

ATL:
Atlas Transformation Language

ATL Starter's Guide
- version 0.1 -

December 2005

by
ATLAS group
LINA & INRIA
Nantes

Content

1	Introduction	1
2	Understanding model transformation	1
3	A simple transformation example	2
4	Creating a new ATL project.....	3
5	Designing a new metamodel.....	4
6	Editing the source model.....	9
7	Programming my first ATL transformation.....	10
7.1	Creation of the ATL file	10
7.2	Creation of a new transformation rule.....	12
8	Creating my first ATL launch configuration	14
8.1	Configuring the ATL Configuration tab.....	14
8.2	Configuring the Model Choice tab.....	15
8.3	The Common tab.....	16
9	Executing my first ATL transformation.....	17
10	Conclusion.....	17
11	References	18
Appendix A	The <i>authors.ecore</i> input file.....	19
Appendix B	The <i>Author2Person.atl</i> file.....	20

Figures List

Figure 1.	An overview of model transformation.....	2
Figure 2.	The Author metamodel	2
Figure 3.	The Person metamodel.....	2
Figure 4.	Overview of the Author to Person ATL transformation.....	3
Figure 5.	The initial ATL perspective	4
Figure 6.	Creation of a new ATL project	5
Figure 7.	Creation of new file	6
Figure 8.	The Author metamodel in KM3 format.....	6
Figure 9.	Injecting a KM3 file into an Ecore metamodel.....	7
Figure 10.	Error during the injection of a KM3 file.....	8
Figure 11.	The generic Ecore file editor	8
Figure 12.	Textual edition of the <i>authors.ecore</i> model file	9
Figure 13.	Creation of new ATL file.....	10

Figure 14. The ATL file wizard	11
Figure 15. An ATL transformation template	12
Figure 16. The Author2Person ATL transformation.....	13
Figure 17. Creating a new launch configuration.....	14
Figure 18. ATL launch configuration – ATL Configuration	15
Figure 19. ATL launch configuration – Model Choice.....	16



1 Introduction

This document is dedicated to ATL beginner users. It aims to provide them with a quick and comprehensive overview of the ATL transformation tool. To this end, it proposes to guide the user through a step-by-step simple ATL example.

It is here assumed that the ATL Development Tools (ADT) have been previously successfully installed. For this purpose, please refer to the ATL Installation Guide [1]. However, this section does not assume any particular ATL knowledge from the user.

The document is organized as follows:

- Section 2 introduces the model transformation problematic and the way model transformation could be achieved using ATL;
- Section 3 describes the transformation example that is going to be developed in the document;
- Section 4 details the creation of a new ATL project;
- Section 5 introduces the way new metamodels can be designed using the ATL Tools Development;
- Section 6 describes the design of a simple input model;
- Section 7 details the design of the ATL transformation;
- Section 8 describes the configuration of the transformation launch configuration;
- Section 9 deals with the transformation execution;
- Finally, Section 10 provides a number of links to advanced ATL resources.

2 Understanding model transformation

In the field of model engineering, models are considered as first class entities. A model has to be defined according to the semantics provided by its metamodel: a model is said to conform to its metamodel. In the same way, a metamodel has to conform to a metamodel. In this three layers architecture (models, metamodels, metamodel), the metamodel usually self-conforms to its own semantics (e.g. it can be defined using its own concepts). Existing metamodels include MOF [2], which has been defined by the OMG, and Ecore [3], which has been introduced with the Eclipse Modelling Framework (EMF) [4].

Considering model as first class entities requires providing a set of tools defining some operations dedicated to models. In this context, model transformation appears to be one of the most useful operations on models. Model transformation therefore aims to provide facilities for generating a model M_b , conforming to a metamodel MM_b , from a model M_a conforming to a metamodel MM_a .

A major feature in model engineering is to consider, as far as possible, all handled items as models. The model transformation itself therefore has to be defined as a model. This transformation model has to conform to a transformation metamodel that defines the model transformation semantics. As other metamodels, the transformation metamodel has, in turn, to conform to the considered metamodel.

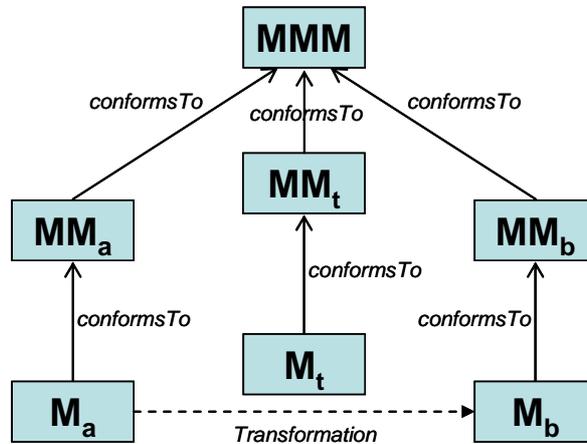


Figure 1. An overview of model transformation

Figure 1 summarizes the full model transformation process. A model M_a , conforming to a metamodel MM_a , is here transformed into a model M_b that conforms to a metamodel MM_b . The transformation is defined by the model transformation model M_t which itself conforms to a model transformation metamodel MM_t . This last metamodel, along with the MM_a and MM_b metamodels, has to conform to a metametamodel (such as MOF or Ecore).

3 A simple transformation example

This section introduces the transformation example that is going to be developed in the document. The aim of this first example is to introduce users with the basic concepts of the ATL programming. To this end, this example considers two similar metamodels, Author (Figure 2) and Person (Figure 3), that both encode data relative to persons.

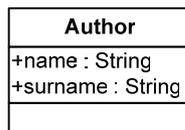


Figure 2. The Author metamodel

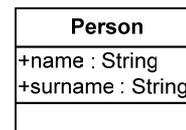


Figure 3. The Person metamodel

Both metamodels are composed of a single eponym element: Author for the Author metamodel and Person for the Person metamodel. Both entities are characterized by the same couple of string properties (name and surname).

The objective is here to design an ATL transformation enabling to generate a Person model from an Author model. The transformation to be designed will have to implement the following (obvious) semantics:

- A distinct Person element is generated for each source Author element;
 - The name of the generated Person has to be initialized with the name of the source Author;
 - The surname of the generated Person has to be initialized with the name of the source Author.

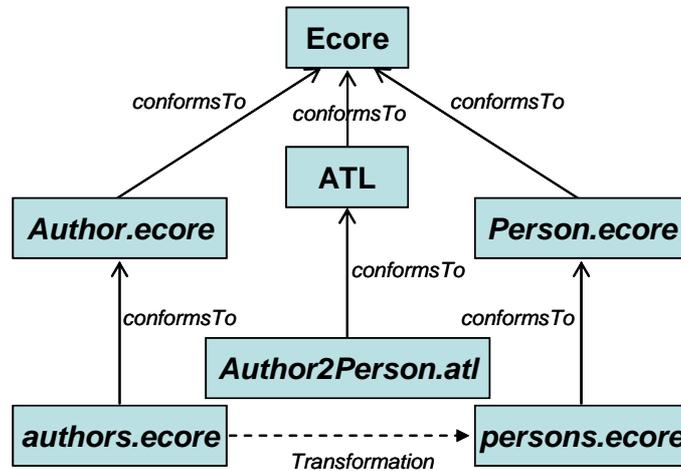


Figure 4. Overview of the Author to Person ATL transformation

Figure 4 provides an overview of the ATL Author to Person transformation process. The figure introduces the name of the files (in italics) that are going to encode the models (*authors.ecore*, *persons.ecore*), the metamodels (*Author.ecore*, *Person.ecore*) and the ATL transformation (*Author2Person.atl*) that will be handled during the design of the Author to Person ATL transformation. These files, along with the way they can be designed in the scope of the ATL development tools, are progressively introduced in the rest of this document.

Note that the transformation to be designed (*Author2Person.atl*) will have to conform to the ATL transformation metamodel. Neither this metamodel, nor the Ecore metamodel, will be directly handled during the design of the Author to Person transformation (they are associated with no file within Figure 4).

The first step in designing an ATL Author to Person transformation is to create a new ATL project. This is the focus of the next section.

