	ATLAS group ***LINA & INRIA*** **Nantes**	

ATL Inventory

Date of preparation: 31/08/06

Revision: 0.1

Rev.	Date (dd/mm/yy)	Author	Checked by	Description
0.1		Freddy Allilaire		Creation of the document

Table of contents

Tables list

Figures list

1. Introduction

ATL, the Atlas Transformation Language, is the ATLAS INRIA & LINA research group's answer to the OMG MOF/QVT RFP. It is a model transformation language specified as both a metamodel and a textual concrete syntax. In the field of Model-Driven Engineering (MDE), ATL provides developers with a mean to specify the way to produce a number of target models from a set of source models.

The ATL language is a hybrid of declarative and imperative programming. The preferred style of transformation writing is the declarative one: it enables to simply express mappings between the source and target model elements. However, ATL also provides imperative constructs in order to ease the specification of mappings that can hardy be expressed declaratively.

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. Besides basic model transformations, ATL defines an additional model querying facility that enables to specify requests onto models. ATL also allows code factorization through the definition of ATL libraries.

Developed over the Eclipse platform, the ATL Integrated Development Environment (IDE) provides a number of standard development tools (syntax highlighting, debugger, etc.) that aim to ease the design of ATL transformations. The ATL development environment also offers a number of additional facilities dedicated to models and metamodels handling. These features include a simple textual notation dedicated to the specification of metamodels, but also a number of standard bridges between common textual syntaxes and their corresponding model representations.

2. An Introduction to Model Transformation

Models are now part of an increasing number of engineering processes (such as software engineering). However, in most cases, they are still confined to a simple documentation role instead of being actively integrated into the engineering process. As opposed to this passive approach, the field of Model-Driven Engineering (MDE) aims to consider models as first class entities. It also considers that the different kinds of handled items (such as the tools, the repositories, etc.) can be viewed and represented as models. The model-driven approach supposes to provide model designers and developers with a set of operations dedicated to the manipulation of models. In this context, model transformation appears to be a central operation for model handling: it aims to make it possible to specify the way to produce a number of target models based on a set of source models. In the scope of the model-driven engineering, it is assumed that model transformations, as any other model-based tool, can be modelled, which means that they have to be considered themselves as models.

This section aims to provide an overview of the main MDE concepts, with a particular focus on model transformation. To this end, it first presents, in Section 2.1, the organisation of the model-driven architecture. This first section addresses the model definition mechanisms that constitute the core of the MDE area: it introduces the notions of models, metamodels and metametamodels, as well as the conformance relation that relates these different artefacts. The second part of the section more particularly deals with model transformation. It provides an overview of the conceptual model transformation architecture and detailed the way this conceptual architecture is matched to the ATL language.

2.1. The Model-Driven Architecture

Models constitute the basic pieces of the model-driven architecture. Indeed, in the field of model-driven engineering, a model is defined according to the semantics of a model of models, also called a *metamodel*. A model that respects the semantics defined by a metamodel is said to *conform* to this metamodel. As an example, Figure 1 illustrates the conformance relation between a Petri net model and the Petri Nets metamodel.

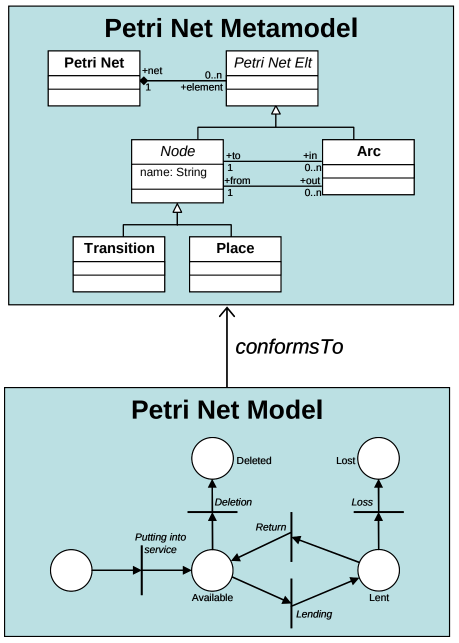

Figure 1. Conformance relation

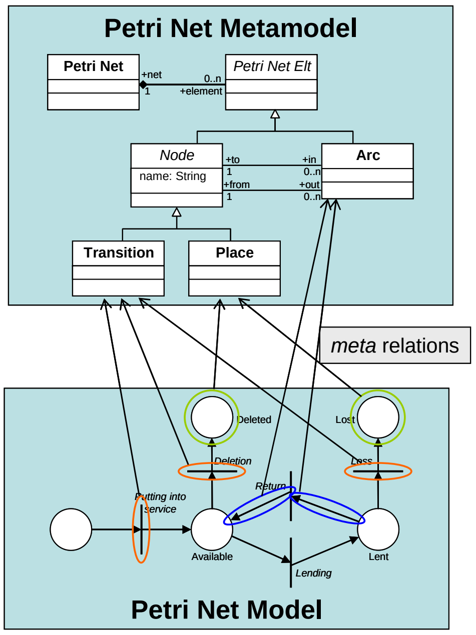

Figure 2. Meta relations

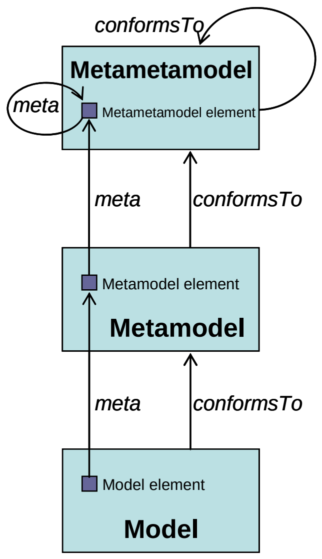

Figure 3. The model-driven architecture

As every model, the described Petri net model is composed of a number of distinct model elements. In the context of a Petri net, these model elements correspond to the places, the transitions and the arcs that compose the model. These different elements, as well as the way they are related, are defined in the scope of the Petri net metamodel. In the same way a model conforms to its metamodel, there exists a relation between the elements of a model and those of its metamodel. This relation, called *meta*, associates each element of a model with the metamodel element it instantiates. Figure 2 illustrates some of the existing meta relations between elements of the Petri net model and those of the Petri net metamodel.

At this stage, it must be recalled that, before being a metamodel, a metamodel is a model. This implies for it to conform to its own metamodel. To this end, the model-driven architecture defines a third modelling level which corresponds to the *metametamodel*, as illustrated in Figure 3.

A metametamodel aims to introduce the semantics that are required to specify metamodels. As a model with its metamodel, a metamodel conforms to the metametamodel. Note that a metametamodel is usually self-defined, which means that it can be specified by means of its own semantics. In such a case, a metametamodel conforms to itself.

Several metametamodel technologies are available. The ATL transformation engine currently provides support for two of these existing technologies: the Meta Object Facilities (MOF 1.4) [40] defined by the OMG and the Ecore metametamodel [41] defined by the Eclipse Modelling Framework (EMF) [42]. This means that ATL is able to handle metamodels that have been specified according to either the MOF or the Ecore semantics.

2.2. Model Transformation

In the scope of model-driven engineering, model transformation aims to provide a mean to specify the way to produce target models from a number of source models. For this purpose, it should enable developers to define the way source model elements must be matched and navigated in order to initialize the target model elements.

Formally, a simple model transformation has to define the way for generating a model M_b, conforming to a metamodel MM_b, from a model M_a conforming to a metamodel MM_a. As previously highlighted, 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 metametamodel.

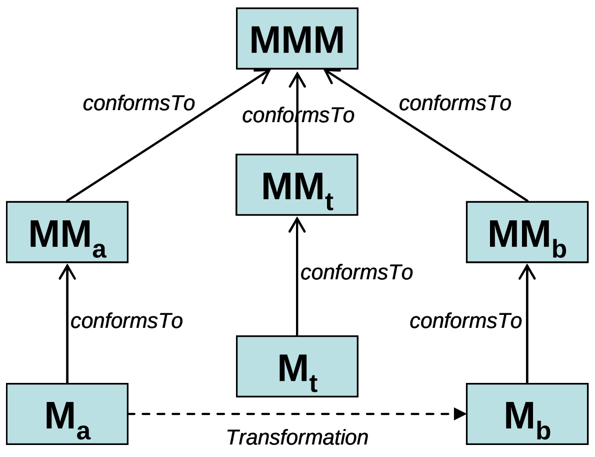

Figure 4. An overview of model transformation

Figure 4 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 MMM (such as MOF or Ecore).

