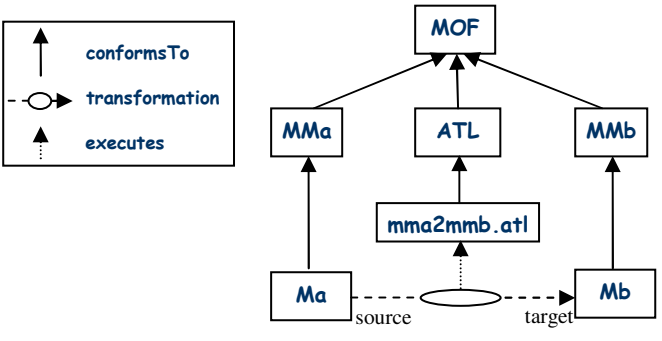

Fig. 1. Overview of ATL transformational approach

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 write-only target models. 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 the ATL language. The syntax of the language is presented based on examples (sections 3.1-3.4). Then in section 3.5 we briefly describe the execution semantics of ATL.

3.1 Overall Structure of Transformation Definitions

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 the transformation module and declares the source and target models. Below we give an example header section:

```
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 models. 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 ([9], 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 on a source metamodel type since target models are not navigable. *Operation* helpers define operations in the context of a model element or in the context of a module. They can have input parameters and can use recursion. *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 constraints about termination and cycles.

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 other elements. To illustrate the syntax of attribute helpers we consider an example.

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

The attribute helper *allAtributes* is used to determine 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 2-7). This is an example of a recursive helper (line 4).

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* (an OCL Boolean expression). A source pattern is evaluated to a set of matches in source models.

The target pattern is composed of a set of *elements*. Every element 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 initialization expression for the feature value. The following snippet shows a simple matched rule in ATL.

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

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 3) specifies that only persistent classes without superclasses will be matched.

The target pattern contains one element of type *Table* (line 5) assigned to the variable *t*. This element has one binding that specifies an expression for initializing 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. For a given match the target elements of the specified types are created in the target model and their features 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.

The feature initialization uses a value resolution algorithm, called *ATL resolve algorithm*. The algorithm is applied on the values of binding expressions. If the value type is primitive, then the value is assigned to the corresponding feature. If its type is a metamodel type or a collection type there are two possibilities:

- if the value is a **target element** it is assigned to the feature;
- if the value is a **source element** it is first resolved into a target element using internal traceability links. The resolution results in an element from

the target model created from the source element by a given rule. After the resolution the target model element becomes the value of the feature;

Thanks to this algorithm, target elements can be linked together using source model navigation.

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 single match as many times as it is referred to by other rules, every 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;

The ATL resolution algorithm 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. The source pattern types in the parent rule may be replaced by their subtypes in the subrule source pattern. The guard of a subrule forms a conjunction with the guard of the parent rule.

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.

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 complex transformation algorithms behind a simple syntax.

However, in some cases complex source-domain or target-domain specific algorithms may be required and it may be difficult to specify a pure declarative solution for them. There are several possible approaches to this issue. We consider two of them:

- allow **native operation calls** to modules written in an arbitrary language. 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;

ATL has an imperative part based on two main constructs:

- **called rules**. A called rule is basically a procedure: it is invoked by name and may take arguments. Its implementation can be native or specified in ATL;
- **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 briefly sketch some aspects of the execution algorithm of ATL transformations. The execution starts by invoking an optional called rule marked as *entry point*. This rule, in turn, may invoke other called rules. Then the algorithm executes the standard matched rules (some of them may contain an action block). Rule matching and rule application are separated in two phases. In the first phase all patterns of the rules are matched against the source model(s). For every match the target elements are created. Traceability links are also created in this phase. In the second phase all the bindings for the created target elements are executed. ATL resolution algorithm and execution of lazy rules are applied if necessary.

The algorithm does not suppose any order in rule matching, target elements creation for a match, and target elements initialization. Action block (if present) must, however, be executed after having applied the declarative part of the rule.

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. Since the source models are read-only, the attribute helper values may be cached. Lazy evaluation and caching improve the performance.

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. This is a consequence of the fact that 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. It is possible to have recursive helpers that do not terminate. In this case the transformation does not terminate either. Called rules use imperative constructs and the termination is not guaranteed. Lazy rules may introduce circular references to each other thus causing non-termination.

4 Case Study: Transforming Class to Relational Models

Because of the lack of space we present a rather simplified version of the case study given in the call for papers of the workshop. For the full version the reader is referred to [5]. The case study requires transformation of simple class models to relational models. The source and target metamodels are shown in Fig. 2.

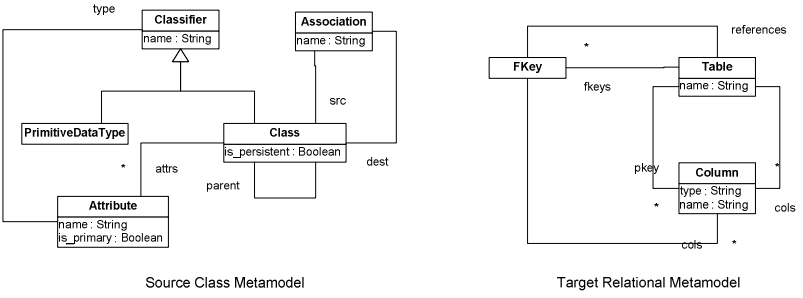

Fig. 2. Source and target metamodels

Classes in the source model have names and a number of attributes. They 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*). Every relational model contains a number of tables. Each table has a number of columns, some of them form a primary key. A table may be associated to zero or more foreign keys. We will focus only on two transformation rules:

- Persistent classes that are roots of an inheritance hierarchy are transformed to tables;
- Table columns are derived from the attributes of a class. Attributes of a primitive type are transformed to a single column. If the attribute is primary it results in a column from the primary key. Attributes of a non-primitive type are transformed to a set of columns derived from the type attributes. This rule is applied recursively until a set of primitive attributes is obtained (flattening);

Below we give the transformation definition for the case study.

