Science of Computer Programming 72 (2008) 31–39

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

ATL: A model transformation tool

Frédéric Jouault [a,*], Freddy Allilaire [a], Jean Bézivin [a], Ivan Kurtev [b]

[a] ATLAS (INRIA & LINA), University of Nantes, 2, rue de la Houssinière BP 92208, 44322, Nantes, France
[b] Software Engineering Group, University of Twente, P.O. Box 217, 7500 AE, Enschede, The Netherlands

ARTICLE INFO

Article history:
Received 30 April 2006
Received in revised form 16 August 2007
Accepted 17 August 2007
Available online 8 May 2008

Keywords:
Model engineering
Model transformation
M2M (model-to-model transformation)

ABSTRACT

In the context of Model Driven Engineering, models are the main development artifacts and model transformations are among the most important operations applied to models. A number of specialized languages have been proposed, aimed at specifying model transformations. Apart from the software engineering properties of transformation languages, the availability of high quality tool support is also of key importance for the industrial adoption and ultimate success of MDE. In this paper we present ATL: a model transformation language and its execution environment based on the Eclipse framework. ATL tools provide support for the major tasks involved in using a language: editing, compiling, executing, and debugging.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Model transformations play an important role in the Model Driven Engineering (MDE) approach. Developing model transformation definitions is expected to become a common task in model driven software development. Software engineers should be supported in performing this task by mature MDE tools and techniques in the same way as they are presently supported by classical IDEs, compilers, and debuggers in their everyday programming 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 model transformation languages have been proposed [1,11,13].

In this paper we describe ATL (ATLAS Transformation Language): a domain-specific language for specifying model-to-model transformations. It is a part of the AMMA (ATLAS Model Management Architecture) platform [2]. ATL is inspired by the OMG QVT requirements [11] and builds upon the OCL formalism [12]. The choice of using OCL is motivated by its wide adoption in MDE and the fact that it is a standard language supported by OMG and the major tool vendors. In that way we capitalize on the existing user familiarity with OCL.

ATL is a hybrid language, i.e. it provides a mix of declarative and imperative constructs. The major language features and the supporting development tools are presented. We informally explain the syntax and semantics of ATL by using a simple case study as an example.

The paper is organized as follows. Section 2 explains the basic concepts related to model transformations and to the operational context of ATL. Section 3 presents the language constructs on the base of examples. Section 4 describes the tool support available for ATL: the ATL virtual machine, the ATL compiler, the IDE based on Eclipse, and the debugger. Section 5 presents a brief comparison with other approaches for model transformations. Section 6 gives conclusions.

* Corresponding author.
E-mail addresses: frederic.jouault@univ-nantes.fr (F. Jouault), freddy.allilaire@univ-nantes.fr (F. Allilaire), jean.bezivin@univ-nantes.fr (J. Bézivin), kurtev@ewi.utwente.nl (I. Kurtev).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.08.002

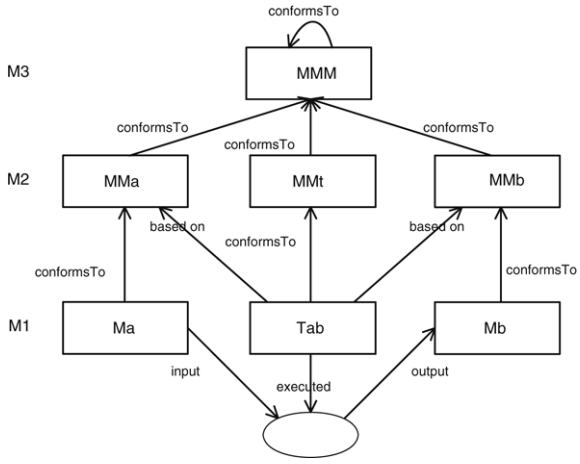

Fig. 1. Model transformation pattern.

2. Models and model transformations

In MDE, models are considered as the unifying concept in IT engineering. Traditionally, models have been used as initial design sketches mainly meant to communicate ideas among developers. MDE promotes models to primary artifacts that drive the whole development process. The notion of model goes beyond the narrow view of semi-formal diagram thus requiring much more precise definitions and modeling languages. In this section we introduce the basic terminology needed for the further discussion.

