

ATL:
Atlas Transformation Language

User Manual
- *version 0.6* -

August 2005

by
ATLAS group
LINA & INRIA
Nantes

Content

1	Introduction	1
2	OCL Types and OCL Expressions in ATL	1
2.1	<i>OCL Primitive Types.....</i>	<i>1</i>
2.1.1	Operations on Primitive Types.....	1
2.1.2	Examples of Operation on Primitive Datatypes	1
3	OCL Collections.....	2
3.1.1	Collection Operations.....	2
3.1.2	Iterator over Collections.....	2
3.1.3	Sequence Operations	3
3.1.4	Set Operations	3
3.1.5	Bag Operations.....	4
3.1.6	Collection Operation Examples.....	4
3.2	<i>OCL Model Elements.....</i>	<i>5</i>
3.2.1	Examples of Class Operations on MOF 1.4	5
3.2.2	Enumerations.....	6
3.3	<i>OCL If Expression.....</i>	<i>6</i>
3.4	<i>OCL Let Expression.....</i>	<i>6</i>
3.5	<i>OCL Comment</i>	<i>7</i>
3.6	<i>OCL Tips and Tricks.....</i>	<i>7</i>
4	ATL Modules	7
4.1	<i>Preparation</i>	<i>8</i>
4.2	<i>Header Section.....</i>	<i>9</i>
4.3	<i>Import Section.....</i>	<i>9</i>
4.4	<i>Helpers.....</i>	<i>9</i>
4.5	<i>Rules.....</i>	<i>11</i>
5	ATL Advanced Features	12
5.1	<i>Queries and the Generation of Text.....</i>	<i>12</i>
5.2	<i>Libraries.....</i>	<i>12</i>
5.3	<i>Complex Headers.....</i>	<i>13</i>
5.4	<i>Rules with Multiple Instantiations</i>	<i>14</i>
5.5	<i>Navigation and Multiple Instantiations.....</i>	<i>14</i>
5.6	<i>Flexible Runtime Instantiation of Target Elements.....</i>	<i>15</i>
5.7	<i>ATL Tips and Tricks.....</i>	<i>16</i>
6	The ATL IDE for Eclipse.....	17
6.1	<i>Installing the ATL IDE for Eclipse.....</i>	<i>17</i>
6.2	<i>Basic Features</i>	<i>17</i>
6.2.1	Perspectives.....	17
6.2.2	Navigator.....	18
6.2.3	Editors	18
6.2.4	Outline.....	18

6.2.5	Console.....	19
6.3	<i>Creating a Project</i>	19
7	Compilation	20
7.1	<i>Setting up a Launch Configuration</i>	20
7.2	<i>Running a Launch Configuration</i>	22
7.3	<i>Debugging</i>	22
7.4	<i>Log File</i>	23
8	References	23
Appendix A:	Example Code	24

Figures List

Figure 1.	The Book metamodel.....	8
Figure 2.	The Publication metamodel	8
Figure 3.	Simple inheritance case	16
Figure 4.	ATL IDE for Eclipse.....	17
Figure 5.	Synchronized Cursors of Outline and ATL Editor	19
Figure 6.	ATL Configuration	21
Figure 7.	ATL Model Configuration.....	22
Figure 8.	The Book metamodel.....	24
Figure 9.	The Publication metamodel	24

1 Introduction

ATL, the Atlas Transformation Language, is the ATLAS INRIA & LINA research group answer to the OMG MOF [6]/QVT RFP [7]. It is a model transformation language specified both as a metamodel and as a textual concrete syntax. It is a hybrid of declarative and imperative. The preferred style of transformation writing is declarative, which means simple mappings can be expressed simply. However, imperative constructs are provided so that some mappings too complex to be declaratively handled can still be specified.

An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models.

2 OCL Types and OCL Expressions in ATL

Since the OCL standard is omnipresent in ATL programs, it is worth while introducing them right at the beginning. ATL is designed and implemented with respect to the OCL standard. Derivations from the standard are indicated with a specific remark.

2.1 OCL Primitive Types

OCL has four basic primitive datatypes:

- Boolean (true, false);
- Integer (1, -5, 2, 34, 26524, ...)
- Real (with the values 1.5, 3.14, ...);
- String ('To be or not to be...').

Furthermore, OCL has the following comparators: <, >, =, =>, =<.

2.1.1 Operations on Primitive Types

Integer and Real have the following operations:

- *, +, -, /, div(), abs(), mod(), max(), min(), sum(), sin(), cos()

Boolean has the operations:

- and, or, xor, not, implies, if-then-else

String has the operations:

- concat(), size(), substring(), toInteger() and toReal()

A difference between OCL and ATL is that in ATL you can use the operator + for the concatenation of Strings.

Early versions of ATL use div instead of / or div() for division

2.1.2 Examples of Operation on Primitive Datatypes

In the following some usage examples of OCL operations on primitive datatypes are illustrated:

- Addition of two integers:

1 + 1

- Further mathematical operations:

`1 - 80 div 2`

- Comparisons of integers:

`1 < 2 and 1 > 2`

- Boolean operations:

`true or false`

3 OCL Collections

Collection is the abstract superclass of Set, OrderedSet, Bag and Sequence. These Collection classes have the following characteristics:

- Set is a collection without duplicates. Set has no order;
- OrderedSet is a collection without duplicates. OrderedSet has an order;
- Bag is a collection in which duplicates are allowed. Bag has no order;
- Sequence is a collection in which duplicates are allowed. Sequence has an order.

3.1.1 Collection Operations

Collection has the following operations:

- The number of elements in the collection self:

`size()`

- The information of whether an object is part of a collection:

`includes()`

- The information of whether an object isn't part of a collection:

`excludes()`

- The number of times that object occurs in the collection self:

`count()`

- The information of whether all objects of a given collection are part of a specific collection:

`includesAll()`

- The information of whether none of the objects of a given collection are part of a specific collection:

`excludesAll()`

- The information if a collection is empty:

`isEmpty()`

- The information if a collection is not empty:

`notEmpty()`

3.1.2 Iterator over Collections

The following iterations over collections are possible:

- The selection of a sub-collection:

```
select()
```

- When specifying a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection) use:

```
collect()
```

- The information of whether an expression is true for all objects of a given collection:

```
forAll()
```

- The addition of all elements of a collection:

```
sum()
```

3.1.3 Sequence Operations

Sequence supports all collection operations and some specific ones:

- Adding an element at the end of a sequence:

```
append()
```

- Inserting an element at a specific point in a sequence:

```
insertAt()
```

- Casting a sequence to a set removes duplicates:

```
asSet()
```

- Casting a sequence to a bag:

```
asBag()
```

- Casting a sequence of collections to a sequence directly containing the children of the subordinate collections:

```
flatten()
```

- An example of the insertion of the element 10 at the second place is:

```
Sequence { 5, 15, 20 } -> insertAt(2,10)
```

which results in:

```
Sequence {5, 10, 15, 20}
```

3.1.4 Set Operations

Set supports all collection operations and some specific ones:

- Adding an element to the set:

```
including()
```

- Removing an element from the set:

```
excluding()
```

- Casting a set to a sequence:

```
asSequence()
```

- Casting a set to a bag:

```
asBag()
```

3.1.5 Bag Operations

Bag supports all collection operations and some specific ones:

- Adding an element to the bag:

```
including()
```

- Removing an element from the bag:

```
excluding()
```

- Casting a bag to a sequence:

```
asSequence()
```

- Casting a bag to a set:

```
asSet()
```

3.1.6 Collection Operation Examples

In the following some operations on collections are illustrated:

- Specifying a sequence literal:

```
Sequence {1, 2, 3}
```

- Is a collection empty?

```
Sequence {1, 2, 3}->isEmpty()
```

- Getting the size of a collection:

```
Sequence {1, 2, 3}->size()
```

Please compare:

```
Sequence {3, 3, 3 }->size()
```

returns 3 while

```
Set {3, 3, 3 }->size()
```

returns 1. This is the case because set eliminates duplicates and sequence not.

