ADT: Eclipse development tools for ATL

Freddy Allilaire (freddy.allilaire@laposte.net) Tarik Idrissi (tarik.idrissi@laposte.net)

Université de Nantes
Faculté de Sciences et Techniques
LINA (Laboratoire d'Informatique de Nantes-Atlantique)
44322 Nantes Cedex 3, France

Abstract

This paper presents our work on the creation of an Integrated Development Environment (IDE) for ATL under Eclipse. ATL is a language for expressing model transformations. The ATL IDE proposes tools that may be found in traditionnal programming languages: syntax highlighting, code completion, wizard for the creation of project or debugger. These developed tools make it possible to create more easily transformations.

Keywords : MDA, ATL, IDE, Eclipse, Model transformation

1. Introduction

The MDA is an OMG initiative in response to the recent new software crisis. If the platform changes were rather rare before, they are much more frequent today. The idea of the MDA is that technologically neutral models could be implemented on a large variety of technologically different platforms in an automated way [1]. However tools supporting MDA basic operations are still rather rare. ATL proposes a partial answer to this situation. It is a language, engine and environment for model transformations developped by the ATLAS team in Nantes [2].

2. Concept

2.1 ATL

What is ATL ?

ATL stands for ATLAS Transformation Language. ATLAS is a recently created INRIA project located at the University of Nantes (LINA) and focusing on data-centric systems.

The ATL transformation language that is being developed by the ATLAS team is a hybrid language, i.e. a mix of declarative and imperative constructions designed to express model transformation as required by any MDA approach. It intends to answer the MOF/QVT RFP issued by OMG. It is described by an abstract syntax tree (a MOF meta-model) and a textual concrete syntax. Any transformation such as PIM (Platform Independent Model) to PSM (Platform Specific Model) can then be written in ATL.

What is an ATL transformation ?

An ATL transformation can be decomposed into three parts : a header, helpers and rules.

The header is used to declare general information such as the module name (it is the transformation name : it must match the file name), the input and output meta-model and potential import of needed libraries.

Helpers are subroutine that are used to avoid code redundancy. We can imagine a helper for translating UML visibility kind to MOF visibility kind in a UML to MOF transformation. It is easy to guess the utility of this helper when we know all UML and MOF elements that define a visibility.

Rules are the heart of ATL transformations because they describe how output elements (based on the output meta-model) are produced from input elements (based on the input meta-model). They are made up of bindings, each one expressing a mapping between an input element and an output element.

The ATL project is part of the GMT Eclipse project. That is why Eclipse has been chosen to be used as an IDE for ATL. In the next part of the paper we are going to present the Eclipse tools for ATL.

2.2 Eclipse

Eclipse was created by OTI and IBM teams responsible for IDE products. Eclipse [3] is a platform that has been designed for building integrated web and application development tooling. The value of the platform is that it encourages rapid development of integrated features based on a **plug-in** model. Eclipse has a wide community of tool developers.

Eclipse provides a common user interface model for working with tools. It is designed to run on multiple operating systems while providing robust integration with each underlying OS.

At the core of Eclipse there is an architecture for dynamic discovery, loading, and running of plug-ins. The platform handles the logistics for finding and running the right code. The platform UI provides a standard user navigation model. Each plug-in can then focus on a small number of well-defined tasks.

Open architecture

The Eclipse platform defines an open architecture so that each plug-in development team can focus on their area of expertise. If the platform is correctly designed, new features can be added without impact to other tools. Newly developed tools can plug into the workbench using well defined hooks called **extension points**.

The platform itself is built in layers of plug-ins, each one defining extensions to the extension points of lower-level plug-ins, and in turn defining their own extension points for further customization. This extension model allows plug-in developers to add a variety of function to the basic tooling platform. The platform manages the complexity of different runtime environments. Plug-in developers can focus on their specific task instead of worrying about these integration issues.

2.3 EMF

EMF [4] (Eclipse Modelling Framework) is a modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF is used to define and implement structured data models. A data model is simply a set of related classes used to handle the data which you want to deal with in your application. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

3. Tools supporting ATL transformation

The ATL project will progressively provide a complete environment based on Eclipse for developing, testing and using model transformation programs. The environment under Eclipse for ATL is called ADT (ATL Development Tooling). Currently, the following items have been developed:

- A project kind and a project creation wizard
- A text editor with syntax highlighting and bracket matching
- A content outline and a property view associated to the editor
- Smart icons for all ATL specific elements
- A builder associated to the project kind
- A perspective
- Compilation error report
- A debugger

The development of the following items has started and will be completed soon :

- Rename refactoring (in near feature)
- Execution error report
- Auto-completion and content assist for the editor

3.1 A project kind, a project creation wizard and a perspective

Let us start by explaining the concepts of project and perspective in Eclipse.