The MDE community has been using the concepts of *model*, *metamodel*, and *metametamodel* for quite some time. A model is a representation of a system. It captures some characteristics of the system and provides knowledge about it. In MDE we are interested in models expressed in precise languages. The conceptual foundation of a modeling language, when expressed as a model, is called *metamodel*. Often, metamodels are considered as definitions of the abstract syntax of modeling languages. The relation between a model expressed in a language and the metamodel of this language is called *conformsTo*. Metamodels are in turn expressed in a modeling language called metamodeling language. Its conceptual foundation is captured in a model called *metametamodel*. Models, metamodels, and metametamodel form a three-level architecture with levels named M1, M2, and M3 respectively. For a precise definition of these concepts based on multi-graphs the reader is referred to [8].

The principles of MDE may be implemented in several standards. For example, OMG proposes a standard metametamodel called Meta Object Facility (MOF) [10]. An example of a metamodel in the context of OMG standards is the UML metamodel.

An MDE development process is a series of transformations over models. Usually, transformations are refinement steps over models that decrease the level of abstraction but other forms may also be found in transformation chains. The goal is to produce a model that contains enough details for automatic generation of executable code. Other transformation scenarios in MDE are refactoring, reverse engineering, data translation between heterogeneous data sources, etc.

In general, model transformations may be implemented in different ways, for example, by using a general purpose programming language. In this paper we focus on transformations expressed in a specialized model transformation language.

Model transformations in MDE follow a common pattern known as *model transformation pattern* shown in Fig. 1 *Tab* is a transformation program which execution results in automatic creation of model *Mb* from *Ma*. These three entities (*Tab*, *Mb*, and *Ma*) are all models conforming to *MMt*, *MMb,* and *MMa* metamodels, respectively. *MMt* corresponds to the abstract syntax of the transformation language.

The three metamodels conform to a metametamodel named *MMM*. In the context of OMG standards, the metametamodel *MMM* is the MOF.

3. ATLAS transformation language

ATL is applied in the context of the transformation pattern shown in Fig. 2. 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 conforming to the *ATL* metamodel. All metamodels conform to the *MOF*.

ATL is a hybrid transformation language. It contains a mixture of declarative and imperative constructs. We encourage a declarative style of specifying transformations. 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 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.

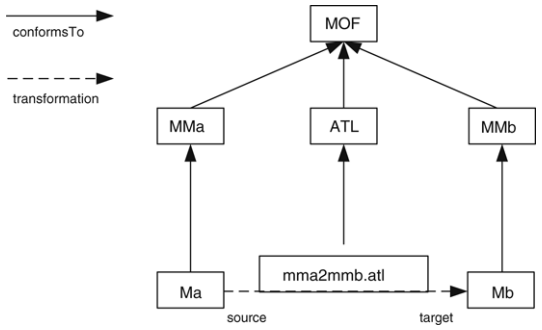

Fig. 2. Overview of the ATL transformational approach.

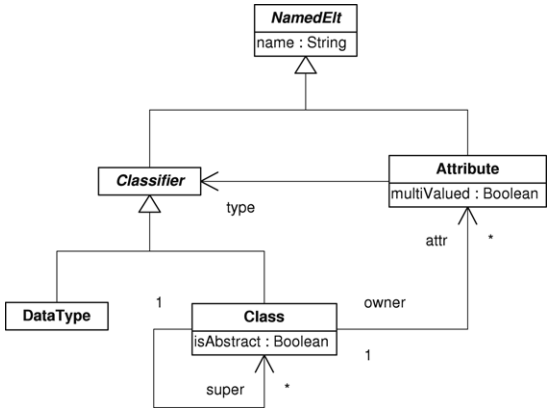

Fig. 3. Class metamodel.

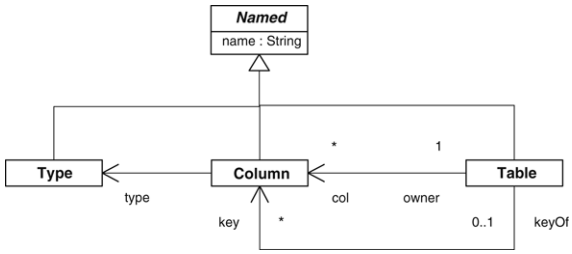

Fig. 4. Relational metamodel.

