Achieving QVTO & ATL Interoperability

An Experience Report on the Realization of a QVTO to ATL Compiler

Alfons Laarman
a.w.laarman@ewi.utwente.nl

Software Engineering Group, University of Twente, The Netherlands

Abstract. With the emergence of a number of model transformation languages the need for interoperability among them increases. The degree at which this interoperability can be achieved between two given languages depends heavily on their paradigms (declarative vs imperative). Previous studies have indicated that the QVT and ATL languages are compatible. In this paper we study the possibility to compile QVT Operational to the ATL virtual machine. We describe our experience of developing such a compiler. The resulting compiled QVT transformations can run on top of existing ATL tools. Thereby we achieve not only QVT/ATL interoperability but also QVT conformance for the ATL tools as defined in the QVT specification.

1 Introduction

Model Driven Engineering (MDE) is an emerging approach for software development gaining more and more attention by the industry and the academia. MDE emphasizes the need for thorough modeling of software systems before their implementation. Implementations can be derived from their models by applying model transformations, possibly in a fully automated way.

Since the adaptation of MDE, several model transformation languages have been defined. As their tools grow more mainstream, the need for interoperability among them increases. Each of these languages has its own application domain and therefore software developers should be able to compare and select the languages for their particular problem. In some cases it might be desirable to combine several transformations in different languages with each other. The degree at which this interoperability can be achieved depends heavily on the language paradigm (declarative vs imperative) [9].

QVT is a defacto standard specification for model transformations with multiple possibilities for implementation. Current tools employ various approaches like compilation to Java (SmartQVT [14]), interpretation (ProceduralQVT [13]), etc. However, the use of these implementations is limited because of their youth [11]. Thus, for practical reasons a user might want to use other languages with better tool support, for example ATL, which is based on a virtual machine architecture. It may be beneficial to compile QVT programs to the ATL virtual

machine. Potential benefits of this form of interoperability are: reusing QVT programs on top of the ATL virtual machine, reusing ATL tools, claiming QVT compliance for the ATL tools and comparing ATL and QVT programs.

In this work, we report our experience of implementing a QVT to ATL compiler. We describe here a design for the compiler, which has been implemented as a proof of concept. This solution will provide significant QVT/ATL interoperability and will prove ATL tools to be QVT Operational conformant.

In the next section, we will state the problem with the current situation in MDE. The following sections will provide background for QVT, ATL and associated tools and languages. In Section 4, the approach, we explain the development method and the technology used to implement the compiler followed by the description of the design and implementation. To test the compiler some example transformations were used. These examples are presented in the discussion section together with the problems we encountered during implementation and with future work. In the last section, we draw conclusions.

2 Problem Statement

With the emergence of a number of model transformation languages the need for interoperability among them increases. Each of these languages has an own application domain and therefore software developers should be able to compare and select the languages for their particular problem. In some cases it might be desirable to combine several transformations in different languages with each other. The degree at which this interoperability can be achieved depends heavily on the language paradigm (declarative vs imperative) [9].

Interoperability between QVT and ATL is desirable for additional reasons: QVT is a defacto standard for model transformations and ATL is not only designed to support QVT transformation scenarios but goes beyond QVT context. ATL does this by supporting scenarios where source and target models are artifacts created with various technologies such as databases, XML documents, etc [11]. Furthermore ATL provides tool support, whereas direct QVT implementations are still in a phase of infancy [11].

Both ATL and QVT specify several languages distributed across multiple layers. Based on language features Czarnecki and Helsen describe similarities between ATL and QVT [4]. Jouault and Kurtev believe ATL and QVT to be interoperable with each other [9]. They made a detailed comparison between the languages using following language properties:

- Relative abstraction level
- Transformation scenarios (model transformation, synchronization and conformance checking)
- Paradigm (declarative, imperative or both)
- Directionality (multidirectional or unidirectional)
- Cardinality (M-to-N or M-to-1)
- Traceability (automatic or user-specified)

There are three different QVT languages, which all differ from the ATL language according to these criteria. Finally they conclude that QVT Operational (one of the three languages specified by [3]) has to highest potential to be interoperable with ATL.

Thus a transformation from QVT Operational (QVTO) to ATL virtual machine can be implemented at relatively low cost. Concrete implementation details for such a transformation are however not provided in [9]. Therefore the hypotheses of [9] still requires investigation. We provided this proof by realizing a compiler implementation.

