Providing an open Virtual-Machine-based QVT implementation

Adolfo Sánchez-Barbudo[1], E. Victor Sánchez[1], Víctor Roldán[1], Antonio Estévez[1], y José Luis Roda[2]

[1] Open Canarias, S.L., C/. Elías Ramos González, 4 - Oficina 304
38001 Santa Cruz de Tenerife, Spain
{adolfosbh,vsanchez,vroldan,aestevez}@opencanarias.com
[2] Universidad de La Laguna, La Laguna, Spain
jlroda@ull.es

Abstract. Now that OMG's MOF 2.0 QVT specification has finally reached official status it will be interesting to witness how much impulse it will represent to the evolution of the Model-Driven Engineering field. The Eclipse platform is developing its own open QVT solutions, but they are not ready yet for industrial production. This paper discusses an Eclipse-based QVT open implementation that Open Canarias has been developing for several years. The distinctive trait of this implementation is that at its core lies a Virtual-Machine model transformation technology, driven by a language named Atomic Transformation Code (ATC). We will explain this QVT solution and the whole process QVT model transformation instances follow until they can be executed, and will also reflect how this virtual-machine approach influences its overall architecture.

Keywords: Model Transformations, QVT, Virtual Machine, ATC.

1 Introduction

The Object Management Group™(OMG™) defines itself as "an international, open membership, not-for-profit computer industry standards consortium". This organization periodically releases public open specifications, many of which become standards that contain recognized best practices that help in driving the industry towards achieving milestones and confronting further challenges. Modeling and Model-Driven Software Development (MDSD) are covered by several OMG specifications gathered together under the Model-Driven Architecture (MDA) umbrella.

By policy, the OMG does not provide complying implementations of its own specifications. Such activity is left to interested third parties or vendors. Among others, this means that accuracy, unambiguity and even just basic functionality for its specifications are not always warranted, even for finalized specifications. Additionally, OMG does not sanction any available solutions to become reference implementations of those specifications.

Similarly, Eclipse "is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. The Eclipse Foundation is a not-for-profit, member supported corporation that hosts the Eclipse projects and helps cultivate both an open source community and an ecosystem of complementary products and services". Eclipse focuses on providing open-source implementation solutions related with the most common problems software and software development faces today. The Eclipse Foundation became member of the OMG in 2007.

Many OMG specifications and Eclipse implementations overlap in their scope and goals. There are even Eclipse projects created with the sole purpose of providing an implementation for a particular OMG specification, such as SBVR, UML2, or OCL. That is why in 2008 a movement to bring both parties closer together was pushed forward. Two OMG-Eclipse joint symposia were celebrated with great success to discuss possible technological synergies that go beyond the mere passive, non-coordinated, current reciprocal complementarity.

As an example of the non-trivial alignment between both organizations, Eclipse's M2M (Model To Model) and M2T (Model To Text) projects include components dedicated to supporting OMG's related specifications, which are the *MOF Query/View/Transformation (QVT)* [10] and *MOF Models to Text* [11]. Of the three languages the QVT specification comprises, a project that attempted to give support to the procedural *Operational Mappings*, which is led by Borland, has been under development as an Eclipse project since at least as early as 2006.

But the QVT specification has not reached official 1.0 status until recently [10]. Furthermore, the whole process has taken about six whole years, and there are still some details that need polishing. Anyways it is now when QVT has to demonstrate its full potential into becoming a true industrial standard, or either be relegated to becoming a referential notation much in the sense as UML has traditionally been used for documentation and the sharing of ideas, as discussed in [15]. The quality of the available QVT implementations will highly determine the success of QVT in the upcoming years.

Back in 2005, when QVT had still to decide between two contending specification proposals, Open Canarias S.L. started a model transformation project whose primary goal was to comply with MDA standards. That meant that the QVT specification had to be supported. A preliminary work on this topic was published in [12]. The fundamental trait of this project is that it is built upon a Virtual-Machine technology, driven by its own byte-code model transformation language named Atomic Transformation Code (ATC)[3] [8].

In this paper we will discuss our QVT implementation project in terms of the different steps involved in the process that ultimately leads to execution of the source QVT model transformations, applied over a set of models. We will deepen on several of the difficulties encountered throughout its development. Some alternatives to different aspects of this work will also be addressed.