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 to it are not allowed. Target model cannot be navigated. A bidirectional transformation is implemented as a couple of transformations: one for each direction.

Source and target models for ATL may be expressed in the XMI OMG serialization format. Source and target metamodels may be expressed also in XMI or in the more convenient KM3 notation [8].

In this section we describe the features of the ATL language. The syntax of the language is presented based on examples (Sections 3.2–3.5). Then in Section 3.6 we briefly describe the execution semantics of ATL. A more detailed and formal specification of the ATL semantics can be found in [4].

3.1. Case study: Transforming class models to relational models

The case study requires transformation program that transforms simple class models to relational models. The source and target metamodels are shown in Figs. 3 and 4, respectively.

In a class model, classes have zero or more attributes and may specialize other classes. The type of attributes is either a primitive datatype or a class.

A relational model contains a set of tables. Every table has zero or more columns. Some of the columns are keys. Every column has a primitive type.

We will focus on two transformation rules described informally in the following way:

- For each class in the source model a table in the target model is created with the same name as the class name;
- For each single valued attribute a column is created in the target model.

The complete case study may be found in the accompanying software bundle.

3.2. 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. Here is an example header section:

```
module Class2Relational;
create OUT : Relational from IN : Class;
```

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 *Class* and *Relational* 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.

3.3. 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 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.

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 operation helpers we consider an example.

```
helper context String def: firstToLower() : String =
  self.substring(1, 1).toLower() + self.substring(2, self.size());
```

The helper *firstToLower* is defined in the context of the *String* type (indicated by the keyword *context*) and its values are strings. The OCL expression used to calculate the value of the helper is given after the '=' symbol. When executed on a given string the helper returns the same string with the first letter not capitalized.

3.4. 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 3.5 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 the source metamodels and from 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 target 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. This rule implements the logic for transforming classes to tables.

```
1. rule Class2Table {
2.   from
3.     c : Class!Class
4.   to
5.     out : Relational !Table (
6.       name <- c.name,
7.       col <- Sequence {key}->union(c.attr->select(e | not e.multiValued)),
```

```
8.        key <- Set {key}
9.      ),
10.
11.     key : Relational!Column (
12.       name <- 'objectId',
13.       type <- thisModule.objectIdType
14.     )
15. }
```

The rule name *Class2Table* is given after the keyword *rule* (line 1). The rule source pattern specifies one variable of type *Class* (line 3). The target pattern contains one element of type *Table* (line 5) assigned to the variable *out*, and one element of type *Column* (line 11) assigned to the variable *key*. These elements have bindings that specify expressions for initializing their features. The symbol '<-' is used to delimit the feature to be initialized (left-hand side) from the initialization expression (right-hand side). For example, line 7 presents a binding that initializes the feature *col* of the table. The value is obtained by obtaining all the columns derived from non multi-valued attributes, and uniting them with the *key* column created in the same rule. The expression in line 7 relies on a resolution algorithm that returns target model elements created from given source model elements. This algorithm is explained in the next section. The rule that creates columns from non multi-valued attributes is given below.

```
1. rule ClassAttribute2Column  {
2.   from
3.     a : Class!Attribute (
4.       a.type.oclIsKindOf(Class!Class) and not a.multiValued
5.     )
6.   to
7.     foreignKey : Relational !Column  (   TODO : Why this is foreignKey ? It should be normal column ?
8.       name <- a.name +  'Id',
9.       type <- thisModule . objectIdType
10.  )
11. }
```

This rule shows an example usage of a guard expression in the source pattern (line 4). It ensures that only non multi-valued attributes will be selected for transformation by this rule.

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 they are triggered.

- *Standard* rules are applied once for every match that can be found in source models. In the presented example only standard matched rules are used;
- *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. Lazy rules are indicated by the keyword *lazy*;
- *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. Rules of this type are indicated by the *unique lazy* keywords.

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 modifications: subtyping target types, adding bindings, replacing bindings, and adding new target elements.

