On the Architectural Alignment of ATL and QVT

Frédéric Jouault, Ivan Kurtev
ATLAS Group,
INRIA and LINA,
University of Nantes
+33 (0) 2 51 12 58 08
{frederic.jouault | ivan.kurtev}@univ-nantes.fr

ABSTRACT

Transforming models is a critical activity in Model Driven Engineering (MDE). With the expected adoption of the OMG QVT standard for model transformation language it is anticipated that the experience in applying model transformations in various cases will increase. However, the QVT standard is just one possible approach to solving model transformation problems. In parallel with the QVT activity many research groups and companies have been working on their own model transformation approaches and languages. It is important for software developers to be able to compare and select the most suitable languages and tools for a particular problem. This paper compares the proposed QVT language and the ATLAS Transformation Language (ATL) as a step in the direction of gathering knowledge about the existing model transformation approaches. The focus is on the major language components (sublanguages and their features, execution tools, etc.) and how they are related. Both languages expose a layered architecture for organizing their components. The paper analyzes the layers and compares them according to various categories. Furthermore, motivations for interoperability between the languages and the related tools are given. Possible solutions for interoperability are identified and discussed.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Specialized Application Languages – *model transformation languages.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *data types, frameworks.* D.2.12 [**Interoperability**]: Tool and Language Interoperability.

General Terms

Standardization, Languages

Keywords

Model engineering, model transformations, QVT, ATL, interoperability

1. INTRODUCTION

Model transformation is a critical component of Model Driven Engineering (MDE). OMG recognized this and accordingly issued the Query/Views/Transformations (QVT) RFP [13] to seek an answer compatible with its MDA standard suite: UML, MOF, OCL, etc. Several formal replies were given by a number of companies and research institutions. They evolved during the last three years into a single proposal [15] by QVT-Merge group consisting of a large number of original submitters. This proposal should be adopted in the near future and we will refer to it as QVT in the rest of this paper. Some other proposals evolved in parallel to the OMG process. Atlas Transformation Language (ATL) [2] [9] is one of them.

ATL and QVT share some common features as they initially shared the same set of requirements defined in QVT RFP. However, actual ATL requirements have changed over time as this language matured. They will not be presented in this paper since they are not part of its focus.

Both ATL and QVT have a similar operational context shown in Figure 1. *Tab* is a transformation program which execution results in automatic creation of *Mb* from *Ma*. These three entities are all models conforming to *MMt*, *MMb*, and *MMa* MOF metamodels respectively. *MMt* corresponds to the abstract syntax of the transformation language. QVT and ATL provide their own metamodels defining their abstract syntaxes.

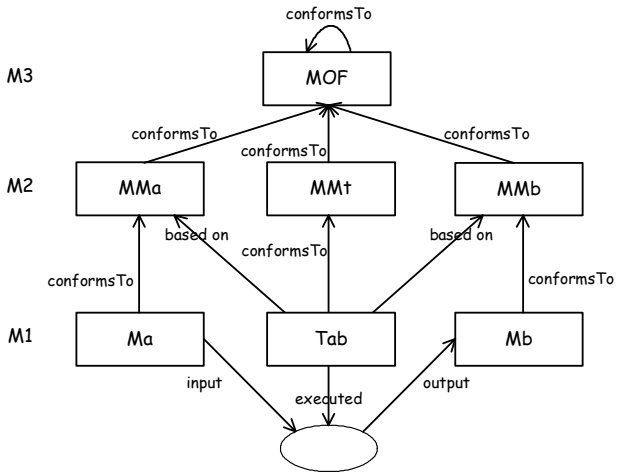

Figure 1 Operational context of ATL and QVT

The adoption of the QVT standard and the expected development of QVT tools should open a possibility for applying model transformations in more non-trivial cases. The experience gained from this should be used for improving the language. Furthermore, we witness other approaches for model transformations along with the QVT. A set of different languages should provide a broader field of experimentation. This motivates us for keeping working on ATL rather than turning it into a QVT implementation.

SAC'06, April, 23-27, 2006, Dijon, France.

As language engineering history has shown, no single language can be adapted to all application domains. We believe it is not different for model engineering and especially for model transformation. Moreover, although simple problems can often be solved using one language, more complex problems sometimes need several. Therefore, it is important to acquire knowledge about the strong and weak points of existing languages and their applicability.