- Nesting sequences:

```
Sequence { Sequence { 2, 3}, Sequence {1, 2, 3}}
```

- Getting the first element of a sequence:

```
Sequence {1, 2, 3}->first()
```

- Getting the last element of a sequence:

```
Sequence {1, 2, 3}->last()
```

- Selecting all elements of a sequence that are smaller than 3:

```
Sequence {1, 2, 3, 4, 5, 6}->select( i | i <= 3)
```

- Rejecting all elements of a sequence that are smaller than 3:

```
Sequence {1, 2, 3, 4, 5, 6}->reject( i | i <= 3)
```

- Collect the names of all MOF classes:

```
MOF!Class.allInstances()->collect(e|e.name)
```

The OCL shorthand expression

```
MOF!Class.allInstances()->collect(name)
```

means semantically the same but is not yet implemented in ATL.

- All numbers in the sequence greater than 2:

```
Sequence{ 12, 13, 12}->forall( i | i>2 )
```

- Exists a number in the sequence that is greater than 2?

```
Sequence{ 12, 13, 12}->exists( i | i>2 )
```

3.2 OCL Model Elements

Model elements are defined in the source and target metamodels. Most metamodels have classes. In ATL the notation "Metamodel!Class" is used to be able to differentiate classes of different metamodels. Hence, it is possible to refer to several metamodels in the same program in ATL. The latter is not the case in OCL.

There are different OCL operations to treat and analyze classes:

- The operation `oclIsTypeOf()` checks if a given instance is an instance of a certain type (and not of one of its subtypes or of other types);
- The operation `oclIsKindOf()` checks if a given instance is an instance of a certain type or of one of its subtypes;
- The operation `allInstances()` returns you all instances of a given Type;
- The operation `oclIsUndefined()` tests if the value of an expression is undefined (e.g. if an attribute with the multiplicity zero to one is void or not. Please note: attributes with the multiplicity n are often represented with collections, which may be empty and not void).

3.2.1 Examples of Class Operations on MOF 1.4

It is very interesting to use OCL expressions in the context of the MOF metamodel [6]. Examples are given in the following.

Please compare:

```
MOF!Attribute.oclIsKindOf(MOF!ModelElement)
```

is true while

```
MOF!Attribute.oclIsTypeOf(MOF!ModelElement)
```

is false.

- Collect the names of all MOF classes:

```
MOF!Class.allInstances()->collect(e|e.name)
```

- Count the number of classes in MOF:

```
MOF!Class.allInstances()->size()
```

- Getting the names of all primitive MOF types by filtering:

```
MOF!DataType.allInstances()
->select(e|e.oclIsTypeOf(MOF!PrimitiveType))
->collect(e|e.name)
```

- Getting the names of all primitive MOF types the simple way:

```
MOF!PrimitiveType.allInstances()->collect(e|e.name)
```


- An enumeration instance in MOF:

```
MOF!VisibilityKind.labels
```

- Getting all (local and inherited) StructuralFeatures of a Class. In the following code example the names of all StructuralFeatures of the class PrimitiveTypes are displayed:

```
MOF!PrimitiveType.findElementsByTypeExtended(MOF!StructuralFeature, true)
->collect(e | e.name)
```

- Getting the names of all classes inheriting from more than one class:

```
MOF!Class.allInstances()
->select(e | e.supertypes->size() > 1)
->collect(e | e.name)
```

3.2.2 Enumerations

Enumerations are defined in the source and target metamodels. In OCL enumeration literals are written using the enumeration type two double points and the value.

The value *female* of the enumeration *Gender* is expressed in OCL in the following way:

```
Gender::female
```

while the current version of ATL uses sharp and the enumeration value but no enumeration type:

```
#female
```

Enumerations can be compared using the equal operator. Supposing *aPerson* is a variable of the type *Person* having the attribute *sex* which is of the enumeration type *Gender*, the following ATL expression is possible:

```
if aPerson.sex = #female
then
    ' Madam '
else
    ' Sir '
endif;
```

3.3 OCL If Expression

In OCL if-clauses are expressed with an if-then-else-endif structure. Neither the else-part nor the endif can be omitted.

An example of an if-clause:

```
if 3 > 2
then
    'three is greater than two'
else
    'this case should never occur'
endif
```

3.4 OCL Let Expression

Let expressions are very useful when debugging. They help displaying the value of an expression by the means of the let variable.

The let command to define variables:

```
let a : Integer = 1 in a + 1
```

Enchaining let expressions:

```
let a : Integer = 1 in let b : Integer = 2 in a + b
```

3.5 OCL Comment

As in the OCL standard, also in ATL comments start with two consecutive hyphens "--" and end at the end of the line.

The ATL editor in Eclipse colours comments with dark green, if the standard configuration is used:

```
-- this is an example of a comment
```

3.6 OCL Tips and Tricks

In C++ and Java, the optimiser stops the evaluation of a logical expression if a false value is followed by a logical AND or a true value is followed by a logical OR. It does not matter if the rest of the logical expression results in an exception or an error because it is not evaluated.

In OCL this is not the case. The expression will always be fully evaluated.

This is why rather than:

```
not person.ocIsUndefined() and person.name='Isabel'
```

you should write:

```
if person.ocIsUndefined()  
then  
    false  
else  
    person.name='Isabel'  
endif
```

Furthermore, you should not write:

```
person.ocIsUndefined() or person.name='Isabel'
```

but rather:

```
if person.ocIsUndefined()  
then  
    true  
else  
    person.name='Isabel'  
endif
```

Furthermore, you should not write:

```
cols->select( person | not person.ocIsUndefined() and person.name='Isabel' )
```

but rather:

```
cols->select( person | not person.ocIsUndefined() )  
    ->select( person | person.name='Isabel' )
```

or use an adequate if expression.

4 ATL Modules

The ATL programs for model to model transformation are called modules. A module file has the header section, the imports section, as well as helpers and rules. For an ATL module only the header section is mandatory. The ATL module is stored in a file with the extension *.atl*.