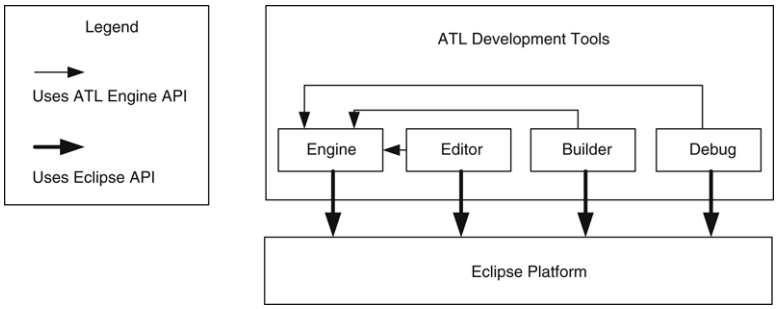

Fig. 5. ATL development tools architecture.

3.5. Imperative features of ATL

Sometimes complex transformation algorithms may be required and it may be difficult to specify a purely declarative solution for them. 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;
- **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.

3.6. 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 are implemented in the current version of ATL, and typically 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 mutually circular references, thus causing non-termination.

4. ATL development tools

ATL is accompanied by a set of tools built on top of the Eclipse platform. Fig. 5 represents the overall architecture of the ATL Development Tools (ADT). ADT is composed of the ATL transformation engine (*Engine* block), and of the ATL Integrated Development Environment (IDE: *Editor*, *Builder* and *Debug* blocks). Complete usage documentation is provided in the ATL user manual. Each block is described in the remaining part of this section.

4.1. Engine

The ATL engine is responsible for dealing with core ATL tasks: compilation and execution. ATL transformations are compiled to programs in specialized byte-code. This byte-code is executed by the ATL Virtual Machine (VM). The VM is specialized in handling models and provides a set of instructions for model manipulation.

The architecture of ATL execution engine is shown in Fig. 6.

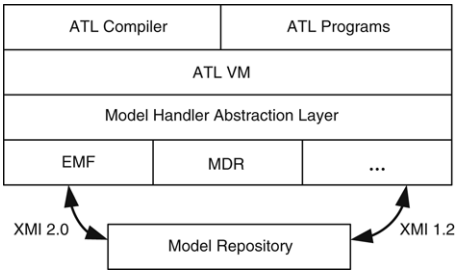

Fig. 6. The architecture of the ATL execution engine.

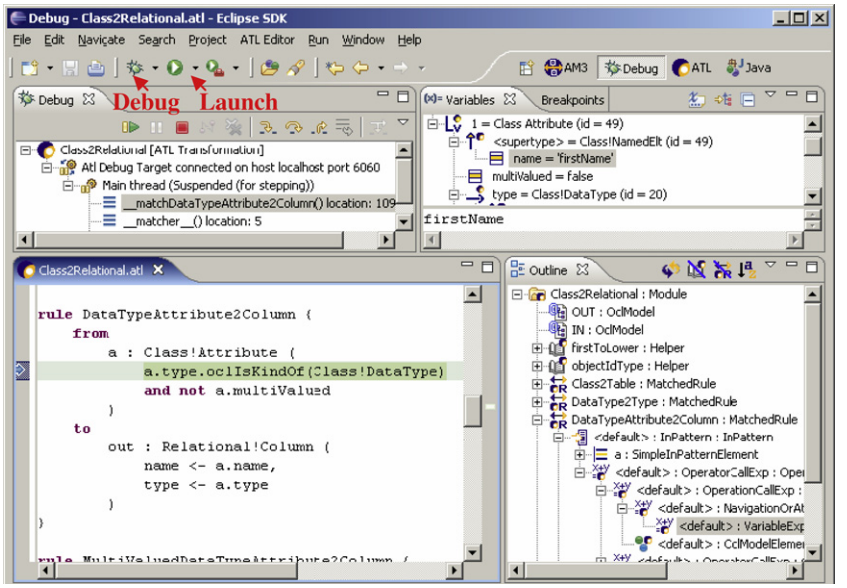

Fig. 7. ATL editor and debugger screenshot.

The VM 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 interfaces for model manipulation. Some examples are Eclipse Modeling Framework (EMF) [3], and MDR [9]. Model repository provides storage facilities for models. Within Eclipse, it corresponds to the workspace file system.

4.2. Editing

The ATL editor supports syntax highlighting, error reporting, and outline view (i.e. tree-based representation of the ATL program). The bottom left part of Fig. 7 shows how the *Class2Relational.atl* example is represented under the editor. The bottom right part of the figure shows the corresponding outline.