As a first step in this direction we compare the ATL and QVT architectures and align them. This architectural alignment clarifies the possibilities for interoperability between the two languages and their tools. We try to achieve interoperability on the basis of model engineering principles, mainly using model transformations between ATL and QVT programs.

The results of our study showed that at a conceptual level it is possible to have interoperation between ATL and QVT based on model transformations on programs written in these languages. Since ATL and QVT provide more than one language level multiple transformations are possible. Not all of them are feasible though. Some are difficult (perhaps impossible) to be implemented due to potential conceptual and abstraction level mismatches. We identified two transformations that would provide useful capabilities at a relatively low price: QVT Operational Mappings to ATL virtual machine and ATL to QVT Operational Mappings.

The paper is organized as follows. Section 2 is a presentation of QVT architecture as described in the specification. Section 3 presents the three levels of ATL architecture. Section 4 shows how both architectures can be aligned. Section 5 proposes QVT-ATL interoperability solutions. Section 6 gives conclusions.

2. QVT ARCHITECTURE

The QVT RFP called for a language capable of expressing queries, views, and transformations over models in the context of MOF 2.0 metamodeling architecture. The QVT RFP was answered by several initial submissions that converged into a single proposal for the QVT standard. In this section we give an overview of the language by focusing on two issues: the language architecture and the conformance points for QVT tools.

2.1 QVT Layered Architecture

QVT operates in the layered MOF-based metamodeling architecture prescribed by OMG (Figure 1). The abstract syntax of QVT is defined as a MOF 2.0 metamodel. This metamodel defines three sublanguages for transforming models. OCL 2.0 [14] is used for querying models. Creation of views on models is not addressed in the proposal.

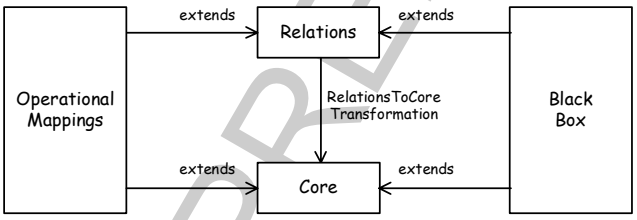

Figure 2 QVT languages layered architecture

The three QVT languages collectively form a hybrid transformation language with declarative and imperative constructs. The languages are named *Relations*, *Core*, and *Operational Mappings*. These languages are organized in a layered architecture shown in Figure 2.

The languages *Relations* and *Core* are declarative languages at two different levels of abstraction. The specification document defines their concrete textual syntax and abstract syntax. In addition, *Relations* language has a graphical syntax. *Operational Mappings* is an imperative language that extends *Relations* and *Core* languages. The three languages are presented briefly in the next section. More details are given in the discussion in section 4.

2.2 QVT Languages

Relations language provides capabilities for specifying transformations as a set of relations among models. Relations contain a set of object patterns. These patterns can be matched against existing model elements, instantiated to model elements in new models, and may be used to apply changes to existing models. The language handles the manipulation of traceability links automatically and hides the related details from the developer.

Core language is a declarative language that is simpler than the *Relations* language. Transformation definitions written in it tends to be longer than the equivalent definitions written in *Relations* language. Traceability links are treated as ordinary model elements. The developer is responsible for creating and using the links. One purpose of the *Core* language is to provide the basis for specifying the semantics of the *Relations* language. The semantics of the *Relations* language is given as a transformation *RelationsToCore*. This transformation may be written in the *Relations* language.

Sometimes it is difficult to provide a complete declarative solution to a given transformation problem. To address this issue the QVT proposes two mechanisms for extending the declarative languages *Relations* and *Core*: a third language called *Operational Mappings* and a mechanism for invoking transformation functionality implemented in an arbitrary language (*Black Box* implementation).

Operational Mappings language extends the *Relations* language with imperative constructs and OCL constructs with side effects. The basic idea in this language is that the object patterns specified in the relations are instantiated by using imperative constructs. In that way the declaratively specified relations are imperatively implemented in the language. The syntax of *Operational Mappings* language provides constructs commonly found in imperative languages (loops, conditions, etc.).

