THALES recommendations for the final OMG standard on Query / Views / Transformations

Benoît Langlois benoit.langlois@thalesgroup.com

Nicolas Farcet nicolas.farcet@thalesgroup.com

THALES Research & Technology France
MIRROR Pilot Program[1]
Domaine de Corbeville
91404, ORSAY CEDEX
FRANCE

August 29, 2003

Abstract

Model transformation is at the core of the OMG's Model Driven Architecture™ (MDA). The current submission process on the OMG MOF 2.0 Query/Views/Transformation (QVT) Request for Proposals (RFP) [1] elicited eight initial submissions and currently, five revised submissions are still competing. IBM has formulated, in response to the eight first submissions, a set of recommendations for the final standard [8]. This paper enhances IBM recommendations with an additional user point of view.

This paper, based on user criteria, presents recommendations gathered from THALES experience. These end-user recommendations are expressed in terms of IBM technical recommendations. Finally, we propose a candidate technical architecture to harmonize the positions of tool developers and end-users.

[1] MIRROR is a THALES Pilot Program on the MDA.

1. Introduction

Model transformation is at the core of the MDA and particularly at the core of the model-driven software engineering chains. The major stake for a user is to capitalize on the software engineering expertise (PIM to PIM, PIM to PSM, PSM to PSM transformations, model quality, documentation…) to build a rationalized software engineering tool chain with the financial objective to increase the ROI (return on investment) of software production. In an industrial environment, the main expectation is to have portable and durable query / view / transformation (QVT) expertise, and this concerns multiple communities of QVT actors. These are QVT designers, designing model transformations, QVT developers, coding rules, and more largely architects, appreciating model transformation impacts on architecture, as well as process engineers, ensuring model transformation-aware methodology applicability.

The MOF 2.0 Query / Views / Transformations (QVT standard) should meet such user requirements to be fully adopted. The OMG MOF 2.0 Query/Views/Transformation (QVT) Request for Proposals (RFP) resulted in eight initial submissions and currently, five revised submissions are still competing. IBM has proposed twelve technical recommendations, in response to these eight initial submissions, to have a standard at the level of its ambitions. THALES, in this paper, proposes seven additional recommendations with a user point of view.

Section 2, on the ground of a set of user analysis criteria, proposes a set of user recommendations. Section 3 analyses IBM technical recommendations with our QVT user standpoint. Finally, section 4 presents a technical architecture for the QVT Standard, meant to address most of these recommendations.

In section 4.1, the proposed reflective and layered architecture is a result of the joint THALES / INRIA / CEA MOTOR project. Additional contact points: Jean Bézivin, INRIA/Irin-Université de Nantes, France, Jean.Bezivin@sciences.univ-nantes.fr; Jean-Marc Jézéquel, INRIA/Irisa-Université de Rennes, France, jezequel@irisa.fr.

2. THALES recommendations

This section proposes a set of user recommendations for the MOF 2.0 QVT RFP. These recommendations synthesize the analysis criteria presented in the next section.

2.1. Analysis criteria

The selection of user analysis criteria for the QVT standard must not directly target the QVT technical standard itself, otherwise it would be another technical analysis, but shall rather focus on quality factors to be satisfied from an end-user point of view. Four quality factors are introduced here: portability, maintainability, utilisability and functionality (ISO 9126 standard terminology, [5]). A further study should analyze all quality factors defined in the ISO 9126 or in the IEEE 830 standards [6].

The four quality factors are presented in the following tables with a short definition, a set of stakes and a set of means satisfying the considered factor quality and a level of requirement (high, medium or low). In the following text, a “QVT work product” is a piece of information

or a physical entity, in relation with QVT, produced or used by the activities of the Query / Views / Transformations Engineering processes[2]. It can be a QVT rule, a QVT library, a QVT COTS's, a QVT model…

Portability	
	Definition: Ability to move a QVT work product from one platform to another. Easiness of this moving. *Stakes*: • [S1.1]: Raise the durability of QVT work products, allowing capitalization. • [S1.2]: Be tool-independent. *Means*: • [M1.1]: The semantics of the QVT standard shall be complete, consistent and unambiguous. • [M1.2]: The tool developers shall respect the QVT standard. *Level of requirement*: High, in order to maintain the ROI of software production across different platforms.

