ATL:
Atlas Transformation Language

Specification of the ATL Virtual Machine

- version 0.1 -

2005

by
ATLAS group
LINA & INRIA
Nantes

Content

Figure List

Table List

1 Introduction

The ATL virtual machine 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.

2 ATL Programming Language Concepts

ATL, the Atlas Transformation Language, is the ATLAS INRIA & LINA research group answer to the OMG MOF [1]/QVT RFP [2]. It is a model transformation language specified both as a metamodel and as a textual concrete syntax. It is a hybrid of declarative and imperative. The preferred style of transformation writing is declarative, which means simple mappings can be expressed simply. However, imperative constructs are provided so that some mappings too complex to be declaratively handled can still be specified.

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

2.1 OCL Types and OCL Expressions in ATL

Since the OCL standard [3] is omnipresent in ATL programs, it is worth while introducing them right at the beginning. ATL is designed and implemented with respect to the OCL standard. Derivations from the standard are indicated with a specific remark.

2.1.1 OCL Primitive Types

OCL has four basic primitive datatypes:

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

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

2.1.1.1 Operations on Primitive Types

Integer and Real have the following operations:

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

Boolean has the operations:

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

String has the operations:

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

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

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

2.1.1.2 Examples of Operation on Primitive Datatypes

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

Addition of two integers:

```
1 + 1
```

Further mathematical operations:

```
1 - 80 div 2
```

```
1 < 2 and 1 > 2
```

Boolean operations:

```
true or false
```

2.1.2 OCL Collections

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

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

2.1.2.1 Collection Operations

Collection has the following operations:

- The number of elements in the collection self:

  ```
  size()
  ```

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

  ```
  includes()
  ```

- The information of whether an object is not part of a collection:

  ```
  excludes()
  ```

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

  ```
  count()
  ```

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

  ```
  includesAll()
  ```

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

  ```
  excludesAll()
  ```

- The information if a collection is empty:

  ```
  isEmpty()
  ```

- The information if a collection is not empty:

  ```
  notEmpty()
  ```

- Iterators over collectionsThe selection of a sub-collection:

  ```
  select()
  ```

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

  ```
  collect()
  ```

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

  ```
  forAll()
  ```

- The addition of all elements of a collection:

  `sum()`

2.1.2.2 Sequence Operations

Sequence supports all collection operations and some specific ones:

- Adding an element at the end of a sequence:

  `append()`

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

  `insertAt()`

- Casting a sequence to a set removes duplicates:

  `asSet()`

- Casting a sequence to a bag:

  `asBag()`

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

  `flatten()`

An example of the insertion of the element 15 at the second place is:

```
Sequence{12, 13, 12}->insertAt(2,15)
```

which results in:

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

2.1.2.3 Set Operations

Set supports all collection operations and some specific ones:

- Adding an element to the set:

  `including()`

- Removing an element from the set:

  `excluding()`

- Casting a set to a sequence:

  `asSequence()`

- Casting a set to a bag:

  `asBag()`

2.1.2.4 Bag Operations

Bag supports all collection operations and some specific ones:

- Adding an element to the bag:

  `including()`

- Removing an element from the bag:

  `excluding()`

- Casting a bag to a sequence:

  `asSequence()`

- Casting a bag to a set:

  `asSet()`

2.1.2.5 Collection Operation Examples

In the following some operations on collections are illustrated:

- Specifying a sequence literal:

  `Sequence{1, 2, 3}`

- Is a collection empty?:

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

- Getting the size of a collection:

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

Please compare:

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

returns 3 while

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

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

- Nesting sequences:

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

- Getting the first element of a sequence:

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

- Getting the last element of a sequence:

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

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

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

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

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

- Collect the names of all MOF classes:

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

The OCL shorthand expression

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

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

- All numbers in the sequence greater than 2:

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

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

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

2.1.3 OCL Model Elements

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

There are different OCL operations to treat and analyze classes:

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

2.1.3.1 Examples of Class Operations on MOF 1.4

It is very interesting to use OCL expressions in the context of the MOF metamodel [1]. Examples are given in the following.

- Please compare:
  `MOF!Attribute.oclIsKindOf(MOF!ModelElement)`

is `true` while
`MOF!Attribute.oclIsTypeOf(MOF!ModelElement)`

is `false`.

- Collect the names of all MOF classes:
  `MOF!Class.allInstances()->collect(e|e.name)`
- Count the number of classes in MOF:
  `MOF!Class.allInstances()->size()`
- Getting the names of all primitive MOF types by filtering:
```
MOF!DataType.allInstances()->
select(e|e.oclIsTypeOf(MOF!PrimitiveType))->collect(e|e.name)
```
- Getting the names of all primitive MOF types the simple way:
  `MOF!PrimitiveType.allInstances()->collect(e|e.name)`
- An enumeration instance in MOF:
  `MOF!VisibilityKind.labels`
- Getting all (local and inherited) StructuralFeatures of a Class. In the following code example the names of all StructuralFeatures of the class PrimitiveTypes are displayed:
```
MOF!PrimitiveType.findElementsByTypeExtended(
      MOF!StructuralFeature, true)->
            collect(e | e.name))
```

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

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

2.1.4 Enumerations

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

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

```
Gender::female
```

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

```
#female
```

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

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

2.1.5 OCL If Expression

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

An example of an if-clause:

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

2.1.6 OCL Let Expression

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

The let command to define variables:

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

Enchaining let expressions:

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

2.1.7 OCL Comment

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

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

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

2.1.8 OCL Tips and Tricks

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

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

This is why rather than:

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

you should write:

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

Furthermore, you should not write:

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

but rather:

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

endif
```

Furthermore, you should not write:

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

but rather:

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

or use an adequate if expression.

2.2 ATL Modules

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

Please note:

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

2.2.1 Preparation

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

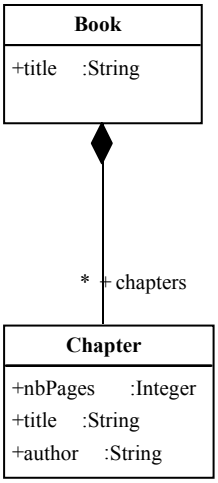

Figure 1. The Book metamodel

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

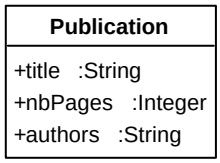

Figure 2. The Publication metamodel

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

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

2.2.2 Header Section

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

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

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

The keyword *module* defines the module name.

The keyword *create* introduces the target models declaration.

The keywords *from* introduces the source models declaration.

Please note:

- The ATL file name (here: *Book2Publication.atl*) has to correspond to the module name (here: Book2Publication) and must end with the extension *.atl*.

2.2.3 Import Section

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

```
uses strings;
```

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

Please note:

- Do not declare imports that are not necessary.
- The *string.atl* library contains many useful functions for Strings but is not necessary for the *Book to Publication* example.

2.2.4 Helpers

Helpers can be used to define (global) variables and functions. Helper functions are OCL expressions. They can call each other (recursion is possible) or they can be called from within rules (see section 2.2.5). In general, they serve to define repetitive pieces of code in one place.

A helper function has the following structure:

It starts with the keyword *helper* and ends with a semicolon. In between is an OCL expression.

- The notion of the term *context* is similar to OCL and it may be compared to an input parameter of a method. A context variable is specified with the help of the ATL path expression *metamodel!element* and is accessible via the self variable. In most cases the elements referred to are classes, i.e. the class Book of the metamodel Book. If no *context* is specified, the module itself is taken as context.
- The function name is introduced with *def :* and is followed by brackets.
- The return type is between a colon and an equals sign that serves as starting point for the function implementation.
- The function finishes with a semicolon.

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

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

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

Otherwise, it concatenates '*and*' with the *authorName*. The *getAuthors* function returns the result of the last iteration containing the concatenation of all *Authors' names*.

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

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

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

The helper function with context is called with:

```
b.getAuthors()
```

while the helper function with parameter is called with:

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

Please note:

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

2.2.5 Rules

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

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

The *source pattern* consists of the keyword *from*, an *in* variable declaration and optionally of a filter. A *filter* is an OCL expression restricting the rule to elements of the source model that satisfy certain constraints. The filter is written behind the in variable declaration and surrounded by brackets.

In the *target pattern* (the *to* part), rules declare to which element(s) of the target model the *source pattern* has to be transformed to. The implementation of the *target pattern* declares the details of the transformation. It may contain one or several *target pattern elements* (see Section 2.3.4).

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

For the *Book to Publication* example, a rule is required that transforms a *Book* to a *Publication*. Only *Books* with more than two pages are considered as *Publications*. The titles of *Books* and *Publications* have to correspond. The *authors* attribute of *Publication* contains all *authors* of all *Chapters*. For the latter, the function *getAuthors* (see Section 2.2.4) and the *getNbPages* can be reused. The number of pages of a *Publication* is the sum of all pages of the *Chapters* of a *Book*.

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

Please note:

- Each rule must have a name that is unique within the module.
- Assignments are separated with comma ",". The last assignment in a block of statements does not have a comma.
- The attribution of values is performed with the left arrow operator "<-".

2.3 ATL Advanced Features

The advanced features will be explained using new examples.

2.3.1 Queries and the Generation of Text

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

ALT query programs must start with a query instantiation which consist of the keyword *query,* a query variable, an equal sign and an OCL expression initializing the query variable. An include section (see Section 2.2.3) with *uses* is optional.

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

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

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

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

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

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

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

2.3.2 Libraries

When writing large programs, it is of advantage to group reusable pieces of code in one place. This is the purpose of ATL libraries. In Section 2.2.3 it is explained how to import existing libraries in a module (and they can be included in a query in the same way), in this section it will be illustrated how to write a library.

Unlike modules, Libraries are ATL programs that are not executable on their own and consist mainly of helper methods. Libraries start with the keyword library, the library name and a semicolon. An include section (see Section 2.2.3) with *uses* is optional. Then helpers follow.

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

```
library GeometryLib;

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

-- and some further geometric global helper variables

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

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

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

      Tuple {
            x = a.x - b.x,
            y = a.y - b.y,
            z = a.z - b.z
      };
 -- and some further geometric helper functions
```

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

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

Examples of usage of the above defined helpers:

```
thisModule.PIDiv180
```

or

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

2.3.3 Complex Headers

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

```
module GeometricalTransformations;

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

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

```
DXF1!Point
```

or

```
DXF2!Point
```

Please note:

- Each metamodel must have a different name (e.g. DFX1 and DXF2) but they may point to the same metamodel file.
- Different source models are separated using commas “,”.

2.3.4 Rules with Multiple Instantiations

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

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

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

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

2.3.5 Navigation and Multiple Instantiations

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

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

This works very well if each source element is mapped to no more than one target element. In case of multiple instantiations, one can use the operation resoveTemp with the name of the target pattern variable in order to distinguish between the different instantiations.

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

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

An example of how to get a table reference is:

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

An example of how to get a key reference is:

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

2.3.6 Flexible Runtime Instantiation of Target Elements

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

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

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

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

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

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

A speciality of (the current version of) ATL is that the *forEach* expression does not only iterate through the *allFiles* sequence, but also through the *allFilesPaths*. This is why each file is assigned with the corresponding path (represented through a sequence of folder names).

3 The Structure of the ATL Virtual Machine

This section specifies an abstract machine. It does not document any particular implementation of the ATL virtual machine. In order to be executed, ATL code, as well as code from other languages, can be compiled into the `asm` format. This file format example is detailed in Section 5.

The present section provides an overall description of an ATL virtual machine. It introduces the data types that have to be supported by the virtual machine, describes its internal runtime data structures as well as the model elements representation and presents the specified instruction set.

3.1 Data Types

All the types manipulated by the ATL virtual machine are considered as objects in that sense it is possible to call operations on them. However, two kinds of types may be distinguished: *primitive* and *composite* types. Primitive types provide support for string, boolean and numerical values. This set defines the core types the virtual machine relies on. Besides these primitive types, the specification of the ATL virtual machine also refers to a set of composite types that includes collections, operation signatures and object references. The representation of these composite types is to be based on the set of primitive types.

The ATL virtual machine expects that nearly all type checking is done prior to run time, typically by a compiler, and does not have to be done by the ATL virtual machine itself. The instruction set of the ATL virtual machine distinguishes its operand types using instructions intended to operate on values of specific types. For instance, *push*, *pushi* and *pushd* are all ATL virtual machine instructions that push a value onto the current operand stack, but each is specialized for its operand type: `string`, `int` and `double`, respectively. For a summary of type support in the ATL virtual machine instruction set, see Section 3.5.

3.1.1 Primitive Types

The set of primitive types supported required for the ATL virtual machine to run is the following:

- the `int` type that encodes integral numerical values.
- the `double` type that encodes real numerical values.
- the `boolean` type that encodes true and false truth values.
- the `string` type.

3.1.2 Composite Types

There are three kinds of composite types: the `collection` type, the operation `signature` type and the `reference` type. Composite values must be encoded by means of primitive types. However, this specification does not mandate any particular representation for these composite types and implementation choices are let to the discretion of the implementors.