Black Box mechanism allows plugging-in and executing external code during transformation execution. This mechanism allows complex algorithms to be im plemented in any programming language and enables reuse of already existing libraries. This makes some parts of the transformation opaque, which brings a potential danger since the functionality is arbitrary and is not controlled by the transformation engine.

2.3 QVT Conformance Points

Figure 2 does not suggest any particular implementation of QVT transformation engine. Tool vendors may choose different strategies. For example, the *Core* language may be supported by an execution engine and the *Relations* transformations may be transformed to equivalent programs written in *Core* language. In that way the engine is capable of executing programs written in both

languages. Another possibility is that only the *Relations* and *Operational Mappings* are supported by a tool. In this case the *Core* language serves simply as a reference point for specifying the semantics of *Relations* language.

These implementation options may produce tools with different capabilities. To denote the capabilities of tools, the QVT proposal defines a set of *QVT conformance points* for tools. Conformance points are organized along two dimensions and form a grid with 12 cells. Table 1 shows the dimensions and the possible conformance points.

		Interoperability Dimension			
Language Dimension		Syntax Executable	XMI Executable	Syntax Exportable	XMI Exportable
	Core				
	Relations				
	Operational Mappings				

Table 1 QVT conformance points for tools

The *Language Dimension* defines three levels corresponding to the three QVT languages: *Core*, *Relations*, and *Operational Mappings*. If a tool conforms to a given level this means that it is capable of executing transformation definitions written in the corresponding language.

The *Interoperability Dimension* is concerned with the form in which a transformation definition is expressed. It defines four levels:

- **Syntax Executable.** A tool can read and execute transformation definitions written in the concrete syntax given in the QVT proposal;
- **XMI Executable.** A tool can read and execute transformation definitions serialized according to the XMI serialization rules (recall that transformation definitions conform to the QVT metamodel and therefore are XMI serializable);
- **Syntax Exportable.** A tool can export transformation definitions in the concrete syntax of the corresponding language;
- **XMI Exportable.** A tool can export transformation definitions in XMI format;

A requirement states that if a tool is *SyntaxExecutable* or *XMIExecutable* for a given language level, it should also be *SyntaxExportable* or *XMIExportable* respectively.

3. ATL ARCHITECTURE

ATL architecture is composed of three layers as shown in Figure 3. They are (in decreasing abstraction level order) ATLAS Model Weaving (AMW) [6], ATL, and ATL Virtual Machine (ATL VM).

ATL provides both declarative and imperative constructs and is therefore a hybrid model transformation language. Compiled ATL programs are executed by ATL VM, which uses a model-oriented instruction set. AMW may optionally be used as a higher abstraction level transformation specification language. These three layers are presented in the coming sections.

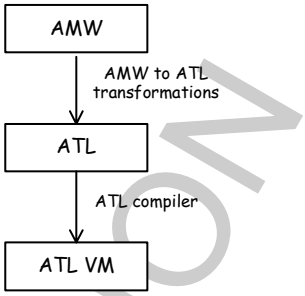

Figure 3 ATL layered architecture

We start by presenting ATL followed by an overview of the virtual machine because they are both central components of ATL model transformation architecture. AMW is not limited to model transformation and can be used in other contexts. It is presented in the last subsection.

3.1 ATLAS Transformation Language

Operational context of ATL corresponds to the scheme presented in Figure 1 with ATL metamodel standing in place of *MMt*. This metamodel corresponds to ATL abstract syntax. Transformation programs are, however, more frequently written using ATL concrete syntax. A parser and a serializer enable to easily obtain either one from the other.

The declarative part of ATL is based on the notion of *matched rule*. Such a rule consists of a source pattern matched over source models and of a target pattern that gets created in target models for every match. Traceability links are automatically created. Rule inheritance and polymorphic rule reference are available. Navigation is performed using OCL expressions.

ATL offers two imperative constructs: *called rule* and *action block*. A called rule is explicitly called, like a procedure, but its body may be composed of a declarative target pattern. Matched rules and called rules may be used together in a single transformation program. Action blocks are sequences of imperative instructions that can be used in either matched or called rules. The recommended style is declarative (i.e. no called rules and no action blocks). Imperative style should only be used when no declarative language construct provides the capabilities required by a particular case.