4 Creating a new ATL project

The first step, after having launched Eclipse, is to move to the ATL perspective. This is simply achieved by selecting ATL in the Window→Open Perspective→Other... menu. The title bar of the Eclipse window shall have changed to "ATL – Eclipse SDK".

Under the ATL perspective, the Eclipse window should look like the window presented in Figure 5. The initial ATL perspective is composed of:

- The Navigator view (on left column), in which will appear the ATL projects;
- The Outline view (on right column), in which will appear an outline of the currently edited transformation, model or metamodel;
- The edition view (the middle top section) aims to display the currently edited file (a transformation, model or metamodel file);
- The bottom section contains different views (Problems, Properties, Error Log and Console). At this stage, the most interesting of these views is the Problems one that will provide information on errors detected at compile-time on edited files.

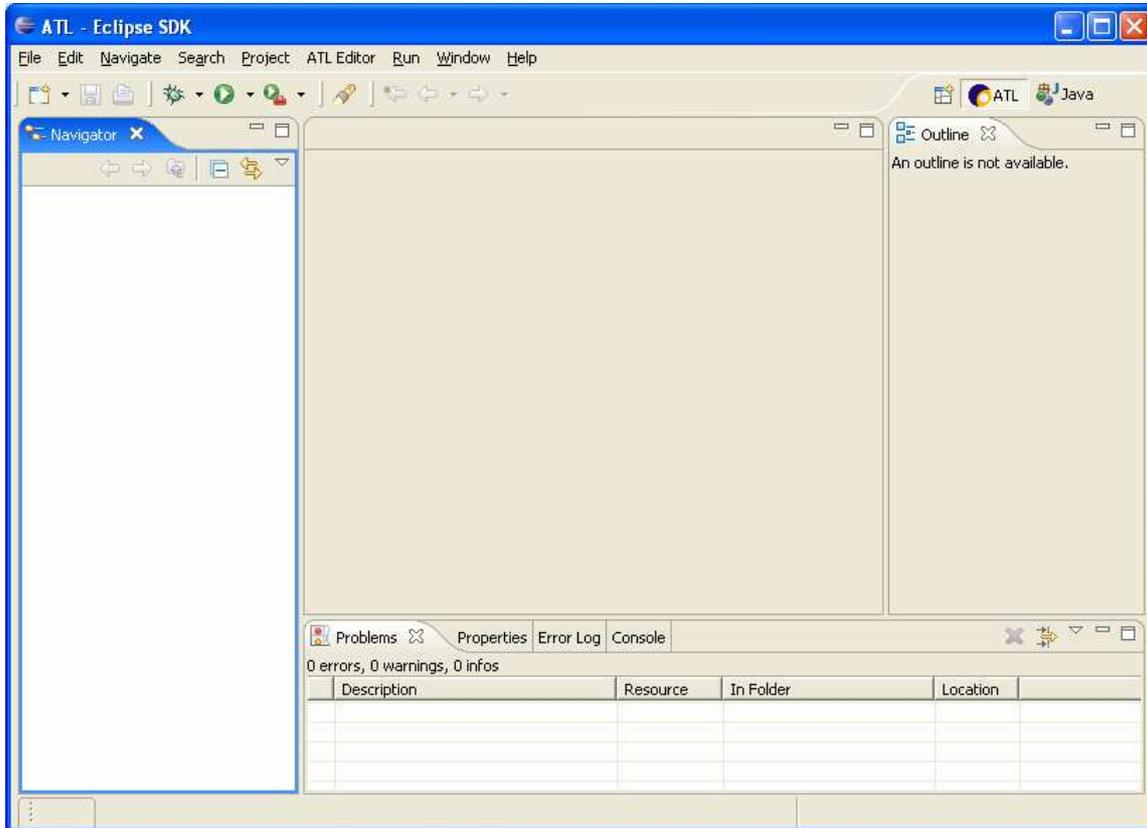


Figure 5. The initial ATL perspective

It is now possible to create a new ATL project. For this purpose, move to the navigator view and select mouse right click→New→ATL Project (see Figure 6): the name of the project has to be entered (for instance "Author2Person"). Once created, the new project shall appear in the Navigator view. After a double-click on the empty project (or after the first operation on this project), a `.project` file will appear within the project. This file is an XML file generated by Eclipse in order to store metadata relative to the created project.

Now the ATL project has been created, next step is to provide the ATL tool suite with a computable version of the considered metamodels. This new step is detailed in the following section.

5 Designing a new metamodel

As highlighted in Section 2, it is required, for a model transformation to run, to have the metamodels of both source and target models (which are here respectively Author and Person). At this stage, these metamodels are only available under a graphical form (see Figure 2 and Figure 3). It is therefore necessary to provide the transformation engine with a computable version of these metamodels.

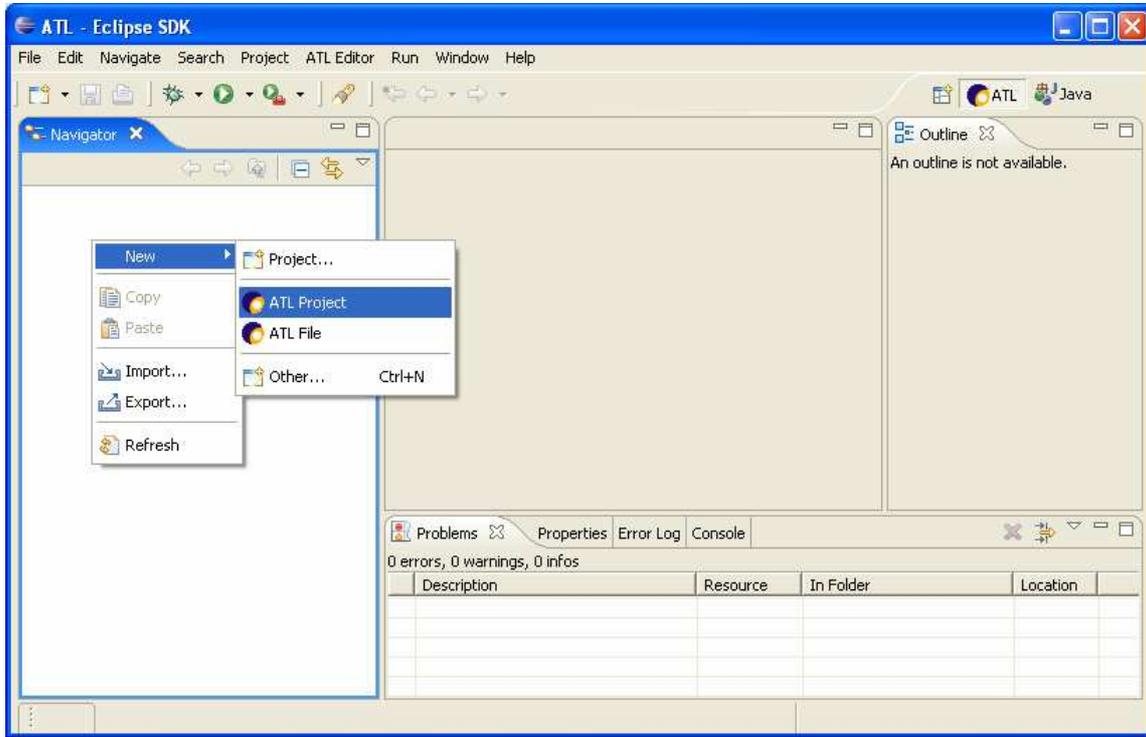


Figure 6. Creation of a new ATL project

EMF provides its own file format (.ecore) for model and metamodels encoding. This format is based on the semantics of the Ecore metamodel [3] and the corresponding files are encoded with XMI 2.0 [5]. Although possible, the manual edition of Ecore metamodels is particularly difficult with EMF. In order to make this common kind of editions easier, the ATL Development Tools include a simple textual notation dedicated to metamodel edition: the Kernel MetaMetaModel (KM3) [6]. This textual notation greatly eases the edition of metamodels. Once edited, KM3 metamodels can be injected into the Ecore format using ADT integrated injectors.

The ATL example currently considered only requires a few notions of KM3. The KM3 textual notation is very close to the Java notation. Thus, a KM3 file is usually used to encode a single metamodel. It is composed of:

- Packages. Each package includes a number of classes;
 - Classes. A class may define attributes as well as references to other classes (for more details on KM3, please refer to the KM3 Manual [6]).