4.3. Building and launching ATL transformations

ATL compiler is automatically called on each ATL file in each ATL project during the Eclipse build process. By default, this process is triggered when a file is modified (e.g., saved).

Executing an ATL transformation requires the declared source and target models and metamodels to be bound to actual models (i.e. XMI files typically ending in *.xmi* or *.ecore*). This is done in the launch configuration wizard. Fig. 8 gives a screenshot of this wizard and shows the correspondence between the user interface and the operational context of the transformation given in Fig. 1. Solid arrows map models and metamodels to their declarations whereas dashed arrows map the declarations to the corresponding files. For instance, the source model *Ma* is declared as model *IN* with metamodel *Class*. The file corresponding to model *IN* is *sample-Class.ecore*. The first tab that can be seen on the top left of Fig. 8 is entitled "ATL Configuration" and is used to specify the location of the transformation to execute. This wizard is accessible from Eclipse *Launch* button (see Fig. 7).

The ATL engine delegates reading and writing models to the underlying model handler. When EMF is used, for instance, source and target models must be in EMF XMI 2.0. They can be edited with EMF reflective editor, which represents models

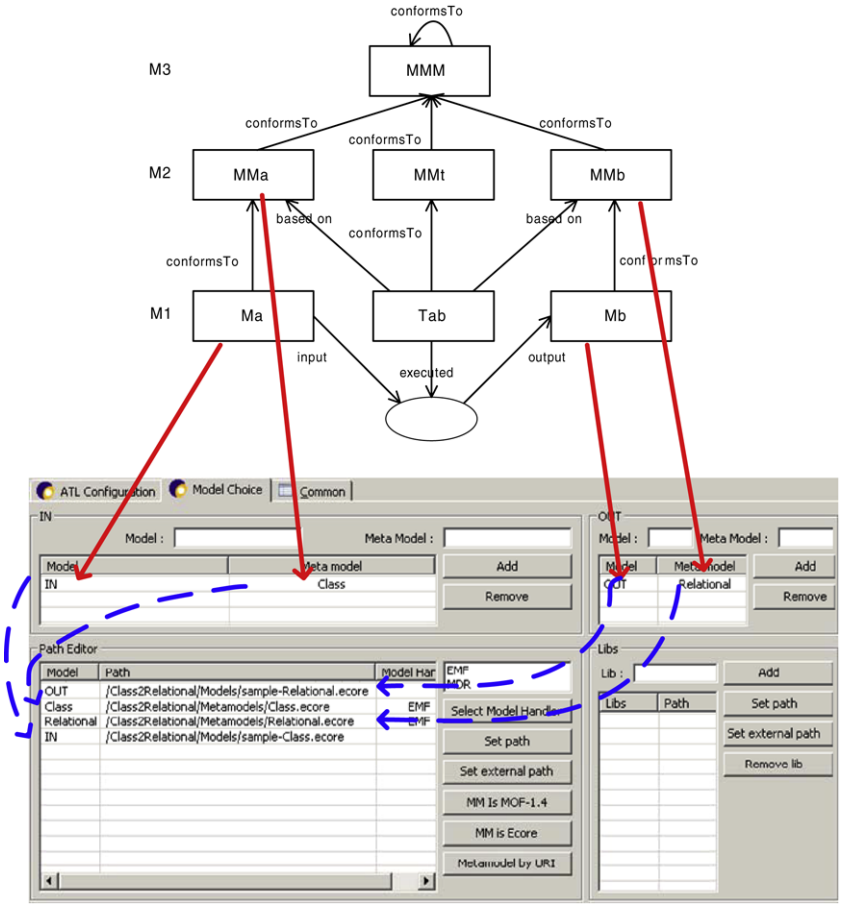

Fig. 8. ATL launch configuration.

as trees. More complex transformation scenarios can use other kinds of formats (e.g., XML, textual). This is illustrated by the CPL2SPL example provided with ATL Development Tools. However, dealing with such formats involves more than model-to-model transformation. It is consequently out of the scope of this paper.