Transformation programs written in ATL are inherently unidirectional. Source models, which are only navigable (i.e. read-only), and target models, which are not navigable (i.e. write-only), are clearly identified at development time.

There are two modes in which the declarative part of an ATL program can operate: *standard* and *refining*. In standard mode, elements are only created when a rule is matched. However, since models cannot be transformed in-place (source models are read-only), transformations that only modify small parts of a model and leave most of the rest unchanged are complex to write in this mode. As a matter of fact, there must be roughly at least one copy rule for each type declared in the metamodel. This is not required in the refining mode where unmatched elements are automatically copied by the engine. In most cases, developers may assume they are actually modifying a source model with the difference that every navigation expression always operates on the original source model.

An execution engine and development tools (code editor, compiler, debugger, etc.) are available on GMT project web site [7].

3.2 Execution Support: ATL VM

We chose to base the current ATL execution engine on a virtual machine architecture as shown in Figure 4. The VM is implemented on top of two *model handlers* (i.e. libraries dealing with models): Eclipse Modeling Framework (EMF) [4] and Netbeans MetaData Repository (MDR) [12]. The VM could also be based on other model handlers as suggested by the "etc." box in Figure 4. ATL compiler works on top of ATL VM and generates ATL programs capable of running on top of it too.

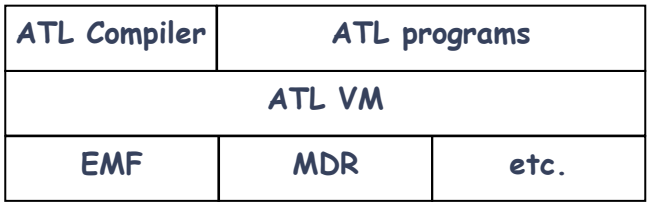

Figure 4 ATL VM architecture

There are several advantages of this approach:

- Extending ATL mostly requires changes in the compiler only;
- Adding the ability to deal with new kinds of models only involves changes in the virtual machine;
- Compiling other languages to ATL bytecode and having them benefit from the same virtual machine tools such as debugger and model handler drivers (e.g. MDR, EMF);

Of course, actual execution of ATL programs could also be implemented by direct interpretation (without intermediate bytecode) or compilation to native code.

ATL VM language is a small imperative instruction set composed of four categories of bytecodes: stack, memory, control flow, and model handling. There are currently only 21 different instructions. VM operations are especially adapted to OCL helpers implementation (see [14], section 7.4.4).

There is also specific support for traceability but it must be used explicitly. Traceability links are indeed not automatically created like they are in ATL. A draft specification of ATL VM is available on GMT [7].

3.3 ATLAS Model Weaving

We saw that ATL virtual machine provides a basic set of constructs, which are sufficient to perform automatic operations on models. Although ATL provides a higher level language for transformation definition it is sometimes necessary to express transformations in even more abstract terms. AMW provides solutions to this issue.

We believe application-independent transformation language abstract syntaxes are not enough for some applications. Some problem domains indeed require specific transformation concepts that cannot always match general purpose syntax. A similar issue was identified by Meyer in [11]. He suggested that the modular structure of a software system should be compatible with the model of the problem domain for which the system is built. In the context of model transformation, this means that transformation abstract syntax should match problem domain transformation concepts.

It should first be noted that *model weaving* [6] is different from *aspect weaving* [10]. Model weaving is about establishing typed links between model elements. Links themselves form a model and *link types* are therefore defined in a metamodel. Weaving links are more abstract than ATL rules because, whereas ATL and ATL VM have fixed semantics, AMW has user-defined semantics. Consequently, *link types* can be adapted to specific application domains. This provides a framework for defining solution domain concepts that match those of a given problem domain.

For instance, in [1] we presented a concrete use of AMW to represent and use mappings between metamodels and UML profiles. Several *link types* are used to: map metamodel classes and profile elements (i.e. UML classes and stereotypes), map their respective properties (attributes and tagged values), identify cases in which multiple elements on one side correspond to a single element on the other side, etc. These *link types* are specific to the problem of mapping metamodels and UML profiles. As a consequence, actual mapping descriptions can be relatively simple compared to their ATL equivalents. We also showed in [1] that such AMW mapping descriptions can be executed. To this end, they are transformed into ATL programs translating between metamodel-based and profile-based models.