ATL is a model transformation language that enables to specify how one (or more) target model can be produced from a set of source models. In other word, ATL introduces a set of concepts that make it possible to describe model transformations.

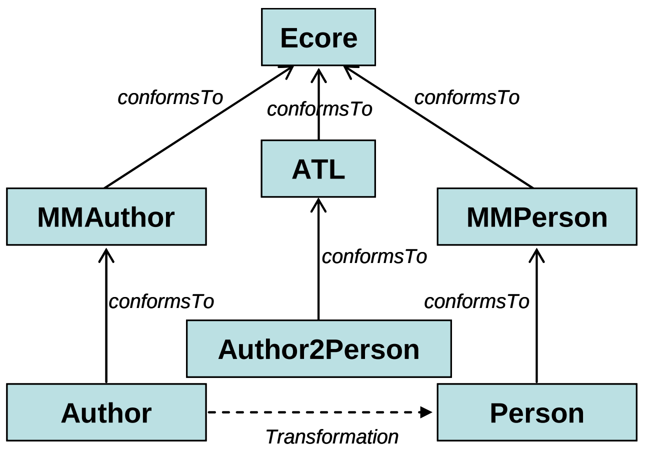

Figure 5. Overview of the Author to Person ATL transformation

Figure 5 provides an overview of the ATL transformation (Author2Person) that enables to generate a Person model, conforming to the metamodel MMPerson, from an Author model that conforms to the metamodel MMAuthor. The designed transformation, which is expressed by means of the ATL language, conforms to the ATL metamodel. In this example, the three metamodels (MMAuthor, MMPerson and ATL) are expressed using the semantics of the Ecore metametamodel.

3. Overview of the Atlas Transformation Language

The ATL language offers ATL developers to design different kinds of ATL units. An ATL unit, whatever its type, is defined in its own distinct ATL file. ATL files are characterized by the *.atl* extension.

As an answer to the OMG MOF/QVT RFP, ATL mainly focus on the model to model transformations. Such model operations can be specified by means of ATL *modules*. Besides modules, the ATL transformation language also enables developers to create model to primitive data type programs. These units are called ATL *queries*. The aim of a query is to compute a primitive value, such as a string or an integer (see Section [43] for further details on the set of ATL primitive data types), from source models. Finally, the ATL language also offers the possibility to develop independent ATL *libraries* that can be imported from the different types of ATL units, including libraries themselves. This provides a convenient way to factorize ATL code that is used in multiple ATL units. Note that the three ATL unit kinds same the share *.atl* extension.

These different ATL units are detailed in the following subsections. This section explains what each kind of unit should be used for, and provides an overview of the content of these different units.

3.1. ATL module

An ATL module corresponds to a model to model transformation. This kind of ATL unit enables ATL developers to specify the way to produce a set of target models from a set of source models. Both source and target models of an ATL module must be “typed” by their respective metamodels. Moreover, an ATL module accepts a fixed number of models as input, and returns a fixed number of target models. As a consequence, an ATL module can not generate an unknown number of similar target models (e.g. models that conform to a same metamodel).

Section 3.1.1 details the structure of an ATL module. Section 3.1.2 presents the two available execution modes for ATL modules. Finally, the execution semantics of the ATL module are briefly introduced in Section 3.1.3

3.1.1. Structure of an ATL module

An ATL module defines a model to model transformation. It is composed of the following elements:

- A header section that defines some attributes that are relative to the transformation module;
- An optional import section that enables to import some existing ATL libraries (see Section 3.3);
- A set of helpers that can be viewed as an ATL equivalent to Java methods;
- A set of rules that defines the way target models are generated from source ones.

Helpers and rules do not belong to specific sections in an ATL transformation. They may be declared in any order with respect to certain conditions. These four distinct element types are now detailed in the following subsections.

3.1.1.1. Header section

The header section defines the name of the transformation module and the name of the variables corresponding to the source and target models. It also encodes the execution mode of the module. The syntax for the header section is defined as follows:

```
module module_name;
create output_models [from|refines] input_models;
```

The keyword *module* introduces the name of the module. Note that the name of the ATL file containing the code of the module has to correspond to the name of this module. For instance, a ModelA2ModelB transformation module has to be defined into the *ModelA2ModelB.atl* file.

The target models declaration is introduced by the *create* keyword, whereas the source models are introduced either by the keyword *from* (in normal mode) or *refines* (in case of refining transformation). The declaration of a model, either a source input or a target one, must conform the scheme *model_name* : *metamodel_name*. It is possible to declare more than one input or output model by simply separating the declared models by a coma. Note that the name of the declared models will be used to identity them. As a consequence, each declared model name has to be unique within the set of declared models (both input and output ones).

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

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

3.1.1.2. Import section

The optional import section enables to declare which ATL libraries (see Section 3.3) have to be imported. The declaration of an ATL library is achieved as follows:

```
uses extensionless_library_file_name;
```

For instance, to import the *strings* library, one would write:

```
uses strings;
```

Note that it is possible to declare several distinct libraries by using several successive *uses* instructions.

3.1.1.3. Helpers

ATL helpers can be viewed as the ATL equivalent to Java methods. They make it possible to define factorized ATL code that can be called from different points of an ATL transformation.

An ATL helper is defined by the following elements:

- a name (which corresponds to the name of the method);
- a context type. The context type defines the context in which this attribute is defined (in the same way a method is defined in the context of given class in object-programming);
- a return value type. Note that, in ATL, each helper must have a return value;
- an ATL expression that represents the code of the ATL helper;
- an optional set of parameters, in which a parameter is identified by a couple (parameter name, parameter type).

As an example, it is possible to consider a helper that returns the maximum of two integer values: the contextual integer and an additional integer value which is passed as parameter. The declaration of such a helper will look like (detail of the helper code is not interesting at this stage):

```
helper context Integer def : max(x : Integer) : Integer = ...;
```

It is also possible to declare a helper that accepts no parameter. This is, for instance, the case for a helper that just multiplies an integer value by two:

```
helper context Integer def : double() : Integer = self * 2;
```

In some cases, it may be interesting to be able to declare an ATL helper without any particular context. This is not possible in ATL since each helper must be associated with a given context. However, the ATL language allows ATL developers to declare helpers within a default context (which corresponds to the ATL module). This is achieved by simply omitting the *context* part of the helper definition. It is possible, by this mean, to provide a new version of the max helper defined above:

```
helper def : max(x1 : Integer, x2 : Integer) : Integer = ...;
```

Note that several helpers may have the same name in a single transformation. However, helpers with a same name must have distinct signatures to be distinguishable by the ATL engine.

The ATL language also makes it possible to define attributes. An attribute helper is a specific kind of helper that accepts no parameters, and that is defined either in the context of the ATL module or of a model element. In the remaining of the present document, the term *attribute* will be specifically used to refer to attribute helpers, whereas the generic term of *helper* will refer to a functional helper.

Thus, the attribute version of the double helper defined above will be declared as follows:

```
helper context Integer def : double : Integer = self * 2;
```

Declaring a functional helper with no parameter or an attribute may appear to be equivalent. It is therefore equivalent from a functional point of view. However, there exists a significant difference between these two approaches when considering the execution semantics. Indeed, compared to the result of a functional helper which is calculated each time the helper is called, the return value of an ATL attribute is computed only once when the value is required for the first time. As a consequence, declaring an ATL attribute is more efficient than defining an ATL helper that will be executed as many times as it is called.

Note that the ATL attributes that are defined in the context of the ATL module are initialized (during the initialization phase, see Section 3.1.3.1 for further details) in the order they have been declared in the ATL file. This implies that the order of declaration of this kind of attribute is of some importance: an attribute defined in the context of the ATL module has to be declared after the other ATL module attributes it depends on for its initialization. A wrong order in the declaration of the ATL module attributes will raise an error during the initialization phase of the ATL program execution.

3.1.1.4. Rules

In ATL, there exist two different kinds of rules that correspond to the two different programming modes provided by ATL (e.g. declarative and imperative programming): the matched rules (declarative programming) and the called rules (imperative programming).

Matched rules. The matched rules constitute the core of an ATL declarative transformation since they make it possible to specify 1) for which kinds of source elements target elements must be generated, and 2) the way the generated target elements have to be initialized. A matched rule is identified by its name. It matches a given type of source model element, and generates one or more kinds of target model elements. The rule specifies the way generated target model elements must be initialized from each matched source model element.