In the scope of the considered ATL example, a metamodel can be associated with a package and an entity with a class.

It is now possible to edit the considered metamodels in the KM3 format. For this purpose, two new files have to be created: *Author.km3* (for the Author metamodel) and *Person.km3* (for the Person metamodel). The creation of a new KM3 file is achieved by selecting the File entry in the New→Other...→Simple folder (see Figure 7). The name (*Author.km3*) and the path of the file to be created have to be entered.

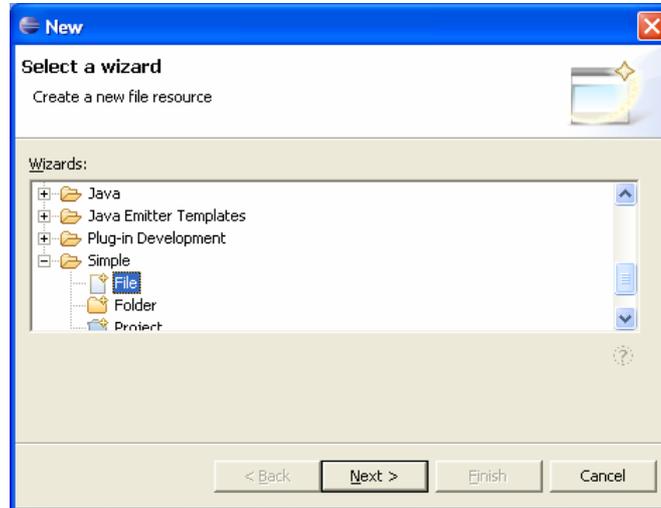


Figure 7. Creation of new file

Once the creation of the file has been validated, the new file appears in the current ATL project (in the Navigator view). It is now ready to be edited (the file is opened by a double-click).

Figure 8 provides a screenshot of the edited Author metamodel. Note that the KM3 file must include a specific package (PrimitiveTypes) that defines all the primitive types that are referred to within the other edited packages. KM3 currently defines four primitive data types: String, Boolean, Integer and Double. In the scope of the Author metamodel, a single primitive type requires to be defined: the String data type. The Person metamodel can be edited in the same way.

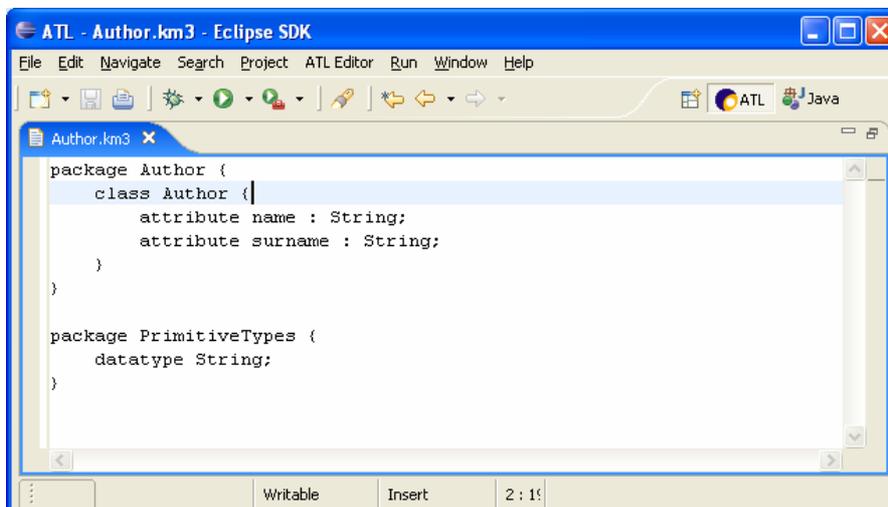


Figure 8. The Author metamodel in KM3 format

Note that, at this stage, the edited KM3 files may include undetected errors. Indeed, a KM3 file is only compiled when it is the target of an injection operation.

Now, the Author and the Person metamodels have been edited in the KM3 format, they can be injected into Ecore files. For this purpose, select mouse right click→Inject KM3 to Ecore metamodel on the *Author.km3* file (see Figure 9). If the file is errorless, this operation will create an *Author.ecore* file in the considered project (in the Navigator view). The *Person.ecore* file can be generated in the same way.

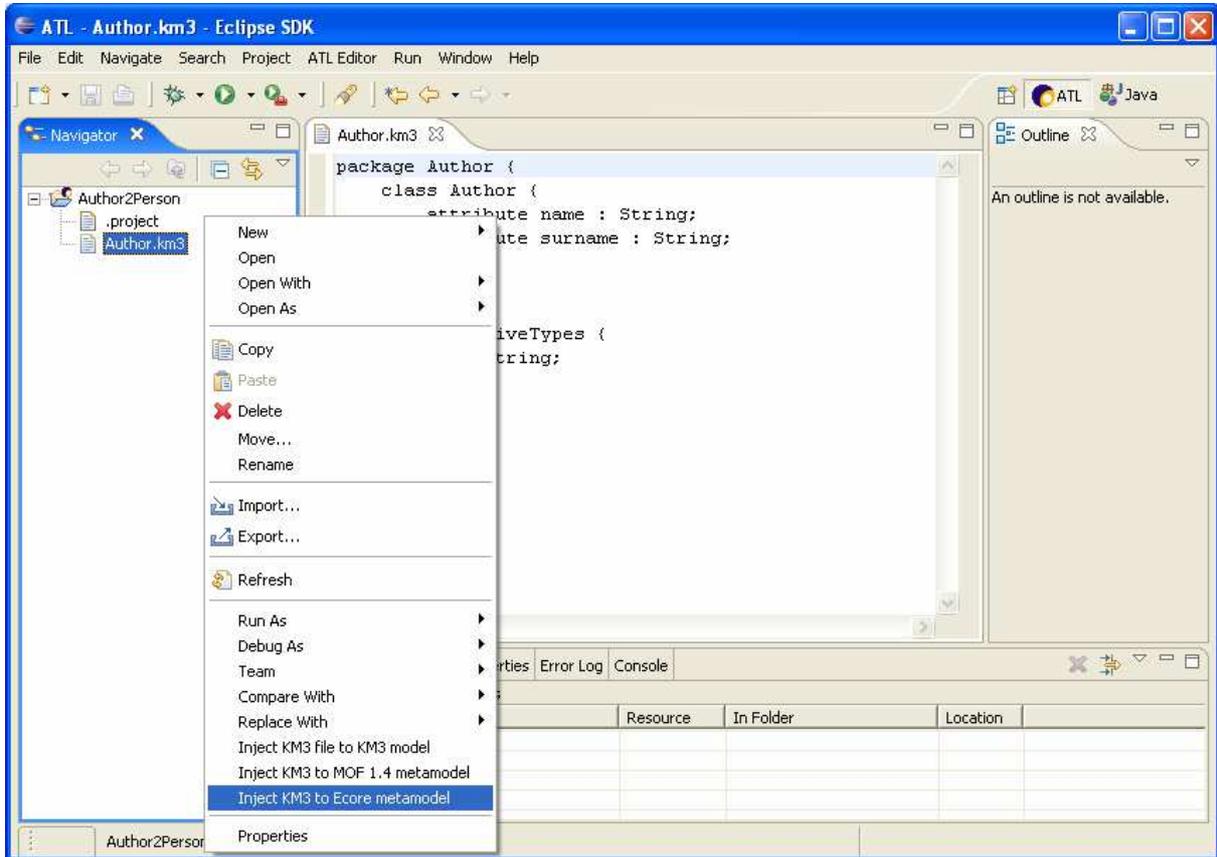


Figure 9. Injecting a KM3 file into an Ecore metamodel

In case the KM3 source file contains any errors at injection time, the Ecore file will not be generated. The detected errors will then be located on the edition view (when opened) and detailed in the Problems view (at the bottom of the perspective). As an example, Figure 10 provides a screenshot of an injection error where the data type of the attribute surname declared on line 4 is unknown.