Maintainability	
	Definition: Ability to make evolve and integrate QVT work products. Effort required to make evolve and integrate QVT work products. *Stakes*: • [S2.1]: Reduce maintenance costs. • [S2.2]: Have durable software engineering chains thanks to their ability to adapt to ever evolving environments, e.g. platforms, standard evolution, model transformation specifications. *Means*: • [M2.1]: Reduce QVT work product production and evolution time. • [M2.2]: Facilitate QVT work product deployment and integration. • [M2.3]: For scalability purposes, have a language easing QVT work product design and realization. This quality factor does not concern only the tools but also the QVT standard. Scalability is representative: the complexity must be reduced as much as possible to express and modify, as simply as possible, QVT work products. Reusability efficiency (generalization / specialization, pattern or libraries usage / design…) is also another aspect to be treated by the QVT standard. *Level of requirement*: High, in order to maintain the ROI and to have durable development platform. This point is key in an industrial context.

[2] This definition is an adaptation of the SPEM work product definition [4].

Usability	
	Definition: Ability to use the QVT language to produce QVT work products. Effort required to produce QVT work products. *Stakes*: • [S3.1]: Improve QVT language expressiveness and efficiency. • [S3.2]: Facilitate the genericity and the customization of the QVT work products. • [S3.3]: Ensure QVT work product composition capability. • [S3.4]: Offer the ability to support multiple semantics[3] for multiple communities of QVT actors (easing production and communication). *Means*: • [M3.1] (satisfying [S3.1] and [S3.2]): Have a language facilitating the learnability and the understandability of the QVT Rules. This aspect concerns only the concrete syntax of a QVT language. • [M3.2] (satisfying [S3.4] that implies a reflexive approach of QVT): Have abstract to concrete syntax transformations, but also transformations from one semantics into another. *Level of requirement*: • [S3.1]: Medium, compared to the previous quality factors. • [S3.2] and [S3.3]: High to improve the ROI. • [S3.4]: High: the QVT standard has to be sufficiently open to cover all QVT aspects of the model software engineering.

Functionality	
	Definition: Ability to actually support QVT work product production and usage. Level of compliance[4]. *Stake*: • [S4.1]: Ensure compliance of the tool with the QVT Standard. • [S4.2]: Ensure interoperability between tools. *Means*: • [M4.1]: Have a complete and unambiguous semantics of execution specification (structure and behavior), for portability (see [S1.2] and [M1.2]) and interoperability between tools. • [M4.2]: Have tools respecting this semantics of execution: the successive states of the model shall be determinist (see [M1.2]). *Level of requirement*: High, for the same reasons than the portability quality factor.

[3] Each developer community expect a QVT language, e.g. object-oriented, functional or logic languages. However, language semantics must not be limited to the single developer point of view. For example, for a QVT design with a model approach, a simple graphical semantics with textual notations based on OCL can be sufficient.

[4] Functionality does not concern here the set of functions proposed to produce and use QVT work products but the capability to support this production and this usage.

2.2. THALES recommendations

Here is listed a first set of user recommendations that should meet the QVT user community expectations about the QVT standard.

THALES Recommendation 1	
[TR1]	**Ensure QVT work product portability**

Users require portability. Portability allows to acquire and diffuse QVT expertise [S2.2] within a heterogeneous but coherent environment [S1.2], [S4.1], [S4.2] and, as a consequence, to have durable QVT work products [S1.1].

In short, on different platforms, the same QVT specification has to produce the same effects.

Would the effects be different, either one of the platform does not respect the QVT standard, or the standard is incomplete and / or inconsistent and / or ambiguous. Or both.