The adaptability of AMW to different problem domains is achieved by providing tools working on a core weaving metamodel defining only the abstract notion of *link type*. This core metamodel can be extended by users. Metamodel extension is a complex operation, which will not be discussed here. The basic idea is that user-defined *link types* have to extend the abstract *link type* concept defined in the core. Transformations to ATL code can then be used to implement *link types'* semantics.

Furthermore, weaving semantics need not even be executable. This is because AMW application domain is actually broader than transformation specification. This is, however, out of the scope of this paper. More information may be found in [6]. An implementation of AMW is available on GMT web site [7].

4. ARCHITECTURES ALIGNMENT

In this section we compare ATL and QVT by aligning the components found in their layered architectures. The alignment of the two architectures aims at finding correspondences between components. However, these language components are complex entities that expose a set of diverse features. An equivalence relation may be established with respect to some features while with respect to others there may be a significant difference. To achieve an alignment we compare the language components on the basis of several categories. The categories are derived from the work of Czarnecki and Helsen [5] and Gardner et. al [8]. Results of the comparison are summarized in Table 2.

Category		ATL	QVT
Abstraction Level of Transformation Specification	Abstractness ↑	AMW	
			Relations
		ATL	Core, Operational Mappings
		VM	

Transformation Scenarios	Model synchronization	See note (1)	Relations, Core
	Conformance checking	See note (1)	Relations, Core
	Model transformation	AMW, ATL, VM	Relations, Core, Operational Mappings
Paradigm	Declarative	AMW	Relations, Core
	Hybrid	ATL	
	Imperative	VM	Operational Mappings
Directionality	Multidirectional	AMW	Relations, Core
	Unidirectional	ATL, VM	Operational Mappings
Cardinality	M-to-N	ATL, AMW, VM	Operational Mappings, Relations and Core (in checkonly mode)
	M-to-1		Relations and Core (in enforce mode, see also note (2))
Traceability	Automatic	ATL	Relations, Operational Mappings
	User-specified	VM	Core
In-place Update		ATL (in refining mode), VM	Relations, Core, Operational Mappings

Table 2 ATL and QVT components alignment

Note (1): Conformance checking and model synchronization can be implemented in ATL by writing transformation programs but there is no language support for it whereas there is such a support in Relations and Core.

Note (2): The QVT specification does not clarify well the cardinality issue in case of enforce mode.

In the table the rows represent categories and their subcategories. For every category/subcategory we identify the languages that belong to it. Table columns correspond to ATL and QVT languages. The categories are explained in the remaining part of this section.

- **Abstraction level of transformation specification**. Transformation definitions expressed in the languages defined in ATL and QVT differ at their level of abstraction. We present an ordering of the languages according to the abstraction level starting from the highest level. We have to state that the classification of the abstractness is not always well defined and sometimes the distinction is not absolute.
    1. *AMW*. We consider AMW as the most abstract among the languages since the relations specified in it have very general meaning not going beyond the simple assertion of a relation. More specific semantics is defined per every concrete domain. In addition, by default the relations are not executable unless there is an ATL transformation that gives an execution semantics;
    2. *Relations*. Relations language is less abstract than AMW since the semantics of the relations is fixed. They can be used in two modes: *enforce* and *checkonly*. Checkonly mode is used to check correspondences among models without changing them. Enforce mode causes changes of existing models and creation of new models;
    3. *ATL*, *Core*, and *Operational Mappings*. Declarative ATL is considered less abstract than Relations since the rules are unidirectional and intended only for transformations. Core language by definition is less abstract than Relations. Operational Mappings is used to refine relations. We consider the three languages at the same level of abstraction although there are some specific differences. For example, Core has to deal explicitly with traceability whereas the other two languages provide a dedicated support for that. Declarative ATL may be considered more abstract than Operational Mappings. However, ATL with imperative features has a lot of overlap with Operational Mappings language;
    4. *ATL VM*. The language of the ATL VM is the least abstract language. It provides only basic model manipulation operations and a set of primitive instructions;
- **Transformation scenarios**. We consider three scenarios:
    - *Model synchronization*. In this scenario two existing models are synchronized according to a given set of relations. Whenever necessary changes are made in the models. This scenario is supported by Core and Relations languages by using *enforce* mode;
    - *Conformance checking*. In this scenario two models are checked if they satisfy a set of relations. No changes are made to the models. This scenario is supported by Core and Relations languages by using *checkonly* mode;
    - *Model transformation*. In this scenario a set of output models is produced from a set of input models. All the components found in QVT and ATL support this scenario;
- **Paradigm**. We recognize *imperative*, *hybrid*, and *declarative* paradigms for defining transformations. As can be seen from the table the QVT languages are either declarative or imperative. However, the language as a whole is considered as hybrid since it allows both styles to be used by using its sublanguages. Moreover, Operational Mappings refines declaratively specified relations and can at a certain extent be considered as a hybrid language;
- **Directionality**. This category indicates the direction in which a transformation definition may be executed. We distinguish between *unidirectional* and *multidirectional* transformations;
- **Cardinality**. Cardinality indicates the number of input and output models for a transformation definition;
- **Traceability**. Traceability links keep a record of correspondences between source and target elements during (and eventually after) the execution of a transformation. In general, there are two ways to deal with traceability: *automatic*