The paper is structured as follows. Section 2 makes a brief overview of the QVT specification, Section 3 details the chosen technology upon which the QVT model transformation engine is based, Section 4 reviews the architecture of the presented QVT implementation solution, in terms of the three stages it comprises. Related work follows in Section 5 and conclusions close this work in Section 6.

2 QVT

The OMG's MOF Query/View/Transformation (QVT) specification defines three different, although complementary, languages to describe model transformations in the scope of the MDA framework. Two of them enjoy a declarative nature, and the third one is imperative. The specification is structured around the architecture depicted in Figure 1. It includes a section explaining a mechanism that enables the opportunity of using black-box invocations.

QVT *Core* (QVTc) is a small declarative language based on pattern matching over a set of variables and evaluating conditions over those variables against a set of models. The language uses a model-based trace mechanism to record what ocurred during a transformation execution. The trace metamodel must be explicitly provided in the definition of the model

[3] ATC-related technology, including the QVT execution project explained here is or will eventually be made available at: http://www.modelset.es/atc/atcdownload.html

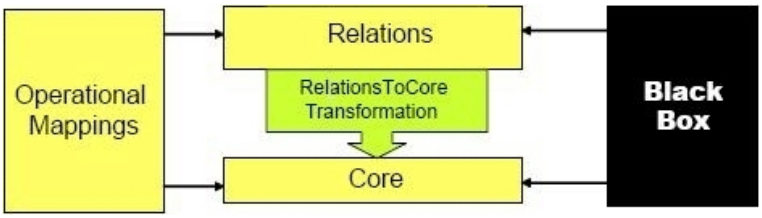

Fig. 1. QVT layered architecture.

transformation, thus it neither depends on the QVTc language nor is implicitly inferred from the contents of such transformation.

The *Relations* (QVTr) language allows us to specify a set of relationships between MOF models. It supports complex object pattern maching. Additionally, it saves the user from having to explicitly manage trace information. Indeed, trace instances are implicitly created and maintained. Although QVTr and QVTc are meant to be semantically equivalent, QVTr is defined at a higher level of abstraction, so it is more user-friendly. The specification even defines a `relToCore` model transformation to translate among both languages' abstract representations. This layered relationship is reflected in the QVT architecture shown in Figure 1.

The QVT specification supports a third language named *Operational Mappings* (QVTo). This imperative language is similar to a traditional procedural programming language, but it also carries a complement of constructs designed to handle (create, modify, etc.) MOF model extents. Like the other two declarative languages, QVTo is heavily based on the declarative (Essential) *Object Constraint Language* (OCL) [16], which is extended by QVTo's own side-effect expressions via the *ImperativeOCL* component.

3 QVT Engine Implementation Infrastructure

In this section, we will describe the technology used at the core of our solution. Its most distinctive aspect is the use of an intermediate, byte-code model transformation language named Atomic Transformation Code which was originally developed precisely to support execution of the QVT specification.

Several projects are picked up and combined in an integral solution capable of executing model transformations specified in the QVT textual concrete syntax. *Eclipse* [6] has been selected as our target platform to implement the solution. Reasons behind this election are unsurprising:

- Eclipse is a powerful open-source platform whose evolution and adoption rate have been growing at a very fast pace, and whose flexibility, extensibility, ease of use, etc. make it an ubiquitous platform.
- Eclipse provides a lot of modeling-related functionality which includes *EMF* [5], as a sound metamodeling infrastructure, or the *MDT-OCL* project, as an implementation of the OCL language.

There are several open-source projects we base our QVT execution project on. A list of related Eclipse and non-Eclipse projects we have used to tackle our solution follows:

- MDT-OCL: The official OCL implementation in Eclipse. OCL is a key piece in any QVT implementation solution, since the three QVT languages rely upon the OCL specification for their definition. Our solution has been aligned with the infrastructure developed for the MDT-OCL project where appropriate.