Note that, at this stage, the generated Ecore files can now be edited by means of the Ecore model editor provided with EMF. This editor is opened with double-click on the targeted Ecore file. Figure 11 provides a screenshot of the edition of the *Person.ecore* metamodel file. The Properties view, available at the bottom of the perspective (see Figure 11) provides information on the element currently selected in the editor view. In Figure 11, the surname attribute of the Person entity is currently selected. The Properties view displays a set of information related to this model element, such as:

- Changeable, which states whether the value of the attribute can be modified, is true;
- EType, the type of the attribute is String;
- EContainingClass, the class in which the attribute is defined, is Person;
- ...

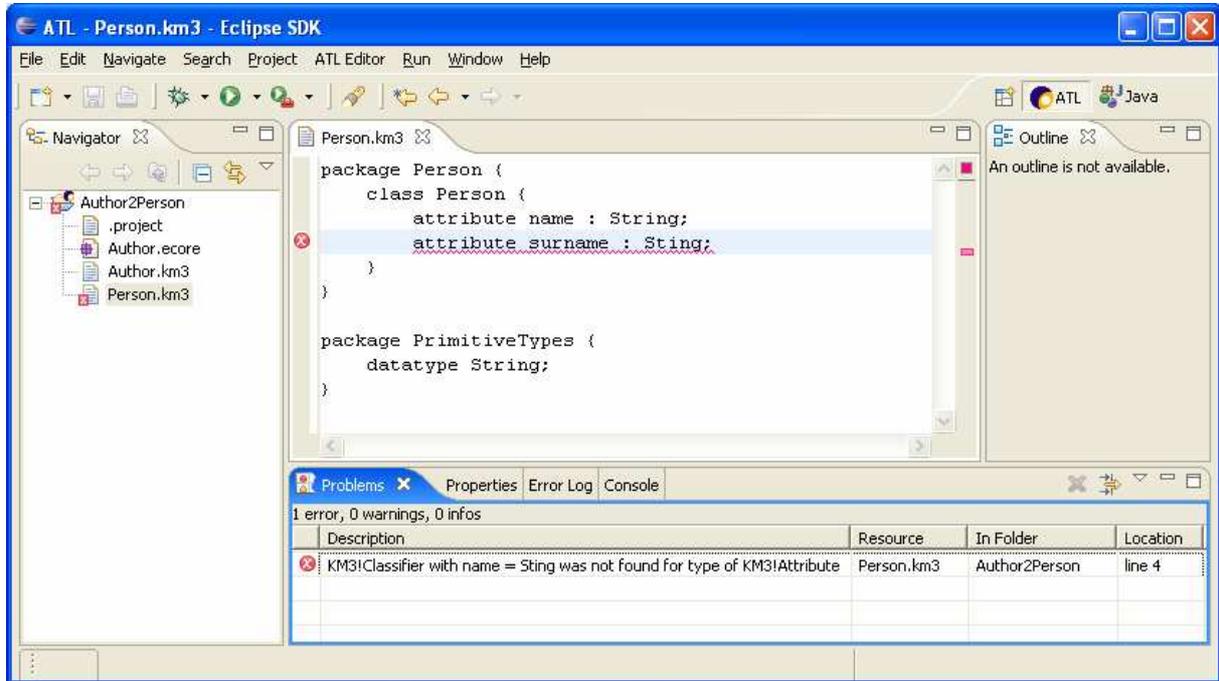


Figure 10. Error during the injection of a KM3 file

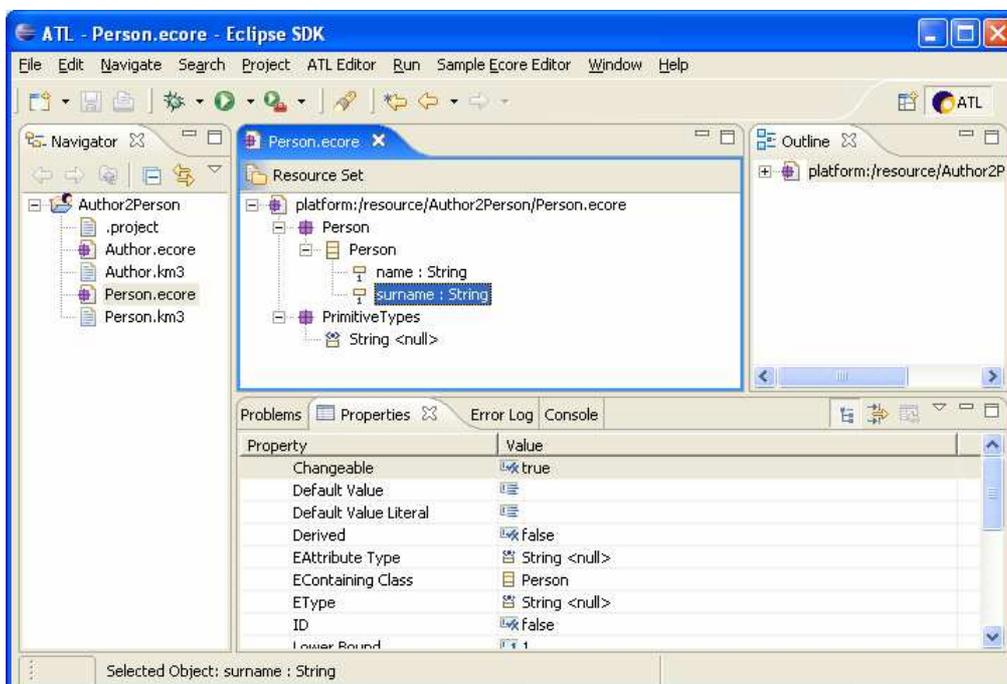


Figure 11. The generic Ecore file editor

The metamodels considered in the scope of this transformation are now available under the computable Ecore format. Next step is to edit an Author model that conforms to the Author metamodel and that will be used as part of the ATL transformation input.

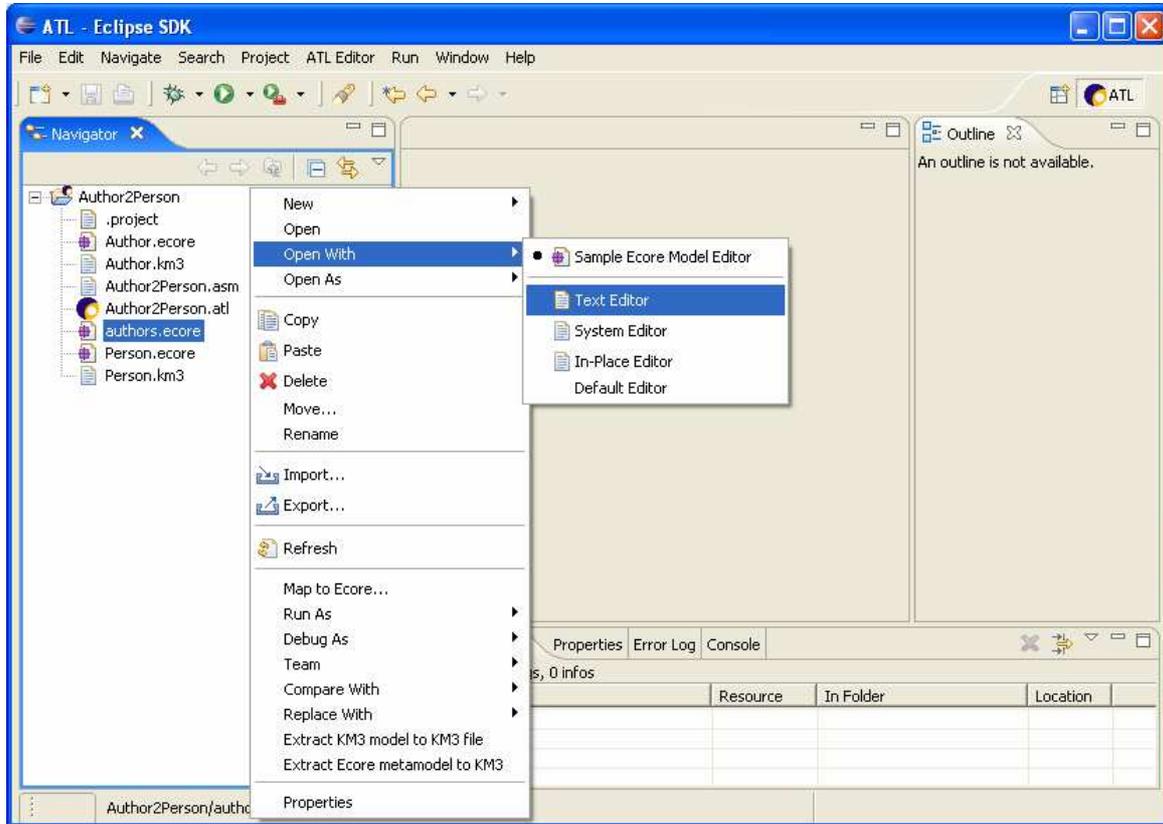


Figure 12. Textual edition of the *authors.ecore* model file

6 Editing the source model

Both input and output metamodels of the transformation are now available in the Ecore format. Executing the transformation example also requires providing an input model that conforms to the designed input metamodel.