and *user-specified*. The automatic way is supported by the language constructs and the execution engine. In the user-specified approach the software engineer is responsible for building and using the data structure for traceability.

- **In-place update**. This is a special case of transformation in which the source and the target models are the same. ATL provides *refining mode* to support the scenario;

As can be seen from the table the relations between the languages are complex. The languages are equivalent in some categories and differ in others. This is due to the relatively complex architecture that comprises not a single language but a set of sublanguages.

5. ACHIEVING INTEROPERABILITY

We saw that although ATL and QVT architectures are different they share many features, differently distributed across layers. This section proposes a framework for interoperability between QVT and ATL based on the alignment considerations given in the previous section. We first give definition and motivations for interoperability between the two languages. Then, we describe some options to actually achieve it.

5.1 Definition and Motivations

For the purpose of this paper we assume the following definition of interoperability: executing programs written in one language with the tools designed for another language. An example would be to execute ATL code on a QVT-compliant engine.

We identified the following four motivations for this kind of interoperability:

1. **Problem-language adequacy**. A given transformation language may be especially adapted to specific problems but unable to solve different problems as efficiently as another language. Some complex problems may even require integration of solutions implemented using different languages. The ability to execute programs written in different languages on a single engine should ease the integration.
2. **Execution**. A given language may have a poor engine to execute programs. The possibility to execute programs on a more mature and optimized engine originally designed for another language could improve the performance. Some languages may not even have any engine. We are, for instance, not aware of any QVT implementation at the time of writing this paper. Moreover, a given model engineering platform may only provide a QVT engine to comply with the standard. Some users may still want to use another language, such as ATL, on top of this platform.
3. **Support tools**. Debuggers, profilers, etc. may not exist for a given language. If it was executed using the environment of a language having more tools, it could benefit from them.
4. **Compliance to standards**. Compliance to standards may sometimes be achieved through interoperability. For instance, if (at least) one QVT language was executable on ATL VM, we would be able to claim two compliance points from Table 1: syntax/XMI executability and exportability of this QVT language.

5.2 Interoperability Options

According to the definition given in the previous section we aim at executing a *source* language on an engine originally built for a *target* language. There are several possibilities to achieve this. Building an interpreter of the *source* language in the *target* language is an example. The target engine could also be extended with necessary source language concepts. We are, however, focusing on another approach in this paper: transforming programs written in *source* language into equivalent programs in *target* language. We believe it is easier to implement transformations between languages rather than interpreters in the context of model transformation. Moreover, modifying the *target* engine may be too difficult, too expensive or simply not be possible in some cases.

In order to study the possible transformations between ATL and QVT languages we place their components in a two dimensional space shown in Figure 5.