Thales Recommendation 2	
[TR2]	**Ensure QVT work product durability**

QVT work products, and especially transformations, are at the core of the software engineering chain. They capitalize a software engineering expertise of many years from many actors being within or outside the enterprise, see [S2.1], [S2.2], [S3.2] and [S3.3]. In a heterogeneous community, portability contributes to the durability of QVT work products [S1.1].

Thales Recommendation 3	
[TR3]	**Ensure QVT work product composition capability**

This point concerns the possibility, at a fine and a large scale, to easily connect a QVT work product to another and also the possibility to connect a QVT work product to some external QVT work product. Connection implies the possibility to express a complex transformation in terms of simpler self-contained transformations. Connection can be resolved statically or dynamically at the runtime.

Thales Recommendation 4	
[TR4]	**Have an open and unified QVT standard**

It is illusive to restrict the QVT standard to only one paradigm [S3.4]. The debate on the imperative vs. declarative style is illustrative. The user has an opportunistic attitude: take the most efficient formalism in response to the problem he has to resolve. The user has also his preferences and his experience: today, most of the developers use a procedural language. The QVT standard shall be open to offer many paradigms, but in a unified way, in order to have portable QVT work products [S1.1], [S4.1] and [S4.2].

Thales Recommendation 5	
[TR5]	**Have the most efficient and expressive QVT language by concern**

A textual language is not adapted for communication. Reciprocally, a graphical language is not adapted for detailed rule description. Actually, the QVT standard has to allow this separation of concerns. Each QVT language shall meet consistent purposes required by one concern in order to have efficient and expressive QVT languages, see [S3.1], [S3.4].

3. IBM recommendations considered from THALES user-driven point of view

This section positions THALES recommendations with respect to IBM recommendations. IBM technical recommendations are considered from THALES more user-driven point of view.

IBM Recommendation 1	
[IR1]	**Support an hybrid approach to transformation definitions**

We agree with this recommendation (see [TR4]) and particularly with the citation of Adam Bosworth. The difference we make is that declarative and imperative styles are two kinds of paradigms. Each paradigm is described with a semantics and it is possible to mix different paradigms to have multi-paradigms, like a declarative / imperative hybrid language[5].

[5] For illustration, see the presentation of the Mozart Programming System [9].

IBM Recommendation 2	
[IR2]	**Provide a simple declarative specification language**

We agree with this recommendation. We may add that a graphical notation can be very expressive and we encourage it in some areas, for instance to express graphically a mapping between concepts. However, a graphical language may be limited in terms of scalability.

IBM Recommendation 3	
[IR3]	**Use declarative queries only**

We agree with this recommendation.

IBM Recommendation 4	
[IR4]	**Provide an abstract syntax for the transformation language**

We fully agree with this and we believe this is the key technical recommendation for the QVT standard. The interest is not only to plug transformations expressed in different language styles, but mainly to have an open and unified standard, allowing portability, interoperability, etc. See our recommendation [TR6], in section 4 below.

IBM Recommendation 5	
[IR5]	**Adopt common terminology**

The need of this recommendation is obvious to clarify the concepts of QVT. It is the starting point to have a complete, consistent and unambiguous semantics, see [M1.1], [M1.2], [M4.1], [M4.2].

IBM Recommendation 6	
[IR6]	**Use the Action Semantics[6] as an interchange format**

We agree with this recommendation, seen as a response to [IR4]: "Imperative parts of rules should be exchanged via UML Action Semantics. The UML 2.0 Action Semantics will be appropriate within the timescale of this RFP. This will provide a standards-based interchange format for imperative specifications of transformation behavior while permitting the use of

[6] UML 2.0 Action Semantics.

particular concrete syntaxes that are appropriate for use in particular development environments" [8].

IBM Recommendation 7	
[IR7]	**Support symmetric rule definitions**

In theory, we agree with this recommendation to avoid redundancy and to have a consistency between [source / target] and [target / source] mappings. In practice, we are not yet convinced of its necessity. This need must be evaluated in reference to the high tool development cost it implies. Moreover, we doubt that it is realistic for an imperative language.