– UMLX [17]: Another official Eclipse project, currently under the *Generative Modeling Technologies* (GMT) section. UMLX provides a lot of common functionality related to the edition of QVT files. Editors for writing QVTc and QVTr model transformations are being provided in the context of this project.
– LPG: The LALR(k) Parser Generator is a non-Eclipse project hosted in Sourceforge that is used by the MDT-OCL project to parse OCL expressions. It provides grammar extensibility mechanisms, an ideal feature for QVT languages.
– Atomic Transformation Code (ATC) [8]: An open, non-official, Eclipse-based project that consists of a byte-code model transformation language and its related virtual-machine engine. ATC has been built on top of Eclipse and the EMF project. This is the project that actually performs the execution of QVT model transformations and the one which confers the Virtual-Machine quality to the whole solution.

4 QVT Execution Process

QVT is a rather complex specification, full of subtle details that make it very difficult to understand and thus implement. We will explain our particular adopted approach that provides a functional, executable implementation of the QVT specification. The first language elected for execution support was QVTo. Despite its extension, it was expected to align much better to the also imperative ATC-based underlying technology than the other two declarative formalisms present in QVT.

The process involved has been partitioned into a sequence of stages for the QVTo language. First, textual model transformations are parsed to obtain their equivalent abstract representation, which can be seen as a model. Once a QVT model is obtained, it can serve as input in a model transformation that converts it into a semantically equivalent version given in terms of the byte-code ATC language (ATC instances are models themselves). The next step is to launch this ATC model transformation as an ideally transparent substitute of the original QVT model transformation, to be executed upon real models. A more thorough explanation on each stage follows.

4.1 QVT Text Files to QVT Models

OMG's QVT specification defines a textual concrete syntax for each of its languages along with their abstract syntax description. The first step towards execution is to link both representations, therefore obtaining an abstract syntax model representation of the textually written source transformation. The following topics are involved in this first stage:

1. Development of a parser capable of generating a QVT output model from QVT textual input.
2. Development of an editor to assist in the programming of QVT model transformations in textual notation (syntax highlighting, context assist, error marking, ...).

These two subtasks were implemented by Open Canarias for the QVTo language due to lack of good open-source available solutions (see Section 5). The GMT-UMLX project provided similar editors for the two other languages, QVTr and QVTc. These editors share the same goals as QVTo's, that is, to parse the original text in order to produce a model in a user-friendly development environment. Therefore it has not been necessary for us to develop them. They also follow a similar alignment with the MDT-OCL project to QVTo's editor in terms of variable environment sharing or LPG grammar extension.

Since QVT is defined as an extension of two other OMG specifications, EMOF (Essential MOF) and Essential OCL, extending EMF (as a legitimate Essential MOF implementation)

and MDT-OCL was required. Additionally, GMT-UMLX provides a useful intrastructure to easily integrate the developed parser into a multi-page editor which offers several views and notations of the same source transformation integrated in a single window. By attaching our QVTo editor to this infrastructure not only we were able to save a lot of ongoing work affecting this and later stages, but also helped the QVTo editor become a more solid and productive environment in which to write QVT transformations.

The mechanisms that drive parsing QVTo input files to generate QVTo output models are based on the same principles MDT-OCL uses to parse OCL textual expressions, and are summarized here:

- Defining an Ecore-Based Metamodel for the QVTo language (which has been contributed to the GMT-UMLX project). This QVTo metamodel extends the Ecore and the Ecore-bound MDT-OCL metamodels.
- Defining a CST Metamodel for the QVTo language, which extends (and hence, reuses) the MDT-OCL CST metamodel. It comprises the definition of CST nodes for QVTo, ImperativeOCL, and some Ecore concepts complementary to those related with OCL.
- Defining a grammar which covers the whole QVTo language. This implies extending the MDT-OCL EssentialOCL grammar and involves the specification of production rules for ImperativeOCL and other QVTo-specific concepts.
- Defining a CST-building infrastructure for creating CST nodes. It uses and extends MDT-OCL's own infrastructure.
- Defining the AST-building infrastructure for creating AST nodes, which also uses and extends MDT-OCL's.
- Defining the validation actions to validate the generated models, which actually extend the MDT-OCL's own validation actions.
- Defining a QVTo Standard Library as an extension of the MDT-OCL Standard Library. This library, as stated in the specification, and as is usual for any programming language's standard library, has been defined in terms of its language. Consequently, in this case it is represented as a model that conforms to the QVTo metamodel.