A matched rule is introduced by the keyword *rule*. It is composed of two mandatory (the source and the target patterns) and two optional (the local variables and the imperative) sections. When defined, the local variable section is introduced by the keyword *using*. It enables to locally declare and initialize a number of local variables (that will only be visible in the scope of the current rule).

The source pattern of a matched rule is defined after the keyword *from*. It enables to specify a model element variable that corresponds to the type of source elements the rule has to match. This type corresponds to an entity of a source metamodel of the transformation. This means that the rule will generate target elements for each source model element that conforms to this matching type. In many cases, the developer will be interested in matching only a subset of the source elements that conform to the matching type. This is simply achieved by specifying an optional condition (expressed as an ATL expression) within the rule source pattern. By this mean, the rule will only generate target elements for the source model elements that both conform to the matching type and verify the specified condition.

The target pattern of a matched rule is introduced by the keyword *to*. It aims to specify the elements to be generated when the source pattern of the rule is matched, and how these generated elements are initialized.

Thus, the target pattern of a matched rule specifies a distinct target pattern element for each target model element the rule has to generate when its source pattern is matched. A target pattern element corresponds to a model element variable declaration associated with its corresponding set of initialization bindings. This model element variable declaration has to correspond to an entity of the target metamodels of the transformation.

Finally, the optional imperative section, introduced by the keyword *do*, makes it possible to specify some imperative code that will be executed after the initialization of the target elements generated by the rule.

As an example, consider the following simple ATL matched rule:

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

This rule, called Author, aims to transform Author source model elements (from the MMAuthor source model) to Person target model elements in the MMPerson target model. This rule only contains the mandatory source and target patterns. The source pattern defines no filter, which means that all Author classes of the source MMAuthor model will be matched by the rule. The rule target pattern contains a single simple target pattern element (called *p*). This target pattern element aims to allocate a Person class of the MMPerson target model for each source model element matched by the source pattern. The features of the generated model element are initialized with the corresponding features of the matched source model element.

Note that a source model element of an ATL transformation should not be matched by more than one ATL matched rule. This implies the source pattern of matched rules to be designed carefully in order to respect this constraint. Moreover, an ATL matched rule can not generate ATL primitive type values.

Called rules. The called rules provide ATL developers with convenient imperative programming facilities. Called rules can be seen as a particular type of helpers: they have to be explicitly called to be executed and they can accept parameters. However, as opposed to helpers, called rules can generate target model elements as matched rules do. A called rule has to be called from an imperative code section, either from a match rule or another called rule.

As a matched rule, a called rule is introduced by the keyword *rule*. As matched rules, called rules may include an optional local variables section. However, since it does not have to match source model elements, a called rule does not include a source pattern. Moreover, its target pattern, which makes it possible to generate target model elements, is also optional. Note that, since the called rule does not match any source model element, the initialization of the target model elements that are generated by the target pattern has to be based on a combination of local variables, parameters and module attributes. The target pattern of a called rule is defined in the same way the target pattern of a matched rule is. It is also introduced by the keyword *to*.

A called rule can also have an imperative section, which is similar to the ones that can be defined within matched rules. Note that this imperative code section is not mandatory: it is possible to specify a called rule that only contains either a target pattern section or an imperative code section.

In order to illustrate the called rule structure, consider the following simple example:

```
rule NewPerson (na: String, s_na: String) {
	to
		p : MMPerson!Person (
			name <- na
		)
	do {
		p.surname <- s_na
	}
}
```

This called rule, named NewPerson, aims to generate Person target model elements. The rule accepts two parameters that correspond to the name and the surname of the Person model element that will be created by the rule execution. The rule has both a target pattern (called *p*) and an imperative code section. The target pattern allocates a Person class each time the rule is called, and initializes the *name* attribute of the allocated model element. The imperative code section is executed after the initialization of the allocated element (see Section 3.1.3.1 for further details on execution semantics). In this example, the imperative code sets the *surname* attribute of the generated Person model element to the value of the parameter *s_na*.

3.1.2. Module execution modes

The ATL execution engine defines two different execution modes for ATL modules. With the default execution mode, the ATL developer has to explicitly specify the way target model elements must be generated from source model elements.

In this scope, the design of a transformation which aims to copy its source model with only a few modifications may prove to be very tiresome. Designing this transformation in default execution mode therefore requires the developer to specify the rules that will generate the modified model elements, but also all the rules that will only copy, without any modification, source to target model elements. The refining execution mode has been designed for this kind of situation: it enables ATL developers to only specify the modifications that have to be performed between the transformation source and target models.

These two execution modes are described in the following subsections.

3.1.2.1. Normal execution mode

The normal execution mode is the ATL module default execution mode. It is associated with the keyword *from* in the module header (see Section 3.1.1.1).

In default execution mode, the ATL developer has to specify, either by matched or called rules, the way to generate each of the expected target model elements. This execution mode suits to most ATL transformations where target models differ from the source ones.

3.1.2.2. Refining execution mode

The refining execution mode has been introduced to ease the programming of refining transformations between similar source and target models. With the refining mode, ATL developers can focus on the ATL code dedicated to the generation of modified target elements. Other model elements (e.g. those that remain unchanged between the source and the target model) are implicitly copied from the source to the target model by the ATL engine.

The refining mode is associated with the keyword *refines* in then header of the ATL module (see Section 3.1.1.1). Granularity of the refining mode is defined at the model element level. This means that the developer will have to specify how to generate a model element as soon as the transformation modifies one of its features (either an attribute or a reference). On the other side, the developer is not required to specify the ATL code that corresponds to the copy of unchanged model elements. This feature may result in important saving of ATL code, which, in the end, makes the programming of refining ATL transformations simpler and easier.

At current time, the refining mode can only be used to transform a single source model into a single target model. Both source and target models must conform to the same metamodel.

Note that, due to current execution semantics of the refining mode (see Section 3.1.3), some specific precautions still have to be taken by developers. Indeed, with current implementation of the ATL engine, to be transformed into a target model element, a source model element has to match one of the following conditions:

- being transformed by a rule explicitly specified by the developer;
- being referred (directly or indirectly) from a transformed source model element.

This means that a source model element will not be copied into its corresponding target model element if:

- no target model element is generated by the explicated transformation rules;

- no explicitly transformed source model element refers, directly or indirectly, this source model element.

As a consequence, it may be useful, when designing an ATL module in refining mode, to specify additional explicit rules in order to make sure that all source model elements are transformed into their corresponding target model elements.

This trap is illustrated by the following example. Consider the SimpleMetamodel metamodel presented in Figure 6: it is composed of a model element A and a model element B. Model element A has a single feature which is a one-to-many reference to element B. Model element B has two features: an attribute called *attributeB* and a zero-to-one reference to element A.

A developer may want to refine a model conforming to the SimpleMetamodel by simply modifying the content of the feature *attributeB* of the model element B.

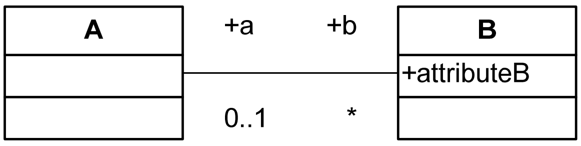

Figure 6. The SimpleMetamodel metamodel

For this purpose, an ATL transformation in refining mode, composed of the following single matched rule may appear to be sufficient:

```
rule B {
	from
		in : SimpleMetamodel!B
	to
		out : SimpleMetamodel!B (
			attributeB <- ...
		)
}
```

As a result, such a transformation will produce a target model only composed of the refined B model elements. The model elements A will not be copied by the transformation since they are neither matched by any explicitly specified transformation rule, nor referred to by the explicitly transformed source model elements.

An approach for correcting this unexpected result may be to initialize the reference *a* of the matched B model elements, so that the pointed A model elements will be transformed. This approach is however not sufficient since the reference *a* has a zero-to-one multiplicity: there may therefore exist some A model elements that will not be pointed by any B model element, and as so, will not be implicitly transformed.

As a consequence, the explicit transformation of the model elements A is here required in order to have the same model elements in the source and the target models. This is achieved by the following couple of matched rules:

```
rule A {
	from
		in : SimpleMetamodel!A
	to
		out : SimpleMetamodel!A (
			b <- in.b
		)
}

rule B {
	from
		in : SimpleMetamodel!B
	to
		out : SimpleMetamodel!B (
			attributeB <- ...,
			a <- in.a
```

```
                    )
}
```

3.1.3. Module execution semantics

This section introduces the basics of the ATL execution semantics. Although designing ATL transformations does not require any particular knowledge on the ATL execution semantics, understanding the way an ATL transformation is processed by the ATL engine can prove to be helpful in certain cases (in particular, when debugging a transformation).