IBM Recommendation 8	
[IR8]	**Support composition and reuse**

We are completely in line with the composition capability and this at different scales: 1) to reduce complexity and for abstraction, 2) to produce and use QVT COTS's, see [TR3].
Concerning reuse, we encourage the use of the template technique to elicit QVT patterns (systematic solutions).

IBM Recommendation 9	
[IR9]	**Support complex transformation scenarios**

We encourage to have M-to-N mappings, supporting the other mapping types (1-to-1, 1-to-N, N-to-1). Concerning symmetric mapping, see our response to [IR7].

IBM Recommendation 10	
[IR10]	**Provide complete examples**

We do not propose examples in this paper.

IBM Recommendation 11	
[IR11]	**Establish requirements on transformation executions**

This point concerns the semantics of execution that has to be explicit, see [M1.1], [M1.2], [M4.1], [M4.2], and especially for the rule engines. This is essential for the portability of the QVT rules.

IBM Recommendation 12	
[IR12]	**Emphasize the tooling aspect**

We suggest the use of the quality factor perspective for eliciting and classifying the use cases allowing to define the requirements that a tool should satisfy.

This analysis shows that: 1) IBM and THALES visions on QVT recommendations are globally aligned, 2) it is necessary to consider user recommendations, going beyond a single technical perspective.

4. Towards a pivot approach

As seen in the compared Thales and IBM recommendations analysis (see sections 2 and 3), we consider that IBM recommendation 4 about an abstract syntax for QVT is central and meets both user and technical expectations about the QVT language. In this section, we present a candidate approach, called the pivot approach, that implements an architecture of the language that complies to this technical recommendation, and that insures a high level of portability, maintainability, usability and functionality quality factors fulfilment.

4.1 A reflective and layered architecture

The next recommendation extends [TR4] ("Have an open and unified QVT standard") and [IR4] ("Provide an abstract syntax for the transformation language") and guarantees [S3.4] ("Offer the ability to support multiple semantics for multiple communities of QVT actors").

THALES Recommendation 6	
[TR6]	**Have a reflective and layered approach**

The best way to have an open, unified [TR4] and durable standard is to adopt a reflective and layered architecture. A layered approach has been introduced by QVT-Partners [2] and OpenQVT [3]. From a bootstrap, the core of QVT, we can construct successive layers of abstract and concrete syntax. An abstract syntax defines a semantics; a concrete syntax defines a concrete representation of an abstract syntax. With a reflective approach, an abstract syntax can be transformed into a concrete syntax and / or in another abstract syntax, a new layer. In order to have a complete and consistent system of QVT languages, each transformation, and particularly between two layers, is described by a mapping[7].

[7] This is an axiomatic approach. In logic, we could say that the bootstrap is a set of axioms; a layer or a concrete syntax corresponds to a theorem; a mapping corresponds to the rules to create a theorem.

The bootstrap. It contains: 1) the primitive types, 2) the first metamodel where 1.a) the Query, the View and the Transformation are explained and described, 1.b) the other fundamental elements, like the packaging, are introduced. The bootstrap contains the core of QVT but also defines the minimal semantics of QVT. The bootstrap is unknown by the end-users.

Note: the dilemma declarative / imperative style is reintroduced here; if there are many possible bootstraps, just one bootstrap shall exist in order to have an unified semantics.

The layers. To have a more concrete presentation of what a layer is, we use the same pattern as the MDA pattern: the bootstrap defines a common language, the QVT-CL, corresponding to a PIM; each new layer defines a specific language, the QVT-SL, corresponding to a PSM. The QVT-CL to QVT-SL transformation is specified by a QVT mapping. This pattern can be reapplied recursively: a QVT-CL defines a new family of language.