Please note:

- To navigate from an element to its attribute, write the name of the element, then “.” and the name of the attribute;
- If an identifier (variable name, metamodel name, model element name, etc.) is in conflict with a keyword, it has to be marked with apostrophes;
- The ATL parser is case sensitive. This concerns the file names as much as the source code itself.

4.1 Preparation

Before an ATL program can be written you must have the target and the source metamodels. For the scope of this manual, the *Book to Publication* transformation example (see Figure 1 for the *Book* metamodel and Figure 2 for the *Publication* metamodel) is used to illustrate ATL. Ecore models can be defined with KM3 [2].

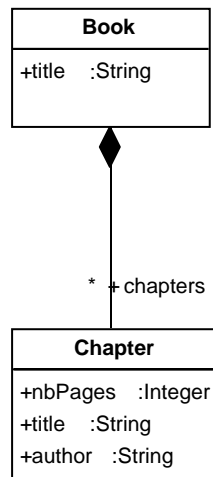


Figure 1. The Book metamodel

The *Book* metamodel contains the class *Book* which contains a set of *Chapters*. *Books* and *Chapters* have *titles*. Additionally, *Chapters* have a number of pages, *nbPages*, and *author*.

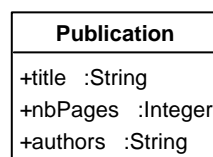


Figure 2. The Publication metamodel

The metamodel *Publication* consists of the class *Publication* which has a *title*, a number of pages, *nbPages*, and *authors*. The attribute *authors* is a String that contains the names of authors separated with the word *and*.

In the following it will be shown step by step how to program an ATL module that transforms *Book* models into *Publication* models. An ATL module begins with the header section and ends with the rules section.

4.2 Header Section

The header section defines the names of the transformation module and the variables of the source and target models.

The following ATL source code represents the header of the *Book2Publication.atl* file, thus the ATL header for the transformation from *Book* to *Publication*:

```
module Book2Publication;  
create OUT : Publication from IN : Book;
```

The keyword *module* defines the module name.

The keyword *create* introduces the target models declaration.

The keywords *from* introduces the source models declaration.

Please note:

- The ATL file name (here: *Book2Publication.atl*) has to correspond to the module name (here: *Book2Publication*) and must end with the extension *.atl*. The same applies to the project configuration. When using the ATL IDE for Eclipse, the module (here: *Book2Publication.atl*), the model variable names (here: *IN* and *OUT*) and model variable names (here: *Publication* and *Book*) have to correspond to the launch configuration (see Section 7).

4.3 Import Section

The import section declares what libraries have to be imported. For instance, to import the *strings* library, one would write:

```
uses strings;
```

The keyword *uses* declares the libraries that have to be imported. There can be several import declarations.

Please note:

- Do not declare imports that are not necessary;
- The *string.atl* library contains many useful functions for Strings but is not necessary for the *Book to Publication* example;
- The launch configuration (see Section 7) must indicate where the imports are to be found. Otherwise imports cannot be uploaded.

4.4 Helpers

Helpers can be used to define (global) variables and functions. Helper functions are OCL expressions. They can call each other (recursion is possible) or they can be called from within rules (see Section 4.5). In general, they serve to define repetitive pieces of code in one place.

A helper function has the following structure:

- It starts with the keyword *helper* and ends with a semicolon. In between is an OCL expression;
- The notion of the term *context* is similar to OCL and it may be compared to an input parameter of a method. A context variable is specified with the help of the ATL path expression *metamodel!element* and is accessible via the self variable. In most cases the elements referred to are classes, e.g. the class *Book* of the metamodel *Book*. If no *context* is specified, the module itself is taken as context;

- The function name is introduced with *def* : and is followed by brackets;
- The return type is between a colon and an equals sign that serves as starting point for the function implementation;
- The function finishes with a semicolon.

For the *Book to Publication* example, a *getAuthor* function has to be implemented. Its task is to iterate over the different *Chapters* of *Book* in order to build a String containing all *Authors' names*. The word *and* separates the *Authors' names*:

```
helper context Book!Book def : getAuthors() : String =
  self.chapters->collect(e | e.author)->asSet()
  ->iterate(authorName; acc : String = '' |
    acc +
    if acc = ''
      then authorName
    else ' and ' + authorName
    endif
  );
```

The context variable *self* is instance of the class *Book!Book*. The *collect* method gets all *author* attributes of all *Chapters*. The *asSet* method eliminates all doublets by converting the sequence to a set. The *iterate* operation is well known from OCL and iterates over a *Collection* while accumulating data. In this case it iterates over the *author* attributes of all *Chapters* of a *Book* and collects the *Authors' names*. The variable *authorName* references the *Author's name* of the current iteration. The variable *acc* serves to accumulate all *authorNames*. The *if* expression returns an instance of *String* containing the *authorName* if *acc* is empty. Otherwise, it concatenates 'and' with the *authorName*. The *getAuthors* function returns the result of the last iteration containing the concatenation of all *Authors' names*.

Often it is possible to either define a context or a parameter variable. The below program with parameter variable does exactly the same as the program above with context variable:

```
helper def : getAuthors(b : Book!Book) : String =
  b.chapters->collect(e | e.author)->asSet()
  ->iterate(authorName; acc : String = '' |
    acc +
    if acc = ''
      then authorName
    else ' and ' + authorName
    endif
  );
```

The mayor difference of the two helper functions is the way that they are called.

The helper function with context is called with:

```
b.getAuthors()
```

while the helper function with parameter is called with:

```
thisModule.getAuthors(b)
```

Please note:

- Each helper function must have a name and a return type definition. There may be functions with the same name but a different context;
- If the context is not otherwise specified, implicitly the module itself is taken as context;
- If the helper has no context defined, the helper has to be called with "thisModule."

4.5 Rules

Rules describe the transformation from a source model to a target model by relating metamodels. Each rule contains a unique name. It is introduced by the keyword *rule* that is followed by the rule's name. Its implementation is surrounded by curly brackets.

In the *source pattern* (*from* part), rules declare which element type of the source model has to be transformed.

The *source pattern* consists of the keyword *from*, an *in* variable declaration and optionally of a filter. A *filter* is an OCL expression restricting the rule execution to elements of the source model that satisfy certain constraints. The filter, when specified, is written behind the *in* variable declaration and surrounded by brackets. When no filter is specified, the rule is executed for each input element whose type corresponds to the *in* variable declaration.

In the *target pattern* (the *to* part), rules declare to which element(s) of the target model the *source pattern* has to be transformed to. The implementation of the *target pattern* declares the details of the transformation. It may contain one or several *target pattern elements* (see Section 5.4).

A *target pattern element* consists of a variable declaration (or more precisely the declaration of the *target pattern variable*) and a sequence of bindings (*assignments*). These bindings consist mainly of left arrow constructs. Usually, an attribute of the target model *out* (on the left side of the arrow) receives a return value of an OCL expression (on the right side of the arrow) that is based on the *in* variable. In this sense, the right side of the arrow may consist of an attribute of the *in* variable or a call to a helper function (which is an OCL expression).