There is no generic process for editing models. Given that the metamodels considered in the present example are quite simple, a basic but quite quick method consists in manually editing a simple Ecore model file. For this purpose, the first step is to create a new file, in the same way the KM3 file have been created in Section 5. The created file must have an *.ecore* extension, for instance *authors.ecore*. Once created, the model file has to be edited. In order to textually edit an Ecore file, select Text Editor in the mouse right click→Open With menu (see Figure 12).

The editor displays an empty textual file. Ecore is based on the XMI 2.0 [5] XML language. As a consequence, the first line must include a generic XML header providing versioning and encoding information:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Next step consists in specifying the root XMI tag. This tag will contain the different Author elements of the input model. It specifies versioning as well as namespace information. Take care here to correctly specify the path to the related Author metamodel:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="Author">
```

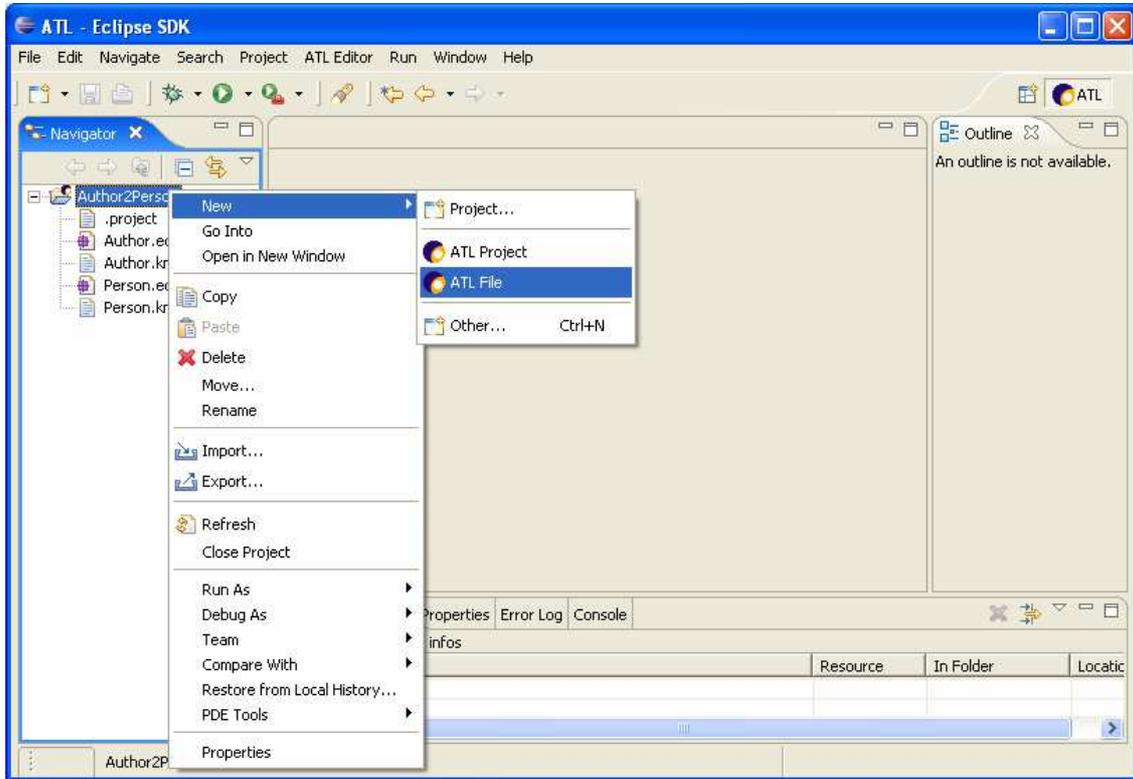


Figure 13. Creation of new ATL file

As a last step, the element of this Author model can be entered within the XML tag. Each entity of the input model here corresponds to an empty Author tag with two XML attributes encoding the name and the surname of Author entity. An example of input model is provided in Appendix A.

Now the input Ecore model has been edited, it can be explored with the graphical Ecore Model Editor. For this purpose, select Sample Ecore Model Editor in the mouse right click→OpenWith menu. Be sure, when trying to open a file with an editor, that this file is not being edited with another editor.

At this stage, both the metamodels and the input model involved in the transformation to be designed are available under the Ecore computable format. It is now possible to focus on the design the ATL transformation itself.

7 Programming my first ATL transformation

This section aims to guide the ATL user through the programming of its first ATL transformation. For this purpose, it introduces the basic ATL concepts that are required in the scope of the considered example. An exhaustive reference to the ATL transformation language can be found in the ATL User Manual [7].

First step in the design of the ATL transformation is the creation of the ATL transformation file. For this purpose, the ATL Development Tools includes a wizard dedicated to the creation of transformation files. Once the file is created, the code of the transformation has to be specified.

7.1 Creation of the ATL file

The ATL file creation wizard is called by selecting New→ATL File as shown in Figure 13. This operation opens the ATL File Wizard.

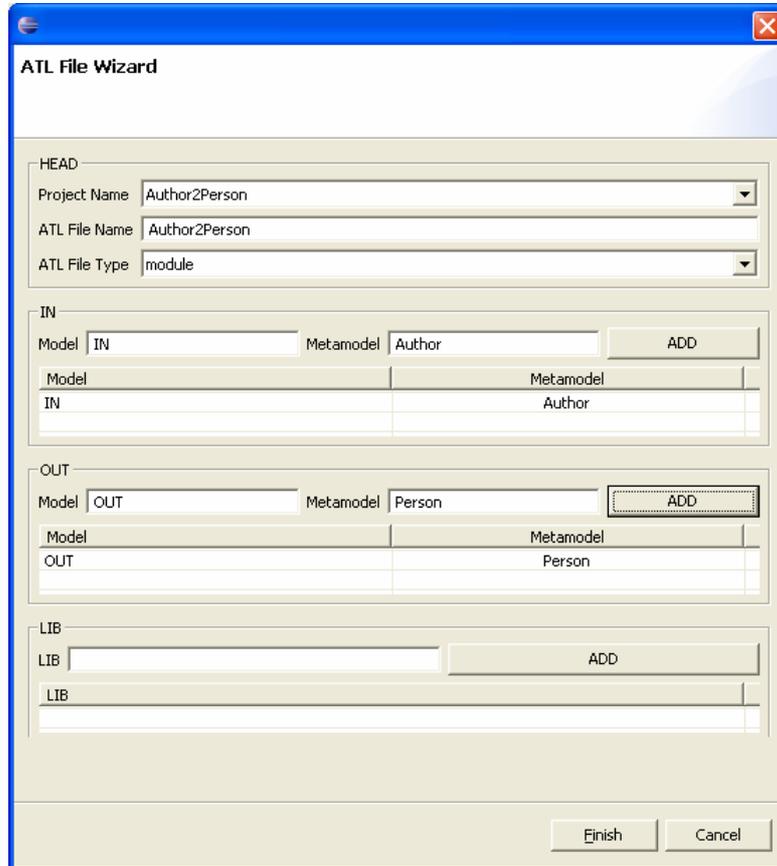


Figure 14. The ATL file wizard

The ATL File Wizard (see Figure 14) enables the ATL developer to specify some information relative to the transformation:

- The ATL project the transformation belongs to;
- The name of the ATL file containing the transformation (without extension);
- The type of the ATL transformation:
 - Module, which corresponds to a classical ATL transformation;
 - Query, enabling to return a primitive value from a model;