The `collection` type simply specifies collections of elements. The elements contained in a collection can be of either a primitive or a complex type. However, the elements of a given collection must be of the same type. No particular assumptions are made on the structure and the semantics of collections that may be implemented for the ATL virtual machine. Thus, a collection implementation can either authorize or prohibit duplicated elements. In the same way, it can be either ordered or unordered. However, the ATL virtual machine requires each collection to supply an element by element iteration facility.

The operation `signature` type has to encode all information enabling to uniquely identify an operation. This information includes: the context for which the operation is defined, the operation name, as well as the number, the type and the order of the operation parameters. Based on encoded data, the ATL virtual machine is to be able to resolve an invoked operation and, therefore, to allocate and initialize a new frame for the operation execution to proceed. In the same manner the structure of collections is not specified, the way operation identification information shall be encoded and resolved is let to the discretion of the implementors.

Besides the `collection` and `signature` types, the ATL virtual machine handles a third composite data type, the references. A reference can be thought of as a pointer to an object. More than one reference to an object may exist. Referred objects can be of either primitive or composite types. It is possible, within a virtual machine, to implement references for all available types. However, the ATL virtual machine specification specifically requires an implementation to provide support for both the `variableref` and the `modeleltref` types.

The ATL virtual machine has to deal with the local variables defined in each allocated frame. These local variables are always fetched and modified via values of type `variableref`. The ATL virtual machine is to be able to resolve these variable references in order to access the local variables of the current frame. As for the collection and the signature types, the scheme enabling to resolve a variable reference into a variable of the local variables set of the current frame is let to the discretion of the implementors.

The ATL virtual machine contains explicit support for model elements. A reference to a model element is considered to have ATL virtual machine type `modeleltref`. Model elements are always operated on, passed, and tested via values of type `modeleltref`. Unlike for model elements, the ATL virtual machine does not requires any specific support for models. Although the ATL virtual machine deals with them, it does not handle model objects in the serialized format (i.e. as arguments of the virtual machine instructions), but refers to them via their name (encoded by means of `string` values).

3.2 Runtime Data Structures

The ATL virtual machine defines various runtime data areas that are used during execution of a transformation.

3.2.1 The pc Register

The ATL virtual machine has a `pc` (program counter) register. At any time, the ATL virtual machine executes the code of a single operation, the current operation. If this operation is not a native method, the `pc` register has to contain the address of the ATL virtual machine instruction currently being executed.

The `pc` register could be managed at either the frame or the virtual machine level. In the second case, the value of the `pc` register has to be memorized each time an operation is invoked so that, once the operation has completed, the execution may be resumed from the instruction following its invocation.

3.2.2 The ATL Virtual Machine Stack

The ATL virtual machine has a private *ATL virtual machine stack*, created at the virtual machine start-up time. The ATL virtual machine stack stores frames. An ATL virtual machine stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in operation invocation and return.

3.3 Frames

A new frame is created each time a non-native operation is invoked. A frame is destroyed when its operation invocation completes. Frames are allocated from the ATL virtual machine stack. Each frame has its own array of local variables and its own operand stack. Only one frame is active at anytime. This frame is referred to as the current frame, and the operation associated to this frame is known as the current operation.

A frame ceases to be the current frame if its operation completes or invokes another operation. When an operation is invoked, a new frame is created and becomes current when control transfers to the new operation. On operation return, the current frame passes back the result of its operation invocation, if any, to the previous frame. The current frame is then discarded as the previous frame becomes the current one.

It may be interesting to allocate a particular kind of frames (that may be called a *native frames*) each time a non-native operation calls a native method. By this mean, it would be possible to provide useful runtime information. When frames are only created for non-native methods, an error occurring in a native method is associated with its calling non-native operation. With native frame allocations, it would be possible to determine whether an error occurred in a native method or in the non-native operation that called it.

3.3.1 Local Variables

Each frame is associated with a set of variables known as its *local variables*. According to the virtual machine implementation, the size of the local variable set may be specified or calculated at runtime. Anyway, the local variable set has to be encoded and embodied, along with the code, in the operation representation that is associated with each non-native frame. Local variables can be associated with data of either primitive or composite types.

The ATL virtual machine uses local variables to pass parameters on operation invocation. The values of the parameters associated with an operation invocation are therefore assigned to local variables. The ATL virtual machine must provide a scheme that enables to identify, among all local parameters, the operation context (`self` in the ATL transformation language) and the subsequently passed parameters.

3.3.2 Operand Stack

Each frame contains a last-in-first-out (LIFO) stack known as its *operand stack*. The operand stack is empty when the frame that contains it is created. The ATL virtual machine supplies instructions to load constants or values from local variables or model elements fields onto the operand stack. Other ATL virtual machine instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack. The operand stack is also used to prepare parameters to be passed to operations and to receive operation results.

As an example, the model handling *get* instruction fetches the value of a model element field. It requires a *modeleltref* to be added on top of the operand stack, pushed there by previous instructions. The *modeleltref* is popped from the operand stack and the value of the fetched field is pushed back onto the operand stack.

Each entry on the operand stack can hold a value of any ATL virtual machine type. Values from the operand stack must be operated upon in ways appropriate to their types. It is not possible, for example, to push two `int` values and subsequently treat them as a *modeleltref*. A small number of ATL virtual machine instructions (the *dup* instructions and *swap*) operate on runtime data areas as raw values without regard to their specific types; these instructions are defined in such a way that they cannot be used to modify or break up individual values.

3.4 Representation of Model Elements

The way model elements are internally managed by ATL virtual machine is not constrained. Implementors are therefore free to use some existing model handling facilities or to provide their own model repository. Custom implementations may be based on various underlying technologies. For instance, one may provide a model repository implementation based on relational database technologies. In this context, a model element allocation may be associated with the creation of a new record. Fetching a model element attribute would therefore correspond to a request on a field of the record associated with the model element.

Implementors may also prefer to use an existing model handling library, such as MDR [4] or EMF [5]. Both include facilities to create, modify and store models and model elements, but they are based on distinct internal representations: MDR uses the MOF 1.4 [1] metamodel whereas EMF is based on Ecore [6]. Both MDR and EMF also offer XMI [7] serialization/deserialization support. However, they do not rely on the same XMI version (XMI 1.2 for MDR and 2.0 for EMF), which implies that a model serialized with MDR can not be deserialized with EMF (and inversely).

3.5 Instruction Set Summary

An ATL virtual machine instruction consists of an *opcode* specifying the operation to be performed followed an optional inline *operand*. Additional arguments or data that may be required by an instruction have to be fetched from the top of operand stack. Some instructions have no inline operands, but still require operands from the operand stack. Some other instructions have no operands at all and consist only of an opcode.

Instructions of the ATL virtual machine can be grouped into three distinct sets: the stack handling instruction set, the control instruction set and the model handling instruction set.

3.5.1 Operand Stack Handling Instructions

The ATL virtual machine provides a number of instructions enabling direct manipulations of the operand stack. These instructions may be sorted into three subgroups: the typed constants pushing instructions, the untyped handling instructions and the loading/storing instructions:

- Pushing a constant onto the operand stack: *push*, *pushi*, *pushd*, *pusht*, *pushf*.
- Untyped manipulations of the operand stack: *pop*, *dup*, *dup_x1*, *swap*.
- Loading/storing a variable from/onto the operand stack: *load*, *store*.

Pushing instructions are specialized instructions. Except for the *push* instruction (dedicated to `string` constants), the name of these instructions identifies the data type it applies on: *pushi* pushes an `int` constant, *pushd* pushes a `double` constant, whereas *pusht* and *pushf* respectively push the true and false `boolean` values. The untyped stack handling instructions provide some basic operations on the operand stack. These instructions proceed in the same way on all primitive and composite types. Thus, a dup instruction, which duplicates the top value of the operand stack, duplicates indifferently an integral value and an operation signature. As the previous instructions, the *load* and *store* instructions are untyped instructions. They perform transfers of values between local variables of the current frame and the operand stack of an ATL virtual machine frame.

3.5.2 Control Instructions

Control instructions cause the ongoing execution to continue from an instruction that may be different from the instruction that follows the control instruction. The ATL virtual machine defines five different control instructions:

- Conditional branch: *if*.
- Unconditional branch: *goto*.
- Iterative execution: *iterate*, *enditerate*.
- Method invocation: *call*.

In the case of a conditional branch, a boolean argument is checked in order to establish whether the execution continues from the next instruction, or whether it is redirected to another instruction. The unconditional branch defines a systematic branch of the execution flow to another instruction.

An iterative execution can be defined by means of the two instructions *iterate* and *enditerate*. Iterative treatments can be nested, and an *enditerate* instruction must be specified for each *iterate* instruction. An iterative treatment corresponds to the set of instructions embraced by an *iterate* and its corresponding *enditerate* instruction. The ATL virtual machine requires iterations to be specified with reference to a collection: the iterative treatment is therefore executed for each element of the collection provided as argument.

The last control instruction is the operation invocation instruction. When called operation is a non-native operation, this instruction systematically redirects the execution to the first instruction of the called operation. Once called operation has completed, execution goes back to the instruction that follows the *call* instruction.

3.5.3 Model Handling Instructions

This last class provides an instruction set dedicated to models and model elements handling. This instruction set also enables the ATL virtual machine to handle objects. There exist five model handling instructions:

- Create a new object: *new*.
- Access objects attributes: *get*, *put*.
- Fetch a model element: *findme*.

- Access the ATL context module element: *getasm*.

The *new*, *get* and *put* instructions provide objects manipulation facilities. Within the ATL virtual machine, all types are considered as objects (see Section 3.1). However, each of these three instructions is only supported by a particular subset of existing objects. The *new* instruction allows allocating both new model elements and new composite type objects, but primitive data type are created by simply pushing values onto the operand stack. The *get* and *put* instructions respectively provides read/write facilities on objects attributes. These two instructions should only be defined for composite objects for which they are meaningful. In the scope of the ATL virtual machine specification, they are defined for both model elements and the ATL context module. Implementors shall note that a custom native type does not have to implement the two instructions together (i.e. a type may be associated with the only *get* instruction).

Besides *get*, *put* and *new*, which define generic facilities for objects manipulation, the model handling instructions set supply two additional instructions, *findme* and *getasm*, that are specifically dedicated to model elements. The *findme* instruction is used to get a reference on a model element, based on the name of the metamodel it belongs to and its own name. In this context, *getasm* can be though as a specialized version of the *findme* instruction that returns a reference to the ATL context module element.

4 The ATL Virtual Machine Instruction Set

The instructions the ATL virtual machine supports can be grouped into three classes: the stack handling instructions, the control instructions and the model handling instructions. The stack handling instruction set defines the instructions enabling to handle a frame operand stack. The control instruction set specifies the instructions that provide control (such as iteration) over the execution flow. The model handling instruction set provides model elements access facilities. This section gives details about the format of each ATL virtual machine instruction and the operation it performs.

4.1 Format of Instruction Description

Each instruction description is presented as follows:

Operation

Short description of the instruction.

Format

mnemonic
operand

Operand stack

..., value1, value2 ⇒ ..., result

This notation shows an operation that begins by having *value2* on top of the operand stack, with *value1* just beneath it. As a result of the execution of the instruction, *value1* and *value2* are popped from the operand stack and replaced by *result* value, which has been calculated by the instruction. The remainder of the operand stack, represented by an ellipsis (...), is unaffected by the execution of the instruction.

Description

A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed the type of results, etc.

Notes

Comments not strictly part of the specification of an instruction are set aside as notes at the end of the description.

4.2 Stack Handling Instructions

4.2.1 The push instruction

Operation

Push `string` constant.

Format

push
s

Operand Stack

... ⇒ ..., s

Description

Push the `string` constant *s* onto the operand stack.

4.2.2 The pushi instruction

Operation

Push `int` constant.

Format

pushi
i

Operand Stack

... ⇒ ..., *i*

Description

Push the `int` constant *i* onto the operand stack.

4.2.3 The pushd instruction

Operation

Push `double` constant.

Format

Pushd
D

Operand Stack

... ⇒ ..., *d*

Description

Push the `double` constant *d* onto the operand stack.

4.2.4 The pusht instruction

Operation

Push *true* `boolean` constant.

Format

pusht

Operand Stack

... ⇒ ..., *true*

Description

Push the *true* `boolean` constant onto the operand stack.

4.2.5 The pushf instruction

Operation

Push *false* `boolean` constant.

Format

pushf

Operand Stack

... ⇒ ..., *false*

Description

Push the *false* `boolean` constant onto the operand stack.

4.2.6 The pop instruction

Operation

Pop the top operand stack value.

Format

pop

Operand Stack

..., value ⇒ ...

Description

Pop the top value from the operand stack.

4.2.7 The store instruction

Operation

Store value into local variable.

Format

store
variableref

Operand Stack

..., value ⇒ ...

