State of the Art of QVT: A Model Transformation Language Standard

Ivan Kurtev

Software Engineering Group, University of Twente, The Netherlands
kurtev@ewi.utwente.nl

Abstract. Query/Views/Transformation (QVT) is the OMG standard language for specifying model transformations in the context of MDA. It is regarded as one of the most important standards since model transformations are proposed as major operations for manipulating models. In the first part of the paper we briefly summarize the typical transformation scenarios that developers encounter in software development and formulate key requirements for each scenario. This allows a comparison between the desirable and the formulated requirements for QVT. Such a comparison helps us to initially evaluate the adequacy of the QVT language.The second part of the paper focuses on the current state of the standard: the language architecture, specification, paradigm, and open issues. The three QVT sublanguages Operational Mappings, Relations, and Core are briefly described. Special attention is given to the currently available and expected tool support.

Keywords: Model transformations, QVT, MDA, MDE.

1 Introduction

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

MDE principles may be applied by using different modeling languages, transformation languages, and tools. One example of such an approach is Model Driven Architecture (MDA) initiative proposed by OMG. MDA distinguishes between platform independent models (PIMs) and platform specific models (PSMs). This classification is motivated by the constant change in implementation technologies and the recurring need to port software from one technology to another. Furthermore, MDA proposes its set of modeling standards: (i) to define models and modeling languages (UML [12], UML profiles, MOF [14]); (ii) to represent and exchange models (XMI) [11]; (iii) to define model constraints (OCL) [16]; (iv) to specify transformations on models. The last operation is proposed as the main way to manipulate models in MDA. The important role of model transformations motivates the effort that OMG took to define a standard language for model transformations aligned with the rest of OMG standards. The result of this effort is the standard QVT MOF 2.0 language [17] which at the time of the writing of this paper is in the final standardization phase.

Transformation technologies are not something new in software engineering. A compiler is actually a transformer that produces an artifact at a lower level of abstraction from another artifact at a higher level of abstraction, possibly expressed in a language that matches the problem domain better. The standardization of XML as an exchange data format gave birth to XSLT, a standard transformation language for XML documents. A similar effort is observed in the domain of Semantic Web. Many more examples may be given from the domain of data engineering, a discipline that is facing hard interoperability and data heterogeneity problems and approaches them by applying data transformations.

In software engineering we witness a stable progress in at least two fields: program transformations and graph transformations. This gives us a valuable insight about the problems we need to tackle and about the advantages and disadvantages of the available techniques.

In the light of this discussion an interesting question emerges. How does OMG derive the QVT standard? What are the transformation scenarios that will be addressed and what kind of properties the language will possess? Unfortunately, a quick look at the QVT Request for Proposals [13] (QVT RFP) document shows that the most important requirements for the language concern its alignment to the existing OMG standards and the software engineering qualities of the language take the role of non-mandatory requirements.

The purpose of this paper is twofold. First we would like to outline a set of transformation scenarios commonly found in software and data engineering. Each scenario naturally poses a set of requirements. They can be compared to the requirements and rationale behind QVT. Second, we present an overview on QVT and the current tool support for this language.

The paper is organized as follows. Section 2 gives a larger context for discussion by considering several well-known transformation scenarios. Section 3 presents the requirements for QVT as described in the QVT RFP. Section 4 explains the overall architecture of the QVT language and briefly describes the three QVT sublanguages: Relations, Core, and Operational Mappings (OM). Section 5 lists the currently available tools for specifying and executing QVT transformations. Section 6 concludes the paper.

2 Transformation Scenarios in Software and Data Engineering

The previous section mentioned that transformations are applied to solve problems in many domains. Those problems, however, generally differ and may pose a set of different requirements. These requirements should be the starting point for the development of a transformation language. In this section we analyze two domains of application of transformations: software development based on the principles of MDE and heterogeneous data translation.

2.1 Model Driven Software Development

Model Driven Software Development (MDSD) applies the principles of MDE in the development of software systems. A system is specified as a set of models that are repetitively refined until a model (models) with enough details to implement the system is obtained. The implementation step should be automated as much as possible by code generation from the models.

When applied in practice, this general scheme of MDSD processes should follow and address some stable general principles and scenarios of software development such as separation of concerns, iterative development, refactoring, reverse engineering, and others. These principles and scenarios take a concrete shape in the context of MDSD and put forward requirements and open questions. In this section we focus on the role of model transformations related to various aspects of MDSD.

Refinement Steps in MDSD. Regardless of the actual development methodology an MDSD process can be seen as a series of refinement steps. More abstract models are transformed into more detailed ones being closer to the actual system. The most important requirement for this refinement process is the semantics preserving property of transformations. Fig. 1 illustrates the process of refinement and the relations of the models to the system.