A project enables to group, organize and manage related elements: in a Java project we can find packages, their respective classes and an associated builder (Java builder) allowing to compile easily each Java class.

A perspective is a visual container for a set of views and editors : in the Java perspective there is among others things a package explorer view (to easily navigate through Java classes) and Java editors (each one editing a Java file with syntax highlighting and other advanced features).

We have defineed our own project kind named “ATL Project” with its specific icons. It is possible to create an ATL project by using the ATL project creation wizard.

An ATL project is automatically associated to the ATL builder. This makes building the ATL transformations contained by an ATL project quite easy: right click on the project and select build. The build operation can even be performed automatically at resource save time if the user activates the appropriate option in his preferences.

Others facilities are offered for an ATL transformation file: deleting or renaming it causes its associated compiled file to be deleted or renamed automatically. When moved in any other folder, its associated compiled file will be moved too to the same folder.

Moreover, ATL defines its own perspective made up of the ATL editor, the ATL outline associated to this editor, the problem view used to report compilation errors.

3.2 Editor

We know how useful can be an editor to help the developer in his task. That is why the first component developed for the ATL language was an editor. The features currently provided by the editor for ATL (that you can see in the middle of **figure 1**) are:

- Syntax highlighting fully customisable through preference pages; the user may change the colour of the editor background, ATL keywords , ATL types and so on.

- Bracket matching: when the user types in an opening bracket, the ATL editor automatically adds its closing and indents the content between those two brackets by offsetting.

The development of the auto-completion feature, that will allow the user for example to dynamically get all the available methods for a variable according to its type, has started and will be available soon. The final purpose is to make that editor support all major features provided by the Eclipse Java editor.

Editors often have corresponding content outliners that provide a structured view of the editor contents and assist the user in navigating through the content of the editor. Thus, like the Java editor, the ATL editor always gets an outline.

For the outline too, the purpose is to make it support all major features supplied by the Java outline. Presently the functionalities related to the outline are the following:

- Selecting an element in the outline causes the area of this element in the editor to be highlighted and the editor vertical rule to be updated (for example an item representing a helper will highlight this one in the editor and will make the editor vertical rule lying down from the helper starting line to its end line).

- When the editor cursor position changes, by scrolling up or down with the keyboard or by clicking with the mouse, the element corresponding to the editor current position is selected in the outline

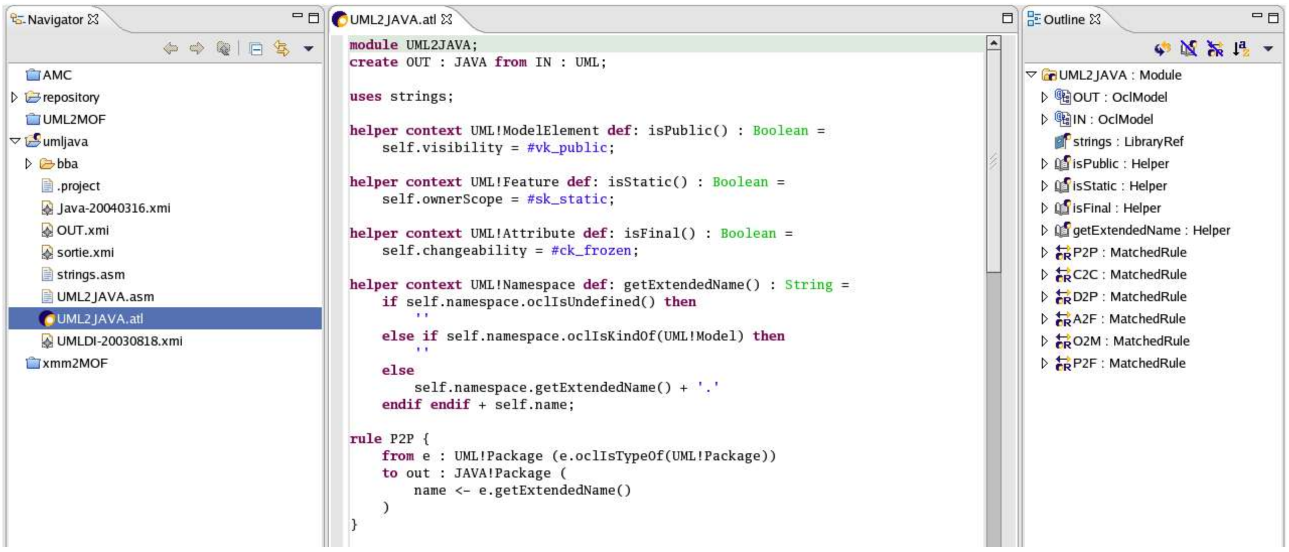
- Clicking on an object in the outline causes information such as its location, its name and so on to be displayed in the property view (in Eclipse a view is a frame used to display information user friendly and from which we can carry out actions).