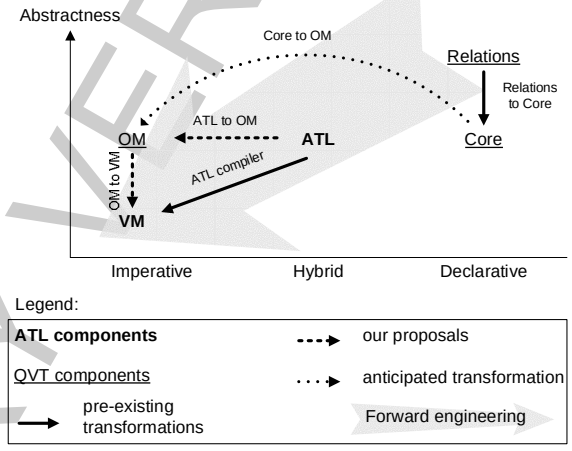

Figure 5 ATL-QVT interoperability through language transformation

The vertical dimension corresponds to the level of abstraction. The horizontal dimension denotes if a component is imperative, hybrid or declarative. We use the information provided in Table 2 to do this. VM stands for ATL Virtual Machine and OM stands for QVT Operational Mappings. AMW is not represented because it has no fixed execution semantics. Once components are placed, we can represent language to language transformations as arrows between them. Two transformations already exist: *Relations to Core* (part of QVT, described in section 2) and *ATL compiler* (part of ATL platform, described in section 3). They are represented as solid arrows. The other four arrows are discussed below.

A transformation from one language to another is traditionally called a compilation. On one hand, compilations are most of the time performed from more abstract to more concrete languages. On the other hand, declarative languages are usually implemented by compilation to imperative languages (machine languages are imperative). Therefore, going from top to bottom and from right to left in Figure 5 roughly corresponds to forward language engineering. Going in this direction, represented here by a large light-gray arrow, is typically easier than going in the reverse engineering direction.

The most concrete and imperative components of ATL and QVT are ATL VM and QVT Operational Mappings respectively. They should, therefore, be the simplest *targets* for us.

We present three interoperability options, corresponding to the dashed arrows in Figure 5:

- **Core-to-OM**. This first option does not deal with QVT-ATL interoperability. We are, nonetheless, considering it because it provides an interesting feature: a path from any QVT language to Operational Mappings, which we have just identified as a possible implementation target. With it, a QVT implementation only needs to provide an Operational Mappings execution engine. Core and Relations languages are transitively executable on it following the transformation chains. We could anticipate that QVT implementers will provide this transformation;
- **OM-to-VM**. This option provides QVT executability on ATL engine. Its implementation would consist in a transformation from QVT Operational Mappings to ATL VM bytecodes. Motivations 1, 2, and 3 would be fulfilled by this approach. Moreover, the combination of ATL VM and *OM-to-VM* transformation forms an engine capable of executing QVT Operational Mappings, thus answering motivation 4. Combined with the *Core-to-OM* transformation this makes all three QVT languages executable on ATL VM;
- **ATL-to-OM**. ATL could be made executable on future QVT engines. This could again be achieved by a transformation, which would translate ATL into QVT Operational Mappings. Motivation 1 and possibly 2 and 3, depending on future QVT engines capabilities (performance and tooling), would be fulfilled by this approach.

Concrete implementation details of these three options are beyond the scope of this paper. We can remark that in each case we go in the forward engineering direction, which corresponds to classical scenarios.

Table 3 summarizes all possible transformations between ATL and QVT components, again with the exception of AMW. Cells of the table stand for transformations. The order in which components are listed is obtained by reading Figure 5 from bottom to top (concrete to abstract) and from left to right (imperative to declarative). Reverse engineering issues are therefore on the upper right part of Table 3. Empty cells correspond to other possible interoperability options that were not discussed here. We only presented the ones we thought were the simplest.

from \ to	**VM**	OM	**ATL**	Core	Relations
VM	N/A	Reverse engineering issues			
OM	OM-to-VM	N/A			
ATL	ATL compiler	ATL-to-OM	N/A		
Core		Core-to-OM		N/A	
Relations				Relations to Core	N/A

Table 3 ATL-QVT interoperability options

Although we anticipate that some QVT implementations will use a *Core-to-OM* transformation, some may only provide Relations and/or Core engines. For this reason, *ATL-to-Relations* and *ATL-to-Core* transformations would also be interesting. They are, however, in the reverse engineering part of Table 3 and therefore probably more complex than the ones we chose to discuss. We have identified two potential issues: transforming imperative parts of ATL into declarative, which is difficult in the general case, and possible declarative concepts mismatch (an ATL rule is not a Relations rule).