3 Background

3.1 QVT Architecture

The Meta Object Facility Query/View/Transformation (MOF QVT) specification [3] is the solution for model transformations in the OMG modeling framework. It is designed to be a standard and does not provide a reference implementation. In the language dimension of the QVT specification we find three different languages: QVTO, QVT Relations (QVTR) and QVT Core. Next to the language dimension a conformance dimension is defined. A tool designer can use this dimension to give a degree of QVT conformance to his transformation language implementation.

QVTO is a completely imperative language, which only supports model transformation scenarios in an unidirectional M-to-N fashion. Mapping operations (see Listing 1) perform the central task of producing output model elements from input model elements. They can be defined on model elements making in an object-oriented language.

```
1 transformation Uml2Rdb(in srcModel:UML, out destModel:RDBMS);
2 main() {
3   srcModel.objects()[Package]->map package2schema();
4 }
5 mapping Package::package2schema() : Schema { .... }
```

Listing 1. Example QVTO code

Traceability is automatic in QVTO [9]. Several language constructs are provided to request trace information in the transformation design. But the execution semantics also rely on the traceability information. For example, if an operation has been executed before with the same input parameters, the second execution will not create any new target model elements, instead it will return those already created. The abstractness of the language is almost at the same level as ATL but probably a bit lower [9].

3.2 ATL Architecture

AtlanMod Transformation Language (ATL) is a modeling platform (language and tools) developed by AtlanMod (INRIA-EMN) [10,13]. The architecture of

ATL mimics the Java Virtual Machine (VM) architecture. First the ATL program is parsed into a model representation, then this is compiled into the assembly format which can be executed by the ATLVM (see Fig. 1).

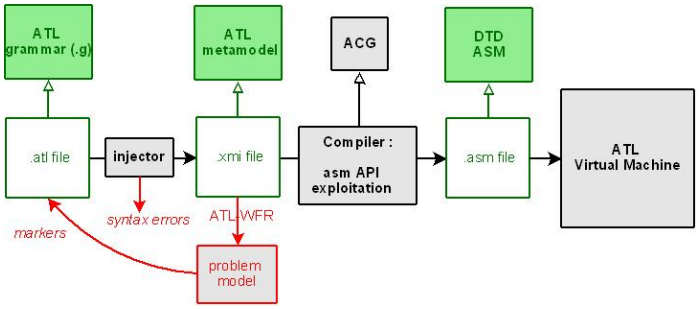

Fig. 1. ATL compilation process (taken from [13])

ATLVM executes an assembly language much like Java byte code and handles models on the basic level of querying and creating model elements [12]. Its paradigm is imperative. Assemblies can contain (virtual) operations which can be defined on model elements (their context), which makes the VM object oriented. The data types handled by the ATLVM consist of the basic primitive types, composite types like in OCL [5] and model element types. The ATLVM is stack-based and understands three kinds of instructions; Operand stack (*push, pop, load, swap and store*), Control (*if, goto, iterate and call*) and Model handling (*create, fetch and get*).

The ATLVM is the basis of the current ATL implementation and is also the target language of the compiler described in this paper. Listing 2 shows an example ATLVM operation named "container" on the element Object from the QVT metamodel. When called, this operation will load the contextual parameter on the stack (line 2) which will subsequently be used as contextual parameter of the operation call "refImmediateComposite()" (line 3).

```
1 context QVT!Object def: container() {
2    load self;
3    call 'J.refImmediateComposite():J';
4 }
```

Listing 2. Example ATLVM code

3.3 DSL Support from ATL

The AtlanMod Model Management Architecture (AMMA), which ATL is part of, also contains some tools for the creation of domain specific languages (DSLs). Language creation can be done by expressing a syntax (abstract and concrete) and sematics. AMMA solves these subtasks using MDE [6]:

– the abstract syntax is captured in a metamodel,
– the concrete syntax is represented as a transformation which maps the concrete syntax elements to the abstract syntax,
– and the semantics for DSL_A can be expressed as a transformation mapping DSL_A constructs to those of an existing DSL_B.

For each of these, AMMA provides a dedicated language:

– KM3 provides a formalized way to specify metamodels with only the most basic concepts [7],
– TCS can specify the textual concrete syntax of a language in terms of the KM3 metamodel [8],
– and lastly ATLVM Code Generator (ACG) is a transformation language which maps the KM3 metamodel elements onto the ATLVM.