The semantics of the two available ATL execution modes, the normal and the refining modes, are introduced in the following subsections.

3.1.3.1. Default mode execution semantics

The execution of an ATL module is organized into three successive phases: a module initialization phase, a matching phase of the source model elements, and a target model elements initialization phase.

The module initialization step corresponds to the first phase of the execution of an ATL module. In this phase, the attributes defined in the context of the transformation module are initialized. Note that the initialization of these module attributes may make use of attributes that are defined in the context of source model elements. This implies these new attributes to be also initialized during the module initialization phase. If an entry point called rule has been defined in the scope of the ATL module, the code of this rule (including target model elements generation) is executed after the initialization of the ATL module attributes.

During the source model elements matching phase, the matching condition of the declared matched rules are tested with the model elements of the module source models. When the matching condition of a matched rule is fulfilled, the ATL engine allocates the set of target model elements that correspond to the target pattern elements declared in the rule. Note that, at this stage, the target model elements are simply allocated: they are initialized during the target model elements initialization phase.

The last phase of the execution of an ATL module corresponds to the initialization of the target model elements that have been generated during the previous step. At this stage, each allocated target model element is initialized by executing the code of the bindings that are associated with the target pattern element the element comes from. Note that this phase allows invocations of the *resolveTemp()* operation that is defined in the context of the ATL module.

The imperative code section that can be specified in the scope of a matched rule is executed once the rule initialization step has completed. This imperative code can trigger the execution of some of the called rules that have been defined in the scope of the ATL module.

3.1.3.2. Refining mode execution semantics

The refining execution mode introduces specific semantics for the implicit generation of copied model elements.

An ATL module executed in refining mode follows the three successive phases of the default execution mode. The execution of the first phase, the module initialization phase, remains unchanged compared to the default execution mode. During the source model elements matching phase, the ATL engine only evaluates the matching conditions of the explicitly specified matched rules. This implies that, at this stage, the only target model elements that are allocated are those that are generated by these explicit transformations rules.

The differences with the default execution mode appear during the execution of the initialization phase of the target model elements. In refining mode, this phase has to deal with the initialization of the explicitly generated target model elements, but also with the allocation and the initialization of the target model elements that are implicitly generated.

For this purpose, each time an already allocated target model element is initialized with a reference to a non-allocated model element, the ATL engine allocates and initializes this new target model element. If the newly created model element also refers to another non-allocated model element, this process is repeated recursively.

Note that with the described semantics, no target model element will be generated for a source model element that is neither matched by an explicit rule, nor referred, directly or indirectly, by an explicitly generated target model element.

3.2. ATL Query

An ATL query consists in a model to primitive type value transformation. An ATL query can be viewed as an operation that computes a primitive value from a set of source models. The most common use of ATL queries is the generation of a textual output (encoded into a string value) from a set of source models. However, ATL queries are not limited to the computation of string values and can also return a numerical or a boolean value.

The following subsections respectively describe the structure and the execution semantics of an ATL query.

3.2.1. Structure of an ATL query

After an optional import section (see Section 3.1.1.2), an ATL query must define a query instantiation. A query instantiation is introduced by the keyword *query* and specifies the way its result must be computed by means of an ATL expression:

```
query query_name = exp;
```

Beside the query instantiation, an ATL query may include a number of helper or attribute definitions. Note that, although an ATL query is not strictly a module, it defines its own kind of default module context. It is therefore possible, for ATL developers, to declare helpers and attributes defined in the context of the module in the scope of an ATL query.

3.2.2. Query execution semantics

As an ATL module, the execution of an ATL query is organized in several successive phases. The first phase is the initialization phase. It corresponds to the initialization phase of the ATL modules (see Section 3.1.3.1) and is dedicated to the initialization of the attributes that are defined in the context of the ATL module.

The second phase of the execution of an ATL query is the computation phase. During this phase, the return value of the query is calculated by executing the declarative code of the *query* element of the ATL query. Note that the helpers that have been defined within the query file can be called at both the initialization and the computation phases.

3.3. ATL Library

The last type of ATL unit is the ATL library. Developing an ATL library enables to define a set of ATL helpers that can be called from different ATL units (modules, but also queries and libraries).

As the other kinds of ATL units, an ATL library can include an optional import section (see Section 3.1.1.2). Besides this import section, an ATL library defines a number of ATL helpers that will be made available in the ATL units that will import the library.

Compared to an ATL module, there exists no default module element for ATL libraries. As a consequence, it is impossible, in libraries, to declare helpers that are defined in the default context of the module. This means that all the helpers defined within an ATL library must be explicitly associated with a given context.

Compared to both modules and queries, an ATL library cannot be executed independently. This currently means that a library is not associated with any initialization step at execution time (as described in Section 3.1.3). Due to this lack of initialization step, attribute helpers cannot be defined within an ATL library.

4. ATL Inventory

4.1. Documentation

4.1.1. Reference manuals

Several reference manuals can be found on the GMT/ATL web site [2][10]. If you need a short overview of ATL, you can read the ATL Presentation Sheet [27]. A user manual [29], starter guide [28] and installation guide [31] provide a complete documentation for ATL.

4.1.1.1. User Manual

The ATL User Manual [29] aims at providing both an exhaustive reference of the ATL transformation language and a comprehensive guide for the users of the ATL IDE. For this purpose, this manual is organized in three main parts: the first part (Section 2 and Section 3) introduces the main concepts of model transformation and provides an overview of the structure and the semantics of the ATL language. The second part (corresponding to Section 4) focuses on the description of the ATL language while the last part (Section 5) deals with the use of the ATL tools.

The detailed structure of the document looks as follows:

- Section 2 provides a short introduction to the model transformation area;
- Section 3 offers an overview of the ATL capabilities;
- Section 4 is dedicated to the description of the ATL language;
- Section 5 describes the IDE that has been developed around the ATL transformation language;
- Section 6 provides ATL programmers with a number of pointers to available ATL resources;
- Finally, Section 7 concludes the document.

4.1.1.2. Starter Guide

This document [28] 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) has been previously successfully installed. For this purpose, please refer to the ATL Installation Guide [31]. However, this document 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.

4.1.1.3. Installation Guide

This document [31] aims to provide a step-by-step guide for installing the Eclipse/EMF distribution of the ATL Development Tools. Section 2 deals with the installation of ADT from binaries available from the download section in GMT/ATL project [4]. Section 3 details ADT installation from its source code (available on the Eclipse CVS repository). Finally, Section 4 provides some links to already available ATL resources, as well as references to the ATL mailing list.

4.1.1.4. Installation from source

This document [32] explains how to install ATL Toolkit from source. This operation is also detailed in the installation guide. We advise you firstly this installation guide.

4.1.1.5. Specification of the ATL Virtual Machine

This document presents the ATL virtual machine [30]. The ATL VM is an abstract computing machine. As the Java Virtual Machine, it is associated with its own particular instruction set. The ATL virtual machine is independent from the ATL transformation language. It only has to deal with the file format in which programs to be executed are compiled to. An XML based file format, the `asm` file format, has been designed for the ATL virtual machine. An `asm` file contains ATL virtual machine instructions (also called *byte codes*).

This document is organized as follows:

- Section 2 provides an overview of the ATL transformation language. It introduces the concepts and the terminology necessary for the understanding of the rest of the document.
- Section 3 gives an overview of the ATL virtual machine architecture.
- Section 4 provides a specification of the instruction set of the ATL virtual machine.
- Section 5 specifies the present asm file format, the XML format used to represent compiled ATL programs.
- Section 6 introduces compilation of code written in the ATL transformation language into the instruction set of the ATL virtual machine.
- Finally, Section 7 introduces a set of possible evolutions for both the ATL virtual machine and the asm format.

4.1.2. ATL References

4.1.2.1. ATL Publications

Several publications concerning ATL have been written by the ATLAS Group, the following list presents the main publications on this topic:

Jouault, F, and Kurtev, I : Transforming Models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica. 2005. (link)

Jouault, F, and Kurtev, I : On the Architectural Alignment of ATL and QVT. In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, chapter Model transformation (MT 2006), pages 1188—1195. 2006. (link)

Kurtev, I, van den Berg, K, and Jouault, F : Rule-based Modularization in Model Transformation Languages illustrated with ATL. In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, chapter Model transformation (MT 2006), pages 1202—1209. 2006. (link)

Bézivin, J, and Jouault, F : Using ATL for Checking Models. In: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia. 2005. (link)

Bézivin, J, Jouault, F, and Valduriez, P : An Eclipse-based IDE for the ATL model transformation language. Research Report LINA, (04.08). 2004. (link)

Di Ruscio, D, Jouault, F, Kurtev, I, Bézivin, J, and Pierantonio, A : Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Research Report LINA, (06.02). 2006. (link)

A complete list of ATL publications can be found on GMT/ATL web site in the “publications section” [6].