This stage poses certain technological challenges. For instance, the MDT-OCL project's design did not originally account for language extensions. QVT is not the only language that builds upon OCL, the same does the M2T language, and others may follow shortly. Enhancements to the MDT-OCL project's language extensibility are being successfully contributed.

A previously attempted alternative to supporting QVTo via ATC consisted of eliciting ATC models directly from QVTo text parsing in a monolithic Java approach. A working prototype was produced very quickly, but was soon discarded in favour of the more flexible, model-aligned, approach explained in this work. The main problems with this alternative were that it was difficult to keep QVTo textual parsing and ATC generation mixed together in a single module.

Increasing language support incrementally was found not always as smooth as desirable. Instead, separation of concerns in this case not only keeps efforts in each stage focused, independently of each other, but also opens the opportunity of independent reuse for each stage. That is, QVT editors for the three languages can be integrated in other projects or tools. Likewise, the QVT to ATC translation infrastructure can be used by tools that produce AST models from QVT textual parsing in order for them to achieve true QVT model transformation execution, or to enjoy an additional alternative execution path.

4.2 QVT Models to ATC Models

Once a QVT model has been output from the compilation of its respective QVT input file via the editor-parser process, the next step would be to make it executable, as illustrated in Figure 2 for QVTo.

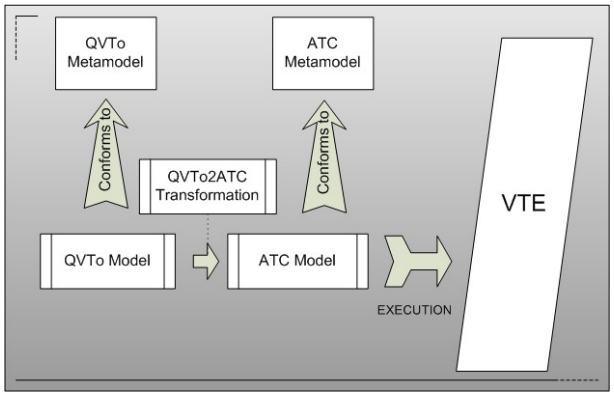

Fig. 2. QVTo translation schema. *VTE* stands for *Virtual Transformation Engine*, and is the engine that executes ATC models. The same process is applied for the QVTc language.

The first alternative that comes to mind would be to develop a transformation engine capable of interpreting and executing QVT models. This is expected to be the most common approach to the problem. However, as it was explained before, our approach involves the use of an intermediate layer, a Virtual-Machine model transformation engine based on the ATC *byte-code* language. This means that QVT models must be translated into ATC prior to execution.

Reflective Model-Driven Engineering [4] is increasingly becoming a powerful instrument in the evolution of the MDE field. Once model transformation instances take the shape of models, they can be treated as any other ordinary model, and thus be subject to the MDA's PIM-PSM evolution approach. This can be seen as a model transformation development process [14]. Since both source and target artifacts on the stage explained in this section are models, a model transformation to automatically produce the target from the source is an adequate choice. We call this kind of model transformations *translation transformations*.

In translating QVTo models into ATC, we could mistakenly assume at first that we face a conversion similar to horizontal, domain mapping transformations, where domain elements from either language are often bound by a nearly one-to-one relationship between each other. This would usually be described via declarative constructs to promote bidirectionality. Instead, a gap in the level of abstraction is encountered, where the one-to-one relationship is unusual. Bridging this gap requires a vertical approach. In this case, bidirectionality, although desirable, is not so important nor easily achievable.

Translation transformations into ATC have been developed for both QVTc and QVTo. These are currently called *QVTc2ATC* and *QVTo2ATC*, respectively. Since execution was required, they had to be developed in a language for which a proven model transformation engine existed and was reliable. We have used a notation close to ATC but of a higher level of abstraction for productivity reasons. Other working language alternatives should be equally valid for this purpose.

Both model transformations have been designed around a common core structure that takes charge of translating MOF constructs (model parameters, operation parameters, and basic transformation structuring), and also OCL expressions. The latter includes the OCL Standard Library, which defines a respectable quantity of operations on basic types for which code capable of translating them into an equivalent ATC manifestation must be developed. Additionally, QVTo defines a Standard Library on its own, and *QVTc2ATC* must account for two different execution modes: *checking* and *enforcement*.