For the *Book to Publication* example, a rule is required that transforms a *Book* to a *Publication*. Only *Books* with more than two pages are considered as *Publications*. The titles of *Books* and *Publications* have to correspond. The *authors* attribute of *Publication* contains all *authors* of all *Chapters*. For the latter, the function *getAuthors* (see Section 4.4) and the *getNbPages* (see Appendix A:) can be reused. The number of pages of a *Publication* is the sum of all pages of the *Chapters* of a *Book*.

```
rule Book2Publication {  
  from  
    b : Book!Book (  
      b.getNbPages() > 2  
    )  
  to  
    out : Publication!Publication (  
      title <- b.title,  
      authors <- b.getAuthors(),  
      nbPages <- b.getNbPages()  
    )  
}
```

Please note:

- Each rule must have a name that is unique within the module;
- When the source pattern includes no conditions, brackets have to be omitted;
- Assignments are separated with comma “,”. The last assignment in a block of statements does not have a comma;
- The attribution of values is performed with the left arrow operator “<-”;
- Both in the *source* and the *target pattern*, attributes and calls of helpers based on the input element have to be prefixed by the name of this input element.

5 ATL Advanced Features

The advanced features are explained using new examples.

5.1 Queries and the Generation of Text

The ATL programs shown so far are all modules. However, there are also ATL query programs. Queries allow to analyse models and to calculate an output that is not necessarily a model. This makes them very handy to generate text or code from a model.

ALT query programs must start with a query instantiation which consist of the keyword *query*, a query variable, an equal sign and an OCL expression initializing the query variable. An include section (see Section 4.3) with *uses* is optional.

In the following example you see an extract of the XQuery2Code program [3] which transforms XQuery models to code. With the *allInstances* function it runs through all elements of a particular element type of the input model. The *collect* function calls the *toString* helper functions and concatenates the *String* values that they return. The *writeTo* function writes the concatenation into a dedicated file.

Please note that there are several *toString* helper functions. During the execution, the helper function with the context type that fits best (here: to the type of the *e* variable) will be chosen for execution.

In general, one can say that this approach simplifies the generation of text or code because the programmer is supported in treating the coding of each metamodel types separately.

```
query XQuery2Code =
  XQuery!XQueryProgram.allInstances()
    ->collect(e | e.toString().writeTo('C:/test.xquery'));

helper context XQuery!XQueryProgram def: toString() : String =
  self.expressions->iterate(e; acc : String = '' |
    acc + e.toString() + '\n'
  );

helper context XQuery!ReturnXPath def: toString() : String =
  '{' + self.value + '}';

helper context XQuery!BooleanExp def: toString() : String =
  self.value;
```

5.2 Libraries

When writing large programs, it is of advantage to group reusable pieces of code in one place. This is the purpose of ATL libraries. In Section 4.3, it is explained how to import existing libraries in a module (and they can be included in a query in the same way), in this paragraph it will be illustrated how to write a library.

Unlike modules, Libraries are ATL programs that are not executable on their own and consist mainly of helper methods. Libraries start with the keyword *library*, the library name and a semicolon. An include section (see Section 4.3) with *uses* is optional. Then helpers follow.

In the following the *GeometryLib* is shown as example:

```
library GeometryLib;

helper def: PIDiv180 : Real = 180.toRadians() / 180;

-- and some further geometric global helper variables
```

```
-- adds two vectors
helper def : forward( a : TupleType(x : Real, y : Real, z : Real),
                      b : TupleType(x : Real, y : Real, z : Real)) :
    TupleType(x : Real, y : Real, z : Real) =

    Tuple {
        x = a.x + b.x,
        y = a.y + b.y,
        z = a.z + b.z
    };

-- subtracts the second from the first vector
helper def : backward(a : TupleType(x : Real, y : Real, z : Real),
                      b : TupleType(x : Real, y : Real, z : Real)) :
    TupleType(x : Real, y : Real, z : Real) =

    Tuple {
        x = a.x - b.x,
        y = a.y - b.y,
        z = a.z - b.z
    };

-- and some further geometric helper functions
```

In the *GeometryLib*, there are several helper functions (e.g. *forward* and *backward*) and global helper variables (e.g. *PIDiv180*) defined that can be used by modules that needs such functionality.

Once a module (or query) has imported the *GeometryLib*, it can call the library's helpers just as if they were defined in the module (or query) itself.

Examples of usage of the above defined helpers:

```
thisModule.PIDiv180

or

thisModule.forward(
    Tuple {x=10, y=10, z=10},
    Tuple {x=10, y=10, z=10}
)
```

5.3 Complex Headers

In ATL modules you may have several source models. Source metamodels may but need not have the same metamodel. Here is an example:

```
module GeometricalTransformations;
create OUT : DXF2 from IN1 : DXF1, IN2 : GeoTrans;
```

In the above case the DXF1 and DXF2 variables refer to same metamodel, namely DXF. DXF has the class Point. One can distinguish if one refers to a Point of the model DXF1 or of the model DXF2 simply by using the path expressions:

```
DXF1!Point
```

or

```
DXF2!Point
```

Please note:

- Each metamodel must have a different name (e.g. DXF1 and DXF2) but they may point to the same metamodel file;

- Different source models are separated using commas “,”.

5.4 Rules with Multiple Instantiations

If for one *source pattern* (the *from* part of a *rule*) several *target pattern elements* (the *to* part of a *rule*) have to be instantiated, we speak of multiple instantiations. Multiple instantiations are defined using several *target pattern elements* in one and the same rule. They are separated using commas “,”.

The Class to Relational transformation has a rule requiring that a *Table* has to be created for each *Class*. Additionally, each *Table* has to have a *key* set containing at least one *key*. In this example the *key* is a *Column* with the name *objectId*. However, Classes of the source metamodel do not have persistent identifiers (surrogates) such as keys. In this sense, for each *Class* not just a *table* but also a *key* column has to be instantiated. Consequently, for the *Class2Table* rule two *target pattern elements* are needed, namely *table* for the creation of the *Table* instance and *new_key* for the creation of the *key*.

Please note that the *key* attribute of *table* (which is a set of columns) can be initialized with *new_key* (which is a *Column*). However, you must not use *new_key.name* or *table.name* in expressions.

```
rule Class2Table {
  from
    c : Class!Class
  to
    table : Relational!Table (
      name <- c.name,
      key <- Set {new_key}
      -- further value assignments
    ),
    new_key : Relational!Column (
      name <- 'objectId'
      -- further value assignments
    )
}
```

5.5 Navigation and Multiple Instantiations

ATL forbids the navigation in the created target model because this implies restrictions on the execution order. Such restrictions could severely hinder optimisation and very probably lead to poor performance.

Consequently, ATL promotes navigation in the source model. This also involves that whenever an attribute of the target model has to be assigned with a reference to an element of the target model not the target element is referenced, but the source element from which the target element has been or will be created.