Description

The *value* on the top of the operand stack is popped from the operand stack, and the local variable referred by *variableref* is set to *value*.

4.2.8 The load instruction

Operation

Load value from local variable.

Format

load
variableref

Operand Stack

... ⇒ ..., value

Description

The *value* of the local variable referred by *variableref* is pushed onto the operand stack.

4.2.9 The swap instruction

Operation

Swap the two top operand stack values.

Format

swap

Operand Stack

..., value2, value1 ⇒ ..., value1, value2

Description

Swap the two top values on the operand stack.

4.2.10 The dup instruction

Operation

Duplicate the top operand stack value.

Format

dup

Operand Stack

..., value ⇒ ..., value, value

Description

Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.

4.2.11 The dup_x1 instruction

Operation

Duplicate the top operand stack value and insert two values down.

Format

dup_x1

Operand Stack

..., value2, value1 ⇒ ..., value1, value2, value1

Description

Duplicate the top value on the operand stack and insert the duplicated value two values down in the operand stack.

4.3 Control Instructions

4.3.1 The if instruction

Operation

Branch if boolean value *b* is true.

Format

if
offset

Operand Stack

..., b ⇒ ...

Description

The value *b* on the top of the operand stack must be of type `boolean`, whereas the *offset* operand must be of type `int`. *b* is popped from the operand stack and checked. If the value is true, the execution proceeds at the *offset* from the address of the opcode of this *if* instruction. The target address must be that of an opcode of an instruction within the operation that contains this *if* instruction. Otherwise, if the boolean value *b* is false, execution proceeds at the address of the instruction following this *if* instruction.

4.3.2 The goto instruction

Operation

Branch always.

Format

goto
offset

Operand Stack

No change.

Description

The *offset* operand must be of type `int`. Execution proceeds at the *offset* from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the operation that contains this *goto* instruction.

4.3.3 The iterate instruction

Operation

Delimitate the beginning of iteration on collection elements.

Format

Iterate

Operand Stack

..., collection ⇒ *...*

Description

collection is popped from the operand stack. Execution proceeds from the opcode following this *iterate* instruction to its corresponding *enditerate* opcode (*iterate* instructions can be nested) for each element of the popped collection. Before each loop entry, the next element of the popped collection is pushed onto the operand stack. After the last loop, execution proceeds from the opcode following the *enditerate* instruction associated with this *iterate* instruction. Within an operation, each *iterate* instruction has to be associated with an *enditerate* instruction.

4.3.4 The enditerate instruction

Operation

Delimitate the end of iteration on collection elements.

Format

enditerate

Operand Stack

No change.

Description

Delimitate the end of an iteration previously opened by an *iterate* instruction. No action is associated with an *enditerate* instruction. Within a given operation, the number of *enditerate* instructions must be equal to the number of *iterate* instructions. An *enditerate* instruction may be specified only if a non-delimited *iterate* instruction has been previously declared in the same operation.

4.3.5 The call instruction

Operation

Call a method.

Format

call
signature

Operand Stack

..., context, [*arg1*, [*arg2 ...*]] ⇒ *...* when called operation has no return value

..., context, [*arg1*, [*arg2 ...*]] ⇒ *result* when called operation has a return value

Description

signature is a reference to an operation, which gives the name as well as the description of the operation (context for which the operation is defined and parameters types). The named operation is resolved according to its name, its context and the number, type and order of its parameters. The *context* must be followed on the operand stack by *nargs* argument values, where the number, type, and order of the values must be consistent with the signature of the selected operation.
The *context* and the *nargs* argument values are popped from the operand stack. If the operation is not native, a new frame is created on the ATL virtual machine for the method being invoked. The context and the argument values are consecutively made the values of local variables of the new frame, with *context* in local variable *0*, *arg1* in local variable *1*, and so one. The new frame is then made current, and the ATL virtual pc is set to the opcode of the first instruction of the operation to be invoked. Execution continues with the first instruction of the operation. When the operation has a return value, it is assigned to the remaining top value of the operand stack once the operation has completed. The return value is then pushed onto the operand stack of the calling frame.
If the called operation is a native method, the *nargs* argument values and context are passed to the code that implements the method.

Notes

The ATL virtual machine instruction set includes no *return*-like instruction. It is however possible to replace it with a *goto* instruction pointing to the last operation instruction.

4.4 Model Handling Instructions

4.4.1 The new instruction

Operation

Create new object.

Format

new

Operand Stack

..., classifier-name, metamodel-name ⇒ ..., reference

Description

This instruction is used to create new model elements and new composite objects. The type of the object (the element) to be created is resolved from *classifier-name* and *metamodel-name*. If the metamodel name is *#native*, the classifier is resolved from the set of supported composite native types. Otherwise, *metamodel-name* has to specify the name of a metamodel involved in the current transformation. The classifier is then resolved from the target models that conform to the metamodel identified by *metamodel-name*. If resolution completes successfully, memory for the new object instance is allocated. As a result, both *classifier-name* and *metamodel-name* are popped from the operand stack and replaced by a *reference* to the allocated object.

Notes

The current version of the ATL language does not enable to run transformations producing several target models that conform to a same metamodel. In such a situation, the model in which the new model element is allocated is undefined (among the models sharing the same metamodel). Future versions of the ATL virtual machine will include an additional instruction for model element allocation. Compared to the *new* instruction, this instruction will accept an additional parameter in order to identify the target model into which the new element will have to be allocated.

4.4.2 The get instruction

Operation

Fetch field from object.

Format

get
name

Operand Stack

..., reference ⇒ ..., value

Description

The *reference* is popped from the operand stack. The referenced field is resolved from *reference* and *name*. The *value* of the resolved field of the object pointed by *reference* is then fetched and pushed onto the operand stack.

Notes

The *get* instruction must be defined for the object pointed by *reference*. The ATL virtual machine internal types for which *get* instruction is defined are model elements and the ATL context module.

4.4.3 The set instruction

Operation

Set field in object.

Format

set
name

Operand Stack

..., reference, value ⇒ ...

Description

The referenced field is resolved from *reference* and *name*. The type of a *value* stored by a *set* instruction must be compatible with the referenced field. The *value* and *reference* are popped from the operand stack. The referenced field of the object pointed by *reference* is then set to *value*.

Notes

The *set* instruction must be defined for the object pointed by *reference*. The ATL virtual machine internal types for which *set* instruction is defined are model elements and the ATL context module.

4.4.4 The findme instruction

Operation

Fetch a classifier.

Format

findme

Operand Stack

..., classifier-name, metamodel-name ⇒ ..., classifier

Description

The classifier is resolved from *classifier-name* and *metamodel-name*. As a result, both *classifier-name* and *metamodel-name* are popped from the operand stack, and replaced by the resolved *classifier*.

4.4.5 The getasm instruction

Operation

Fetch the *asm* element.

Format

getasm

Operand Stack

... ⇒ ..., asm

Description

Push *asm*, a reference to the ATL Context Module element, onto the operand stack.

5 The asm File Format

This section describes the ATL virtual machine asm file format. This format is the one which is interpreted by the current reference implementation of the ATL virtual machine. This reference implementation does not fully meet the abstract specification provided in Section 3 and Section 4. However, the asm file format includes all information that may be required to build an implementation of the ATL virtual machine that complies with this abstract specification. ATL virtual machine implementations that aim to execute ATL transformations compiled into asm files have to be able to read the asm file format and correctly perform the operations specified therein.

Each asm file contains the definition of either an ATL transformation or an ATL query. This section mainly focuses on the asm encoding of ATL transformations. An asm file is an XML-based textual file. Its main elements are introduced in Section 5.1. The remaining sections provide a detailed specification of these elements.

5.1 Overview of the asm File Structure

An asm file is a textual file containing some XML data which encode an ATL transformation. This section provides an overview of the main XML elements that may be found in asm files. A complete DTD representation of the structure of asm files is provided in Appendix I.

An asm file is composed of a single asm structure:

```
<!DOCTYPE asm [
  <!ELEMENT asm       (cp, field*, operation+)>
  <!ELEMENT cp        (constant*)>
  <!ELEMENT field     (...)>
  <!ELEMENT operation (...)>
  ...
  <!ATTLIST asm name CDATA>
]>
```

The asm structure is an ordered structure that contains the transformation constant pool (cp), followed by a set of field (that may be empty) and one or more operation. The asm structure also has a name attribute. The value of this attribute is a constant pool index pointing to the constant pool entry that stores the transformation name (see Section 5.2, The Constant Pool).

The constant pool can be viewed as a storage space for the constant values that are handled in an asm file. It contains a set of constant pool entries. Each constant pool entry encodes a constant value. A single constant pool entry can be referred by multiple constant occurrences. Section 5.2 specifies the type and the format of the data that are stored in the constant pool.

Specified fields are those that are associated with the ATL context module. As opposed to constant pool entries, fields are not ordered and have to be accessed by their name. The field elements are described in Section 5.3.

Finally, the asm structure contains a non empty set of operations. The operation set specifies the instructions to be executed by the ATL virtual machine in order to carry out the compiled transformation. Each executable asm file contains at least one operation, the *main* operation, which is the entry point of the transformation program. The structure of an operation element is described in Section 5.4.

5.2 The Constant Pool

The constant pool stores all the constant values, whatever their type and the kind of information they encode (transformation name, operation signatures, offset values, etc.), that appear in an asm file. Within the constant pool, each constant value is hold by an indexed constant pool entry (a constant element). In the asm file format, constant tags embed no indexing information. Constant pool entries are therefore indexed according to the order in which they are defined within the asm file. Thus, the first constant element is

associated with the 0 index, the second with 1 and so on… A constant pool index is considered as valid if it is greater than 0 and less than the number of constant pool entries minus one. Note that the constant pool size is not specified in the asm file format: it has to be calculated by the ATL virtual machine.

The constant pool is defined by the following cp structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT cp (constant*)>
  <!ELEMENT constant EMPTY>
  ...
  <!ATTLIST constant value CDATA>
]>
```

The constant pool is composed of constant elements. These constant elements have a value attribute that contains the constant value. This value is encoded as a string. The different kinds of constant data stored in the constant pool, and the context in which they appear, are:

- The transformation name (in the asm element).
- The names of the manipulated ATL virtual machine types (in push instructions, see Section 5.4.3).
- The asm encoding of the ATL virtual machine types (in field, context and parameter elements, see Section 5.4.3).
- The names of the transformation input and output metamodels (in push instructions, see Section 5.4.3).
- The names of the transformation input and output models (in push instructions, see Section 5.4.3).
- The names of the model elements handled by the transformation (in push instructions, see Section 5.4.3).
- The operation signatures (in call instructions, see Section 5.2.2).
- The names of locally defined non-native operations (in operation elements, see Section 5.4).
- The handled offset values (in goto and if instructions, see Section 5.4.3).
- The local variables names (in lve elements, see Section 5.4.5).
- The asm encoding of the location of source code expressions (in lne elements, see Section 5.4.4).

Several occurrences of a same constant in an asm file are represented by the single entry in the constant pool. Note that a given constant pool entry may represent different data types. For instance, the “3” constant can represent both an offset value and a local variable table index. An executable asm file has to include a *main* operation. As a consequence, the constant pool of such an asm file will at least include a constant pool entry containing the “main” value.

The asm file format assumes specific encoding schemes for data types, operation signatures and expression locations. The following subsections detail these particular encodings.

5.2.1 Data Types Encoding

The asm file format defines an internal encoding for the types of the elements it handles. These values, encoded as strings, are used to specify data types (i.e. primitive, composite as well as model elements types) for the tags field, context and parameter. The field tag defines an attribute of the ATL context module. The context tag specifies the element type for which an operation is defined, whereas the parameter one defines both name and type of an operation parameter. Table 1 summarizes the encoding of the data types an ATL virtual machine is required to implement.

Type Name	Type Encoding
string	S
int	I
double	D
boolean	B
Model element	M<metamodel-name>!<modelelt-name>;
Collection(<type>)	C<type>

Table 1. asm encoding of the ATL virtual machine required types

The ATL virtual machine requires model elements to be handled by means of the modeleltref composite type. Encoding for such elements both includes the name of the pointed model element and the name of its metamodel. For instance, a Node model element defined in the scope of the XML metamodel is encoded as MXML!Node;.

The composite variableref type is implemented, in the asm file format, as an integral value that indexes an entry of the local variable table. However, as the other data types, variableref values are stored as strings within the constant pool.

Encoding of the abstract collection type includes the type of the elements contained in the collection. Thus, a collection of int value is encoded by CI. In the same way, a collection of XML Node model elements is encoded by CMXML!Node;.

5.2.2 Operation Signatures Encoding

The ATL virtual machine signature type is used by the *call* instructions to identify the operation to be invoked. A signature therefore has to encode all information that may be required by the virtual machine to match an operation call to its corresponding operation definition. In the asm file format, the signature type is encoded by means of a string and relies on the type encoding scheme specified in the previous section.

An asm signature encodes the following information: the operation name, the context in which the operation is defined, the operation return type and the number, the type and the order of the operation parameters. This information is encoded in the following string:
<context-type>.<operation-name>(<param-type1><param-type2>...<param-typen>):<return-type>

Considering this encoding scheme, an operation named *toBoolean*, which is defined within a string context, that returns a boolean value and that accepts no parameters is encoded by S.toBoolean():B. In the same way, the signature of the *getAttrVal* operation, defined in the context of an XML element, which accepts a string parameter (encoding the attribute name) and returns a string is MXML!Element;.getAttrVal(S):S.

5.2.3 Expressions Location Encoding

Within each defined operation, the linenumbertable structure encodes the bindings between source code expressions and ATL virtual machine instructions. Each linenumbertable entry binds a set of consecutive virtual machine instructions to a portion of the source code (see Section 5.4.4).

The asm file format defines a specific string encoding to identify source code portions. This encoding defines both the portion start line and start column and its end line and end column. An asm expression location encoding string respects the following format:
startline:startcolumn-endline:endcolumn

Considering this format, the string 21:41-21:57 identifies the source code portion that is comprised between the columns 41 and 57 of the line 21 of the source code file.

5.3 The Fields

An asm structure can include some field definitions. Fields can be viewed as local variables that are directly associated with the ATL context module. A field is defined by the following structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT field EMPTY>
  ...
  <!ATTLIST field name CDATA>
  <!ATTLIST field type CDATA>
]>
```

Fields have a `name` and a `type` attributes. These attributes contain valid constant pool indexes. Value of the `name` attributes indexes a variable name constant, whereas the `type` attribute value points to a data type encoding constant.