The full list of publications made by the ATLAS Group is available on the ATL site [3].

4.1.2.2. ATL External References

The ATL community is growing, there are more and more users. This implies also a growing number of presentations, papers, use cases, etc. that refer to ATL. We have started to take inventory of these references. There are accessible on the dedicated section on GMT/ATL [11] and ATLAS/ATL [12] web sites.

4.1.2.3. AMMA/ATL Workshop on Model Engineering

Since 2005, the ATLAS Group has organized at University of Nantes a workshop called AWME (AMMA/ATL Workshop on Model Engineering). This workshop is an occasion for some of the users of AMMA/ATL to meet and to exchange ideas. This meeting is also an opportunity to present some of the planned extensions to these tools. Finally it is an occasion to express suggestions and wishes for future directions of the AMMA Platform.

Presentations made during the first AWME [20] and the second AWME [21] are reachable online.

4.1.3. ATL Wiki

A wiki is “a piece of server software that allows users to freely create and edit Web page content using any Web browser” [44].

The ATL documentation on the Eclipse Wiki provides a dynamic documentation where contributors can participate. Several topics are already handled:

4.1.3.1. Standard Library

The ATL standard library [34] consists of the types built into ATL and of their operations. It is based on the OCL 2.0 standard library as described in chapter 11 of the UML 2.0 OCL Final Adopted specification [45]. However, some extensions have been made.

4.1.3.2. Troubleshooter

An ATL troubleshooter [35] has been created to locate and fix sources of trouble. Various topics are already dealed, for example: ATL Virtual Machine Troubles, ATL Called Rules, and UML2 Profiles.

4.1.3.3. FAQ

It is the official ATL FAQ [36] but for the moment, the entries from the old ATL FAQ [22] have not been copied here yet.

The following list gives an overview of these FAQ:

- How does ATL deal with primitive types?
- How do I convert an Ecore metamodel to KM3?
- Is it possible to have an EMF based input model and an MDR based output model?
- How close is ATL navigation language from the OCL 2.0 standard?
- Metamodels declared in the header (after the `create` keyword) of transformation models have to be named. What name should be used?

4.1.3.4. How to

This page contains a list of problems related to ATL usage. It gives hints towards solutions and links to examples.

The following list gives a glimpse of dealing topics:

- How do I generate text from models?
- How do I launch transformations programmatically?
- How do I use extern parameters in ATL transformation?

4.1.3.5. AM3 Ant Task

This page describes the ant tasks [46] provided by AM3. AM3 Ant Tasks [38] provide tasks to load and save models as well as run ATL transformations.

am3.loadModel is used to load a model. This model may be a terminal model or a metamodel. The metametamodels are typically not loaded with this task since they come bundled with a model handler.

am3.saveModel is used to save a model.

am3.atl executes an ATL transformation. The models used by a transformation are referenced by their IDs as defined at their loading time (see AM3_Ant_Tasks#am3.loadModel).

There is an exception for metametamodels, which are typically already available from the model handler. A metametamodel is referenced by a string composed of a percent character (i.e. '%') followed by the name of the model handler. For instance: '%EMF' refers to Ecore and '%MDR' refers to MOF 1.4.

Other specific tasks are also available.

4.1.3.6. Wish List

This page [39] contains a list of requested features for the ATL language and execution engine. They are sorted in three lists depending on their status: unimplemented, being currently implemented, and being currently tested. This is a place where you may add your own requests.

4.2. ATL Examples

The ATL examples are published on the GMT/ATL web site in the ATL Transformations Zoo section [7].

4.2.1. The ATL Transformation Zoo

The GMT project [1] hosts an initial experiment for building a collection of model transformations. This collection of transformations already offers an interesting landscape of application domains of the model transformation. It therefore represents an interesting starting point for studying what should be the content of transformation libraries and identifying some of the issues large scale transformation libraries will have to face. This section offers an overview of the current content of the collection of ATL transformation examples. For this purpose, it provides an overview of the list of transformations that are currently available on the GMT web site. This short description scheme, which includes the short name, full name and the short description of the transformation, will be presented as in the following pattern:

Short Name. *Full Name*.
Short Description.

Ant2Maven. *From Ant to Maven*.
This transformation aims to generate a Maven project from a simplified Ant model.

ATL2BindingDebugger. *Refining ATL with bindings debugging instructions*.
The ATL2BindingDebugger transformation adds a debug instruction to each binding in an ATL transformation. Source and target models conform to ATL metamodel.

ATL2BindingDebugger. *Refining ATL with bindings debugging instructions*.
The ATL2BindingDebugger transformation adds a debug instruction to each binding in an ATL transformation. Source and target models conform to ATL metamodel.

ATL2Problem. *From ATL to Problem*.
This transformation generates a Problem model that contains the list of non-structural errors (along with additional warnings) that have been detected within the source ATL model. The transformation assumes the input ATL model is structurally correct, as those that have passed a syntactic analysis (for instance, a reference defined with cardinality [1-1] should not be undefined).

ATL2Tracer. *Refining ATL with traceability generation instruction*.
This transformation generates enriches the source ATL transformation with a number of instructions dedicated to the generation of a target traceability model. Source and target models conform to ATL metamodel.

BibTeXML2DocBook. *From BibTeXML to DocBook*.
The BibTeXML to DocBook transformation generates a structured DocBook document presenting the bilbliographic entries contained by the source BibTeXML model.

Book2Publication. *From simple Book to simple Publication*.
The Book to Publication transformation generates a Publication model from a Book model composed of an ordered set of chapters. The number of pages of the generated Publication corresponds to the sum of all pages of the chapters of the source Book model.

Class2Relational. *From Class to Relational*.
This simplified transformation generates a Relational model from a Class model.

DSL2EMF. *From Domain Specific Languages to Eclipse Modelling Framework*.
This transformation aims to generate an EMF metamodel from a simplified DSL metamodel. From the source DSL metamodel, the transformation generates a KM3 metamodel from which an EMF metamodel can be obtained by means of the KM32EMF transformation.

EMF2DSL. *From Eclipse Modelling Framework to Domain Specific Languages*.
This transformation aims to generate a DSL metamodel from a source EMF metamodel. First step consist in obtaining a KM3 metamodel corresponding to the source EMF metamodel with the EMF2KM3 transformation. The transformation can then generate a DSL metamodel from the obtained KM3 metamodel.

EMF2KM3. *From Eclipse Modelling Framework to Kernel MetaMetaModel*.
This transformation generates a KM3 model from an Ecore model.

GeometricalTransformations. *From Geometrical data and Geometrical transformation to Geometrical Data*.
This transformation performs geometrical transformations. It generates a DXF-based Geometrical Data model from the couple of source models encoding DXF-based Geometrical Data and the Geometrical Transformations (rotate, move, explode) to be applied to these data.

Grafcet2PetriNet. *From Grafcet to Petri Net*.

This transformation generates a Petri Net model from a Grafcet model.

JavaSource2Table. *From Java Source to Table*.
This transformation aims to generate a Table, which summarizes how many times each method declared in the Java source code is called within the definition of any declared method, from a simplified Java source code model. The considered source JavaSource model focuses on method declarations and calls.

KM32ATLCopier. *From Kernel MetaMetaModel to ATL copier of KM3 models*.
The KM32ATLCopier transformation creates an ATL transformation dedicated to the copy of a KM3 model (e.g. that generates a copy of its source KM3 model). Source model conforms to KM3 metamodel and target model conforms to ATL metamodel.

KM32DOT. *From Kernel MetaMetaModel to DOT*.
The KM3 to DOT transformation generates a DOT graphical class diagram from a KM3 metamodel description. KM3 is a textual concrete syntax to describe metamodels.

KM32EMF. *From Kernel MetaMetaModel to Eclipse Modelling Framework*.
This transformation generates an Ecore model from a source KM3 model.

KM32Metrics. *From Kernel MetaMetaModel to Metrics*.
This transformation generates a Metrics model from a source KM3 metamodel. The generated Metrics model includes both very basic metrics (such as number of classes, attributes, etc.) and more complex ones (such as the number of inheritance graphs, of inheritance trees).