- It is possible to sort and filter the content of the outline, e.g. hide helpers or rules

The specificity of the ATL outline is that while Java outline only contains the class, its attributes, methods and subclasses, ATL outline contains the whole Abstract Syntax Tree (AST). It is then possible to navigate through basic elements such as expressions, operators and so on. This is due to the fact that the ATL outline was not only designed to support traditional outline facilities but also to play an important role in the debugging task. This needs some explanations. Breakpoints are very important in the debugging process and in many languages as Java, breakpoints are placed on lines. In ATL, breakpoints may be placed on almost any element of the AST. And in that respect, the ATL outline will play an essential role. Indeed, adding a breakpoint can be done by right clicking on an element and selecting "A dd breakpoint". This al lows fine grained control of the execution process.

The ATL outline (that you can see on the right of **figure 1**) is also designed to support rename refactoring. This feature is not completely available yet but the principle will be the same as for breakpoints: selecting an element in the AST, right clicking and selecting "rename" will open a frame enabling to carry out the refactoring.

The outline is designed according to the MVC (Model View Controller) pattern. As for the Model part it is not made up of traditional Java classes but is an EMF based Model. The ATL meta-model is loaded by EMF providing then tools that enable to deal easily with ATL model elements.



```
module UML2JAVA;
create OUT : JAVA from IN : UML;

uses strings;

helper context UML!ModelElement def: isPublic() : Boolean =
    self.visibility = #vk_public;

helper context UML!Feature def: isStatic() : Boolean =
    self.ownerScope = #sk_static;

helper context UML!Attribute def: isFinal() : Boolean =
    self.changeability = #ck_frozen;

helper context UML!Namespace def: getExtendedName() : String =
    if self.namespace.oclIsUndefined() then
        ''
    else if self.namespace.oclIsKindOf(UML!Model) then
        ''
    else
        self.namespace.getExtendedName() + '.'
    endif endif + self.name;

rule P2P {
    from e : UML!Package (e.oclIsTypeOf(UML!Package))
    to out : JAVA!Package (
        name <- e.getExtendedName()
    )
}
```

```
UML2JAVA : Module
  OUT : OclModel
  IN : OclModel
  strings : LibraryRef
  isPublic : Helper
  isStatic : Helper
  isFinal : Helper
  getExtendedName : Helper
  P2P : MatchedRule
  C2C : MatchedRule
  D2P : MatchedRule
  A2F : MatchedRule
  O2M : MatchedRule
  P2F : MatchedRule
```

Figure 1 – ATL Editor with its content outline

3.3 Execution of transformation

In order to execute ATL transformations, there are two ways like for Java programs : run and debug.

Debug plug-ins of Eclipse allow to extend the plateform in order to launch correctly your program, with parameters of the user.

Before launching a transformation, one needs to create a launch configuration. One should specify parameters required for the execution of the transformation. Then it is possible to run a transformation using a generic ATL launch configuration that derives most of the launch parameters from the ATL project and the workbench preferences. It is however possible to override the derived parameters or specify additional arguments.

There is two different ATL launchers: ATL configuration and Remote ATL configuration.

ATL configuration

The ATL Configuration defines the transformation to be executed. This configuration needs the name of the project where your transformation is located, the name of your transformation and the model handler (for the moment MDR or EMF). This configuration needs also the name and the path of the models used in the transformation (INPUT MODEL, OUTPUT MODEL, INPUT METAMODEL, OUTPUT METAMODEL and LIBS). For the moment, the name of your model should be the same between configuration and ATL transformation file. But soon, thanks to the repository, this binding will be automatically achieved.

Example : *create OUT : Java from IN : UML;* in your transformation implies in this tab: the couple IN (model) and UML (metamodel) in the IN part and the couple OUT (model) and Java (metamodel) in the OUT part.

Currently, we are working on the repository where all metamodels will be stored, the metamodels needed will be found in the repository and link with launch configuration automatically made.