As a matter of fact, when a source element is assigned to an attribute of the target model, this associates the attribute with the default target pattern (i.e. the first target pattern) of the rule that matches the source element.

This works very well if each source element is mapped to no more than one target element. In case of multiple instantiations, one can use the operation *resolveTemp* with the name of the target pattern variable in order to distinguish between the different instantiations. The *resolveTemp* operation is defined in the context of the ATL module, which implies that is called as follows: *thisModule.resolveTemp(...)*.

The operation accepts two parameters:

- An input model element that is matched by the rule in which the target pattern is defined;
- A string that identifies the name of the target pattern variable.

In the class to table transformation example for each class a table and a key will be generated:

```
rule Class2Table {
  from
    c : Class!Class
  to
    table : Relational!Table (
      name <- c.name,
      key <- Set {new_key}
      -- further value assignments
    ),
    new_key : Relational!Column (
      name <- 'objectId'
      -- further value assignments
    )
}
```

An example of how to get a key reference is:

```
thisModule.resolveTemp(
  Class!Class->allInstances()->
    select(c | c.name = 'NameOfTheClassCorrespondingToTheSearchedKey'),
  'new_key'
)
```

An example of how to get a table reference is:

```
thisModule.resolveTemp(
  Class!Class->allInstances()->
    select(c | c.name = 'NameOfTheClassCorrespondingToTheSearchedTable'),
  'table'
)
```

Note that “table” is the default target pattern of the `Class2Table` rule. As a consequence, it is not necessary to call the `resolveTemp` operation to access it: it is implicitly referred to by the input model element matched by the `Class2Table` rule.

5.6 Flexible Runtime Instantiation of Target Elements

With the possibilities introduced so far it was only possible to create a fixed number of target elements per rule. The *foreach* expression allows to instantiate as many instances of target elements per rule as needed at runtime. Hence, the number of target elements may be undefined at compilation time.

With the help of an iterator, the *foreach* operation iterates through a collection of elements and instantiates a new target element of the specified type for each occurrence in the collection.

The *foreach* expression is also called a *foreach* target pattern element. It starts with a target pattern variable, a semicolon, the keyword *distinct* and the model element type that has to be instantiated for each iteration. This is followed by the keyword *foreach* and the iterator declaration in brackets. The iterator declaration comprises the iterator name, the keyword *in* and an expression of type collection.

In the list to folder transformation a tree-structured file folder has to be mapped to a flat list of files. More precisely, for the root element of a tree, a list must be generated. This list must contain all children of the folder's root. The number of children is not limited. This is a typical use case for a *foreach* expression:

```
rule Root2List {
  from
    f : Folder!File (
      f.folder.oclIsUndefined()
    )
  using {
```

```

allFiles : Sequence(Folder!File) =
    Folder!File.allInstances()->
        select(e | e.oclIsTypeOf(Folder!File))->asSequence();
allFilesPaths : Sequence(Sequence(String)) =
    allFiles->collect(e | e.folder.getPaths());
}
to
out : List!List (
    name <- f.name
),
fi : distinct List!File foreach(singleFile in allFiles) (
    name <- singleFile.name,
    path <- allFilesPaths,
    list <- out
)
}

```

With (the current version of) ATL, when a collection is assigned to an attribute/reference of the target element of a *foreach* expression (such as *path*), the size of the assigned collection (*allFilesPaths* in the rule) has to be the same than the one of the reference collection of the *foreach* expression (here *allFiles*). As a matter of fact, the *foreach* expression does not only iterate through the reference collection (the *allFiles* sequence), but also through the assigned ones (*allFilesPaths*). This is why, in this example, each file of *allFiles* is associated with its corresponding path in the sequence of folder names (*allFilesPath*).

Building, in the *using* statement, a collection of collections, enables to assign a collection to a property of each element of the reference collection of the *foreach* expression. Assigning a single element (instead of a collection) can simply be achieved by composing a collection of elements (whose size will still have to be equal to the one of the *foreach* reference collection).

5.7 ATL Tips and Tricks

This section aims to highlight some common problems and errors that may be experienced while starting programming with ATL.

In ATL, an element of the input model should not be matched more than once. At present time, this constraint is not verified at compile time, and this kind of errors can lead to unexpected results. A typical case of multiple matching of an input model element appears with the definition, in the input metamodel, of an inheritance link in which the parent entity is not abstract. Figure 3 provides a simple example of this kind of situation.

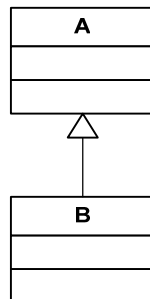


Figure 3. Simple inheritance case

The multiple matching problem appears here when trying to respectively match A and B elements by means of two distinct rules (*ruleA* and *ruleB*). With an intuitive source pattern such as $a : \text{MM!A}$, *ruleA* will match purely A elements as well as B elements. Since these last ones are also matched by *ruleB*, this raises a multiple matching problem. To solve the problem, the developer has to ensure that *ruleA*

only matches purely A elements. This is achieved by filtering, in the source pattern of *ruleA*, the type of the elements to be matched by the rule:

```
rule ruleA {
  from
    a : MM!A (
      a.oc1IsTypeOf(MM!A)
    )
  ...
}
```

The OCL function *oc1IsTypeOf* here tests whether the input model element is an instance of the metamodel element passed as parameter.

6 The ATL IDE for Eclipse

Eclipse is an open universal tool platform for software development and in particular for the construction of IDEs, integrated development environments. It has been chosen to build the ATL IDE [1] with.

6.1 Installing the ATL IDE for Eclipse

For the installation of the ATL IDE for Eclipse, please consult the ATL installation instructions in [4].

6.2 Basic Features

The Eclipse environment contains a set of tools and features which have been adapted and extended to best suit the needs of ATL development. The principal work environment is called workbench (see Figure 4) that contains several subwindows (views) and toolkits.