KM32Problem. *From Kernel MetaMetaModel to Problem*.
This transformation generates a Problem model from a source KM3 metamodel The generated Problem model contains the list of non-structural errors (along with additional warnings) that have been detected within the input KM3 metamodel. The transformation assumes the input KM3 metamodel is structurally correct, as those that have passed a syntactic analysis (for instance, a reference defined with cardinality [1-1] should not be undefined). It may therefore fails when executed on a KM3 metamodel produced from a MOF metamodel that has not been checked.

Make2Ant. *From Make to Ant*.
This transformation aims to generate an Ant project from simplified Make model.

Maven2Ant. *From Maven to Ant*.
This transformation aims to generate an Ant Maven project from a simplified Maven model.

MOF2UML. *From MOF to UML class diagram*.
This transformation aims to generate a UML class diagram from a source MOF model. The transformation is based on the UML Profile for MOF OMG specification. The transformation leads to some loss of information since some properties of the source MOF classes have no UML equivalent.

Monitor2Semaphore. *From Hoares Monitor to Disjkstras Semaphore*.
The Monitor to Semaphore transformation aims to produce a program a program containing Hoares monitors definitions into an equivalent program in which synchronization operations are based on Dijkstras semaphores. The transformation strictly follows the operational semantic of monitors described in terms of semaphores.

MSExcelExtractor. *From Microsoft Office Excel to text*.
This transformation aims to generate a Microsoft Office Excel model, composed of a single workbook, from an XML Excel file that conforms to a simplified subset of the SpreadsheetML XML dialect (which is the one used by Microsoft to import/export Excel workbook's data in XML since the 2003 version of Microsoft Office).

MSExcelInjector. *From text to Microsoft Office Excel*.
This transformation aims to generate an XML Excel file (conforming to a simplified subset of the SpreadsheetML XML dialect) composed of a single workbook from the source Microsoft Office model.

MSExcel2SoftwareQualityControl. *From Microsoft Office Excel to Software Quality Control*.
This transformation aims to generate a Software Quality Control model, which provides a simple structure for managing software quality controls (such as bug tracking) from a Microsoft Office Excel workbook that contains information about quality controls of a software product.

MySQL2KM3. *From MySQL to Kernel MetaMetaModel*.
The MySQL to KM3 transformation generates a KM3 metamodel from a relational database model corresponding to a MySQL database.

PathExpression2PetriNet. *From Path Expression to Petri Net*.
This transformation generates a Path Expression from a simple source Petri Net.

PetriNet2Grafcet. *From Petri Net to Grafcet*.
This transformation generates a Petri Net from a source Grafcet.

PetriNet2PathExpression. *From Petri Net to Path Expression*.
This transformation generates a Petri Net from a source Path Expression.

Public2Private. *From UML public attributes to UML private attributes*.
This transformation aims to make all public attributes of a UML 1.4 model private. Getters and setters are also created appropriately (i.e. no setter for a frozen attribute).

Software Quality Control to Bugzilla. *From Software Quality Control to Buzilla*.
This transformation aims to generate a simple Bugzilla model (Bugzilla being a free bug tracking system developed by the Mozilla Foundation) from a SoftwareQualityControl model.

Software Quality Control to MantisBugTracker. *From Software Quality Control to Mantis Bug tracker*.
This transformation aims to generate a simple Mantis Bug Tracker model (Mantis Bug Tracker being a free bug tracking system based on PHP and MySQL) from a SoftwareQualityControl model.

Table2MSExcel. *From Table to Microsoft Office Excel*.
This transformation generates a Microsoft Office Excel workbook containing a representation of the source Table model. The considered Microsoft Office Excel metamodel is based on a simplified subset of the SpreadsheetML XML dialect which is the one used by Microsoft to import/export Excel workbooks data in XML since the 2003 version of Microsoft Office.

UML2Amble. *From UML to Amble*.
This transformation generates a distributed Amble program from a specific UML model that specifies the different aspects of a distributed program. Local variables and methods of the concurrent processes managed by an Amble program must be specified as an UML class diagram in which each class is associated with a given process, and the transitions between classes represent the existing communication channels. Each process is also described by a dedicated UML state machine in which the transitions correspond to the set of Amble transitions ("condition", "receipt", "receipt_any").

UML2Java. *From UML Class Diagram to Java*.
This transformation aims to generate a simplified Java model (mainly dealing with the package reference, the attributes and the methods of the classes) from a source UML class diagram.

UML2MOF. *From UML Class Diagram to MOF*.
This transformation aims to generate a MOF model from a UML class diagram.

UMLActivityDiagram2MSProject. *From UML Activity Diagram to Microsoft Office Project*.
The UML2MSProject transformation generates a MS Project from a loop free UML activity diagram (describing some tasks series). The transformation is based on a simplified subset of the UML State Machine metamodel. This transformation produces a project defined in conformance to a limited subset of XML format loaded by MS Project.

UMLDI2SVG. *From UML Diagram Interchange to Scalable Vector Graphics*.
This transformation aims to generate a SVG model from a source UML Diagram Interchange.

UMLModelCopy. *UML Model Copy*.
This transformation aims to copy the source UML model into a similar UML model.

UMLModelMerge. *Merging of two UML models*.
This transformation aims to merge a "merge" UML model with the main source UML model. The UML model produced by the transformation contains the result of this merging. The merging is limited to copying elements that can occur in UML Class Diagrams (including Action Semantics).

UMLAssociationAttributes. *UML Association attributes*.

This transformation aims to introduce Attributes (with Java initial values) for each Association End.

UMLAccessors. *UML Accessors*.
This transformation introduces accessor Operations/Methods (with Java bodies) for each public Attribute. The public Attributes are then made private. This transformation is also used in refactoring as "Encapsulate Field".

UMLObservers. *UML Observers*.
This transformation aims to introduce the Java infrastructure to implement the "Observer", "Observable" and "subscribe" Stereotypes. Any instances of "Observer" Classes that have a "subscribe" Association to an "Observable" Class, will automatically be subscribed to the "Observable" instance that they are linked to. In addition, all setter Methods of the "Observable" Class will be updated to do a notify as well. An "update" Operation/Method will be introduced for each "Observer" Class, which is configured to invoke any "onXYZChange" Operations, where "XYZ" is the name of the Attribute that is changed.

UMLSingleton. *UML Singleton class refinement*.
This simple transformation adds a static "getInstance" Operation/Method and a static "instance" Attribute to each Class with a "Singleton" Stereotype.

UMLApplet. *UML Applet*.
This transformation aims to introduce a Java Applet infrastructure for each Class with an "Applet" stereotype.

UMLDataTypesSingleton. *UML data types*.
This transformation aims to replace the OCL types with the corresponding Java types.

UMLAsyncMethods. *UML asynchronous methods refinement*.
This simple transformation aims to wrap the Method body of each Operation with an "asynchronous" Stereotype inside a Thread.

XSLT2XQuery. *From XSLT to XQuery*.
This transformation aims to generate an XQuery from a source XSLT model. Source XSLT models conform to a subset of the XSLT specification.

4.2.2. A Classification of the ATL Transformations Zoo

This section presents a proposal for a classification of the model transformations contained in the ATL transformation zoo.

4.2.2.1. Semantic Bridges

A number of the already available transformations provide semantic bridges between metamodels. Some of these transformations deal with metamodels that handle very close concepts. For instance, in the model engineering area, this is the case of the UML2MOF, DSL2EMF and EMF2KM3 transformations. This is also the purpose of the Monitor2Semaphore transformation which, in the field of program synchronization; enables to move from a monitor-based synchronization to a semaphore-based one.

Some other transformations define semantic bridges between different technical spaces. This is the case of the two transformations Class2Relational and SimpleClass2SimpleRDBMS that enable to move from an object-oriented model to a relational database representation. Another kind of semantic bridge that is available in the transformation repository makes it possible to produce programs from a model. This is the purpose of both the UML2Java and UML2Amble transformations, which respectively enables to get in a Java and an Amble program from an UML model. Note that there exist other examples of semantic bridges, such as the Grafcet2PetriNet or the PathExpression2PetriNet transformation.

Preliminary observations among the community of ATL developers have shown an interesting level of reusability for many of the semantic bridge transformations. This is particularly true for semantic bridges defined in the model engineering area and that enable to move from a modeling technology to another (UML, MOF, DSL, etc.).