Remote ATL configuration

This launching configuration is useful when you want to launch the debuggee (the debugged program) out of Eclipse (for example, if your transformation needs injector or extractor, the ATL configuration cannot be used for the moment but it is possible with Remote ATL configuration). Otherwise, you can use the ATL configuration when you have a project with input and output models, input and output metamodels; it is easier to use than remote ATL configuration.

This configuration just needs the name of the project and the name of the ATL transformation and connection properties : hostname (for example: localhost) and port.

If all the parameters are given (project, transformation, model handler, input and output model), you can apply the configuration, else a message error at the top of the dialog displays what is wrong in your configuration (for example, the path of your model is empty).
When the configuration is completed, the launcher is ready to do its work. ATL Transformation can be runned or debugged.

Running your programs

In run mode, the program executes, but the execution may not be suspended or examined.

When the transformation has finished its work, output model has been created or updated. In the next version of the ADT, execution errors will be displayed in the Eclipse console.

Debugging your programs

In debug mode, execution may be suspended and resumed, breakpoints added, and you can see the variables of the stackframes.

There are several important views in the debug perspective.

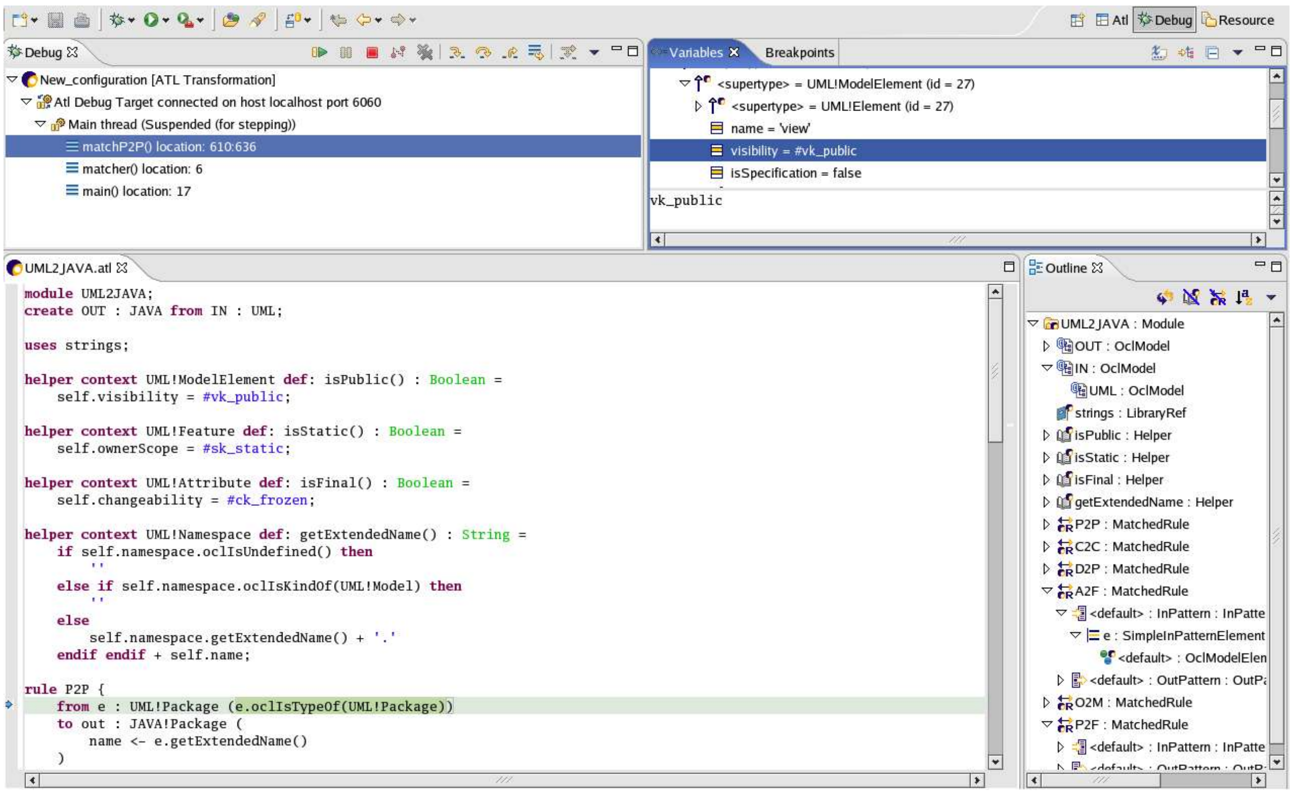
In the **Debug view** (top left of **figure 2**), you see the different configurations launched with their state. For each configuration, there is the debug target, threads and stackframes. At the beginning, the transformation is suspended. When the breakpoint is hit, execution is suspended. Notice that the process is still active (not terminated) in the Debug view.
You can step through the code with several action. With the action **Step Over**, the execution will continue at the next line in the same rule. You can also use **Step Into** or **Step Return** to step through the code. You can end a debugging with action Terminate, to step over the code until the transformation completes or **Resume** to allow the program to run until the next breakpoint is encountered or until the program is completed. When an exception is raised, you can see it on the debug view.