4.4. Debugging

ATL transformations may be debugged using the same launch configuration used for launching. The only difference is that we now use the *Debug* button (see Fig. 7) instead of *Launch*. Transformations can be executed step-by-step or run normally. In this case, execution stops when an error occurs or a breakpoint is reached. The current context (i.e. values of variables) may be analyzed using the variable view (see the top right part of Fig. 7). It enables simple navigation in source and target models from the current context (rule or helper).

5. Related work

In the last couple of years we have observed a number of proposals for model transformation languages. Some of them are a response to the QVT RFP issued by OMG [11]. ATL is applicable in QVT transformation scenarios where transformation definitions are specified on the base of MOF metamodels. ATL is able to perform other transformation scenarios going beyond the QVT context where source and target models are artifacts created with various technologies such as databases, XML documents, etc. For this purpose ATL relies on AMMA platform features that are beyond the scope of this paper. A comparison between ATL and the last QVT proposal may be found in [6].

Currently, the available tools for executing QVT transformations are still immature. For example, Borland Together Architect 2006 R2 supports an older version of the QVT Operational Mappings specification and not all the features are implemented. Similarly, ModelMorf only partially supports the Relational QVT language. ATL is independent from the specification of the QVT languages and is able to perform the model-to-model transformations in the QVT operational context.

Furthermore, the ATL tool supports most of the planned features of the ATL language as specified in Table 2 of [7]. There are only three notable exceptions (i.e. features not implemented yet): static type checking, helpers in the context of collection types, and more strongly typed resolution, which depends on static type checking.

An important class of transformation approaches relies on graph transformations techniques [1,13]. 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.

6. 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. ATL allows both imperative and declarative approaches to be used in transformation definitions depending on the problem at hand.

ATL is currently used or evaluated on more than 100 sites, academic or industrial. There is a library of ATL transformations available in open source from the M2M Eclipse project. 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 rapidly growing ATL user community that provides a valuable feedback. More than eighty different scenarios accounting for more than one hundred and sixty individual transformations are available on ATL M2M website [5].

Acknowledgements

The present work is partially supported by the IST European MODELPLEX project (MODELing solution for comPLEX software systems, FP6-IP 34081), the OpenEmbeDD project, and the Usine Logicielle project of the System@tic Paris Region Cluster.

Appendix. Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2007.08.002. The archive contains a ready-to-use bundle of ATL Development Tools (including Eclipse EMF [3], and ADT). It also includes ATL starter guide and ATL user manual.

References

[1] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, A. Vizhanyo, The Design of a simple language for graph transformations, Journal in Software and System Modeling (2005) (submitted for publication).
[2] J. Bézivin, F. Jouault, P. Rosenthal, P. Valduriez, Modeling in the large and modeling in the small, in: MDAFA'2004, in: LNCS, vol. 3599, Springer-Verlag, 2004, pp. 33–46.
[3] F. Budinsky, D. Steinberg, R. Raymond Ellersick, E. Ed Merks, S.A. Brodsky, T.J. Grose, Eclipse Modeling Framework, Addison Wesley, 2003.
[4] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bezivin, A. Pieearantonio, Extending AMMA for supporting dynamic semantics specifications of DSLs. LINA Research Report number 06.02, University of Nantes, April, 2006.
[5] Eclipse project, ATL Home Page. http://www.eclipse.org/m2m/atl/.
[6] F. Jouault, I. Kurtev, On the architectural alignment of ATL and QVT, in: Proceedings of ACM Symposium on Applied Computing, SAC 06, Model Transformation Track, Dijon, Bourgogne, France, April 2006.
[7] F. Jouault, I. Kurtev, Transforming models with ATL, in: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, 2005.
[8] F. Jouault, J. Bézivin, KM3: A DSL for metamodel specification FMOODS 2006, Bologna, Italy, 14–16 June 2006.
[9] Netbeans meta data repository (MDR). http://mdr.netbeans.org.
[10] OMG/MOF meta object facility (MOF) specification, OMG Document AD/97-08-14, September 1997. Available from: www.omg.org.
[11] OMG/RFP/QVT MOF 2.0 query/views/transformations RFP, OMG document ad/2002-04-10. Available from: www.omg.org.
[12] OMG. Object constraint language (OCL), OMG Document ptc/03-10-14, 2003.
[13] D. Varró, G. Varró, A. Pataricza, Designing the automatic transformation of visual languages, Journal of Science of Computer Programming 44 (2002) 205–227.