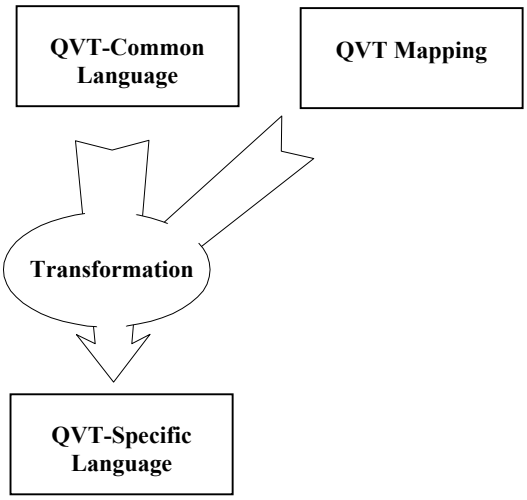

Figure 1. QVT-Language Pattern

This solution offers a good compromise for the tool developers: if a tool language respects the semantics of an abstract syntax, this tool language can be transformed into another target language (and reciprocally). For the users, the comformance to this common semantics is a guarantee of portability. See [TR1] and [TR2].

THALES Recommendation 7	
[TR7]	**Isolate the access to the model repository**

A repository of models, such as a model database, shall be operated through a query / view / transformation facade API. This layer, that we call the QVT-Repository API, contains all services required to interact with a model repository. The following figure exemplifies a reflective layered architecture, including a repository layer.

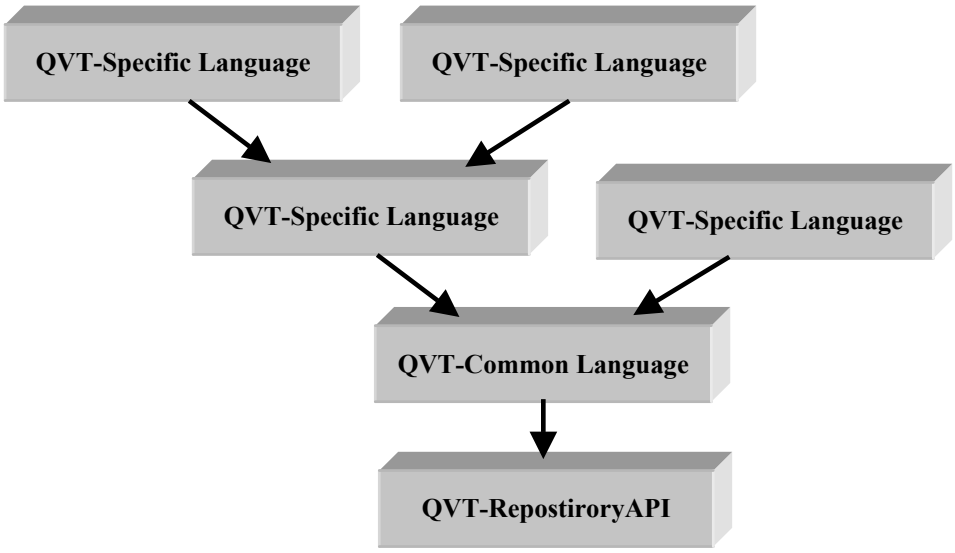

Figure 2. Layered QVT-Language Approach

4.2 The Pivot technique

In response to [TR6], we propose the pivot technique. The major interest of this approach is to satisfy the quality factor of portability and particularly to guarantee a semantics of execution, see [TR1] and [M1.1]. A good example is Java: thanks to its virtual machine, under Windows or under Unix, the behavior is the same. For our concern, the QVT-CL (the bootstrap) constitutes the pivot and all other QVT-SL shall be semantically projected on the pivot (a mapping from the QVT-SL to the QVT-CL guarantees this projection).

Note: to be precise, a chain of semantics mappings from a QVT-SL 1 to a QVT-SL 2 (an horizontal mapping) is not strictly equivalent to a QVT-SL1 to a QVT-SL2 mapping via the pivot (this is a vertical mapping, see the note 6).

The architecture we propose is organized in four layers.

The QVT Language layer, also called Metamodel layer. This layer characterizes the foundational common concepts and semantics that the various QVT languages will have to map onto. This semantics is organized in two levels: common semantics and advanced semantics. The purpose of having these two levels is to provide the possibility to compose transformations with heterogeneous QVT languages.