The ATL language is also written in KM3, TCS and ACG [6]. The process of executing an ATL definition involves two steps: (1) parsing of the definition's syntax using TCS and (2) running the ACG definition on the resulting model (which conforms to the ATL metamodel as expressed in KM3).

Contrary to what Fig. 1 suggests, the ACG definition is not executed directly. Fig. 2 shows how ACG definitions are compiled to (ATLVM) assemblies. For this reason the performance of compiling ATL transformation definitions is more than satisfactory. Even on the largest existing ATL transformations [2], the compilation is done almost instantly when the definition is edited and saved.

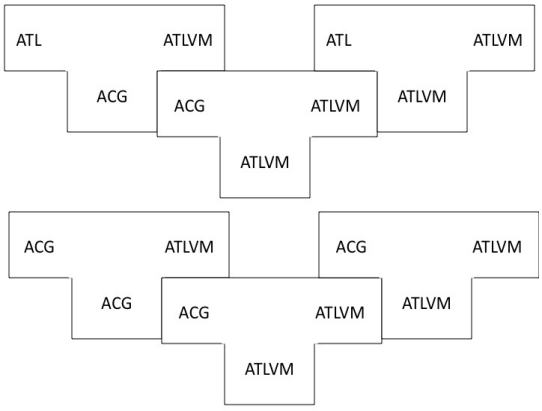

Fig. 2. ACG translation and bootstrap

4 Approach

By implementing a QVTO to ATL compiler, we can achieve interoperability and establish ATL conformance to QVT. Such a compiler can be built using any

compiler design framework/technology of preference. In the previous section, we showed how the AMMA tools can be used to define DSLs that can be executed in ATL environments (read: on the ATLVM). Since these tools have proven to provide a satisfactory performance for ATL and integrate needly into the MDE environment, they are the primary candidate for implementing the compiler.

With AMMA the implementation comes down to defining a metamodel and grammar for the syntax and expressing the semantics in ACG. It is a straightforward task to define the QVTO syntax in KM3 and TCS. In fact, we even skipped implementing the syntax and used the parser and metamodel from SmartQVT[14], which conforms to the QVT specification.

Thus, we only had to implement the semantics in ACG and use that as the QVT transformation compiler (see Fig. 3, QVTO2ATLVM[1]). The transformation definition on the left is the model of a parsed QVTO transformation. The resulting assembly (transformation definition on the right) could then directly be executed on the ATLVM just like ATL transformations. Moreover, compatibility with ATL transformations can be achieved through means of superimposition [15]. In effect, this approach is not subordinate to a QVTO to ATL solution (with subsequent step to ATLVM). It is less complicated, however, since the ATLVM is at a lower abstraction level [9].

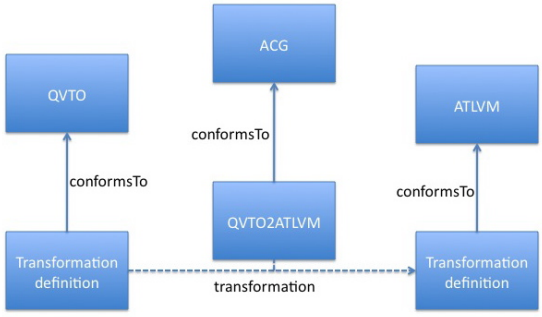

Fig. 3. The QVTO to ATLVM compiler in ACG

The compiler semantics can be derived from the QVT specification. Like any specification this one contains some points which are ambiguous or incomplete. Whenever we encountered impreciseness in the specification we tried to resolve it by discussing it with the communities of other (open source) implementations (especially [13]). The results of this process are presented in Section 6.3. We aimed to implement the full QVTO specification. To verify whether we achieved this goal, a representative set of QVTO transformations should run on the compiler. To this end the demo transformations from [14] are used.

[1] For simplicity, again we omit the fact that the ACG transformation is transformed to ATLVM first

5 QVTO to ATL Compiler

In this section, we will detail the QVTO to ATL compiler. Since ACG is a very specific syntax [1], we will refrain from listing code samples in ACG. To illustrate the way in which we defined the QVT semantics in ATL we will use pseudo code which abstracts from the stack-based virtual machine implementation by using variables. Like any good compiler implementation, the compiler will provide the semantics for each construct of QVTO (or each type in the abstract syntax tree (AST)). So the pseudo code will define templates for QVTO constructs.