5.4 The Operations

An `asm` file defines a set of `operation`. Executable `asm` files have a program entry point which must be implemented by a *main* operation. Each operation contains a sequence of instructions that will have to be interpreted by the ATL virtual machine when the operation will be invoked. An operation is defined by the following structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT operation (context, parameters, code, linenumbertable,
localvariabletable)>
  <!ELEMENT context EMPTY>
  <!ELEMENT parameters (...)>
  <!ELEMENT code (...)>
  <!ELEMENT linenumbertable (...)>
  <!ELEMENT localvariabletable (...)>
  ...
]>
```

An `operation` is an ordered structure in which are specified a `context`, some `parameters`, some instructions (within the `code` element), a `linenumbertable` and a `localvariabletable`. The `context` element defines the context in which the operation is defined. The set of `parameters` encodes both type and name of the operation parameters. The `code` element contains an ordered sequence of instructions that implements the treatment associated with the operation. The `linenumbertable` structure defines bindings between the instructions of the `code` element and the expressions that appear in the source code. Finally, the `localvariabletable` stores the local variables that are defined for the operation. Note that, as opposed to the operation signature encoding, the `asm` file format does not include return type information to its operation definition.

These different elements are detailed in the following subsections: Section 5.4.1 defines the structure of the `context` element, Section 5.4.2 details the `parameters` definition, Section 5.4.3 describes the content of the `code` element and Section 5.4.4 and 5.4.5 respectively specify the structure of the `linenumbertable` and `localvariabletable` elements.

5.4.1 The Context

An operation context element specifies the context in which the operation is defined. A `context` element is defined by the following structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT context EMPTY>
  ...
  <!ATTLIST context type CDATA>
]>
```

Each `operation` is associated with a unique `context`. This context is defined by the `type` attribute that points to a data type encoding entry of the constant pool.

5.4.2 The Parameters

Each operation is associated with an ordered set of parameters. This ordered set encodes the parameter declaration order. The parameter set is empty for operations that accept no parameters. A parameter set is defined by the following structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT parameters (parameter*)>
  <!ELEMENT parameter EMPTY>
  ...
  <!ATTLIST parameter name CDATA>
  <!ATTLIST parameter type CDATA>
]>
```

Each parameter entry (encoded by a `parameter` element) has a `name` and a `type` attributes. Both attributes contain the index of a constant pool entry. The `name` attribute points to a constant pool entry that contains a variable name (the name of the parameter). The entry the `type` attribute refers to contains a data type encoding constant, which defines the parameter type. Parameter tags embed no ordering information. Within an `asm` file, the parameters order is encoded by the order in which the `parameter` tags appear in the `parameters` element.

5.4.3 The Code

The treatments that are performed by an operation are defined within the operation `code` element. The `code` of an operation is defined by the following structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT code (push, pushi, pushd, pushf, pusht, pop, store, load, swap, dup,
dup_x1, if, goto, iterate, enditerate, call, new, get, set, findme, getasm)*>
  <!ELEMENT push       EMPTY>
  <!ELEMENT pushi      EMPTY>
  <!ELEMENT pushd      EMPTY>
  <!ELEMENT pushf      EMPTY>
  <!ELEMENT pusht      EMPTY>
  <!ELEMENT pop        EMPTY>
  <!ELEMENT store      EMPTY>
  <!ELEMENT load       EMPTY>
  <!ELEMENT swap       EMPTY>
  <!ELEMENT dup        EMPTY>
  <!ELEMENT dup_x1     EMPTY>
  <!ELEMENT if         EMPTY>
  <!ELEMENT goto       EMPTY>
  <!ELEMENT iterate    EMPTY>
  <!ELEMENT enditerate EMPTY>
  <!ELEMENT call       EMPTY>
  <!ELEMENT new        EMPTY>
  <!ELEMENT get        EMPTY>
  <!ELEMENT set        EMPTY>
  <!ELEMENT findme     EMPTY>
  <!ELEMENT getasm     EMPTY>
  ...
  <!ATTLIST push  arg CDATA>
  <!ATTLIST pushi arg CDATA>
  <!ATTLIST pushd arg CDATA>
  <!ATTLIST store arg CDATA>
```

```
  <!ATTLIST load  arg CDATA>
  <!ATTLIST if    arg CDATA>
  <!ATTLIST goto  arg CDATA>
  <!ATTLIST call  arg CDATA>
  <!ATTLIST get   arg CDATA>
  <!ATTLIST set   arg CDATA>
]>
```

The `code` part of an operation definition contains a sequence of instructions among those that have been defined in the ATL virtual machine instruction set (see Section 4). Each defined instruction has its counterpart (in the form of an instruction element) in the `asm` file format. The instruction tags embed no ordering information. Within an `asm` file, instructions are ordered according to their order of appearance in the `code` part. They are numbered from 0 to the number of instructions minus one.

Some instructions tags (*push*, *pushi*, *pushd*, *store*, *load*, *if*, *goto*, *call*, *get* and *set*) have an `arg` attribute. This attribute contains the inline parameter of the instruction. Its value is an index that targets a constant pool entry. According to the instruction, the attribute points to different kinds of constant pool entries:

- Push instructions. Identified constant pool entry contains either a string constant (*push*), an integral constant (*pushi*) or a double constant (*pushd*).
- *load* and *store* instructions. Identified constant pool entry contains an index value for the local variable table.
- *if* and *goto* instructions. Identified constant pool entry contains an offset value. The constant pool stores absolute offset values. An *offset* value refers to the instruction number *offset* of the local instructions stack (instructions are numbered from 0).
- *get* and *set* instructions. Identified constant pool entry contains the name of an object attribute.
- *call* instructions. Identified constant pool entry contains an operation signature value.

5.4.4 The Line Number Table

The `linenumbertable` structure defines the bindings between the stack instructions of an operation and the corresponding code within the source file. This structure encodes useful information for implementers that wish to provide debugging facilities along with the ATL virtual machine. The `linenumbertable` is defined by the following structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT linenumbertable (lne*)>
  <!ELEMENT lne EMPTY>
  ...
  <!ATTLIST lne id    CDATA>
  <!ATTLIST lne begin CDATA>
  <!ATTLIST lne end   CDATA>
]>
```

The line number table is composed of line number entries (`lne` elements). Each entry specifies a binding between an instructions sequence from the operation stack and a part of the source code file.

Each entry has an id `id`, a `begin` and an `end` attributes. The `id` attribute identifies a source file portion. The attribute points to an expression location constant pool entry. The `begin` and `end` attributes make it possible to identify the sequence of instructions that are associated with the identified portion of the source code file. They respectively specify the number of the first and the last instructions of this sequence within the `code` element. The `code` instructions are numbered from 0 to the number of instructions minus one. The `begin` and `end` attributes do not refer to constant pool entries, but directly encode the instruction number values.

5.4.5 The Local Variable Table

The `localvariabletable` structure is similar to the Java local variable table. It encodes the visibility of each local variable within the operation. It also specifies the bindings between these local variables and the variable slots that are used to host them. The `localvariabletable` is defined by the following structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT localvariabletable (lve+)>
  <!ELEMENT lve EMPTY>
  ...
  <!ATTLIST lve slot  CDATA>
  <!ATTLIST lve name  CDATA>
  <!ATTLIST lve begin CDATA>
  <!ATTLIST lve end   CDATA>
]>
```

The local variable table is composed of local variable entries (`lve` elements). A local variable table contains at least one entry: the object that defines the context in which the operation has been called.

Each table entry is associated with a single local variable defined for the current operation. Each entry has a `slot`, a `name`, a `begin` and an `end` attributes. The `slot` attribute specifies the slot number the local variable is assigned to. The slot is identifies by an integral value (stored as a string). Note that this value is not a reference to the `asm` constant pool. A single slot can successively hold different variables of an operation providing that these variables are defined for distinct portions of the operation code. The `name` attribute defines the local variable name. This attribute contains a constant pool index that points to a variable name constant. Finally, the `begin` and `end` attributes make it possible to identify the sequence of instructions for which the local variable is defined. They respectively specify the number of the first and the last instructions of this sequence within the `code` element. The `code` instructions are numbered from 0 to the number of instructions minus one. As for the `slot` attribute, these integral values do not refer to constant pool entries.

Whatever the considered operation, the first slot of the local variable table is associated with the context object of the operation. Subsequent slots are associated with the operation parameters in the order they have been defined. Thus, the first parameter is associated with slot 1, the second parameter with slot 2 and so on… Following additional slots are used to hold the locally declared variables. The number of these additional slots has to be at least equal to the maximum number of simultaneously defined variables (i.e. variables that are defined over a same sequence of instructions).

6 Compiling ATL for the ATL Virtual Machine

This section aims to introduce compilation techniques for the ATL virtual machine. The ATL virtual machine has been primarily designed to support the ATL transformation language. This section therefore concentrates on compiling source code written in the ATL transformation language for the `asm` file format described in Section 5.

However, the ATL virtual machine does not assume that the instructions it executes are generated from ATL code. One may be interested in compiling source code written in another programming/transformation language to the ATL virtual machine. Understanding how the reference compiler utilizes the ATL virtual machine can therefore prove to be useful to prospective compiler writers, as well as to those trying to understand the ATL virtual machine itself.

A large part of this section is devoted to the way ATL code is compiled into the `asm` format. The compilation technique the referenced ATL compiler implementation is based on is also detailed.

6.1 Format of Examples

The format of the examples presented in this section should be familiar to anyone who has read assembly code. Each instruction takes the form:

```
<index> <opcode> [<operand>] [<comment>]
```

The `<index>` is the index of the opcode of the instruction in the array that contains the bytes of ATL virtual machine code for this method. Alternatively, the `<index>` may be thought of as a byte offset from the beginning of the method. The `<opcode>` is the mnemonic for the instruction's opcode. The optional `<operand>` is the operand of the instruction and the optional `<comment>` is given in the XML comment format:

```
   8  <pushi arg="100">       <!-- Push int constant 100 -->
```

The `<index>` prefacing each instruction may be used as the target of a control transfer instruction. For instance, a `<goto arg="8">` instruction transfers control to the instruction at index 8.

6.2 Compiling ATL Code to the `asm` File Format

This section aims to introduce some of the basic compilation techniques that are used to build `asm` files from ATL code. For this purpose, some common instruction patterns (such as control instructions, iterative blocks and operation invocations) are identified and the way they can be compiled to the `asm` format is discussed.

The additional data types are required to compile the ATL language are introduced in a first subsection. Sections 6.2.2 to 6.2.5 deal with the compilation of the supported data types. Compilation of conditional expression and iterative blocks are respectively detailed in Sections 6.2.6 and 6.2.7. Finally, Sections 6.2.8 and 6.2.9 illustrate compilation of operation invocations.

6.2.1 Additional Types

The ATL virtual machine specification requires implementations to provide support for the set of virtual machine data types (specified in Section 3.1). Executing transformations with the ATL virtual machine may require this set of virtual machine types to be enriched by means of language specific types.

The ATL transformation language has been built upon the OCL language [3]. Within ATL, the helpers, but also the rules filters and the initializations of allocated elements are expressed by means of OCL expressions. Since they appear in ATL transformations, OCL data types have to be supported by ATL virtual machines that aim to execute ATL transformations. Besides OCL-related types, ATL also defines its own specific types. These custom types, as well as OCL ones, are described in the following paragraphs.