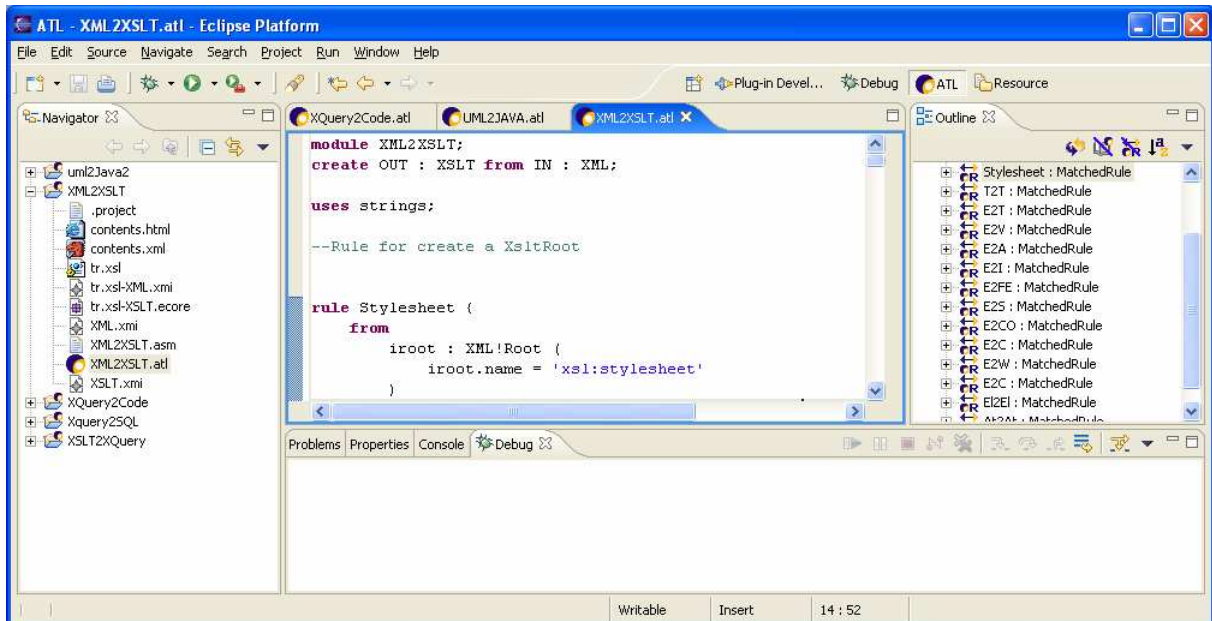



Figure 4. ATL IDE for Eclipse

6.2.1 Perspectives

In Eclipse, the notion of perspective refers to a workbench configuration that is arranged in order to optimise the handling of a certain task. For ATL development there are two perspectives of

	ATL Documentations	
	ATL User Manual	Date 23/08/2005

importance, namely the ATL perspective and the debug perspective. Depending on your Eclipse workbench these perspectives may already be available via buttons in the thumb index on the top right hand side of your workbench. Otherwise a left mouse click on the perspective icon (displaying a window with subwindows and a yellow cross, accessible on the top right hand side of your workbench) leads you to a context menu from which *Other* and then *ATL* (for the ATL development) or *Debug* (for the ATL execution in debug mode) can be chosen to obtain the corresponding perspective. If ATL is not within the selectable perspectives, the ATL plugin has not been successfully installed (please reinstall, see Section 6.1).

Please note:

- Only the perspectives ATL or debug should be used for developing projects with ATL or KM3.

6.2.2 Navigator

The Eclipse navigator shows you the files of your workbench. With a right mouse click on a file you obtain a context menu showing you possible actions, e.g. the possibility to open a file in a text editor, to display an ATL file with the ATL editor, to debug or run an ATL file or to import the Ecore model from a KM3 file.

6.2.3 Editors

Eclipse facilitates the development of powerful source editors. Concerning the ATL development, an ATL editor has been implemented that performs syntax highlighting, runtime parsing, compilation and error detection. Modifications of the ATL file that lead to a syntactically correct ATL program will immediately trigger the compilation and thus the generation of a new assembler file (at latest when the file is saved). An assembler file has the extension *.asm* and contains the compiled code of the corresponding ATL file.

Please note:

- For the display of ATL files the ATL editor is recommended.

6.2.4 Outline

The Eclipse outline view gives an overview over the structural elements of the active file in the Eclipse editor with which it is to be synchronized. In this sense the ATL editor is in permanent communication with the outline view. Adding the code for a new structural element such as a rule or a helper context function in the ATL editor will automatically lead to a corresponding addition in the outline view (at latest when the file is saved). Furthermore, the cursors in the ATL editor and the outline view will always point to the same structural element (see Figure 5). If the cursor is moved in one (meaning the ADT editor or the outline), the other will replace its cursor correspondingly.

In the outline view it is possible to set breakpoints for debugging that are marked in the ATL editor with green points.

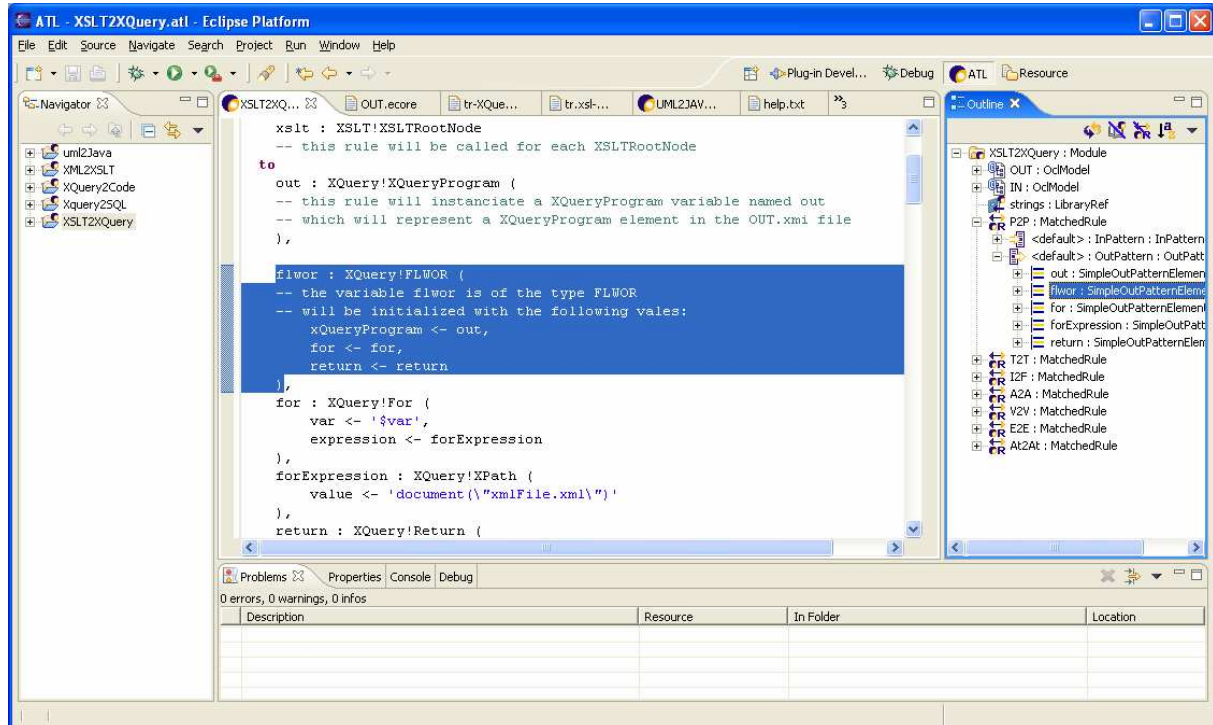


Figure 5. Synchronized Cursors of Outline and ATL Editor

6.2.5 Console

The Eclipse console shows the output of programs that have been triggered from subordinate runtime workbenches. This is a helpful feature for debugging ATL programs that make use of the *debug* operation (see Section 7.3). For the creation a subordinate runtime workbench, use the right mouse button to pop up the context menu and choose *Run* in the submenu *Run*. In the run configuration view select *Run-Time Workbench* and click the *New* button. Then select the newly created instance and trigger the subordinate runtime workbench with the button *Run*. As a consequence, the standard output of programs running in the subordinate runtime workbench will be displayed in the console of the original workbench.