Listing 3 shows such a template for the construct *MappingOperation*. It will be executed for each *MappingOperation* **m** found in the AST of the QVTO program that is being compiled. When executed, a virtual machine operation will be emitted (line 2). It will have a context, a name and parameters, which are all derived from **m** using OCL queries (in bold). For understanding these queries it is highly recommended to read about the QVTO abstract syntax in [3].

In pseudo code, we sometimes summarize large computations with natural language in italics. If these operations are not trivial, they will be detailed in a subsequent listing. Square brackets are used inside the pseudo code to indicate code generation from queried model elements. So line 3 will call the template for whatever construct is found in the when property of **m**. The benefit of the pseudo code is that it links compile and runtime using variables and thereby it can represent the compiler internals compactly. For example, on line 3, the result of the square brackets is emitted code (compile time) and the result from executing this code (runtime) is subsequently assigned to the variable "executable". In the accompanied text to the code listings, QVT constructs are written in *italics*.

```
1  MappingOperation m {
2     operation context m.context name m.name params m.ownedParameter  {
3        executable = [ m.when ]
4        if executable then
5           result = [ m.bodySection ]
6        else
7           strict = [ m.calledStrict ]
8           if strict
9              Raise exception
10       endif
11       return result
12    }
13 }
```

Listing 3. Example pseudo code

The following subsections will provide pseudo code templates for all QVTO constructs except those that have relatively simple semantics and can easily be handled with low-level ATLVM instructions.

5.1 OperationalTransformation

An *OperationalTransformation* represents a model transformation. It has a name and a set of model parameters (*in*, *out* and *inout*) as its interface. This concept can directly be represented using the ATLVM assembly language. The ATLVM

has built-in library support and can combine several of these transformations, like the QVT semantics prescribe. Since the ATLVM supports M-to-N transformations all the model parameters can be handled (*in*, *out* and *inout*). *IntermediateProperties* and *IntermediateModels* are features of transformations that can be used as structure for temporary data storage at transformation execution. They can easily be mapped to (assembly-language) global variables of a corresponding type in the ATLVM.

5.2 MappingOperation

A *MappingOperation* is responsible for mapping input model elements to output model elements. Its semantics are sensitive to trace information. Therefore to store and retrieve trace information we created several methods that the compiler generates for each compiled transformation. The interface used to query this information is shown in Table 1. The Trace object that is returned by the "findTrace" functions has members for the called mapping, the inputs, the outputs and the returned result.

Table 1. Interface for trace information

Function	**Description**
Trace:findTrace(*inputs*, [*reversed*])	returns the set of outputs (if any) that are mapped to the inputs or vice versa (if reversed=true)
Trace:findTrace(*mapping*, *inputs*, [*reversed*])	returns the set of outputs (if any) that are mapped by *mapping* to the inputs or vice versa (if reversed=true)
putTrace(*mapping*, *inputs*, *outputs*)	stores a trace of *inputs* to *outputs* in *mapping*

Depending on the trace information a *MappingOperation* can be combined in several ways with other mapping operations (*inheritance*, *disjunction* and *merging*). It can also have a precondition (*when* clause) and/or post condition (*where* clause). A mapping operation should have at least one input (a *context* or *in* parameter[2]) and one *out* parameter (*result*). In addition, it may contain several other *in*, *out* or *inout* parameters.

Since ATLVM and QVTO share the imperative object-oriented paradigm, mapping operations can be directly mapped on assembly operations. The return value of assembly operations can be used to handle *result*. The values of other *out* and *inout* parameters will have to be passed back to the calling operation by global assembly variables (this will be discussed in more detail later). In Listing 4, we show the general structure of an assembly operation that implements the semantics of a *MappingOperation* **m**.

[2] There is an ambiguity in the specification which we will discuss in Section 6.3

```
1  operation context m.context name m.name params m.ownedParameter {
2     Check parameter type conformance
3     Check and call disjunctions
4     executable = [ m.when ]
5     if executable then
6        trace = findTrace(m, m.ownedParameter.select(type=in))
7        if trace
8           return trace.result
9        Retrieve inherited values for out parameters
10       [ m.initSection ]
11       Instantiate uninitialized model elements for all out parameters
12       putTrace(m,  m.ownedParameter.select(type=in),
13                    m.ownedParameter.select(type=out))
14       Call inherited operations and assign to result
15       result = [ m.bodySection ]
16       [ m.endSection ]
17       Call merged operations and assign to result
18       executable = [ m.where ]
19       if !executable
20          Raise exception
21    else
22       strict = [ m.calledStrict ]
23       if strict
24          Raise exception
25    endif
26    return result
27 }
```