Execution support for QVTr models has not started yet. However, QVTc is acknowledged as a lower-level version of QVTr. A model transformation named *relToCore* is even defined in the QVT specification to translate QVTr models into QVTc models, a process analogous to the more general Virtual-Machine concept that has been applied in this work. Figure 3 shows the path QVTr models traverse until execution in our particular solution.

The problem with this *relToCore* transformation is that it is not written in terms of the QVTc language, which would have allowed us to translate it into ATC via *QVTc2ATC* and then be able to apply the production chain of Figure 3. Instead, it is written in QVTr, which means we have to find a suitable QVTr model transformation engine, such as, perhaps, MOMENT-QVT [13], that must be capable of producing the QVTc equivalent version of *relToCore* via the same *relToCore* transformation.

But such models cannot exist since *relToCore* is currently not well defined. The specification states that in QVTr "only first order sets are allowed, i.e., Elements cannot have type set of sets", but unfortunately, the transformation makes use of these collections of collections constructs at some point. We don't know yet if there exist workarounds to this limitation. We believe the QVT specification maintainers will solve this issue in a reasonable amount of time. Otherwise, we could default to developing a direct *QVTr2ATC* model transformation that would transform QVTr models into semantically equivalent ATC counterparts.

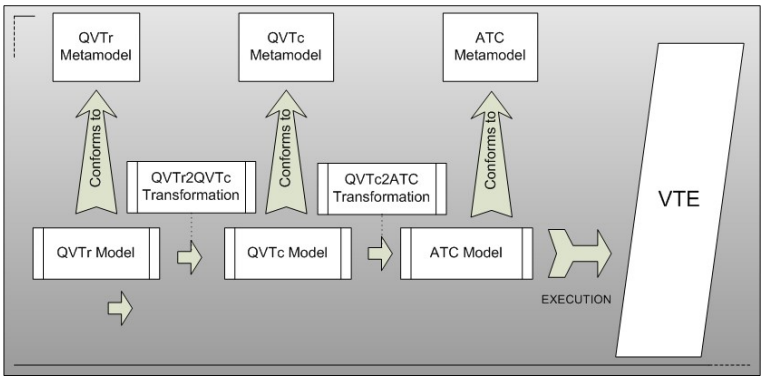

Fig. 3. QVTr translation schema.

4.3 QVT Execution

Now we already know how a translated ATC model transformation is obtained. This is semantically equivalent to the original QVT transformation defined in its textual concrete syntax. The third, final step would be to execute it upon concrete models. The Virtual Transformation Engine (VTE) will ensure that the ATC model's defined activities are properly and orderly executed.

Therefore, this execution stage consists essentially of providing means to specify which models will be involved in the transformation execution. We have developed a generic transformation launcher, which comes with a user interface (depicted in Figure 4) in which such models can be entered in a user-friendly mode. It has been designed as a generic launcher, to which any model transformation engine, even different versions of the same engine, can be bound. A binding for VTE is provided.

Other automated alternatives are worth considering, such as defining a model transformation chain that might represent part of a Software Product Line, in which models to be

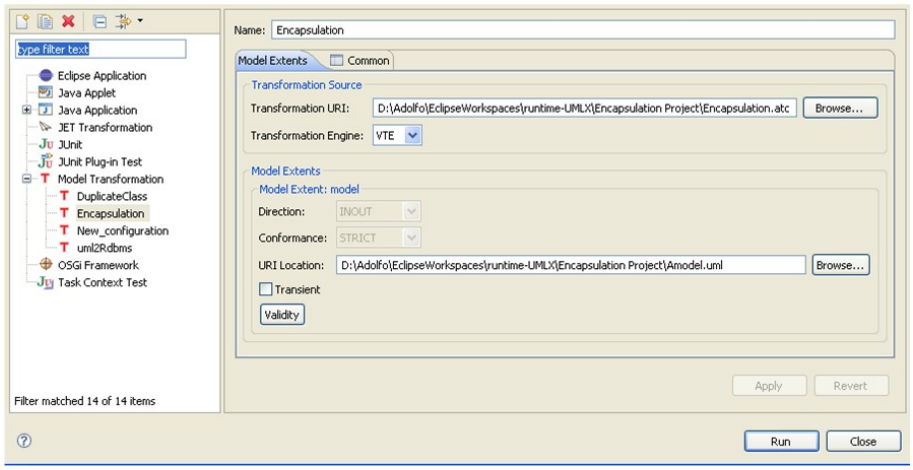

Fig. 4. VTE-bound model transformation launcher UI snapshot.