- The *common semantics* is what every QVT language shall support. Composition of transformations shall be expressed with this common semantics as public provided and required interfaces..
- The *advanced semantics* is an extended range of semantics that QVT languages may support to represent private QVT behavior (i.e. not the public interfaces and not in the semantics of the QVT-CL).

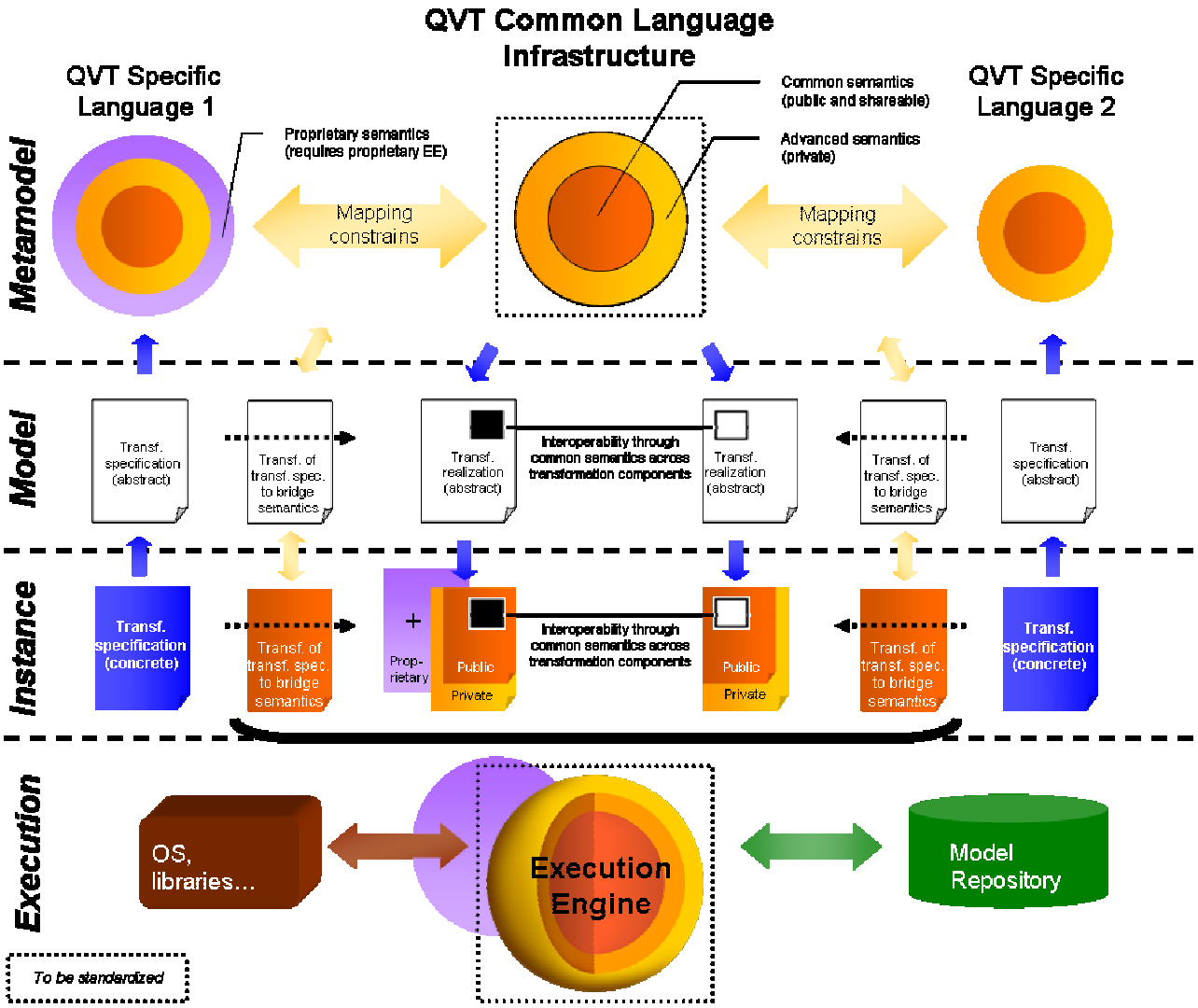

Figure 3. *QVT Common Language Infrastructure*

We think that at least these two levels of semantics shall be standardized in order to provide the ability to support multiple languages and allow the interoperability and composability of QVT specifications.