Listing 4. Definition of the semantics of a mapping operation

Disjunction is a way of combining several operations with distinctive preconditions. The semantics of executing disjunctive operations (line 3) are defined in Listing 5.

```
1  for each disjuction in m.disjunction do
2     executable = [ disjunction.when ]
3     if executable then
4        result = call disjunction
5        return result
6     endif
7  endfor
```

Listing 5. Definition of disjunction semantics

Semantics for operation inheritance require the inheriting operation to pass the values of its out parameters to the operation it inherits from. This is done on line 9 and 14. The value passing can be implemented using global variables or by adding extra (hidden) parameters to the operation. The following pseudo code does not discriminate against either method and refers to the passed values as a property inheritedValue (Listings 6 and 7). Merging semantics can be defined analogously.

```
1  if m.inheritedValue
2     m.result = m.inheritedValue
```

Listing 6. Retrieve inherited values for out parameters

```
1  for each inherited in m.inherited
2     inherited.inheritedValue = m.result
3     result = call inherited
4     m.result = result
5  endfor
```

Listing 7. Call inherited operations and assign to result

For the sake of simplicity only the result parameter is considered as out parameter. This can be easily extended to all output parameters by using a composite type. Also we did not take into account multiple result values. This would require value exchanges between operation callee and caller (also for *inheritance, merging* and *disjunction*) and can be implemented using global variables or composite values in the return value.

5.3 ImperativeCallExp

An *ImperativeCallExp* calls operations. Multiple *result* values and values of *out/inout* parameters can be passed as described in the above paragraph. They should however be processed after an operation call completes. Such a processing would involve decomposing the composite type or global variables and assigning the values to the appropriate variables in the operation call.

5.4 HelperOperation, ConstructorOperation and EntryOperation.

The semantics of these specialized operations do not include any other concepts than the mapping operations, therefore their definition can all be derived from the definition of the mapping operation. Except for an additional constraint of the helper operation, which states that no model element can be created in the output models. This constraint can be implemented using an OCL query on the model of the QVTO program at compile time.

Operation overriding has been defined for all kinds of operations. We did not feel the need to implement the semantics, because the ATLVM already overrides operations with the same signature.

5.5 ResolveExp

A *ResolveExp* has several executing semantics that have in common a source element for which trace links are searched and resulting (zero or more) target element(s). So the following properties of resolve are optional.

- an *inMapping*, specifying through which operation the trace link should have been created.
- a *one* flag, indicating that only one result should be returned
- an *inverse* flag, indicating that the resolve is from target to source model
- a *deferred* flag, indicating that the resolving process should be delayed until after the transformation execution.

Listing 8 shows the definition which supports all these semantics for a resolve r.

```
1  ResolveExp r {
2     if r.deferred then
3        Store resolve parameters and assigned property
4        return null
5     endif
6     if r.inMapping then
7        result = findTrace(r.inMapping, r.source, r.inverse)
```

```
8       else
9           result = findTrace(r.source, r.inverse)
10      endif
11      if r.one
12          result = result.first()
13      Push result on top of the stack
14  }
```

Listing 8. Definition of resolve semantics

Line 3 handles deferred resolve semantics. Of course the result of this deferred resolve should end up somewhere in (one of) the target model at the end of the transformation execution. Therefore the QVTO specification defines the deferred assignment as the variable or model property assignment directly underneath the resolve operation[3].

The target (left-hand-side) of the deferred assignment needs to be stored for the delayed resolve execution. In the words of the specification: "the execution engine stores the following information for the future variable: the source object, the function representing the filtering expression and the property or the variable reference to be assigned." After the *EntryOperation* of the transformation is finished we can reuse the semantics definition in lines 6 to 13 on this stored information to execute the deferred resolve.

Listing 8 abstracts from the way the trace information is stored, like we already did in the mapping operation definition. Later we give the definition of the interface that we used so far and give possible solutions for storing of the trace information.