The *OM-to-ATL* transformation may also be considered as a mean to provide execution support for QVT Operational Mappings on an ATL engine. This transformation is considered as a reverse engineering issue in Table 3 because its source (i.e. Operational Mappings) is imperative while its target (i.e. ATL) is hybrid. However, restricting the target ATL model to imperative constructs should make this transformation relatively simple: in this case, no declarative construct needs to be inferred from imperative code.

The solutions discussed in this section are actual examples of using model engineering techniques (i.e. model transformation) to achieve interoperability.

6. CONCLUSIONS

In this paper we described the architectures of QVT and ATL. We analyzed how they can be aligned on several categories (abstractness, paradigms, directionality, cardinality, etc.). The results of the analysis were used to show how ATL programs could be executed on future QVT engines and how QVT languages (Relations, Core and Operational Mappings) could reciprocally be executed on ATL virtual machine. To achieve this interoperability, model engineering principles were applied, mainly model transformations, to model transformation languages themselves. We discussed advantages and feasibility of some options that would provide more features at a lower cost: QVT Operational Mappings to ATL virtual machine and ATL to QVT Operational Mappings transformations.

The results show that it is possible to have a reasonable interoperability between two, in some respects, rather different transformation languages. The interoperability issues were discussed at a more conceptual level. To actually implement the proposed options we need a further comparison between the QVT and ATL requirements and the detailed (i.e. non-architectural) design choices.

The framework depicted in section 5, especially as it is represented in Figure 5, is based on general considerations. We expect that it is usable for analyzing the interoperability options among other transformation languages.

7. ACKNOWLEDGMENTS

This work has been partially supported by ModelWare, IST European project 511731. We would like to thank the members of the ATLAS team for the useful feedback on the ideas presented in the paper.

8. REFERENCES

[1] Abouzahra, A., Bézivin, J., Didonet Del Fabro, M., and Jouault, F., A Practical Approach to Bridging Domain Specific Languages with UML profiles, In: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05, San Diego, California, USA, 2005

[2] Bézivin, J., Dupé, G., Jouault, F., Pitette, G., and Rougui, J. E., First experiments with the ATL model transformation language: Transforming XSLT into XQuery, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, Anaheim, CA, USA, 2003

[3] Bézivin, J., Jouault, F., Rosenthal, P. and Valduriez, P., Modeling in the Large and Modeling in the Small, Lecture Notes in Computer Science, Volume 3599, Aug 2005, Pages 33 – 46

[4] Budinsky, F., Steinberg, D., Raymond Ellersick, R., Ed Merks, E., Brodsky, S. A. and Grose, T. J., Eclipse Modeling Framework, Addison Wesley, 2003

[5] Czarnecki, K., Helsen, S. Classification of model transformation approaches. OOPSLA2003 Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA, 2003

[6] Didonet Del Fabro, M., Bézivin, J., Jouault, F., and Valduriez, P., Applying Generic Model Management to Data Mapping, to appear in the proceedings of the Journées Bases de Données Avancées (BDA05), 2005

[7] Eclipse Foundation, Generative Model Transformer (GMT) Project, http://www.eclipse.org/gmt/

[8] Gardner, T., Griffin, C., Koehler, J., and Hauser, R. A review of OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards the final standard. 1st International Workshop on Metamodeling for MDA, York, UK, 2003

[9] Jouault, F., and Kurtev, I., Transforming Models with ATL, to appear in proceedings of Model Transformations in Practice Workshop, October 3rd 2005, part of the MoDELS 2005 Conference

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ, Lecture Notes in Computer Science, Volume 2072, Jan 2001, Page 327

[11] Meyer, B., Object-oriented software construction, Second edition, Prentice Hall PTR, 1997

[12] Netbeans Meta Data Repository (MDR). http://mdr.netbeans.org

[13] OMG, MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10 (2002)

[14] OMG. Object Constraint Language (OCL), OMG Document ptc/03-10-14

[15] OMG, Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10), OMG Document ad/2005-07-01 (2005)