The OCL language defines four different implementations of the ATL abstract `collection`. These OCL-based types are listed in Table 2 along with their respective encoding within the `asm` file format. Note that, in OCL as in ATL, these types all inherit from an abstract collection type.

Type Name	Type Encoding
Sequence<type>	Q<type>
Bag<type>	G<type>
Set<type>	E<type>
OrderedSet<type>	O<type>

Table 2. `asm` **encoding of OCL implementations of the** `collection` **type**

The four collection types defined within OCL provide all combinations between ordered/non-ordered and with/without duplicates:

- The `bag` type. It defines non-ordered collections with duplicates.
- The `set` type. It defines non-ordered collections without duplicates.
- The `sequence` type. It defines ordered collections with duplicates.
- The `orderedset` type. It defines ordered collections without duplicates.

Whatever their type, collection types are encoded in `asm` by their own encoding followed by the encoding of the contained elements. As an example, a sequence of integral numerical value is encoded as `QI`. In the same way, an ordered set of Node elements (defined within the XML metamodel) is encoded as `QMXML!Node;`.

Besides collection types, OCL defines two additional data types. Table 3 lists these remaining OCL-based types along with their respective `asm` encoding.

Type Name	Type Encoding
Tuple(<type1>, …, <typen>)	T<type1>…<typen>
EnumLiteral	Z

Table 3. `asm` **encoding of OCL types**

The `tuple` type defines signatures of multi-typed sequences. A `tuple` is encoded by associating its own encoding with those of elements that compose it (in the order they are presented by the signature). For instance, a tuple respectively containing an integral numerical value, a Node model element and a boolean value is encoded as `TIMXML!Node;B`.

The `EnumLiteral` type corresponds to the ATL enumeration type (see Section 2.1.4). In the scope of ATL, enumerations are specified in source and target metamodels, and the properties they define can be referred within the ATL code.

The ATL transformation language also defines some specific types that do not belong to the set of types required by the ATL virtual machine to run. These types are listed in Table 4 with their encoding in the `asm` file format.

Type Name	Type Encoding
Model	L
ATL context module	A
Transient Link	NTransientLink;
Transient Link Set	NTransientLinkSet;

Table 4. `asm` **encoding of custom ATL types**

The `model` type is associated with models. An ATL transformation mainly deals with its input and output models. The `ATL context module` type is associated with the transformation itself. For each ATL transformation execution, there exists a one and only instance of `ATL context module`. This element makes it possible for a transformation being executed to access elements of the transformation module (such as the transformation fields).

Finally, compilation of ATL code for the `asm` file format also takes advantage of the transient link type (encoded as `NTransientLink;`). This last type does not belong of the ATL language definition. This means that ATL programmers cannot define transient link variables within their ATL transformations. The

transient link type can be viewed as a linkage type internal to this ATL virtual machine implementation. A transient link makes it possible to link a given input model element matched by a rule to its previously created corresponding output model element(s). Thus, each distinct match of a single rule is represented by its own transient link. The transient link type comes along with the transient link set type (which is encoded by `NTransientLinkSet;`). This internal type defines convenient facilities for handling transient link collections.

6.2.2 Compiling Primitive Literals

The ATL language handles both primitive and composite data types. The four primitive ATL types are `Integer`, `Double`, `String` and `Boolean`. On the other hand, collections and model elements form the set of composite types. In this section, we introduce the compilation of both collections and primitive data types to the `asm` file format. Compilation of model elements handling is addressed in the next section.

Compiling primitive data types in achieved in a very simple way. Such data are therefore created and initialized by simply pushing, by means of the appropriate virtual machine instruction, a constant of the corresponding type onto the operand stack. Thus, the instruction `<push arg="10">` creates a new string which value is stored by constant pool entry 10. In the same way, assuming that the constant pool entry 11 contains an integral constant, the instruction `<pushi arg="11">` can be used to allocate a new integral value.

6.2.3 Compiling Composite Literals

Compilation of composite literals, such as collections, is not much more complex. Composite data types are required to be created before being initialized. Concerning the collection data type, this means that the ATL virtual machine first allocates an empty collection and then initializes it by means of successive element insertions. In the scope of the ATL virtual machine, collections are allocated by means of the *new* instruction. Inserting elements into the newly created collection can then be achieved by calling the native *including* operation defined for the collection. As an example, it is possible to consider the following declaration of an integer set with the ATL language:

```
let int_set: Set(Integer) = Set{8, 15, 6}
```

This simple declaration can be compiled into the following instruction set:

```
0  <push arg="78"/>       <!-- Push string constant "Set" -->
1  <push arg="9"/>        <!-- Push string constant "#native" -->
2  <new/>
3  <pushi arg="79"/>      <!-- Push int constant 8 -->
4  <call arg="46"/>       <!-- Call CJ.including(J):CJ operation -->
5  <pushi arg="27"/>      <!-- Push int constant 15 -->
6  <call arg="46"/>       <!-- Call CJ.including(J):CJ operation -->
7  <pushi arg="56"/>      <!-- Push int constant 6 -->
8  <call arg="46"/>       <!-- Call CJ.including(J):CJ operation -->
```

The first step consists in allocating a new empty `Set`. For this purpose, the strings “Set” and “#native” are successively pushed onto the operand stack (instructions 0 and 1). Whereas “#native” encodes the metamodel of the element to be created, “Set” refers to its classifier type. Instruction 2 uses these two strings to create a new empty set. As a result, the set is pushed on top of the operand stack. At this stage, the allocated set still requires to be initialized. That is the purpose of the remaining instructions. Each element of the declared collection is pushed onto the operand stack in turn and the *including* operation associated with the considered collection type is called for each of them. In the considered example, the considered collection is composed of integers. Initialization of the allocated set therefore consists in pushing an integer onto the operand stack (instructions 3, 5, 7) and calling the set *including* operation (instructions 4, 6, 8) with the set as context and the pushed integer as argument. Note that an *including* operation pushes the modified collection back onto the operand stack, so that it is still present on top of the stack once the operation has completed.

In the scope of the ATL transformation language, tuples are considered as another composite data type. As a collection, a tuple has to be created before being initialized. While a tuple allocation is performed by means

of the *new* instruction, its initialization is achieved by successively assigning values to its elements (with the *set* object handling instruction). As an example, the following declaration of an integer set with the ATL language may be considered:

```
let a_tuple: TupleType(a:Integer, b:String, c:Real, d:Boolean) =
      Tuple{a=8, b='Hello', c=15.9, d=true};
```

This tuple declaration can be compiled into the following instruction set:

```
   0  <push arg="121"/>       <!-- Push string constant "Tuple" -->
   1  <push arg="9"/>         <!-- Push string constant "#native" -->
   2  <new/>
   3  <dup/>
   4  <pushi arg="108"/>      <!-- Push int constant 8 -->
   5  <set arg="122"/>        <!-- Set "a" object property -->
   6  <dup/>
   7  <push arg="123"/>       <!-- Push string constant "Hello" -->
   8  <set arg="124"/>        <!-- Set "b" object property -->
   9  <dup/>
  10  <pushd arg="125"/>      <!-- Push double constant 15.9 -->
  11  <set arg="126"/>        <!-- Set "c" object property -->
  12  <pusht/>                <!-- Push true constant -->
  13  <set arg="127"/>        <!-- Set "d" object property -->
```

The first step aims to allocate a new tuple structure. This is achieved by instructions 0 to 2. They first push two string constants, “Tuple” and “#native” (which respectively encode the classifier and the metamodel of the object to be created), onto the operand stack. The new instruction is then called with these two parameters.

Once a new tuple object has been allocated, its elements can be initialized. The different elements can be initialized in any order. For each of them, the initial value is first pushed onto the operand stack and then assign to its corresponding property. In the example considered here, the initial values are provided as constants. Thus, assignment of the “a” property is achieved by pushing an integral constant onto the operand stack (instruction 4) and assigning it to the tuple “a” property (instruction 5). Initialization of the “b” property is achieved in the same way, by pushing a string value onto the operand stack (instruction 7). Note that the different *dup* instructions aim to duplicate the reference to the created tuple (on the operand stack) before it is consumed by the *set* instruction.

Finally, ATL deals with a last composite data type which corresponds to OCL enumerations. Although enumerations can not be defined within an ATL module, the ATL code can refer to enumerated properties. As the other composite types, a reference to an enumeration needs to be allocated prior to be initialized. As for the tuple data type, these two steps are based on the *new* and the *set* object handling instructions. In the following example, “sex” is a property of a `Person` model element whose possible values are “#male” or “#female”.

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

Assuming that the “sex” property of the `Person` model element is stored in local variable 1 (see Section 6.2.4 for further details on model elements compilation), this ATL code can be compiled into the following instruction set:

```
   0  <load arg="28"/>        <!-- Load local variable 1 -->
   1  <push arg="144"/>       <!-- Push string constant "EnumLiteral" -->
   2  <push arg="9"/>         <!-- Push string constant "#native" -->
   3  <new/>
   4  <push arg="145"/>       <!-- Push string constant "female" -->
   5  <set arg="58"/>         <!-- Set "name" object property -->
   6  ...
```

The first instruction loads the model element property onto the operand stack. The enumeration object is then allocated. For this purpose, two strings constants, “EnumLiteral” for the classifier name and “#native” for the metamodel, are successively pushed onto the operand stack (instructions 1, 2). The enumeration object is then allocated with a *new* instruction that accept the two previously pushed strings as parameters (instruction 3). The enumeration can then be initialized: the “female” string constant is pushed onto the operand stack (instruction 4) and assigned to the object “name” property (instruction 5). Now the enumeration data is initialized, the conditional expression can be further compiled (see Section 6.2.6 for details).

6.2.4 Compiling Model Elements

Among composite types, the ATL language provides support for model elements. As composite data, model elements are compiled into the `asm` format similarly to collections. This means that a model element is created and initialized during two distinct steps. There is however a significant difference in ATL between collections and model elements: whereas ATL allows programmers to create new collections, it does not provide any facilities for model element allocation. As a matter of fact, the model element allocations implicitly defined within an ATL transformation are made explicit (i.e. encoded into the `asm` file) at compile time.

In order to illustrate the way model element handling is compiled, we introduce an ATL rule which simply copies an input manual BibTeX entry to an equivalent output element:

```
rule Manual2Manual {
      from
        i : BibTeX!Manual
      to
        o : BibTeX!Manual (
            title <- i.title,
            id <- i.id
      )
}
```

Depending on the way a transformation is compiled into `asm`, the allocation and the initialisation of a model element can be distributed among several operations. This is, for instance, the case with the current reference compiler: a first operation allocates a new output element for each matched input elements and a second operation is dedicated to the initialization of previously created output elements (see Section 6.3 for further details). However, for simplicity sake, we present here both allocation and initialisation steps within a single instruction set. Assuming that the input element is stored in local variable 1, the rule can be compiled to:

```
   0  <push arg="22"/>       <!-- Push string constant "Manual" -->
   1  <push arg="23"/>       <!-- Push string constant "BibTeX" -->
   2  <new/>
   3  <store arg="96"/>      <!-- Store into local variable 2 -->
   4  <load arg="96"/>       <!-- Load local variable 2 -->
   5  <load arg="49"/>       <!-- Load local variable 1 -->
   6  <get arg="75"/>        <!-- Fetch "title" object property -->
   7  <set arg="75"/>        <!-- Set "title" object property -->
   8  <load arg="96"/>       <!-- Load local variable 2 -->
   9  <load arg="49"/>       <!-- Load local variable 1 -->
  10  <get arg="124"/>       <!-- Fetch "id" object property -->
  11  <set arg="124"/>       <!-- Set "id" object property -->
```

Similarly to the composite data types, the first step here consists in allocating a new output model element. This allocation follows the same scheme: two strings, respectively encoding the name of the element classifier and the name of its metamodel, are successively pushed onto the operand stack before the *new* instruction to be called (instructions 0 to 2). Created model element is then stored in local variable 2 (instruction 3). The remaining instructions are dedicated to the initialisation of the model element.

The created output model element is loaded onto the operand stack (instruction 4). The input model element is then pushed over it (instruction 5). Its `title` attribute is fetched by instruction 6. It is pushed back onto the

operand stack. This value can then be set to the title attribute of the previously (by instruction 4) pushed output model element (instruction 7). Initialisation of the `id` attribute follows the same process.

Note that initialisation of non-primitive attributes (i.e. links to other model elements) is slightly more complex. It requires calling the *__resolve__* operation (see Section 6.3.3 for further details).

6.2.5 Compiling Transient Links

The transient link type is not part of the ATL language. It represents a virtual machine internal facility to encode links between matched source elements and allocated target elements. Transient links explicitly capture the relations between the input and output elements of a rule. Explicit definition of transient links is performed at compile time.