In the **Editor view** (bottom left of **figure 2**), you can see the code being debugged, the breakpoint. On each step, you see the highlighted text changed.

The **Variable view** (top right of **figure 2**) displays the values of the variables in the selected stack frame. You can expand the tree in the Variables view. The variables in the Variable view will change when you step in the Debug view.

The **Outline view** is useful in debug context to put a breakpoint. In fact, breakpoints are added with this view, they are added on the element selected.

The **Breakpoint view** lists all the breakpoints you have set in the workbench projects. Breakpoints can be enabled, disabled, deleted, added and located.



```
New_configuration [ATL Transformation]
  Atl Debug Target connected on host localhost port 6060
    Main thread (Suspended (for stepping))
      matchP2P() location: 610:636
      matcher() location: 6
      main() location: 17
```

```
<supertype> = UML!ModelElement (id = 27)
  <supertype> = UML!Element (id = 27)
  name = 'view'
  visibility = #vk_public
  isSpecification = false
vk_public
```

```
module UML2JAVA;
create OUT : JAVA from IN : UML;

uses strings;

helper context UML!ModelElement def: isPublic() : Boolean =
    self.visibility = #vk_public;

helper context UML!Feature def: isStatic() : Boolean =
    self.ownerScope = #sk_static;

helper context UML!Attribute def: isFinal() : Boolean =
    self.changeability = #ck_frozen;

helper context UML!Namespace def: getExtendedName() : String =
    if self.namespace.oclIsUndefined() then
        ''
    else if self.namespace.oclIsKindOf(UML!Model) then
        ''
    else
        self.namespace.getExtendedName() + '.'
    endif endif + self.name;

rule P2P {
    from e : UML!Package (e.oclIsTypeOf(UML!Package))
    to out : JAVA!Package (
        name <- e.getExtendedName()
    )
```

```
UML2JAVA : Module
  OUT : OclModel
  IN : OclModel
    UML : OclModel
  strings : LibraryRef
  isPublic : Helper
  isStatic : Helper
  isFinal : Helper
  getExtendedName : Helper
  P2P : MatchedRule
  C2C : MatchedRule
  D2P : MatchedRule
  A2F : MatchedRule
    <default> : InPattern : InPatte
      e : SimpleInPatternElement
        <default> : OclModelElen
    <default> : OutPattern : OutPa
  O2M : MatchedRule
  P2F : MatchedRule
    <default> : InPattern : InPatte
```

Figure 2 – ATL Debug perspective with the ATL Editor, Debug, Variable and Outline view

4. Conclusion

The goal of the ATL Development Tooling (ADT) is to simplify and assist the creative task of ATL programming and debugging. ATL is a transformation language and Eclipse allows us to develop an environment powerful and complete for ATL. But ADT should rapidly grow up because it will be necessary to meet more and more user requirements. For example, a repository will be created and accessible under Eclipse. Metamodels or transformations will be found in this MDA component repository. Issues of naming, remote access, encapsulation and typing will also be handled by this repository. Programming model transformations is a hard task, but we also have to seriously handle many other related tasks like reusing, specifiying and checking transformations. This paper has briefly presented the curent state of tools supporting transformation development and debugging, which is an important part of the work of the transformation programmer.

5. Acknowledgements

We thank Jean Bézivin, Frédéric Jouault and the members of the ATL Project for multiple advices and comments on this work.

Reference

[1] J Bézivin : From Object-Composition to Model-Transformation with the MDA. TOOLSUSA 2001, Santa Barbara, USA2001
http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/TOOLS.USA.pdf
[2] J Bézivin, G Dupé, F Jouault, G Pitette, and E J Rougui : First experiments with the ATL model transformation language: Transforming XSLT into XQuery
http://www.softmetaware.com/oopsla2003/bezivin.pdf
[3] Eclipse Platform Technical Overview
http://www.Eclipse.org/whitepapers/Eclipse-overview.pdf
[4] Using EMF http://www.Eclipse.org/articles/Article-Using EMF/using-emf.html