5.6 TryExp, RaiseExp, BreakExp and ContinueExp

Try blocks and raise expressions are the basis of an exception handling mechanism. Like the operation return (*ReturnExp*), these can be implemented using normal control instructions (*if* and *goto*) and extra checks around operation calls. A simplified typeless example of exception handling is shown in Listing 9. A similar approach can be used to implement both *BreakExp* and *ContinueExp*.

```
1  global exception = null;
2  operation context global name op1() {
3      call op2()
4      if (exception != null) Handle exception
5  }
6  operation context global name op2() {
7      call op3()
8      if (exception != null) goto endBody
9      ...
10 endBody: }
11 operation context global name op3() {
12     ...
13     exception = "Error in op3";
14     jump endBody
15     ...
16 endBody: }
```

Listing 9. Exception handling in the ATLVM

[3] For variable assignments this causes ambiguity (see Section 6.3)

6 Discussion

The definitions, described in the preceding section, have actually been implemented. In the next subsection, we evaluate the implementation. A list of successfully executed example transformations, illustrates to what extent the goals have been met. Thereafter the limitations of the current compiler are discussed. Finally the ambiguities in the QVT specification are explained and the possibilities for future work are examined.

6.1 Evaluation

The following example transformations were successfully executed on the ATL environment using the QVTO to ATL compiler. These examples use all of the expressive features of the QVTO language. Therefore, we can conclude that the goal of implementing a significant portion of the QVTO specification has been achieved.

- UML22RDBMS[4] is a classic MDT example that uses a large part of the QVTO constructs, including resolve and iteration expressions. It features configuration parameters like intermediate models and aliases. Intermediate properties and models are also used.
- UML2Ecore[4] features inheritance of mapping operations, complex OCL expressions, assertions, mapping calls with strict semantics and more.
- Ecore2EMOF[4] features many advanced QVTO constructs.
- An industrial case from Obeo features at least the same complexity as Ecore2EMOF.

Especially the last three examples show that the QVTO to ATL compiler can be used for industrial applications. Furthermore, Kurtev and Jouault [9] expected that the solution presented here could be achieved at "relatively low cost". Our experience confirms that this is indeed the case, since this effort was realized in the short time of one month by one person. Comparing this to the considerate amount of time that is normally spent on implementing solutions as complete as this, we consider their hypothesis validated.

6.2 Limitations

While we did not encounter any QVTO construct which could not be expressed in ATL, the compiler still has some limitations. Regarding the modeling architecture, ATL takes a different view of models and transformations than QVT. QVT specifies that transformation combination is a matter of the transformation specification, while ATL solves this in the engine. For example in QVT we have to specify the model types with complete reference to the model, while an ATL transformation only specifies a label which is binded by the engine. The same could be said about transformation extension (QVT) and superimposition

[4] From the SmartQVT project [14].

(ATL) [15]. To implement this in ATL we had to remove this information from the transformation definition and place it in the engines configuration.

These limitations had some effect on the examples shown in the previous subsection. The definitions had to be adapted to match the ATL architecture on the described points. However, this solution does not limit the functionality of the QVT to ATL compiler.

6.3 The QVT specification

Making an implementation of a specification allows to expose its peculiarities. This work will result in some feedback on the QVT specification. We were not able to find the following problems elsewhere (the numbers refer to sections in [3]):

8.2 It is not specified what would be the result of writing to an *in* parameter withing an *ImperativeOperation* and model element creation inside a *HelperOperation*.

8.2 There is no distinction between statements and expressions in QVT. A break is represented by an *BreakExp*, which indicates it is an expression in the metamodel. Semantically it behaves like a statement, this seems to be acknowledged by the specification by the following sentence: "A break expression ... is used in the body of imperative loop expressions (while and for expressions). A break expression cannot be directly owned by a non imperative expression; like the side-effect free OCL iterate expression." This does however not make *BreakExp* semantically a statement in all cases, *while* can be used inside OCL expressions. This makes compiler development more difficult, because inside expressions we cannot have access to the stack contents.

8.2.1.15 "A mapping operation is an operation implementing a mapping between one or more source model elements into one or more target model elements". While example A.2.4 defines a mapping without any input parameter.

8.2.1.20 "Unless isVirtual is true this invocation is virtual". Logically this should be "false".

8.2.1.22 Deferred assignments can have both a variable and a property as left-hand-side (target). In the case of a variable, the only meaningful semantics would be to find property assignments that use the variable. Otherwise the deferred resolve will not end up in the target model. This however would make the language partly declarative even though it is defined as "completely imperative" ([3] chapter 8). The same problem arises when a deferred assignment is used inside an expression.