 - Library, which defines a set of ATL functions;
- The input and output models, along with their respective metamodels;
- The ATL libraries that are required by the transformation.

The present transformation is part of the created Author2Person ATL project. The file containing the transformation can be named Author2Person (which will lead to the creation of the *Author2Person.atl* file). Finally, this example corresponds to a classical ATL transformation, and the default type (module) has to be selected.

Next step corresponds to the specification of the transformation input and output models. In the IN section, the field Model enables to define the variable name that will correspond to the input model. By convention, the input model of an ATL transformation is often called “IN” (in the same way, its output model is called “OUT”). The field Metamodel contains the name of the metamodel of this input model.

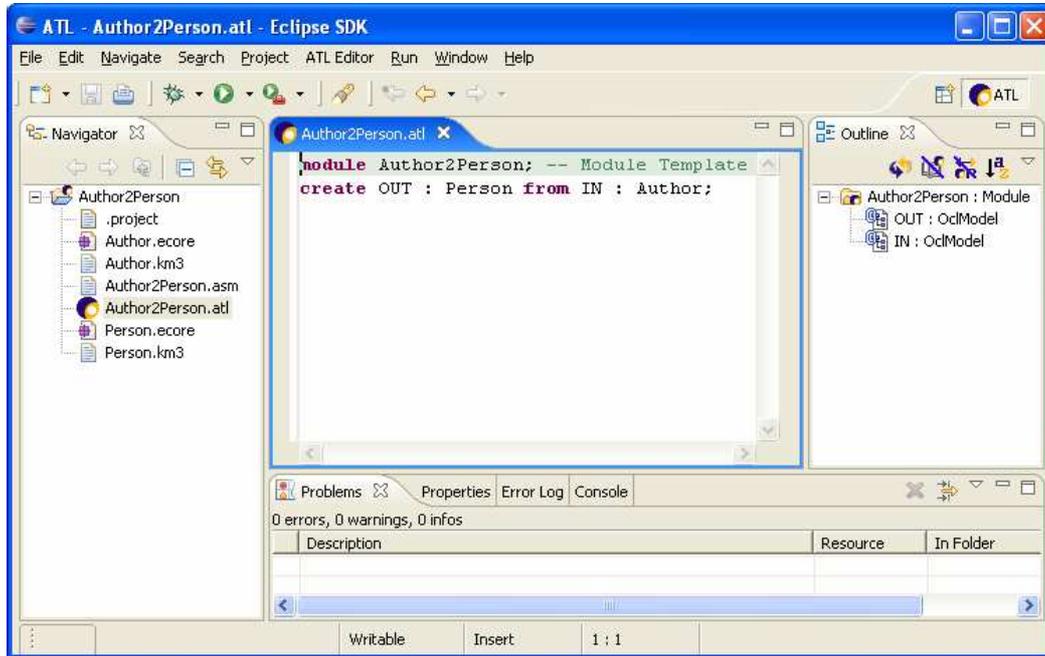


Figure 15. An ATL transformation template

Once these two fields have been filled, the specified couple can be validated by means of the Add button. The output model (OUT section) must be specified in the same way that the input one was.

At this stage, it is possible to note that an ATL transformation may accept several input or output models. Although this feature is out of the scope of this document, it may be noted that, as the entered values represent variable names, each of them must be unique (within the set of declared models).

The last step is to specify the name of the libraries that will be required for the transformation to run. The design of the current transformation does not require any specific library (see the ATL User Manual for further details [7]). The creation of the ATL file can then be validated (by clicking on Finish).

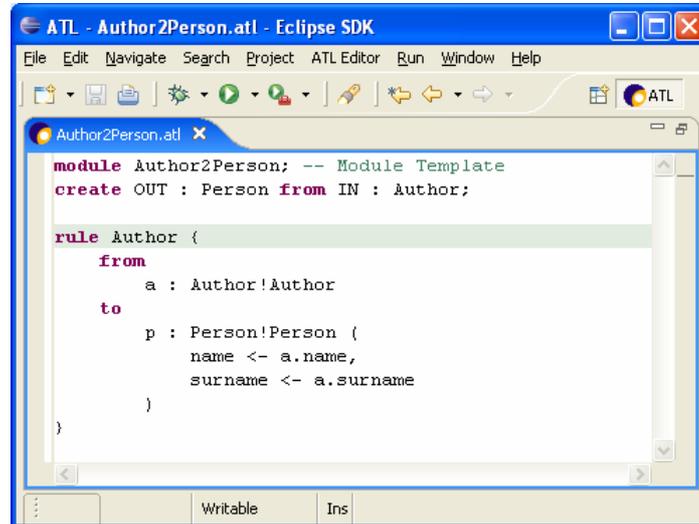
As a result of the ATL File Wizard, a transformation ATL file (*Author2Person.atl*) is created in the ATL project. This file contains the code of the ATL transformation. The ATL editor can be opened by double-clicking on the ATL file. Figure 15 presents a screenshot of the edition of the *Author2Person.atl* file. An ATL module (e.g. an ATL transformation) always starts by a module header. This header corresponds to the declaration of the module name (line 1) followed by the declaration of the input and output models, with their respective metamodels (line 2).

When an ATL file is edited with the ATL editor, the Outline view (right column) provides an overview of the content of the edited file. At this stage, the outline only displays a Module element (named Author2Person) for which two OclModel elements (e.g. two models), named IN and OUT, are defined.

Besides the transformation file, the ATL File Wizard creates an additional new file: the transformation ASM file (*Author2Person.asm*). This new file contains the bytecode corresponding to the ATL transformation. This bytecode is encoded into an XML format and is updated as the transformation file evolves.

7.2 Creation of a new transformation rule

Once the ATL transformation file has been created, the code of the transformation to be designed can be specified. For this purpose, this section quickly introduces the basic notions of the ATL language (see the ATL User Manual for further details [7]).