```
1.  module SimpleClass2SimpleRDBMS;
2.  create OUT : SimpleRDBMS from IN : SimpleClass;
3.
4.  helper context SimpleClass!Class def :
5.    flattenedAttributes : Sequence(Sequence(SimpleClass!Attribute)) =
6.      self.attrs->collect(a |
7.       if a.type.oclIsKindOf(SimpleClass!PrimitiveDataType) then Sequence {a}
8.         else a.type.flattenedAttributes->collect (t | t->prepend(a))
9.        endif
10.     )->flatten();
11.
12. rule PersistentClass2Table{
13.   from
14.     c : SimpleClass!Class (c.is_persistent and c.parent.oclIsUndefined())
15.   to t : SimpleRDBMS!Table (
16.       name <- c.name,
17.       cols <- c.flattenedAttributes,
18.       pkey <- c.flattenedAttributes->select(t | t->last().is_primary)
19.   )
20. }
21.
22. unique lazy rule AttributeTrace2Column {
23.   from trace : Sequence(SimpleClass!Attribute)
24.   to col : SimpleRDBMS!Column (
25.       name <- trace->iterate(a; acc : String = '' |
26.                 acc + if acc = '' then '' else '_' endif + a.name),
27.       type <- trace->last().type
28.    )
29. }
```

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. The decoration part is based on the helper *flattenedAttributes*. In the helper every class generates a sequence of traces derived from its attributes. Every trace is a sequence of attributes and will be transformed to a column. If an attribute is of a primitive type then the trace is the attribute itself (line 7). If an attribute is of a non-primitive type then it results in a set of traces derived from the traces of its type by prepending the attribute to every trace (line 8). The traces represent the paths to the primitive attributes for a given class after application of flattening.

Transformation rules use the result of the decoration part to create the elements in the target model. 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 line 17 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 contained in the *flattenedAttributes* helper. The value of the helper is resolved according to the ATL resolution algorithm. The resolution requires finding a rule that transforms the value of the expression into target model elements. In this case we have an implicit invocation of a transformation rule. The only suitable rule is *AttributeTrace2Column* unique lazy rule. This rule transforms traces to columns.

Furthermore the slot *pkey* contains the primary key of the table. Primary key is a subset of all the columns of the table. The columns in the key are created from the traces whose last element is a primary attribute (line 18). Similarly to the previous slot we have an implicit invocation of *AttributeTrace2Column* rule. This rule may be triggered multiple times over the same source. Since it is a unique lazy rule the invocations after the first time will return the same result.

It must be noted that this implementation relies on features of ATL that are not implemented yet. Current compiler does not fully support lazy rules, rules with multiple source elements, and source elements that are of OCL types (e.g. sequences). A working solution is available on the Eclipse GMT project site [5].

5 ATL Tools