8.2.2.8 *SwitchExp* is specified to be sensitive to a *ContinueExp* and *BreakExp*, but the semantics is not given.

8.2.2.8 The alternative notation for switch statements is not elaborated on.

6.4 Future Work

A QVTR to ATLVM compiler has also been implemented. It is included in the Eclipse Model to Model Project [13]. This compiler is not documented well enough yet to be included in this experience report.

Although not all the constructs from the QVTR specification are semantically supported, enough are present to interpret the SimpleUML to RDBMS reference example. One of the concepts still missing is collection template support. It has been delayed mainly because we do not a have practical example of its use yet. The *enforce* mode is on active development. The experience reveals the implementation to be feasible.

7 Conclusion

In this paper, we studied in detail the interoperability between QVT and ATL. We focused on QVT Operational. Because of the layered architecture of ATL that uses a virtual machine to execute transformations, we were able to define this imperative language on top of it. We have designed, implemented and described a compiler, based on model transformation techniques (ACG). The result is that we can now specify QVTO programs and run them in the ATL environment.

The examples show that the current compiler status can cope with industrial transformation demand. Our results show that ATL is QVT conformant as defined in the QVT specification [3]. At the same time the benefits of running multiple languages on top of one transformation execution engine can be enjoyed. The ATL engine provides a model encoding technologies independent view on models and this can now be used in QVT transformations. Furthermore QVT and ATL transformations could extent each other using predefined coupling mechanism like black boxes.

This status should allow a transformation designer to: interchangeably use the languages, fairly compare transformations in different languages and reuse the ATL tools. Furthermore, ATL can be considered QVT compliant, the compiler implementation is an extra proof of this. In our experience, the MDE paradigm, which the AMMA tool set provides, to parse and compile (actually transform) is very efficient for (transformation) language definitions. It was certainly able to compile a large extent of the QVT languages. Initial results from a QVTR to ATL compiler also confirm this [13].

8 Acknowledgements

First of all, I thank the people at Obeo for giving my an interesting assignment and allowing me to work with them. Former colleague Quentin Glineur helped by discussing the details of his work (Quentin worked on a QVT Relations counterpart). I am grateful for the many hours he could find in his busy life to help and for the kind patience he showed when explaining his (more difficult) work.

Ivan Kurtev offered his help for writing this paper. Due to his effort, this work could be realized and I am grateful for his advice and guidance. Frédéric Jouault gave useful comments while the QVTO compiler was being implemented.

References

1. Quentin Glineur Alfons Laarman and Freddy Allilaire. The acg wiki, http://wiki.eclipse.org/acg, Nov 2007.
2. Atl transformations zoo, http://www.eclipse.org/m2m/atl/atlTransformations/.
3. Wim Bast, Mariano Belaunde, Xavier Blanc, Keith Duddy, Catherine Griffin, Simon Helsen, Michael Lawley, Michael Murphree, Sreedhar Reddy, Shane Sendall, Jim Steel, Laurence Tratt, R. Venkatesh, and Didier Vojtisek. MOF QVT final adopted specification, Nov 2005. OMG document ttfamily ptc/05-11-01.
4. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.
5. Object Management Group. UML 2.0 OCL specification, Oct 2003. Final Adopted Specification (ptc/03-10-14).
6. F. Jouault, J. Bézivin, C. Consel, I. Kurtev, and F. Latry. Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD)*, Nantes, France, July 2006.
7. Frédéric Jouault and Jean Bézivin. Km3: a dsl for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, pages 171–185, Bologna, Italy, 2006.
8. Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs: a dsl for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM.
9. Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied omputing*, pages 1188–1195, New York, NY, USA, 2006. ACM.
10. Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Lecture Notes in Computer Science*, pages 128–138, Heidelberg, 2006. Springer Berlin.
11. Ivan Kurtev. State of the art of qvt: A model transformation language standard. pages 377–393, 2008.
12. Specification of the atl virtual machine, http://www.eclipse.org/m2m/atl/doc/ATL_VMSpecification%5Bv00.01%5D.pdf, nov 2005. Version 0.1.
13. The eclipse m2m project website, http://www.eclipse.org/m2m.
14. The smartqvt project, http://smartqvt.elibel.tm.fr, 'download section' – 'examples'.
15. Dennis Wagelaar. Composition techniques for rule-based model transformation languages. *Theory and Practice of Model Transformations*, pages 152–167, 2008.