In order to illustrate the way this data type may be compiled, we may consider again the `Manual2Manual` rule defined in previous section. This simple rule associates an input model element (`i`) to a single output model element (`o`). Each match of the `Manual2Manual` rule may lead to the creation of a distinct transient link. Assuming the input model element is stored in local variable `1`, the creation of such a new element may be compiled by the following instruction set:

```
   0  <push arg="31"/>       <!-- Push string constant "TransientLink" -->
   1  <push arg="9"/>        <!-- Push string constant "#native" -->
   2  <new/>
   3  <dup>
   4  <push arg="32"/>       <!-- Push string constant "Manual2Manual" -->
   5  <call arg="33"/>       <!--   Call   NTransientLink;.setRule(MATL!Rule;):V
operation -->
   6  <dup>
   7  <push arg="34"/>       <!-- Push string constant "i" -->
   8  <load arg="28"/>       <!-- Load local variable 1 -->
   9  <call arg="35"/>       <!--  Call  NTransientLink;.addSourceElement(SJ):V
operation -->
  10  <push arg="36"/>       <!-- Push string constant "o" -->
  11  <push arg="22"/>       <!-- Push string constant "Manual" -->
  12  <push arg="23"/>       <!-- Push string constant "BibTeX" -->
  13  <new/>
  14  <call arg="37"/>       <!--  Call  NTransientLink;.addTargetElement(SJ):V
operation -->
```

The first step consists in allocating a new `TransientLink`. For this purpose, the strings “TransientLink” and “#native” are successively pushed onto the operand stack (instructions 0 and 1). Whereas “#native” encodes the metamodel of the element to be created, “TransientLink” refers to its classifier type. Instruction 2 uses these two strings to create a new transient link. The name of the rule is then pushed onto the operand stack (instruction 4) and set to the allocated transient link by calling the *setRule* operation (instruction 5).

Next step is to initialize both input and output elements of the transient link. The input element name is pushed on top of the operand stack by instruction 7. The input model element can then be loaded from local variable 1 (instruction 8) and the *addSourceElement* operation called with these two data as arguments (instruction 9). The name of the output element is in turn pushed onto the operand stack (instruction 10). Instructions 11 to 13 then create a new empty output model element and the *addTargetElement* operation is called with both the element name and the allocated model element as arguments (instruction 14).

The transient link data type is defined along with the transient link set type. It provides a set of facilities for storing and retrieving transient links. Assuming that local variable 1 contains an already initialized transient link, the following instruction set illustrates the creation and the initialization of such a structure:

```
   0  <push arg="19"/>       <!-- Push string constant "TransientLinkSet" -->
   1  <push arg="9"/>        <!-- Push string constant "#native" -->
   2  <new/>
   3  <load arg="28"/>       <!-- Load local variable 1 -->
   4  <call arg="38"/>       <!--                                          Call
NTransientLinkSet;.addLink(NTransientLink;):V operation -->
```

This example shows that a transient link set is allocated in the same way collections are. The only difference is the classifier name provided to the new instruction (which here is “TransientLinkSet”). One a transient link has been loaded on top of the operand stack (instruction 3), it can be inserted into the transient link set by calling the *addLink* operation.

6.2.6 Compiling a Conditional Expression

The ATL language provides programmers with a conditional expression represented by the `if-then-else-endif` keywords. As opposed to a traditional programming language (such as Java or C), the ATL `if` instruction corresponds to an expression and, as such, must be associated with a value in any cases (i.e. whatever the result of the condition test). As a consequence, ATL `if` instructions always have to include an `else` clause.

For the purpose of illustrating the compilation of an ATL `if-then-else-endif` sequence, we consider a simple conditional expression which returns two distinct integral constants according to the value of a given integral variable:

```
if val > 10
      then 1
      else 2
endif;
```

This ATL conditional expression can be compiled into the following ATL virtual machine instructions set:

```
   0  <load arg="21"/>       <!-- Load local variable 1 -->
   1  <pushi arg="54"/>      <!-- Push int constant 10 -->
   2  <call arg="55"/>       <!-- Call J.>(J):J operation -->
   3  <if arg="56"/>         <!—- Conditional branch to instruction 6 -->
   4  <pushi arg="32"/>      <!-- Push int constant 2 -->
   5  <goto arg="57"/>       <!—- Branch to instruction 7 -->
   6  <pushi arg="21"/>      <!-- Push int constant 1 -->
```

The first step is here to evaluate the `if` boolean expression. Assuming that the `val` variable implied in this boolean expression is stored within local variable 1, it is loaded on top of the operand stack (instruction 0). The integral constant that constitutes the other part of the boolean expression is pushed in its turn onto the operand stack (instruction 1). Next instruction can then invoke the > operation with `val` as context and `10` as parameter (instruction 2). As a result, this operation pushes a boolean value on top of the operand stack. Returned boolean value is then tested by a conditional branch (instruction 3). If the value is true (i.e. the condition is fulfilled), the execution flow is redirected to instruction 6 which pushes the integral constant 2 onto the operand stack. Otherwise, if the boolean value is false, the execution proceeds from the next instruction (instruction 4) which pushes the integral constant 1 on top of the operand stack. Next instruction (instruction 5) then redirects execution to the instruction directly following the true condition block. In this example, execution is redirected to instruction 7 which follows the instruction 6 that constitutes the true condition block.

This simple example illustrates that it is possible for a branch (i.e. a *goto* or an *if* virtual machine instruction) to target the instruction *n* in an operation composed of *n* instructions numbered from 0 to *n*-1. Such a branch will result in resuming the operation being executed. If this operation has a return value, the value on top of the operand stack when branch is performed is the one to be returned. In the case a conditional expression constitutes the body of an iterative treatment, the corresponding branch will target the virtual machine *enditerate* instruction that is associated with the on going iterative treatment (see Section 6.2.7, compilation of the `select` ATL instruction).

6.2.7 Compiling an Iterative Block

As with the OCL language, an iterative treatment is associated in ATL with an iteration over a collection of elements. This implies an iterative block to be executed for each element of the collection it refers to. An iterative block is typically compiled into a set of ATL virtual machine instructions embraced by the *iterate*/*enditerate* couple of virtual machine instructions (see Sections 4.3.3 and 4.3.4). As ATL iterations, the virtual machine *iterate* instruction is defined according to a reference collection, which is taken on top of the

operand stack. Prior to each iteration, a new element of the reference collection is pushed onto the operand stack. The *enditerate* instruction only aims to mark the end of an iterative block (it is not associated with any particular treatment).

The ATL language defines three main iterative instructions: `select`, `collect` and `iterate`. The `select` instruction enables to select a subset of the reference collection. Elements of this reference collection are filtered according to a boolean expression. A `select` instruction produces an output collection which size is equal or less than the one of the reference collection. These elements moreover have the same type that those of the reference collection. As opposed to the `select` one, the `collect` instruction produces a homogeneous `Sequence` of elements whose cardinal is equal to the reference collection one. Although the returned sequence is to be homogeneous, the type of its elements may be different from the one of reference collection elements. A `collect` may, for instance, be used on a model elements collection to compile the values of a given property of these elements. It can also be used on any collection of cardinal *n* to easily produce a new `Sequence` composed of *n* identical constant values.

We propose here to illustrate the compilation of both `select` and `collect` ATL instructions. We first consider an example making use of the `select` function. We introduce, in this scope, a set of integral values:

```
let int_set: Set(Integer) = Set{8, 15, 6, 3, 19}
```

It is now possible to operate a selection on `int_set` in order to build the subset of values that are greater than 10. This selection corresponds to the following ATL statement:

```
int_set->select(e | e > 10);
```

This simple ATL selection can be compiled into the following ATL virtual machine instructions set:

```
 0  <push arg="31"/>       <!-- Push string constant "Set" -->
 1  <push arg="9"/>        <!-- Push string constant "#native" -->
 2  <new/>
 3  <load arg="21"/>       <!-- Load local variable 1 -->
 4  <iterate/>
 5  <store arg="32"/>      <!-- Store into local variable 2 -->
 6  <load arg="32"/>       <!-- Load local variable 2 -->
 7  <pushi arg="46"/>      <!-- Push int constant 10 -->
 8  <call arg="47"/>       <!-- Call J.>(J):J operation -->
 9  <call arg="48"/>       <!-- Call B.not():B operation -->
10  <if arg="49"/>         <!-- Conditional branch to instruction 13 -->
11  <load arg="32"/>       <!-- Load local variable 2 -->
12  <call arg="50"/>       <!-- Call CJ.including(J):CJ operation -->
13  <enditerate/>
```

Performing a `select` operation on an ATL set results in a new set that contains the only elements that fulfil the condition specified with the boolean expression. The first step therefore consists in instantiating a new empty set. This is achieved with instructions 0 to 2. Assuming that the reference set is stored within local variable 1, it can be then loaded on top of the operand stack (instruction 3). The *iterate* instruction uses this set as its reference collection.

Prior to each iteration, a new element of the collection is implicitly pushed onto the operand stack by the ATL virtual machine. This element is saved into local variable 2 and made available again on top of the operand stack by instructions 5 and 6. Next step corresponds to the evaluation of the ATL boolean expression (`e > 10`). For this purpose, the 10 integral constant is pushed onto the operand stack (instruction 7) and the > operation is called with the current element as context and the integral constant as parameter (instruction 8). This invocation returns a boolean value on top of the operand stack. Next instruction calls the `not` operation with the previously returned boolean value as context (instruction 9). This new invocation pushes a new boolean value onto the operand stack. This value is then tested by a conditional branch (instruction 10). If the value is `true` (i.e. the ATL condition is not fulfilled), the execution flow is redirected to instruction 13 which corresponds to the end of the current iteration. Otherwise, if the ATL condition is fulfilled, execution proceeds from the next instruction (instruction 11). This instruction loads the current collection element onto

the operand stack. This element can then be inserted into the produced set by calling the `including` operation with the output collection as context and the current element as parameter (instruction 12). Note that:

- The context output collection is initially pushed onto the operand stack by instruction 3.
- The `including` operation pushes the updated collection onto the operand stack so that it is not necessary to push it again for the next iteration.

In order to illustrate the compilation of the `collect` instruction, we now introduce another example. For this purpose, we consider `entry_set`, a set of model elements of type `TitledEntry` that represent those entries of a `BibTeX` file that include a `title` attribute. It is possible to operate a `collect` on this `entry_set` in order to build a sequence containing the titles that appear in the considered BibTeX file. This operation corresponds to the following ATL statement:

```
entry_set->collect(e | e.title);
```

This simple ATL instruction can be compiled into the following ATL virtual machine instructions set:

```
0  <push arg="31"/>       <!-- Push string constant "Sequence" -->
1  <push arg="9"/>        <!-- Push string constant "#native" -->
2  <new/>
3  <load arg="21"/>       <!-- Load local variable 1 -->
4  <iterate/>
5  <store arg="32"/>      <!-- Store into local variable 2 -->
6  <load arg="32"/>       <!-- Load local variable 2 -->
7  <get arg="45"/>        <!-- Fetch "title" object property -->
8  <call arg="46"/>       <!-- Call CJ.including(J):CJ operation -->
9  <enditerate/>
```

When applied to an ATL collection, a `collect` instruction produces a `Sequence` that contains as many elements as the reference collection. As for a `select` compilation, the first step consists in instantiating a new empty sequence. This is achieved with instructions 0 to 2. Assuming that the reference set is stored within local variable 1, it can be then loaded on top of the operand stack (instruction 3). The *iterate* instruction uses this set as its reference collection.

Similarly to the `select` compilation, a new element of the collection is implicitly pushed onto the operand stack by the ATL virtual machine priori to each iteration. This element is saved into local variable 2 and made available again on top of the operand stack by instructions 5 and 6.

Next step aims to retrieve/build the element to be inserted in the sequence to be returned. In our example, this element corresponds to the `title` of the current BibTeX entry (i.e. the current element of the reference collection). This string value is fetched by means of the *get* instruction (that applies to the model element on top of the operand stack) and pushed back onto the operand stack (instruction 7). The title can then be inserted into the produced sequence by calling the `including` operation with the output collection as context and the fetched string as parameter (instruction 8).

The last iterative instruction provided by the ATL language is the `iterate` instruction (which should not be confused with its homonym virtual machine instruction). This last instruction is more flexible than the `select` and `collect` ones. It therefore allows programmers to:

- Declare (and initialize) the data type of the value to be returned by the iterative treatement. For instance, `iterate` makes it possible to return a primitive ATL data type (such as `String`, `Integer`, etc.) as result of the iterative treatment.
- Specify what iterative treatment to perform.