4.2.2.2. *Bridges to Graphical Representations*

Models have been represented graphically for long times. These graphical representations are more and more considered as alternative user-friendly representations of models encoded by means of hardly readable formats (such as XMI). The ability to move from textual models to graphical representations therefore represents a significant issue in the field of model engineering. The ATL transformation repository currently contains two transformations that provide bridges to graphical representations: KM32DOT and UMLDI2SVG.

KM32DOT enables to produce a DOT graphical representation from a KM3 metamodel. Note that the existing bridges between the KM3 notation and the main model handler technologies (EMF, MDR) make it possible to get a DOT representation of a model that is encoded using any of the supported modeling technologies. This could simply be achieved by chaining the appropriate semantic bridge transformation (for instance, EMF2KM3) with the KM32DOT transformation. The UMLDI2SVG transformation enables to produce a SVG graphical representation of an UML model that includes diagram interchange information.

Although they must be further developed, transformations defining bridges to graphical representations have a strong reusability potential. It may be noted here that the increasing number of available bridges between modeling technologies makes it possible to produce a graphical representation of a model without requiring the development of a specific transformation between the considered modeling technology and the graphical format.

4.2.2.3. *From Tool to Tool Transformations*

The ATL transformation repository contains a number of tool to tool transformations. These transformations aim to move from the source tool’s semantics to the semantics of a target tool. Compared to semantic bridge transformations, tool to tool transformations mainly deal with tool specific formats. Tool to tool transformations have been developed in many different domains. Thus, three transformations have been developed in the field of build tools. These transformations define some bridges between the formats of the Ant, Maven and Make build tools. Bug tracking is also well represented with three transformations too. Bridges are provided between Software Quality Control and both Bugzilla and Mantis Bug Tracker. An additional transformation enables to generate a Software Quality Control project from an Excel tab.

Other tool to tool transformations include a bridge between the MySQL database system and the KM3 notation (although this one may also be considered as a semantic bridge transformation), a transformation between a simple table abstract format and the MSExcel semantics, and a transformation from UML activity diagrams to an MSProject project.

Reusability potential of tool to tool transformations, compared to the one of semantic bridges, may appear limited. This is due to the fact that these transformations handle tool specific formats, which usually include their own internal concepts and data types, whereas the semantic bridges mainly deal with accepted norms or de facto standards which are often supported by different tools. As a consequence, tool to tool transformations will mainly be reused by developers that target the same source and target tools.

4.2.2.4. *Calculation Transformations*

Calculation transformations include all the transformations that either perform measures or apply operations to their source model. The transformation repository currently contains different calculation transformations. As ATL2Problem and KM32Problem, some of them have been developed to be part of the ATL development environment. These two transformations aim to produce a diagnostic (encoded as a Problem metamodel) from respectively an ATL and a KM3 model. In the scope of the ATL tools, they are used for the analysis of developed ATL transformations and KM3 metamodels. By this mean, it is possible to provide developers with some useful information on the errors/warnings contained by the edited file (either an ATL transformation or a KM3 metamodel).

There exist other calculation transformations. Thus, the KM32Metrics transformation has been developed to compute a number of metrics (such as the number of classes, of associations, etc.) over a source KM3 model. Similarly, the JavaSource2Table transformation builds a table that provides a summary of the number of method calls within each declared method of a source Java program. Finally, the transformation named Geometrical Transformations enables to apply a number of geometrical transformations (rotations, etc.) onto a source 3D geometrical model.

Experience has shown that, compared to the previous transformation families, the reusability, in different contexts, of many of the calculation transformations is quite limited. As part of an explanation, it must be noted that a number of the target models produced by these transformations (such as Problem and Metrics models) correspond to final results of computations that cannot be reused as source of a large number of transformations. However, when correctly designed, the measurements metamodels these transformations rely on exhibit a very interesting reusability potential. As an example, the Problem metamodel is already used to encode problems that are detected in both ATL and KM3 models, and is generic enough to represent problems of any other model.

Another point regarding the Problem and the Metric metamodels is that the different elements (Metrics and Problem) defined by these metamodels are independent the ones from the others. This means that a Problem element from a Problem model is not linked to the other Problem elements of the model. As a consequence, developers that are interested in defining their own versions of the transformations generating such models (ATL2Problem, KM32Metrics, etc.) can easily modify the transformations available in the repository without having to pay specific attention to the associations between the generated model elements.

4.2.2.5. Miscellaneous Transformations

There exist a number of miscellaneous transformations that do not fit the previously described groups. Among these remaining transformations, it is possible to identify a number of simple didactic transformations (Book2Publication, Public2Private, etc.) that aim to illustrate some particular features of the ATL language. As an example, the transformation Public2Private provides an example of the use of the ATL refining transformation mode.

Miscellaneous transformations also include a variety of other transformations. Among these lasts, it is possible to identify a number of transformations that aim to achieve some refinements of their source model. Thus, whereas ATL2Tracer and ATL2BindingDebugger define refinements of an ATL model, UMLAccessors, UMLSingleton, etc. provide refinements for UML models. The reusability level of the miscellaneous transformations highly varies according to the considered transformation.

4.3. How to find some helps and to contribute to the ATL community

4.3.1. ATL Mailing List

The ATL discussions occur on the ATL mailing list [8]. If you want discuss, ask and/or answer questions concerning ATL, you can subscribe to this list by using the following link: http://groups.yahoo.com/subscribe/atl_discussion. Lot of questions has been already posted (nearly 2000). Maybe your question has been already answered; a search engine is available to check this.

4.3.2. Bug reporting / Patch providing

If you have a problem/bug you can also take a look to bugzilla to know if someone else has already identified it. If not, you should not hesitate to contribute and report it. To check this, please use this link: https://bugs.eclipse.org/bugs/buglist.cgi?bug_status=NEW&bug_status=ASSIGNED&bug_status=REOPENED&product=GMT&component=ATL&order=Reuse+same+sort+as+last+time.

If you fix a bug, you can also provide a patch by using Bugzilla. It will be integrated as soon as possible in the ATL source code.

4.3.3. Provide ATL examples

The ATL transformations Zoo [7] is open to contributors, like ATL and AMMA platform in general. All external contributions are welcome.

There are two possibilities to publish your scenario(s):

- Your scenario(s) will be hosted on the Eclipse server
- Your scenario(s) will be hosted on an external server

In the documentation or in the comments of the transformation itself, you may mention your identity and company by mail, URL, etc. In this way you may get back some advice from interested people. More details are available here [54].

4.4. Download

4.4.1. GMT

Release builds of the ATL Development Toolkit (ADT) are available on the GMT/ATL web site [4]. The ATL sources are published on the Eclipse server [9]. If you need more details to install ADT, please refer to the ATL Installation Guide [31].

4.4.2. ATL Bundle

An ATL bundle [13] has been created. It contains a ready-to-use bundle of ATL Development Tools (including Eclipse, EMF, and ADT). It also includes ATL starter guide, user manual, and installation guide. The installation guide explains how to install the ATL development tools by downloading the latest version from the Eclipse GMT project. It is not necessary to follow the installation guide to use this bundle. This software requires Windows OS (2000 or XP) and JRE (Java Runtime Environment) version 1.4 or later.

4.4.2.1. Getting Started

- Unzip this bundle and launch eclipse.exe in the eclipse folder.

- The Eclipse environment is preconfigured for usage of ATL tools. There are three projects included in the working space: Class2Relational (this is a transformation example partially described in the paper), CPL2SPL, and KM32Problem. Each project is accompanied by a separate README file. Please follow the instructions in these files to understand the purpose and the content of the projects and how to execute them. Class2Relational is the simplest example and CPL2SPL is the most complex. We suggest that the user starts with the simplest example and go to the more complex ones.

- Examples are meant to be easily executed without a significant knowledge about ATL and the environment.

4.4.2.2. Package Structure

The package contains two subfolders named "eclipse" and "doc". The first one contains a preconfigured distribution of Eclipse and ATL tools. The second folder contains the language and environment documentation.

4.5. AMMA Platform

The ATL project has allowed broadening the view of Model Driven Development (MDD). Model Transformations are absolutely necessary to any application of MDD. However they are probably not sufficient. We need other operations as well. The AMMA (ATLAS Model Management Architecture) [14] platform is a model management platform designed and developed by the ATLAS Team, INRIA. In the AMMA platform, in addition to ATL, some new projects are being developed. One is the ATLAS model Weaver (AMW). Another one is the ATLAS MegaModel Management Tool (AM3). A last one is the ATP (ATLAS Technical Projectors), a set of injectors and extractors to/from other technical spaces. ATL, AMW, AM3 and ATP are presently the essential part of the AMMA platform. Three of them: AM3 [25] [15], AMW [16] [24], and ATL are available as GMT [1] [19] components. All these tools are built on top of the Eclipse Modeling Framework (EMF). Documentation and open source software related to various components of the AMMA platform will be regularly made available.