These two levels of semantics apply mapping rules in order to guaranty the appropriate correspondence of the QVT-SL semantics with the QVT-CL. Depending on the kind of semantics the specific language provides, the correspondence may be:

- *Inclusion*: The specific language has a semantics range that is fully supported by the common and advanced semantics. This is the case of the QVT Specific Language 2 in Figure 3.
- *Proprietary extensions:* Here, the specific language semantics range does not exactly map onto the common and advanced semantics. For instance, if the QVT-CL supports only single-inheritance, then a specific QVT-SL multiple-inheritance semantics has to be adapted onto QVT-CL through mapping. This is the case of the QVT Specific Language 1 in Figure 3: an additional level of proprietary semantics has been

represented. Note that it will require proprietary extensions into the execution layer of the overall architecture presented below.

The abstract syntax layer of QVT expression, also called model layer. Every QVT expression (queries, views and transformations) shall conform to a QVT language, in fact to a QVT-SL, defined in the previous layer, the QVT Language layer. For each QVT expression, there is an abstract syntax and a concrete syntax. This layer concerns the abstract syntax of a QVT expression, in other words its semantics. However, this QVT expression has to be mapped, with a transformation, in the QVT-CL, the pivot, and respect the common and advanced semantics. This mapping is a semantic mapping. There is a 2-dimensional mapping: vertical with the QVT-SL metamodel and horizontal with the QVT-CL abstract syntax, respectively called "Transformation specification (abstract)" and "Transformation realization (abstract)" in Figure 3.

The concrete syntax layer of QVT expression, also called instance layer. This layer contains the QVT expressions in their concrete syntax. For example if the QVT Specific Language 1 were *Objecteering J* language then the QVT expression would be the actual *J* source code. There is also here a 2-dimensional mapping. However, the "Transformation realization" is here expressed by two entities: a) one for its public and private parts (both conform to the common and advanced semantics defined in the metamodel layer and are likely expressed in terms of bytecode or other intermediary concrete language), and b) one for the proprietary semantics part of the language specification. The public part allows composition of the transformation initially expressed with QVT Specific Language 1 with an other QVT expression expressed, in Figure 3, in QVT Specific Language 2. Moreover, in our architecture, both entities shall be executable on the virtual machine described in the execution layer below. In addition, for a reason of consistency, the horizontal mapping of this layer shall be semantically compliant with horizontal mapping of the previous layer. This very transformation illustrates the ability of our architecture to be reflexive (i.e. its internal transformations are also standard, executable, and conform to the architecture itself). Here, this transformation is expressed solely in terms of the common semantics described in the QVT Language layer, but this is not mandatory.

Execution. This layer defines how we view a common virtual machine for QVT execution. The virtual machine shall support and execute both the common and the advanced semantics (likely expressed in terms of bytecode as defined at the previous layer). The virtual machine exposes an execution surface that needs to be standardized. To maintain tool-neutrality and technology-independence of the virtual machine implementation itself, interfaces with model repositories (for Q and V) as well as interfaces with operating systems, and any required libraries shall also be standardized. Finally, proprietary semantics has to be supported by an additional virtual machine (unless this semantics was "down-mapped", for e.g. through compilation, to the common and advanced semantics – indeed, the specific language compiler can do this semantics adaptation, with information loss, though). Means of integration between both virtual machines shall also be standard if we want to maintain full portability. In this case, it is possible to deliver a proprietary virtual machine that installs along the standard one to support parts of the execution of a transformation specification expressed in a specific language that has some proprietary semantics.