In order to illustrate the compilation of the `iterate` ATL instruction, we consider `int_set` as a set of integral values. It is therefore possible to compute the sum of the values contained in `int_set` by means of the following ATL instruction:

```
int_set->iterate(e; ret : Integer = 0 | ret + e);
```

As a reminder, `e` identifies the current element of the reference collection. `ret : Integer = 0` corresponds to the declaration and the initialization of the variable to be returned (`ret`). Finally, `ret + e` specifies the operation to be iteratively performed. This ATL instruction can be compiled into the following ATL virtual machine instructions set:

```
  0  <pushi arg="7"/>        <!-- Push int constant 0 -->
  1  <store arg="32"/>       <!-- Store into local variable 2 -->
  2  <load arg="21"/>        <!-- Load local variable 1 -->
  3  <iterate/>
  4  <store arg="67"/>       <!-- Store into local variable 3 -->
  5  <load arg="32"/>        <!-- Load local variable 2 -->
  6  <load arg="67"/>        <!-- Load local variable 3 -->
  7  <call arg="68"/>        <!-- Call J.+(J):J operation -->
  8  <store arg="32"/>       <!-- Store into local variable 2 -->
  9  <enditerate/>
 10  <load arg="32"/>        <!-- Load local variable 2 -->
```

Similarly to the previously described ATL iterative instructions, the first step consists in allocating and initializing the variable used to store the value to be returned. However, compared to the other iterative instructions, this variable can be here of any ATL types. In this example, this step consists in initializing the integral value `ret` to 0. This is achieved by instruction 0. Initilialized value is then stored in local variable 2 (instruction 1). Assuming that the reference set is stored within local variable 1, it is then loaded on top of the operand stack (instruction 2). The *iterate* instruction (instruction 3) uses this set as its reference collection.

A new element of the reference collection is implicitly pushed onto the operand stack at the beginning of each iteration. It is saved into local variable 3 by instruction 4. Instructions 5 and 6 respectively load the value to be returned and the current element on top of the operand stack. The operation + is then called with the value to be returned as context and the current element of the reference collection as parameter (instruction 7). As a result, the operation pushes the computed value onto the operand stack. This value is stored in local variable 2 as the value to be returned (instruction 8). Once the iteration has completed, the content of local variable 2 is loaded onto the operand stack (instruction 10).

6.2.8 Receiving Arguments

As opposed to Java, the ATL language does not define context-less operation calls facilities. Whereas `static` Java methods do not have any instance, all invoked operations in ATL are to be associated with a specific context. As a consequence, there exist a one and only way to receive arguments for ATL operations.

The *n* arguments that are passed to an invoked operation are stored in the `localvariabletable` structure of the stack frame created for the new operation. The table entry 0 is reserved to hold a reference to the context in which the operation is called and the arguments are stored in local variables numbered from 1 to *n*. The arguments are received in the order they were passed. As an example, consider the code of the following ATL helper:

```
helper
      def: firstOfTwo(i: Integer, j: Integer) : Integer = i;
```

This helper is defined in the context of the ATL context module and simply returns the value of its first argument. It compiles to:

```
Operation int firstOfTwo(int,int)
   0  <load arg="1">          <!-- Load local variable 1 -->
```

The value of the operation first argument is loaded on top of the operand stack. This top value is the one to be returned when the operation completes.

6.2.9 Invoking an Operation

The ATL language defines a one and only way to invoke operations. ATL invocations are implemented by the *call* instruction (see Section 4.3.5), which takes as argument an index to a constant pool entry containing the full signature of the invoked operation. This operation signature provides the type of context for which the operation is defined, the operation name, its return type and the number, the order and the type of its arguments. The `test` helper, defined below, provides an operation invocation example:

```
helper def: test() : Integer =
      thisModule.firstOfTwo(12, 13);
```

This new helper, which is also defined in the context of the ATL context module, calls the `firstOfTwo` helper with two integral numerical constants. It compiles to:

```
Operation test()
   0  <getasm>                <!-- Push asm module -->
   1  <pushi arg="31">        <!-- Push int constant 12 -->
   3  <pushi arg="32">        <!-- Push int constant 13 -->
   4  <call arg="46">         <!-- Call A.firstOfTwo(II):I operation -->
```

First step of the operation invocation process is pushing operation context and arguments onto the top of the operand stack. This is achieved by first pushing a reference to the ATL context module onto the operand stack. Operation invocation's arguments, `int` values 12 and 13, are pushed in their turn onto the operand stack. Once the frame associated with the `firstOfTwo` operation is created, the context and the arguments passed to the operation invocation become the initial values of the new frame's local variables. Thus, the local variable 0 of the invoked `firstOfTwo` operation will receive the reference to the ATL context module whereas local variables 1 and 2 will be respectively initialized to the 12 and 13 `int` values.

When the invoked operation completes, its `int` return value is pushed onto the operand stack of the invoking operation (`test`). Note that the ATL virtual machine does not define any return-like instruction. As a consequence, an operation always completes with its last instruction. An early return instruction can however be implemented by means of a `goto` instruction pointing to the last instruction of the operation.

If the invoked operation has no return value (i.e. returns a void element), the operand stack of its invoking operation remains unchanged when it completes.

The operand of the *call* instruction (in the example, the runtime constant pool index #4) is a symbolic reference (the operation signature) to a method of an instance. This reference is stored in the constant pool and resolved at run time to determine the actual operation location.

6.3 Compiling an ATL Transformation

This section describes the way an ATL transformation is compiled into the `asm` format for the current reference implementation of the ATL virtual machine. Compiling an ATL transformation consists in generating constant pool, a set of fields and a set of operations from the ATL source code elements (helpers and rules). The following subsections describe the way these source elements are compiled into `asm` elements.

6.3.1 Compiling Helpers

The ATL language enables to define two different kinds of helpers: attribute and function helpers. An attribute helper can be though as a variable global to the ATL module. This variable has to be initialized at the transformation start-up. A function helper specifies a treatment which is executed each time it is invoked.

According to their type, helpers are compiled differently. Thus, two `asm` elements are generated for each attribute helper: a homonym field and an initialization operation. The field aims to store the value associated with the attribute helper whereas the operation specifies the code for the field initialization. Such a generated operation is named *__init<helper-name>*. The context of an attribute helper has to refer to the ATL module but the generated operation accepts the same parameters that the helper it is associated with. The following attribute helper declaration may be considered as an example:

```
helper def: size_max : Integer = 15;
```

This helper declaration compiles into the following `asm` field:

```
<field name="5" type="6"/>          <!-- size_max : I -->
```

Moreover, the initialization of the field is performed by this generated operation:

```
<operation name="21">               <!-- __initsize_max -->
      <context type="8"/>           <!-- A -->
      <parameters>
      </parameters>
      <code>
            <load arg="9"/>         <!-- 0 -->
            <pushi arg="22"/>       <!-- 15 -->
            <set arg="5"/>          <!-- size_max -->
      </code>
      <localvariabletable>
            <lve slot="0" name="20" begin="0" end="2"/><!-- self -->
      </localvariabletable>
</operation>
```

Note that an attribute helper may accept other attribute helpers as parameters. Cross-references, as well as recursivity, are obviously prohibited in attribute helper definitions. However, a compiler should be able to manage valid dependencies between among attribute helpers. The reference compiler currently does not manage these dependencies. Attribute helpers are therefore required to be declared in the right order (with respect to existing dependencies).

Compared to attribute helpers, function helpers are simply compiled into homonym operations. Generated operations are defined with the same context and parameters that their corresponding ATL helpers. An example is provided by the `firstOfTwo` helper introduced in Section 6.2.8:

```
helper
      def: firstOfTwo(i: Integer, j: Integer) : Integer = ...;
```

The context of this helper is the ATL module (since no context is specified), it accepts two integral parameters and returns an integral value. It is compiled into the following `asm` structure:

```
<operation name="67">                   <!-- firstOfTwo -->
      <context type="8"/>               <!-- A -->
      <parameters>
            <parameter name="34" type="6"/>    <!-- 1 : I -->
            <parameter name="54" type="6"/>    <!-- 2 : I -->
      </parameters>
      <code>
            ...
      </code>
      <localvariabletable>
            <lve slot="0" name="20" begin="0" end="0"/><!-- self -->
            <lve slot="1" name="40" begin="0" end="0"/><!-- i -->
            <lve slot="2" name="69" begin="0" end="0"/><!-- j -->
      </localvariabletable>
</operation>
```

Note that entries 1 and 2 of the local variable table respectively contain to the helper parameters `i` and `j`.

6.3.2 Compiling Rules

A rule includes a filter and an instantiation parts. The filter defines a condition that has to be fulfilled by input model elements in order to generate output model elements. The instantiation part specifies the way newly

allocated output model elements are initialized. Compilation of an ATL rule results in inserting two new operations into the `asm` file. These operations are the rule matching operation and the rule instantiation operation. They are respectively named *__match<rule-name>* and *__apply<rule-name>*. Both are defined within the ATL module context.

The matching operation checks, for each source model element which has a meta relation with the metamodel element associated with the rule source pattern, whether it verifies the filter or not. A target metamodel element is instantiated for each source model element validating the filter. Matching operations have no parameters. A matching operation is called once for all the input model elements targeted by its associated rule. It iterates over this set of input model elements. For each of them, it tests the condition specified within the in pattern of the rule. If the current model element fulfils this condition, the matching operation allocates a new empty target model element. A new transient link, which associates the rule, the input and the output model elements, is then allocated and inserted into the ATL module transient link set. The matching operation generated for the `Manual2Manual` operation introduced in Section 6.2.4 is provided in Appendix II.

The instantiation operation aims to initialize the properties of the output model elements instantiated by the corresponding matching operation. It is called for each generated output model element. The instantiation operation accepts a transient link as argument. This transient link is one of the links generated by the corresponding matching operation. It associates an empty allocated output model element with its input model element. For this purpose, the operation fetches the properties of the input model element and builds new values that are assigned to the output model element properties. To this end, the *__resolve__* operation is invoked to resolve references to other input model elements (see Section 6.3.3 for further details on the *__resolve__* operation). The instantiation operation generated for the `Manual2Manual` operation introduced in Section 6.2.4 is provided in Appendix III.

6.3.3 Compiling a Transformation

An ATL transformation is composed of helper and rule definitions. Previous sections have introduced the way the different elements that compose an ATL transformation are compiled into the `asm` format. We describe here the `asm` elements that are generated for an ATL transformation.

Compilation of an ATL transformation module results in two `asm` fields (*links* and *col*) and three new operations (*main*, *__matcher__* and *__exec__*). The *links* field contains a transient link set which is used by the ATL module to store the transient links that are generated by the matching operations. The *col* field

The *main* operation is the transformation entry point. It first initializes the *col* field. It then calls the attribute helper initialization operations (*__init<helper-name>*), before calling the *__matcher__* operation and finally the *__exec__* one. The *__matcher__* operation initializes the *links* field and calls the *__match<rule-name>* operation associated with each defined rule. Finally, the *__exec__* operation invokes the *__apply<rule-name>* operations.

Most internally generated operations have their name escaped in order to avoid name collisions with defined helpers and rules. As an exception, the ATL to `asm` reference compiler does not escape the name of the *main* operation at present time. This behaviour has no incidence on rules naming since the name of operations that are associated with rules are either prefixed by *__match* or *__apply*. However, since a function helper is associated with a homonym operation, ATL programs are assumed not to contain any function helper named “main”. Naming of attribute helpers remains unconstrained (names of the operations associated with this kind of helpers are prefixed with *__init*).

Two additional operations may be generated according to the transformation code: the *__resolve__* and *resolveTemp* operations (the reference compiler currently generates them even when they are not required). Both are defined within the ATL module context. The *__resolve__* is called by instantiation operations, once all output model elements have been allocated. It aims to resolve references between input model elements. Indeed, at the instantiation stage, the output model elements are initialized by executing the assignments specified in the output patterns of each rule. With the ATL language, output model element properties are computed from input model element ones. In case the considered input property is a reference to another input model element, it has to be resolved into the corresponding allocated output model element. This corresponding output model element is the default output element generated by the rule that has matched the input model element.

The information that is required to achieve the resolution is stored within the transient link set associated with the *links* field of the ATL module. The *getLinkBySourceElement* operation (defined in the context of a transient link set) makes it possible to obtain a transient link from its source model element. The *getTargetFromSource* operation, defined in the context of a transient link, then enables to get the transient link default target element from its source element.

The *__resolve__* operation accepts an object as parameter and returns an object. The parameter is a property of an input model element whereas the returned value is the corresponding data in the output model:

- When the parameter is a simple attribute (i.e. as opposed to a model element reference), *__resolve__* returns the value of this parameter.
- If the parameter is a reference to an input model element, the reference is resolved and the corresponding output element is returned. In case the rule that matches the pointed input model element produces more than one output model element, the *__resolve__* operation returns an instance of the first output element of the considered rule (which is considered as the default output model element).
- Finally, if the parameter is an instance of a collection type, the operation returns a collection. The returned collection is build by recursively calling the *__resolve__* operation on each element of the collection provided as a parameter.

As opposed to the *__resolve__* operation, which may only be invoked from the code being generated by the compiler, the *resolveTemp* operation is currently defined in the ATL language. In an ATL file, this operation can be called from the instantiation expressions of the rules output patterns (and from helpers invoked from such expressions). As *__resolve__*, the *resolveTemp* operation aims to resolve references between input model elements during the instantiation step. However, compared to *__resolve__*, this new operation also allows resolving references to non-default output model elements of the rules. For this purpose, it accepts an additional string parameter that identifies the output model element the reference has to be resolved to. This parameter is passed to the *getNamedTargetFromSource* operation (defined in the context of a transient link) in order to get an identified output model element (as opposed to the default one) of the transient link.