```
module Author2Person; -- Module Template
create OUT : Person from IN : Author;

rule Author {
  from
    a : Author!Author
  to
    p : Person!Person {
      name <- a.name,
      surname <- a.surname
    }
}
```

Figure 16. The Author2Person ATL transformation

Besides its header, an ATL module is composed of a set of ATL rules. Each rule defines the way an input element (that is a given type of entity of the input model) is transformed into a target element (a given type of entity of the output model). Both the input and the output elements of a rule are identified by a couple (metamodel_name, entity_name).

A rule is composed of an InPattern and an OutPattern. The InPattern declares a typed variable which corresponds to the rule input element. During the execution of the ATL transformation, this variable will correspond to the source element currently being matched.

The OutPattern declares a typed variable which corresponds to the rule output element. The OutPattern also specifies a set of Binding elements. A Binding describes how a given feature (an attribute or a reference) of the target element is initialized. This initialization must be specified as an OCL expression [8]. The present example only requires expressions enabling to access the input element features.

The considered transformation example aims to transform Author elements (from the Author metamodel) into Person elements (from the Person metamodel). A single rule, matching Author elements into Person elements, is therefore sufficient. This rule will be composed of an InPattern in which the source entity Author is declared. This source entity is typed by Author!Author (its couple *metamodel_name!entity_name*). The OutPattern of the defined rule declares, in the same way, a Person element from the Person metamodel (Person!Person). This OutPattern also defines two bindings that specify the initialization of the Person elements that will be generated by the rule. According to the semantics that have been defined in Section 3, the name and surname of created Person elements have to be copied from those of the corresponding source Author element.

Figure 16 provides an overview of the full code of the Author2Person ATL transformation (also available in Appendix B). Note that the Outline view of the edited ATL transformation is updated as the transformation is edited, so that it provides an up-to-date overview of the structure of the transformation.

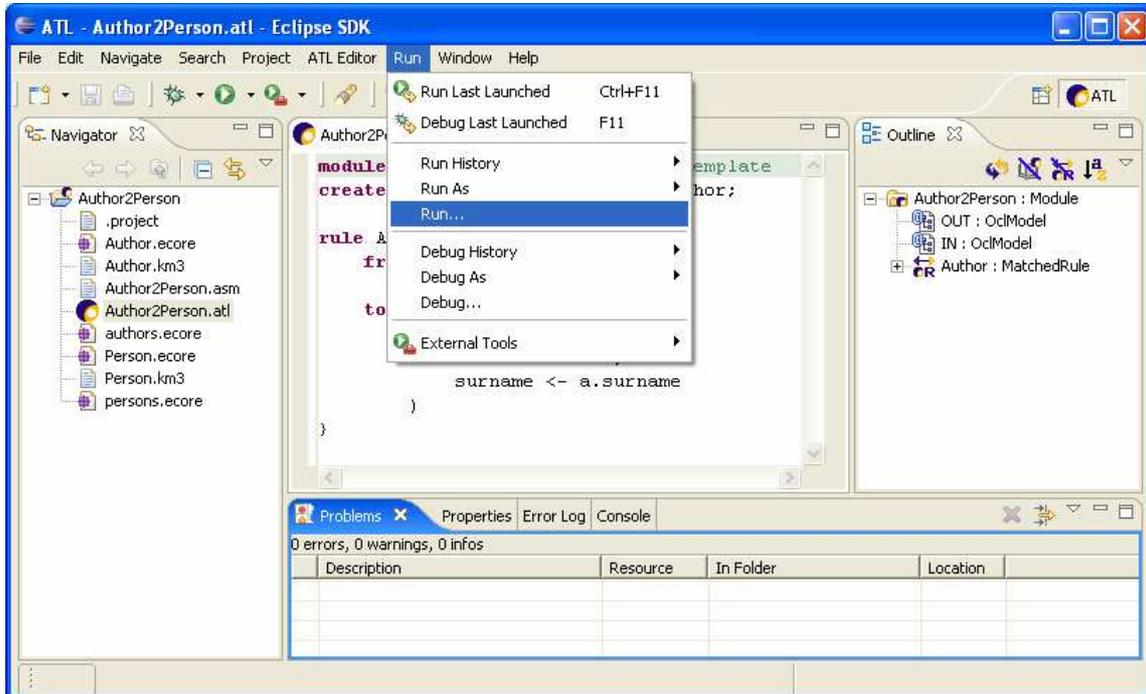


Figure 17. Creating a new launch configuration

8 Creating my first ATL launch configuration

Before being able to execute the designed ATL transformation, it is necessary to configure its associated launch configuration. An ATL launch configuration aims to resume all the information that is required to execute an ATL transformation. This mainly includes the path of the files involved in the transformation (the transformation file, the input and output models, the input and output metamodels, the libraries).

In order to create a new launch configuration, select the Run... item in the Run menu of the Eclipse window (see Figure 17). This opens a new Run window. Select here the ATL Transformation item (in the left column) and click on New to create a new ATL launch configuration.

The ATL launch configuration window (see Figure 18) is composed of three distinct tabs: ATL Configuration, Model Choice and Common. In the top of the configuration window, the user has to enter the name of the created configuration.

8.1 Configuring the ATL Configuration tab

The ATL launch configuration window (see Figure 18) is composed of three distinct tabs: ATL Configuration, Model Choice and Common. The ATL Configuration tab enables to specify the project that contains the transformation to be executed and the path of the ATL transformation file. The considered project is here the *Author2Person* project, and the path of the designed ATL transformation is */Author2Person/Author2Person.atl*.

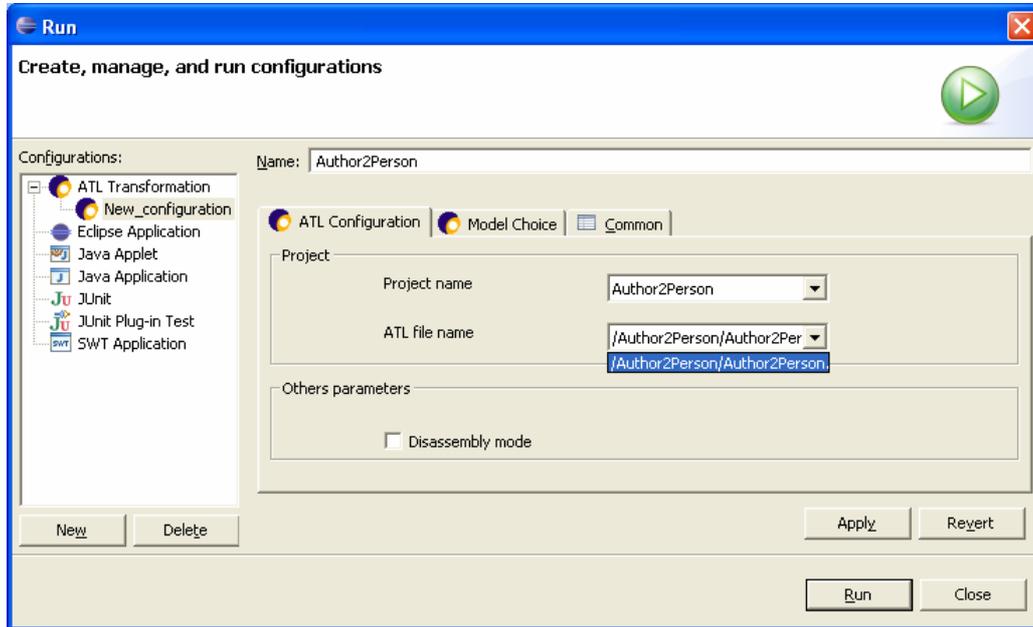


Figure 18. ATL launch configuration – ATL Configuration

8.2 Configuring the Model Choice tab

The main tab of ATL launch configuration is the Model Choice tab (see Figure 19). This tab enables to specify the name of the variables corresponding to involved models and metamodels (within the top IN and OUT parts), along with the path to the corresponding files (in the Path Editor section). It also enables to specify the name and the path of the libraries used by the transformation (in the Libs section).

The name of the input model variable (“IN”) must be entered in the Model field of the IN section. The name of its metamodel variable (“Author”) must be specified in the corresponding Meta Model field. This couple of variable names can be validated by means of the Add button. The output model variable and its corresponding Person metamodel must be specified in a similar way in the OUT section. Note that the variable names specified here must correspond to the variable names that appear in the ATL file. Once they have been validated, both the model and its metamodel appear in the table of the Path Editor section.

In order to complete the launch configuration, the user must specify a path for each element of the Path Editor table. A path can be specified through different means:

- The Set path button enables to select a file that belongs to a currently opened project;
- The Set external path button enables to select a file from the file system;
- The Metamodel by URI button enables to select a metamodel that have been already loaded by an EMF plug-in;
- The MM Is MOF-1.4 button enables to set the metamodel to MOF 1.4 using the version loaded by the ATL engine;
- Finally, the MM is Ecore button enables to set the metamodel to Ecore.

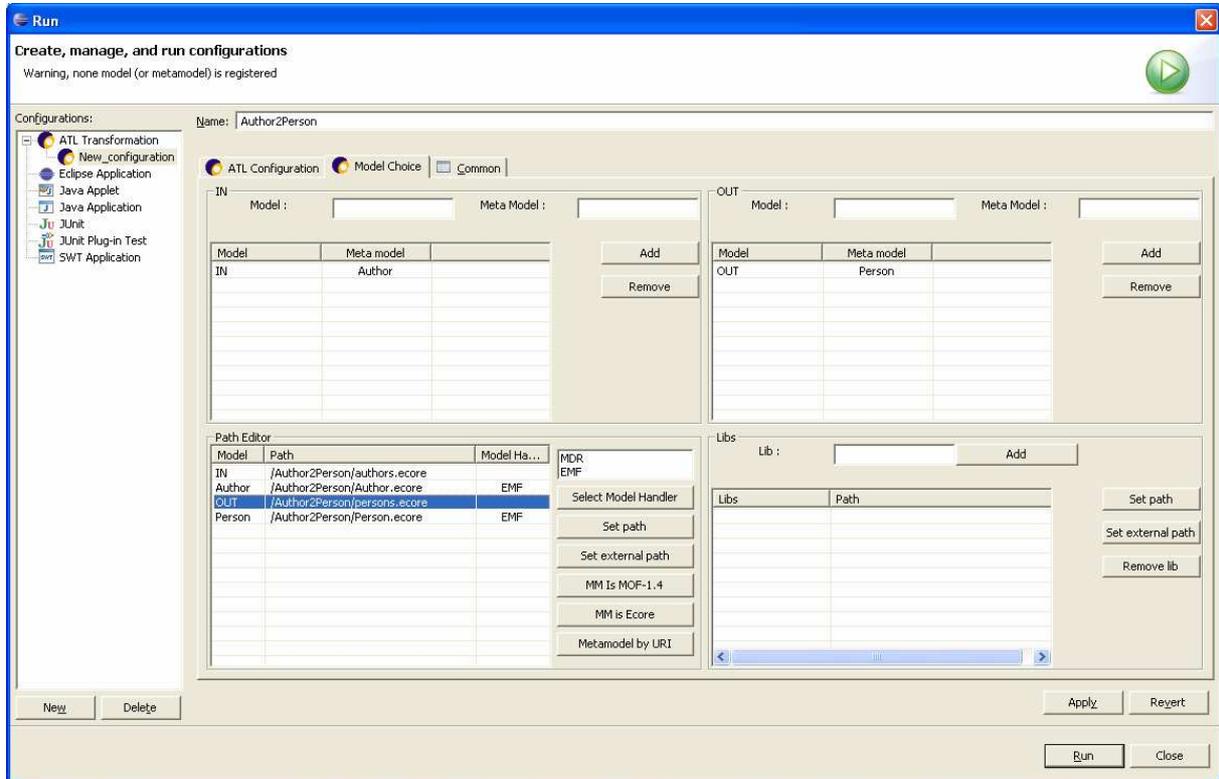


Figure 19. ATL launch configuration – Model Choice

Note that, whereas for the input model as well as for both input and output metamodels, the user just has to select an existing file, he, or she, has to enter the name of the file to be generated for the transformation output model. The extension given to this file name must be consistent with the model handler that is used for the output metamodel (see below). When using the EMF model handler, the output model file should be associated with an *.ecore* file extension.

The Path Editor also enables ATL users to select the model handler that is associated with each declared metamodel. Default value for the model handler is EMF. It can be modified by selecting and model handler (EMF for an Ecore metamodel, or MDR [9] for a MOF 1.4 metamodel) in the box on top of the buttons column. The selection must be validated by means of the Set Model Handler button. Note that, when setting the metamodel either to MOF 1.4 or Ecore (using the dedicated buttons), the model handler is automatically set to the MDR (for MOF 1.4) or EMF (for Ecore).