We think that this architecture responds to the requirements of multiple communities (support for multiple languages, see [S3.4] and [TR4]), transformation composition (common semantics supported by all specific languages, see [TR3]), tool-neutrality (portable virtual machine, see [TR1]). However such an architecture remains large and complex to standardize. A similar architecture has already been standardized however, which means that this is feasible. This is the *Common Language Infrastructure* which is very similar but has a larger scope, not targeted to model transformation. It was standardized at ECMA (and now at ISO) by Microsoft, HP, and Intel [7], along with C#, one concrete language that fits perfectly with the semantics executed by the virtual machine.

In our case however, the core semantics shall target query, view and model transformation. In particular, it shall abstract model repository access [TR7]. The question now arises of what coverage has to be addressed by the common and advanced semantics defined in the metamodel layer? Likely an imperative semantics because it is easier to map declarative language onto imperative ones than the opposite. UML Action Specification Language could be a one candidate for such an imperative semantics.

5. Conclusion

This paper stresses the necessity to formulate a set of user recommendations. Have clearly emerged the portability and durability needs, required to maintain an industrial model engineering tool chain and to maintain, better to improve, the ROI of software production. Another user recommendation has been formulated to have an open and unified QVT standard to support all needs of query, view and transformation for the model engineering and to support different semantics for multiple communities of QVT actors. We globally agree with the IBM recommendations, as a set of technical recommendations for the final standard. For us, the central point is to have an abstract syntax to formulate QVT expressions (recommendation 4). Considering a reflective and layered architecture for an open and unified QVT standard, we have introduced the pivot technique guaranteeing a semantics of execution. Taking into account this proposal, the QVT standard may precise this semantics and the architecture associated. We strongly believe that it is the most consensual approach for an agreement among tool developers for satisfying the user communities.

Acknowledgment

We would like to thank the members of the MOTOR project, Jean Bézivin, Jean-Marc Jézéquel, Didier Vojtisek, Daniel Exertier, Madeleine Faugère and Hubert Dubois; Serge Salicki, the MIRROR pilot program manager and Laurent Rioux, leader representative at OMG for MIRROR project.

The MOTOR Project is a project of the CARROLL research program, launched by THALES and two French public research laboratories: CEA (Commissariat à l'Energie Atomique) and INRIA (Institut National de Recherche en Informatique et en Automatique). The joint program aims at developing software engineering and middleware technologies. Over the coming years, the goal of CARROLL is to spearhead research that is focused on pinpointing increasingly competitive software developments for large-scale and embedded systems, the likes of which are vital in today's ever more demanding and complex software environment.

The results will thus be of valuable worth not only to THALES, but also to software editors and other industry players. More information on CARROLL can be found at www.carroll-research.org.

References

[1] OMG/RFP/QVT. MOF 2.0, Query / Views / Transformation RFP. OMG Document ad/2002-04-10, October 2002.

[2] OMG / MOF 2.0, Query / Views / Transformation ad/2002-04-10, Initial Submission, Version 1.0 2003/03/03, QVT-Partners.

[3] OMG / MOF 2.0, Query / Views / Transformation ad/2002-04-10, Initial Submission, Version 1.0, 2003/03/03, OpenQVT.

[4] OMG / SPEM. Software Process Engineering Process Metamodel (SPEM). OMG Document ad/formal/02-11-14, version 1.0, November 2002.

[5] ISO/IEC TR 9126 (1991). International Organization for Standardization, Geneva. An international standard for quality factors.

[6] IEEE Guide to software requirements specifications. ANSI / IEEE Std. 830-1993 or 1998.

[7] ECMA and ISO/IEC C# and Common Language Infrastructure Standards:

- ECMA (December 2001): 1st edition of the C# and CLI standards as ECMA-334 and ECMA-335, respectively. More information: http://www.ecma-international.org/activities/Languages/ECMA CLI Presentation.ppt
- ISO (April 2003): ISO/IEC 23270 (C#), ISO/IEC 23271 (CLI)

[8] Tracy Gardner, Catherine Griffin, Jana Koehler, Rainer Hauser. *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*, July 21, 2003.

[9] Mozart-Oz, The Mozart Programming System. http://www.mozart-oz.org/