7 Future developments

In this document, we have presented the ATL virtual machine specification along with an XML-based byte code file format (the `asm` file format). In order to illustrate these specifications, we detailed in Section 6, a possible compiling scheme of the ATL language to the `asm` file format.

In this section, we introduce a set of possible future evolutions for both the ATL virtual machine and the `asm` file format. These possible evolutions have been sorted into two groups: those concerning the ATL virtual machine instruction set and those that are related to the `asm` file format.

7.1 Evolving the ATL Virtual Machine Instruction Set

The ATL virtual machine instruction set was primarily designed to provide an execution support for the ATL transformation language. This has obviously influenced the initial design of the virtual machine instruction set which is not large enough to support a broad set of languages. In particular, the ATL virtual machine instruction set does not define any object deletion facilities. On the other hand, some other instructions, which are supported in an inefficient way, should be redefined to improve performance. We describe, from these observations, a couple of possible future evolutions for the ATL virtual machine instruction set.

7.1.1 Support for Object Deletion

Since the ATL language does not require any object deletion facilities, the virtual machine instruction presently does not define any deletion instruction. However, object removal facilities are required by many other languages specifications and a *delete* instruction should be added soon to the ATL virtual machine instruction set. This *delete* instruction will most likely follow the following specification:

Operation

Delete an object.

Format

delete

Operand Stack

..., reference ⇒ *...*

Description

The object to be removed may be either a model element or an instance of a composite data type. The referenced object is deleted. The *reference* is popped from the operand stack.

7.1.2 Improvement of *new* and *findme* instructions

The *new* and *findme* instructions have a very similar behaviour. Both return an object reference. With the *findme* instruction, the returned object is to be a model element. Both accept two parameters which define the metamodel and the classifier (or type) of the object to be allocated/fetched. These parameters must be available on the top of the operand stack of the current frame. As a consequence, executing a *new* or a *findme* instruction implies the former execution of at least two additional instructions in order to compute/push/load the *new*/*findme* parameters onto the operand stack. Providing the instruction with the full type of the object to be allocated/fetched as an inline parameter would be a more efficient solution. The instructions of the ATL virtual machine instruction set are however limited to an only inline argument. The proposed solution then requires both the object metamodel and type to be encoded within a single argument. This could be achieved by means of a string-based type encoding similar to the data type encoding scheme defined for the `asm` file format (see Section 5.2.1). Thus, with respect to the `asm` type encoding scheme, the allocation a new `Node` model element of the `XML` metamodel would be performed by means of a single instruction: `<new arg="MXML!Node;">`. This *new* instruction will most likely follow the following specification:

Operation

Create new object.

Format

New
type-encoding

Operand Stack

... ⇒ ..., reference

Description

This instruction is used to create new model elements and new composite objects. The type of the object (the element) is resolved from the inline *type-encoding* argument. If resolution completes successfully, memory for the new object instance is allocated. As a result, *type-encoding* is popped from the operand stack and replaced by a *reference* to the allocated object.

7.2 Evolving the `asm` File Format

As the ATL virtual machine instruction set, the `asm` file format has been primarily designed to encode compiled ATL transformations. At the present time, it is already generic enough to support the encoding of different languages. However, some points may still be improved in order to provide language compilers with even more encoding options.

The `asm` format makes it possible to define a set of fields. These fields are used to store variables. As opposed to the operations, they currently cannot be associated with a specific context. As a matter of fact, they all have to be associated with a default context. In the scope of the ATL language, this context is the ATL module. It may be interesting to allow the declaration of fields in different context. For this purpose, the context in which a field is defined has to be encoded into the `asm` format. This context may be encoded in the same way that the operation context, as illustrated by the following extract of then `asm` structure:

```
<!DOCTYPE asm [
  ...
  <!ELEMENT field (context)>
  <!ELEMENT context EMPTY>
  ...
  <!ATTLIST field name CDATA>
  <!ATTLIST field type CDATA>
  <!ATTLIST context type CDATA>
]>
```

With such a definition, the *col* field generated for an ATL transformation (see Section N) would be encoded by the following `asm` structure:

```
<field name="3" type="4">          <!-- col : J; -->
      <context type="8"/>          <!-- A -->
</field>
```

Encoding the context of fields into the `asm` format would provide support for some possible ATL language extensions. We may, for instance, allow ATL programmers to associate attribute helpers with input model elements. These attribute helpers would therefore be encoded in `asm` as fields that would be defined into the context of input model elements.

Finally, the `asm` format does not encode the operations return type. As this kind of information may be useful, it is possible to include it into the operation definition along with the operation context and parameters. As for the context element, the only required information is the type of the returned value. This information could be encoded in the `asm` structure in the following way:

```
<!DOCTYPE asm [
```

```
  ...
  <!ELEMENT operation (context, parameters, return, code, linenumbertable,
localvariabletable)>
  <!ELEMENT context EMPTY>
  <!ELEMENT parameters (...)>
  <!ELEMENT return EMPTY>
  <!ELEMENT code (...)>
  <!ELEMENT linenumbertable (...)>
  <!ELEMENT localvariabletable (...)>
  ...
  <!ATTLIST return type CDATA>
]>
```

8 References

[1] OMG/MOF Meta Object Facility (MOF). Version 1.4. formal/2002-04-03, 2002.

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

[3] OMG/OCL Specification, ptc/03-10-14. October 2003.

[4] SUN, netBeans project, MDR - Metadata Repository. http://mdr.netbeans.org/.

[5] F. Budinsky, and D. Steinberg, and E. Merks, and R. Ellersick, and T. J. Grose: Eclipse Modeling Framework. Addison-Wesley.

[6] F. Budinsky, and D. Steinberg, and E. Merks, and R. Ellersick, and T. J. Grose: Eclipse Modeling Framework, Chapter 5 *Ecore Modeling Concepts*, Addison-Wesley.

[7] OMG/XMI XML Model Interchange (XMI) OMG Document AD/98-10-05, October 1998.

I. `asm` File DTD

```
<!DOCTYPE asm [

  <!ELEMENT asm (cp, field*, operation+)>
  <!ELEMENT cp (constant*)>
  <!ELEMENT constant EMPTY>
  <!ELEMENT field EMPTY>
  <!ELEMENT operation (context, parameters, code, linenumbertable,
localvariabletable)>
  <!ELEMENT context EMPTY>
  <!ELEMENT parameters (parameter*)>
  <!ELEMENT parameter EMPTY>
  <!ELEMENT code (push, pushi, pushd, pushf, pusht, pop, store, load, swap, dup,
dup_x1, if, goto, iterate, enditerate, call, new, get, set, findme, getasm)*>
  <!ELEMENT push EMPTY>
  <!ELEMENT pushi EMPTY>
  <!ELEMENT pushd EMPTY>
  <!ELEMENT pushf EMPTY>
  <!ELEMENT pusht EMPTY>
  <!ELEMENT pop EMPTY>
  <!ELEMENT store EMPTY>
  <!ELEMENT load EMPTY>
  <!ELEMENT swap EMPTY>
  <!ELEMENT dup EMPTY>
  <!ELEMENT dup_x1 EMPTY>
  <!ELEMENT if EMPTY>
  <!ELEMENT goto EMPTY>
  <!ELEMENT iterate EMPTY>
  <!ELEMENT enditerate EMPTY>
  <!ELEMENT call EMPTY>
  <!ELEMENT new EMPTY>
  <!ELEMENT get EMPTY>
  <!ELEMENT set EMPTY>
  <!ELEMENT findme EMPTY>
  <!ELEMENT getasm EMPTY>
  <!ELEMENT linenumbertable (lne*)>
  <!ELEMENT lne EMPTY>
  <!ELEMENT localvariabletable (lve+)>
  <!ELEMENT lve EMPTY>

  <!ATTLIST asm name CDATA>
  <!ATTLIST constant value CDATA>
  <!ATTLIST field name CDATA>
  <!ATTLIST field type CDATA>
  <!ATTLIST context type CDATA>
  <!ATTLIST parameter name CDATA>
  <!ATTLIST parameter type CDATA>
  <!ATTLIST push arg CDATA>
  <!ATTLIST pushi arg CDATA>
  <!ATTLIST pushd arg CDATA>
  <!ATTLIST store arg CDATA>
  <!ATTLIST load arg CDATA>
  <!ATTLIST if arg CDATA>
  <!ATTLIST goto arg CDATA>
  <!ATTLIST call arg CDATA>
  <!ATTLIST get arg CDATA>
```

```
  <!ATTLIST set arg CDATA>
  <!ATTLIST lne id CDATA>
  <!ATTLIST lne begin CDATA>
  <!ATTLIST lne end CDATA>
  <!ATTLIST lve slot CDATA>
  <!ATTLIST lve name CDATA>
  <!ATTLIST lve begin CDATA>
  <!ATTLIST lve end CDATA>

]>
```

II. `asm` code of `__matchManual2Manual` operation

```
<operation name="27">  <!-- __matchManual2Manual -->
  <context type="8"/>  <!-- A -->
  <parameters>
  </parameters>
  <code>
      <push arg="28"/> <!-- Manual -->
      <push arg="29"/> <!-- BibTeX -->
      <findme/>
      <push arg="30"/> <!-- Sequence -->
      <push arg="11"/> <!-- #native -->
      <new/>
      <swap/>
      <dup_x1/>
      <push arg="31"/> <!-- IN -->
      <call arg="32"/> <!-- MMOF!Classifier;.allInstancesFrom(S):QJ -->
      <call arg="33"/> <!-- CJ.union(CJ):CJ -->
      <swap/>
      <pop/>
      <iterate/>
      <store arg="34"/><!-- 1 -->
      <pusht/>
      <call arg="35"/> <!-- B.not():B -->
      <if arg="36"/>   <!-- 37 -->
      <load arg="9"/>  <!-- 0 -->
      <get arg="1"/>   <!-- links -->
      <push arg="37"/> <!-- TransientLink -->
      <push arg="11"/> <!-- #native -->
      <new/>
      <dup/>
      <push arg="38"/> <!-- Manual2Manual -->
      <call arg="39"/> <!-- NTransientLink;.setRule(MATL!Rule;):V -->
      <dup/>
      <push arg="40"/>       <!-- i -->
      <load arg="34"/>       <!-- 1 -->
      <call arg="41"/> <!-- NTransientLink;.addSourceElement(SJ):V -->
      <dup/>
      <push arg="42"/> <!-- o -->
      <push arg="28"/> <!-- Manual -->
      <push arg="29"/> <!-- BibTeX -->
      <new/>
      <call arg="43"/> <!-- NTransientLink;.addTargetElement(SJ):V -->
      <call arg="44"/> <!-- NTransientLinkSet;.addLink(NTransientLink;):V -->
      <enditerate/>
  </code>
  <localvariabletable>
      <lve slot="1" name="40" begin="14" end="36"/><!-- i -->
      <lve slot="0" name="20" begin="0" end="37"/><!-- self -->
  </localvariabletable>
</operation>
```

III. `asm` code of `__applyManual2Manual` operation

```
<operation name="161">       <!-- __applyManual2Manual -->
  <context type="8"/>        <!-- A -->
  <parameters>
      <parameter name="34" type="162"/><!-- 1 : NTransientLink; -->
  </parameters>
  <code>
      <load arg="34"/>       <!-- 1 -->
      <push arg="40"/>       <!-- i -->
      <call arg="163"/>      <!-- NTransientLink;.getSourceElement(S):J -->
      <store arg="54"/>      <!-- 2 -->
      <load arg="34"/>       <!-- 1 -->
      <push arg="42"/>       <!-- o -->
      <call arg="164"/>      <!-- NTransientLink;.getTargetElement(S):J -->
      <store arg="100"/>     <!-- 3 -->
      <load arg="100"/>      <!-- 3 -->
      <dup/>
      <load arg="9"/>        <!-- 0 -->
      <load arg="54"/>       <!-- 2 -->
      <get arg="79"/>        <!-- title -->
      <call arg="55"/>       <!-- A.__resolve__(J):J -->
      <set arg="79"/>        <!-- title -->
      <dup/>
      <load arg="9"/>        <!-- 0 -->
      <load arg="54"/>       <!-- 2 -->
      <get arg="165"/>       <!-- id -->
      <call arg="55"/>       <!-- A.__resolve__(J):J -->
      <set arg="165"/>       <!-- id -->
      <pop/>
  </code>
  <localvariabletable>
      <lve slot="2" name="40" begin="3" end="21"/><!-- i -->
      <lve slot="3" name="42" begin="7" end="21"/><!-- o -->
      <lve slot="0" name="20" begin="0" end="21"/><!-- self -->
      <lve slot="1" name="172" begin="0" end="21"/><!-- link -->
  </localvariabletable>
</operation>
```