ATL is accompanied by a set of tools that include the ATL transformation engine, the ATL integrated development environment (IDE) based on Eclipse, and the ATL debugger. ATL transformations are compiled to programs in specialized byte-code. Byte-code is executed by the ATL virtual machine. The virtual machine is specialized in handling models and provides a set of instructions for model manipulation.

The architecture of ATL execution engine is shown in Fig. 3. The virtual machine may run on top of various model management systems. To isolate the VM 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 are components that provide programming interface for model manipulation. Some examples are Eclipse Modeling Framework (EMF) [4] and MDR [7]. Model repository provides storage facilities for models.

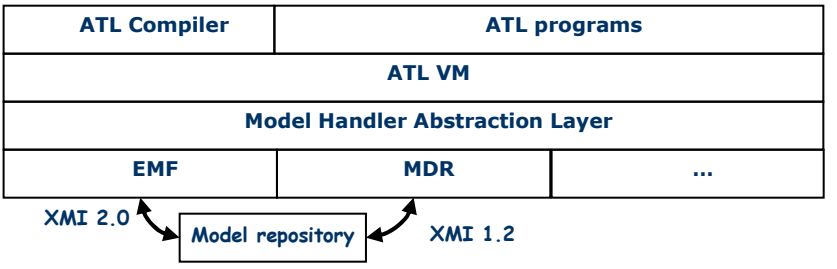

Fig. 3. The architecture of the ATL execution engine

The current ATL IDE is built on top of Eclipse platform. 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.

Table 1 presents a summary of the features of the current ATL compiler and some features planned for future extensions. Stars indicate the supported features. An explanation of some of the features is given after the table.

Table 1. ATL features summary

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		*
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			*
Refining mode (1)			*(basic)	*(improved)
Traceability			internal	external
Imperative part	ATL called rules			*
	native called rules			*
	action blocks			*
OCL type checking			Dynamic	Static (following the specification)

(1) 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. 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 the target model, as if a default copying rule was present.

6 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 [10]. As we explained in Section 2 ATL is applicable in QVT transformation scenarios where transformation definitions are specified on the base of MOF metamodels [8]. 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. A comparison between ATL and the last QVT proposal may be found in [6].

Another class of transformation approaches relies on graph transformations theory [1][11]. 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.

In [3] we present an 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.

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.

7 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., Jouault, F., and Touzet, D. An Introduction to the ATLAS Model Management Architecture. Research Report LINA, (05-01)
[3] 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
[4] Budinsky, F., Steinberg, D., Raymond Ellersick, R., Ed Merks, E., Brodsky, S. A., Grose, T. J. Eclipse Modeling Framework, Addison Wesley, 2003
[5] Eclipse Foundation, Generative Model Transformer Project, http://www.eclipse.org/gmt/
[6] Jouault, F., Kurtev, I. On the Architectural Alignment of ATL and QVT. Proceedings of ACM SAC 2006, Track on Model Transformations, Dijon, France, 2006, to appear
[7] Netbeans Meta Data Repository (MDR). http://mdr.netbeans.org
[8] OMG. Meta Object Facility (MOF) Specification, version 1.4, OMG Document formal/2002-04-03
[9] OMG. Object Constraint Language (OCL). OMG Document ptc/03-10-14
[10] OMG. MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10, 2002
[11] 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