4.5.1. AM3

The AM3 (**ATLAS MegaModel Management**) project [15][25] is a GMT component which is being developed by the ATLAS Team [53], INRIA [48]. The aim of the GMT project is to produce a set of prototypes in the area of Model Driven Engineering (MDE). The AM3 project acts in this way by providing a practical support for modeling in the large, i.e. dealing with global resource management in a model-engineering environment. AM3 is part of the AMMA platform.

The goal of AM3 (ATLAS MegaModel Management) is to provide a practical support for modeling in the large, i.e. managing global resources in the field of MDE (Model-Driven Engineering). These global resources are usually heterogeneous and distributed. To access them without increasing the accidental complexity of MDE, we need to invent new ways to create, store, view, access, and modify the global entities that may be involved in developing a solution. To this intent, the notion of a megamodel (i.e. a model which elements are themselves models) is being used. In order to achieve this overall goal, AM3 provides a set of tools and artifacts that implement our Global Model Management (GMM) approach which is based on the concept of "megamodel".

4.5.2. AMW

The AMW (**ATLAS Model Weaver**) project [16][24] is a GMT component developed by the ATLAS Team, INRIA. AMW is part of the AMMA platform.

The AMW project supports the creation of different kinds of links between model elements. The links are saved in a **weaving model**. The weaving model conforms to an extensible weaving metamodel.
Weaving models can be used in different application scenarios, such as tool interoperability, transformation specification, traceability, model merging.

4.5.3. TCS

TCS (Textual Concrete Syntax) [18] is a DSL for the specification of Textual Concrete Syntaxes in Model Enginnering. A presentation is available here made by ATLAS INRIA and SODIUS [52].

4.5.4. KM3

KM3 (Kernel Meta Meta Model) [17] is a neutral language to write metamodels [50] and to define Domain Specific Languages [51]. KM3 has been defined at INRIA.

The reference manual that presents the language may be found at: [33].

There is an evolutive library of metamodels written in KM3. This library of open source metamodels is called: AtlanticZoo [49]. The contribution to this zoo is open and welcome. A wiki page called "How to contribute to the Atlantic zoo" [47] explain the process. This library contains about 230 KM3 metamodels and has been named a Zoo. This collection of KM3 metamodels is intended for experimental purposes. On the same site there are also a number of "mirror zoos" containing metamodels written in other languages like MOF, UML, Prolog, VB, etc.

5. References

[1] GMT Web Site: http://www.eclipse.org/gmt/

[2] GMT/ATL Web Site: http://www.eclipse.org/gmt/atl/

[3] The ATL Home Page: http://www.sciences.univ-nantes.fr/lina/atl/

[4] ATL Download Page: http://www.eclipse.org/gmt/atl/download/

[5] ATL Wiki: http://wiki.eclipse.org/index.php/ATL

[6] ATL Publications: http://www.eclipse.org/gmt/atl/publication.php

[7] ATL Transformations Zoo: http://www.eclipse.org/gmt/atl/atlTransformations/

[8] ATL Mailing List: http://groups.yahoo.com/group/atl_discussion/

[9] ATL Source Code: http://dev.eclipse.org/viewcvs/indextech.cgi/org.eclipse.gmt/ATL

[10] ATL Documentation: http://www.eclipse.org/gmt/atl/doc/

[11] ATL External References on GMT: http://www.eclipse.org/gmt/atl/references/index.php

[12] ATL External References on ATL Home Page: http://www.sciences.univ-nantes.fr/lina/atl/references/Refs_to_ATL

[13] ATL Bundle: http://www.sciences.univ-nantes.fr/lina/atl/atldemo/adt/

[14] AMMA Wiki: http://wiki.eclipse.org/index.php/AMMA

[15] AM3 Wiki: http://wiki.eclipse.org/index.php/AM3

[16] AMW Wiki: http://wiki.eclipse.org/index.php/AMW

[17] KM3 Wiki: http://wiki.eclipse.org/index.php/KM3

[18] TCS Wiki: http://wiki.eclipse.org/index.php/TCS

[19] GMT Wiki: http://wiki.eclipse.org/index.php/GMT

[20] AMMA/ATL Workshop on Model Engineering First Edition (AWME): http://www.sciences.univ-nantes.fr/lina/atl/Events/AWME

[21] AMMA/ATL Workshop on Model Engineering Second Edition (AWME2): http://www.sciences.univ-nantes.fr/lina/atl/Events/AWME2

[22] Old ATL FAQ: http://www.eclipse.org/gmt/atl/faq/

[23] ATL News: http://www.eclipse.org/gmt/atl/news/index.php

[24] AMW Home Page: http://www.eclipse.org/gmt/amw/

[25] AM3 Home Page: http://www.eclipse.org/gmt/am3/

[26] AM3 Zoos: http://www.eclipse.org/gmt/am3/zoos/

[27] ATL Presentation Sheet: http://www.eclipse.org/gmt/atl/doc/ATL_PresentationSheet.pdf

[28] ATL Starter Guide: http://www.eclipse.org/gmt/atl/doc/ATL_Starter_Guide.pdf

[29] ATL User Manual: http://www.eclipse.org/gmt/atl/doc/ATL_User_Manual%5Bv0.7%5D.pdf

[30] ATL VM Specification: http://www.eclipse.org/gmt/atl/doc/ATL_VMSpecification%5Bv00.01%5D.pdf

[31] ATL Installation Guide: http://www.eclipse.org/gmt/atl/doc/ATL_Installation_Guide%5Bv0.1%5D.pdf

[32] ADT Installation From Source: http://www.eclipse.org/gmt/atl/doc/ADTInstallation%5B0.02%5D.pdf

[33] KM3 User Manual: http://www.eclipse.org/gmt/atl/doc/KernelMetaMetaModel%5Bv00.06%5D.pdf

[34] ATL Standard Library: http://wiki.eclipse.org/index.php/ATL_Standard_Library

[35] ATL Troubleshooter: http://wiki.eclipse.org/index.php/ATL_Language_Troubleshooter

[36] ATL FAQ: http://wiki.eclipse.org/index.php/ATL_FAQ

[37] ATL How To: http://wiki.eclipse.org/index.php/ATL_Howtos

[38] AM3 Ant Tasks Documentation: http://wiki.eclipse.org/index.php/AM3_Ant_Tasks

[39] ATL Wish List: http://wiki.eclipse.org/index.php/ATL_Wish_List

[40] OMG/MOF Meta Object Facility (MOF) 1.4. Final Adopted Specification Document. formal/02-04-03, 2002

[41] Budinsky, F., Steinberg, D., Ellersick, R., Grose, T. Eclipse Modeling Framework, Chapter 5 "Ecore Modeling Concepts". Addison Wesley Professional. ISBN: 0131425420, 2004

[42] Budinsky, F., Steinberg, D., Ellersick, R., Grose, T. Eclipse Modeling Framework. Addison Wesley Professional. ISBN: 0131425420, 2004

[43] OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP. October 2002

[44] Wiki definition: http://wiki.org/wiki.cgi?WhatIsWiki

[45] UML 2.0 OCL Final Adopted specification: http://www.omg.org/cgi-bin/doc?ptc/2003-10-14

[46] Ant: http://en.wikipedia.org/wiki/Apache_Ant

[47] How to contribute to the Atlantic zoo: http://wiki.eclipse.org/index.php/KM3/How_to_Contribute_to_Atlantic_Zoo

[48] INRIA: http://en.wikipedia.org/wiki/INRIA

[49] AtlanticZoo: http://www.eclipse.org/gmt/am3/zoos/

[50] Metamodel Definition: http://en.wikipedia.org/wiki/Meta-model

[51] Domain Specific Languages (DSL) definition: http://en.wikipedia.org/wiki/Domain_Specific_Language

[52] TCS Presentation by SODIUS and ATLAS Group (INRIA and LINA): http://www.eclipse.org/gmt/am3/tcs/doc/TCS_Presentation(SODIUS_INRIA).pdf

[53] ATLAS Team: http://www.inria.fr/recherche/equipes/atlas.fr.html

[54] How To Contribute to The ATL Transformations Zoo: http://wiki.eclipse.org/index.php/ATL/How_to_Contribute_to_ATL_Transformations_Zoo

Appendix