transformed are always the same, thus their locations, along with that of the transformation to execute, do not change. A chain of several model transformations may then be defined to be executed sequentially as long as user input is minimized or not required at all.

The complete QVT execution process comprising the three stages explained throughout this work is summarized for QVTo in Figure 5.

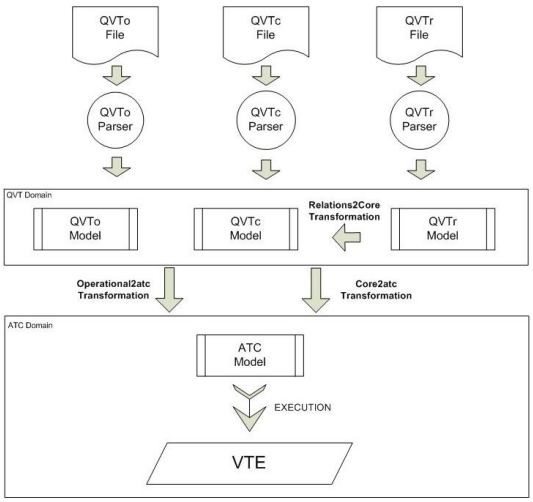

Fig. 5. QVT Execution.

5 Related Work

Other available Eclipse-based QVT implementations exist, such as ikv++'s *medini QVT* [1], a powerful QVTr model transformation engine, or the aforementioned MOMENT-QVT

[13]. The SmartQVT model transformation engine [2] covers the QVTo language and is maintained by some of the people in charge of the QVT specification. In the scope of Eclipse's M2M project, there are two separate project initiatives to bring support to QVTr and QVTc (by Obeo) and QVTo (by Borland). Concerning QVTo, collaboration between Open Canarias and Borland currently involves exchanging ideas and sharing test cases. Borland has announced that it will refactor its QVTo solution so it becomes capable of generating QVTo conforming models out from parsed text.

Eclipse has its own official byte-code model transformation language, the ATL-Virtual Machine, toward which several languages are being compiled, including QVT [9], with an approach that shares many commonalities with ours. Not surprisingly, the group behind this approach is working to extend compatibility to other languages, just like the ATC technology has done (see [7], [3]). The importance of these achievements is that work on either infrastructure may be easily made interoperable with the other by simply providing a translation transformation that bridges between both low-level model transformation languages.

Finally, Open Canarias has applied the same idea of obtaining models via text parsing for the OMG's MOF Models To Text specification [11], also known as *MOF2Text*. This provides ways to transform models into text artifacts. These artifacts can be interpreted as code files, deployment specifications, documents, etc. Similar to what QVT does, the *MOF2Text* syntax is based on that of the OCL language, complemented with its own set of extensions. This means that translation into ATC will reuse the OCL to ATC module already written for QVT.

MTL, a similar component maintained by Obeo has recently started development in the Eclipse's M2T official project. We are also sharing ideas and implementation with M2T's Obeo implementors.

6 Conclusions

In this paper a Virtual-Machine-based QVT engine implementation has been described in terms of design and technology integration. The solution is subdivided into three main parts. These are firstly source QVT text editing-parsing to obtain equivalent abstract syntax trees, which are represented as models. Secondly, model transformations to automatically translate them into the byte-code language named ATC that drives the model transformation Virtual Machine. Finally, these semantically equivalent ATC models can be executed, the user being hidden from the inner details by the UI.

Work is currently very advanced, medium-sized model transformations can usually be smoothly programmed and executed. However, there are still some unfinished areas, such as both the OCL Standard Library, which concerns the three QVT languages, and the QVTo Standard Library. Fortunately there are not currently any technological barriers to achieve full support on that matter. Also, translation support for sophisticated modular component reuse such as QVTo's mapping operations inheriting each other, or even refining QVTr Relations, has not started yet.