6.3 Creating a Project

An ATL project is created with the following actions:


In the Eclipse *File* menu choose the item *New*.

In the appearing view choose *ATL Project* within the *ATL* folder and click on *Next*.

Give the project a sensible name (by concatenating the source model name, the character 2 and the target model name, e.g. XSLT2XQuery) and push the *Finish* button.

Eclipse creates for each project a new folder with the project name. This folder can be opened with a double click. At the beginning it contains but a *.project* file that the ATL programmer may ignore. However, for the basic transformation scenario you need an ATL program in the form of an *.atl* file that should bear the same name as the project itself, the source and the target metamodels in EMF [5] or MDR [9], formatted in XMI [8]. For testing the transformation at least one source model should be available. For the development of ATL programs, the ATL editor is recommended (see Section 6.2.3).

The creation of an empty file is performed with the following steps:

	ATL Documentations	
	ATL User Manual	Date 23/08/2005

- Choose the item *New* in the Eclipse *File* menu or in the Navigator context menu;
- Select the item *File*;
- In the appearing view enter the file name and the intended extension (e.g. *xmi*, *.atl* or *.km3*);
- Make sure that the correct project folder is marked;
- Push the *Finish* button.

7 Compilation

In the Eclipse environment the compilation of an ATL program to an ASM bytecode file is automatically done in the background (e.g. when saving the ATL program).

7.1 Setting up a Launch Configuration

The ATL debug and run environment can be set up once you have created the ATL file and the corresponding metamodel and model files. To obtain the launch configuration view you can use the context menu in the navigator:

A right mouse click in the navigator view opens the context menu.

In the context menu choose *Run...* in the submenu of *Run* or choose *Debug...* in the submenu of *Debug*.

Having reached the launch configuration view (see Figure 6), an ATL transformation is created by selecting the *ATL Transformation* instance in the configuration tree at the left hand side of the view and pushing the *New* button below. Give to the configuration the same name as to its project and choose the corresponding *.atl* file.

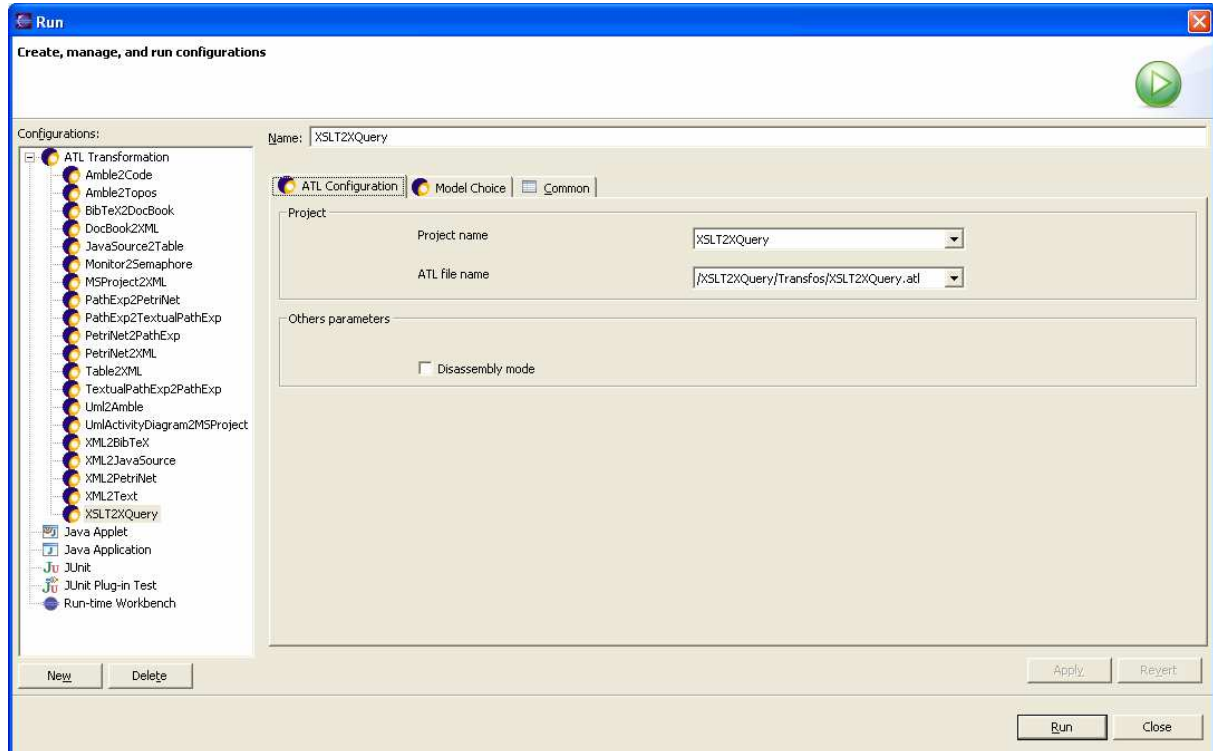


Figure 6. ATL Configuration

The next step to take is to specify the metamodels and the models. To do so, pick the *Model Choice* panel in the configuration view (see Figure 7) and follow these steps:

Define the IN metamodel(s) and model variables by writing their names in the designated fields of the IN box add press the *Add* button. For the model and the metamodel names use the same variable name as specified in your ATL file. The added variables will be displayed in the path editor table.

Define the OUT metamodel and model variables by writing their names in the designated fields of the OUT box add press the *Add* button. For the model and the metamodel names use the same variable name as specified in your ATL file. For The added variables will be displayed in the path editor table.

In the path editor select the different variables and attribute the corresponding files. Push the *Set path* button and specify their name. If the path is not within the workspace, use *Set external path*. For input and output metamodels, use *Set Model Handler* to select the model handler (EMF or MDR) to be used (default value is EMF). In case some of the registered metamodels are either MOF 1.4 or Ecore, the two last buttons, *MM is MOF-1.4* and *MM is Ecore*, enable to associate the concerned metamodel files with the corresponding ATL internal representation of the MOF/Ecore metamodels.

Define the libraries used in the *Libs* panel. Please use the corresponding *.asm* files and make sure that the naming corresponds to the import section of your ATL program.