In the scope of the considered example, the path configuration only makes use of the Set path option since all model and metamodel files are defined within the Author2Person ATL project. Thus, the paths for the input and the output metamodels respectively correspond to the paths to *Author.ecore* and *Person.ecore* files. Similarly, the path to the input model corresponds to the path to the *authors.ecore* file. Finally, a path for the output model file can be defined in the Author2Person project. This output file can be named *persons.ecore*. Since both the input and output metamodels are defined according to the EMF semantics, there is no need to modify the model handlers they are associated with.

8.3 The Common tab

The last tab of the ATL launch configuration, Common, enables to specify advanced options. Default configuration of these options shall enable to run most of the ATL transformations. Detailing these advanced options is not interesting in the scope of this document. However, interested users will find further details on the subject in the ATL User Manual [7].

	ATL Documentations	
	ATL Starter's Guide	Date 07/12/2005

Once the launch configuration has been correctly fulfilled, it has to be saved with the Apply button. The transformation can then be executed.

9 Executing my first ATL transformation

Once the launch configuration of a transformation has been correctly fulfilled, it can be run as many times as needed without requiring any change to the configuration. To run the transformation, just go back to the Run window, select the created transformation in the ATL Transformation folder (on the left column) and click on the Run button.

The ATL engine tries to serialize the output file whether the transformation contains errors or not. Thus, in case of errors, the content of the serialized file may provide useful information on the type of these errors. However, this also means that a transformation should have failed although an output model file has been generated. At this stage, it is therefore advised to check the result of the run transformation.

There exists several ways to check whether a transformation has completed without error or not:

- The simplest way to make sure that a transformation has completed without any error is to check whether the transformation has generated error messages. At present time, this could only be achieved by executing ATL transformations from the Eclipse workbench (this requires ATL to be installed from sources). In this context, the error messages that are generated by the execution of an ATL transformation will be displayed within the Eclipse Console view (by default, at the bottom of the Eclipse window).
- While working on a transformation that generates an Ecore model, there exists a simple way to check whether this output model is well-structured. Indeed, if the EMF Sample Ecore Model Editor (mouse right click→Open With→Sample Ecore Model Editor) allows to explore the content of the output model file, the targeted file is well-structured. This however does neither ensures that the transformation has completed without any error (some errors do not affect the structure of the transformation result), nor means that the generated result is correct (it may contain non structural errors).
- Finally, wherever the ATL transformation is launched from (Eclipse or workbench), and whatever the model handler of the output model (EMF or MDR), it is always possible to explore the result file using the basic textual editor (mouse right click→Open With→Text Editor).

ATL developers can note that the ATL Development Tools include debugging facilities that aims to ease the transformation development process. The description of the debugging tools is out of the scope of this introducing document. However, further details on the debugging functionalities can be found in the ATL User Manual [7].

10 Conclusion

This document has introduced the ATL transformation language through the presentation of the different steps of the design of a simple ATL transformation example. For simplicity sake, the example presented in this document only makes use of a small subset of the full ATL facilities. There exists a number of documents providing detailed information on the different aspects of the ATL transformation tools. Thus, users who are interested in discovering the full potential of the ATL transformation language shall refer to the ATL User Manual [7]. A detailed description of the KM3 textual notation is also available in the KM3 Manual [7].

	ATL Documentations	
	ATL Starter's Guide	Date 07/12/2005

Moreover, a large number of ATL transformations examples (from very simple to complex ones) are available on the GMT project site [9], in the ATL Transformations section. The proposed transformation examples cover many different fields, from geometrical transformations to bridges between existing build tools.

Finally, there exists an ATL discussion board [11] enabling the ATL community to share information about ATL and the related model management tools.

11 References

- [1] The ATL Installation Guide. Available from the GMT project [10], section ATL Documentation.
- [2] OMG/MOF Meta Object Facility (MOF) 1.4. Final Adopted Specification Document. formal/02-04-03, 2002.
- [3] Budinsky, F., Steinberg, D., Ellersick, R., Grose, T. Eclipse Modeling Framework, Chapter 5 "Ecore Modeling Concepts". Addison Wesley Professional. ISBN: 0131425420, 2004.
- [4] Budinsky, F., Steinberg, D., Ellersick, R., Grose, T. Eclipse Modeling Framework. Addison Wesley Professional. ISBN: 0131425420, 2004.
- [5] OMG/XMI XML Model Interchange (XMI) 2.0. Adopted Specification. formal/03-05-02, 2003.
- [6] The Kernel MetaMetaModel (KM3) Manual. Available from the GMT project [10], section ATL Documentation.
- [7] The ATL User Manual. Available from the GMT project [10], section ATL Documentation.
- [8] OMG/OCL Object Constraint Language (OCL) 2.0. OMG Final Adopted Specification. ptc/03-10-14, 2003.
- [9] The netbeans Metadata Repository (MDR) project. <http://mdr.netbeans.org/>.
- [10] The Generative Model Transformer (GMT) project. <http://eclipse.org/gmt/>.
- [11] The ATL mailing list. http://groups.yahoo.com/group/atl_discussion/.

	ATL Documentations	
	ATL Starter's Guide	Date 07/12/2005

Appendix A The *authors.ecore* input file

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="Author">
3     <Author name="David" surname="Touzet"/>
4     <Author name="Freddy" surname="Allilaire"/>
5 </xmi:XMI>
```



Appendix B The *Author2Person.atl* file

```
1  module Author2Person; -- Module Template
2  create OUT : Person from IN : Author;
3
4  rule Author {
5      from
6          a : Author!Author
7      to
8          p : Person!Person (
9              name <- a.name,
10             surname <- a.surname
11         )
12 }
```