Ideally, translation transformations should be writable in terms of a higher-level language than the byte-code, such as QVTo, for maintenance or even evolution purposes. For instance, we can write the *QVTo2ATC* in the same QVTo notation, and then use its model representation obtained via the editor-parser as source for the executable version of QVTo2ATC model transformation written in ATC, to generate an identical or similar version as an output. The semantical equivalence between the output artifact and the handwritten QVTo2ATC version could serve as a powerful test case to assert correctness of the procedure and to further enhance and extend the transformation itself.

Acknownledgements

Thank you Verónica Bollati, Nuria Tejera, Dr. Edward Willink, Christian Damus and Orlando Ávila for your invaluable insight and contributions to this whole project.

References

1. ikv++ homepage. http://www.ikv.de.
2. SmartQVT, a MOF 2.0 Operational Language Model Transformation Implementation. http://smartqvt.elibel.tm.fr/.
3. O. Avila-García, A. E. García, E. V. S. Rebull, and J. L. R. García. Using software product lines to manage model families in model-driven engineering. In *SAC 2007: Proceedings of the 2007 ACM Symposium on Applied Computing, track on Model Transformation*, pages 1006–1011. ACM Press, Mar 2007.
4. J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective Model Driven Engineering. In *Int. Conf. on UML*, pages 175–189, 2003. Available at http:/www.lina.sciences.univ-nantes.fr/Publications/2003/BFJLP03.
5. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework (EMF)*. Addison Wesley, Aug 2003.
6. D. Carlson. *Eclipse Distilled*. Addison Wesley, 2005.
7. J. S. Cuadrado, E. V. Sánchez, J. G. Molina, and A. Estévez. RubyTL a ATC: un caso real de transformacin de transformaciones. In *DSDM'07: Proceedings del IV Taller Sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicationes*, Sep 2007.
8. A. Estévez, J. Padrón, E. V. Sánchez, and J. L. Roda. ATC: A low-level model transformation language. In *MDEIS 2006: Proceedings of the 2nd International Workshop on Model Driven Enterprise Information Systems*, May 2006.
9. F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *1st track on MT - SAC 2006: Proceedings of the Symposium on Applied Computing*. ACM Press, Apr 2006.
10. OMG. MOF 2.0 Query/View/Transformation specification. Formal Specification formal/2008-04-03, OMG, Apr 2008. http://www.omg.org/docs/formal/08-04-03.pdf.
11. OMG. MOF Model to Text Transformation Language. Technical Report formal/2008-01-16, OMG, Jan 2008. http://www.omg.org/docs/formal/08-01-16.pdf.
12. J. Padrón, J. D. García, E. V. Sánchez, and A. Estévez. Implementación de un motor de transformaciones con soporte MOF 2.0 QVT. In *DSDM'05: Proceedings of the II Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones*, Sep 2005. Available at http://www.dsic.upv.es/workshops/dsdm05/files/07-Padron.pdf.
13. P. Queralt, L. Hoyos, A. Boronat, J. A. Carsí, and I. Ramos. Un motor de transformación de modelos con soporte para el lenguaje QVT Relations. In *DSDM'06: Proceedings of the III Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones*, Oct 2006. Available at http://www.dsic.upv.es/workshops/dsdm06/files/dsdm06-14-Queralt.pdf.
14. E. V. S. Rebull, O. Avila-García, J. L. R. García, and A. E. García. Applying a model-driven approach to model transformation development. In *MDEIS'2007: Proceedings of the III International Workshop on Model-Driven Enterprise Information Systems*, Jun 2007.
15. E. V. Sánchez, V. Roldán, A. Estévez, and J. L. Roda. Making a case for supporting a byte-code model transformation approach. In *Symposium on Eclipse Open Source Software and OMG Open Specifications*, Jun 2008.
16. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison Wesley, 1998.
17. E. D. Willink. On re-use of OCL for QVT model-checking editors. Technical report, Eclipse UMLX Project, Jun 2007. Available at http://www.eclipse.org/gmt/umlx/doc/MDD-TIF07/MDD-TIF07-QVTEditors.pdf.