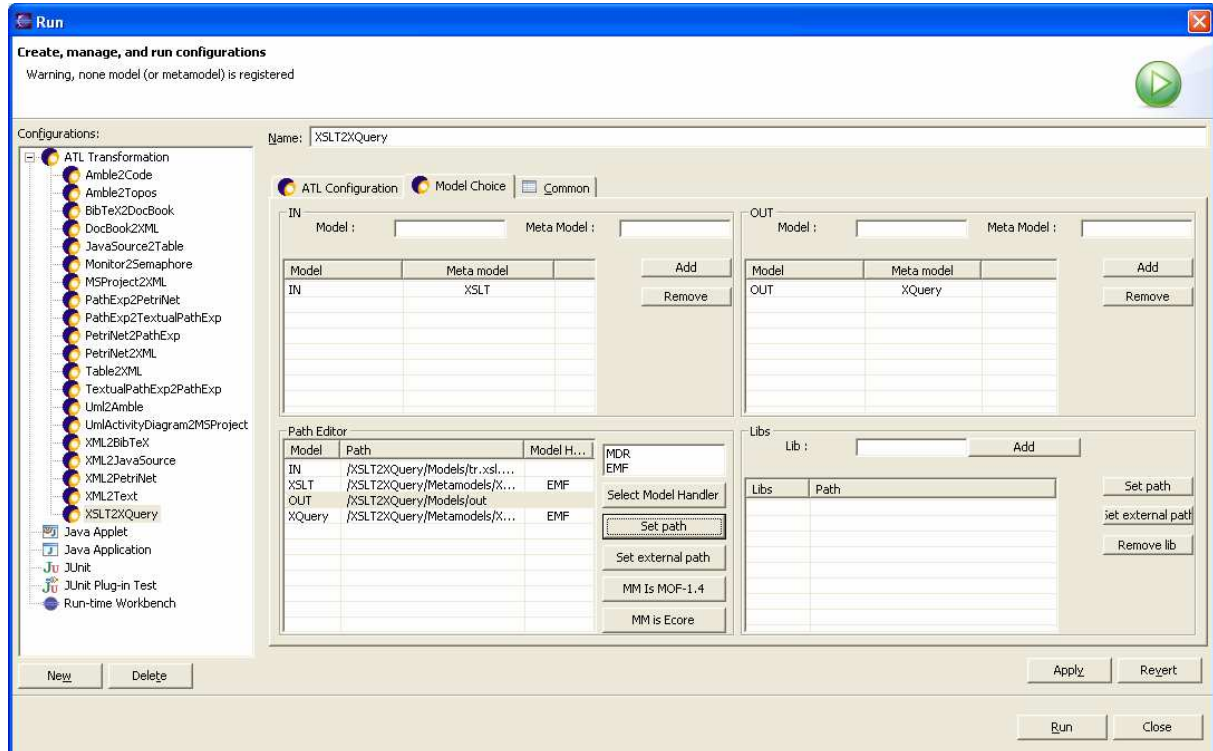


Figure 7. ATL Model Configuration

7.2 Running a Launch Configuration

ATL programs can be launched from the navigator:

- Right mouse click in the navigator view;
- In the context menu choose *Run...* in the submenu of *Run*;
- Choose the configuration needed and run it by pressing the *Run* button on the right bottom of the view.

Another possibility to trigger the execution of an ATL program is to use the run icon (depicting a white play forward sign on a green background) displayed in the bar underneath the main menu.

7.3 Debugging

For debug actions it is recommended to switch to the debug perspective. Additional to the editor and the outline, it contains the debug window that displays the processes (terminated or not) and the control window that lists all chosen *breakpoints* and *debug variables* of ATL programs.

An essential instrument for debugging is the partial execution step by step or with breakpoints. The latter are set with a right mouse button click on a structural element (e.g. a rule) in the outline view and then selecting the “add breakpoint” item. To trigger the partial execution use the debug operation by clicking the debug icon.

You may additionally use the standard output in connection with the *debug* operation as a means for debugging.

It is possible to add debug expressions in your code:

```
expression.debug( 'message' )
```

will print:

message: <expression-value>

This is an example for using *debug* in your code:

```
rule Book2Publication {
  from
    b : Book!Book (
      b.getSumPages() > 2
      -- only Books with more than 2 pages are publications
    )
  to
    out : Publication!Publication (
      title <- b.title,
      authors <- b.getAuthors(),
      nbPages <- b.getSumPages().debug( 'sum pages' )
    )
}
```

Depending on the model data the *debug* operation will print:

```
sum pages : 50
sum pages : 12
sum pages : 519
```

7.4 Log File

ADT log files are contained in your workspace log file (your_workspace/.metadata/.log).

You can access the error log file via the *Help* menu. Choose *About Eclipse Platform*, then *Configuration Details* and finally *View Error Log*.

8 References

- [1] Allilaire, F., Idrissi, T. ADT: Eclipse Development Tools for ATL. EWMDA-2, Kent, September 2004
- [2] ATLAS group, KM3: Kernel MetaMetaModel Manual. 2004
- [3] ATLAS Group, ATL Transformation Example: XSLT2XQuery. 2004
- [4] ATLAS group, Installation of ADT from source. 2004. http://www.sciences.univ-nantes.fr/lina/atl/www/papers/ATL/ATL_Documentation/ADTInstallation.pdf
- [5] IBM, Eclipse project, EMF – Documentation. <http://www.eclipse.org/emf/>
- [6] OMG/MOF Meta Object Facility (MOF). Version 1.4. formal/2002-04-03, 2002
- [7] Object Management Group: OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP. October 2002
- [8] OMG/XMI XML Model Interchange (XMI) OMG Document AD/98-10-05, October 1998. <http://www.omg.org>
- [9] SUN, netBeans project, MDR - Metadata Repository. <http://mdr.netbeans.org/>

Appendix A: Example Code

In this manual, the *Book to Publication* transformation example (see Figure 8 for the *Book* metamodel and Figure 9 for the *Publication* metamodel) serves to illustrate the use of ATL.

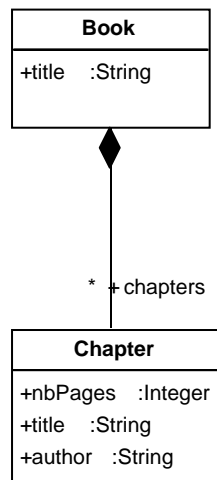


Figure 8. The Book metamodel

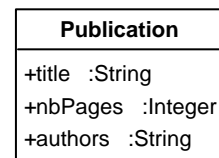


Figure 9. The Publication metamodel

The *Book to Publication* code transforms *Books* with more than two pages to *Publications*. The titles of *Books* and *Publications* have to correspond. The *authors* attribute of *Publication* contains all *authors* of all *Chapters*. The number of pages of a *Publication* is the sum of all pages of the *Chapters* of a *Book*.

```


module Book2Publication;
create OUT : Publication from IN : Book;

helper context Book!Book def : getAuthors() : String =
  self.chapters->collect(e | e.author)->asSet()
  ->iterate(authorName; acc : String = '' |
    acc +
    if acc = ''
      then authorName
    else ' and ' + authorName
    endif
  );

helper context Book!Book def : getNbPages() : Integer =
  self.chapters->collect(f|f.nbPages)
  ->iterate(pages; acc : Integer = 0 |
    acc + pages
  );

rule Book2Publication {
  from
    b : Book!Book (
      b.getNbPages() > 2
    )
  to
    out : Publication!Publication (

```

	ATL Documentations	
	ATL User Manual	Date 23/08/2005

```

    title <- b.title,
      authors <- b.getAuthors(),
      nbPages <- b.getNbPages()
  )
}

```