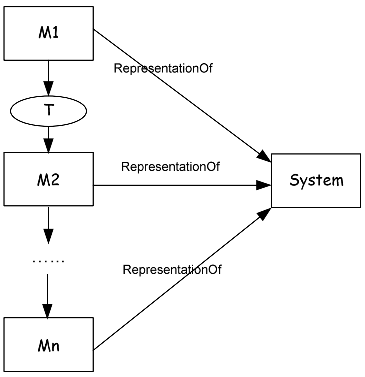

Fig. 1. Refinement of models in MDSD and their relation to the system to be implemented

The vertical dimension denotes the refinement from more abstract to more concrete models. Since all models are representations of the same system every transformation step should preserve the intended meaning of the source model and eventually bring new details. The refinement steps may encode useful design knowledge based on design and architectural patterns, idioms related to a particular implementation technology, and standard transformations such as UML to Java or UML to J2EE. Semantics preservation should ensure that the produced system will behave as it is specified in the models.

Separation of Concerns. The principle of separation of concerns helps in managing the complexity in development of large software systems. The application of this principle in MDSD leads to more than one model of the system developed from different points of interest. These models may be refined independently from each other along a single track as shown in Fig. 2. At a certain moment these models (or code) must be integrated to obtain a complete system.

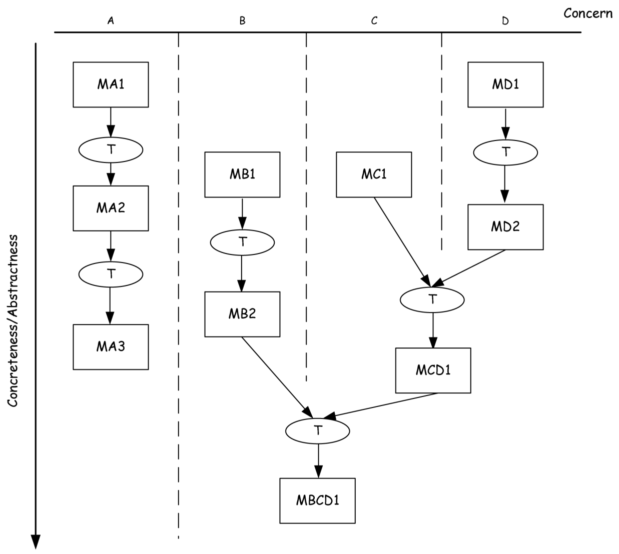

Fig. 2. Refinement and composition of models representing different concerns

Fig. 2 shows an example organization of models in a two-dimensional space. The vertical dimension indicates the level of abstraction of models. The horizontal dimension indicates that the models may be separated according to the problem they solve or the point of interest taken to develop a model. Such points of interest are known as *concerns*. Fig. 2 shows a horizontal dimension with four concerns: *A*, *B*, *C*, and *D*. At a certain stage models of different concerns may be composed. Composition of models is treated as a transformation that takes at least two input models and generates an output model. Both the refinement and composition transformations must be semantics preserving since the resulting models represent the same system as the source models.

Two issues arise in relation to the principle of separation of concerns. The first one is the consistency between models belonging to various concerns. Models of different concerns should be treated separately but ultimately they represent the same underlying system. Therefore, the independent changes over the models should not produce inconsistent results.

The second issue is the composition of models which is a special kind of transformation with at least two input models. The composition problems may expose specificities that may require a specialized language optimized for composition tasks [3].

Iterative Development and System Evolution. Contemporary software development methods promote iterative processes to manage complexity and to deal with identification of system inadequacy at an earlier stage of development. Every new iteration changes (adds to) the functionality of the system. Changes may also occur when the system evolves due to changed requirements during the maintenance phase. The impact of a change on a system developed according to the MDE principles requires changes to the existing models and integration of the newly developed models with existing ones. Since the system is developed as a series of transformations over models a change in one model must be propagated through the rest. The propagation may be in two directions: to models derived from the changed model and to models from which the changed model is derived.

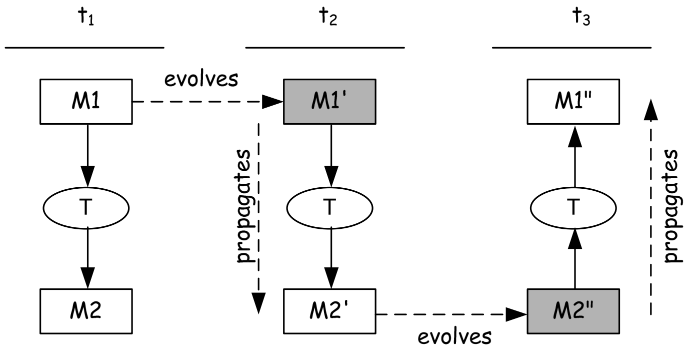

Fig. 3. Evolution and change propagation in MDE

Fig. 3 shows three moments in which models sequentially evolve. After the initial transformation is executed at moment *t1*, subsequent changes of the source and target models (at moments *t2* and *t3*) may require forward and backward change propagation. Two problems arise here: how to identify the required changes and how to apply them on existing models at a low cost.

The first problem is known as *traceability problem.* A trace allows a software artifact to be related to its predecessors that were developed during earlier phases of development. For example, a Java class may be traced back to its design class, analysis class, and ultimately to the requirement that motivates its presence in the system. In the case of model transformations a trace would relate elements in the source model to the created elements in the target model. By transitivity, traces may be detected over the chain of transformations. If a model element is changed, traces help in detecting the changes in the model elements derived from it and ultimately in the system code. Traceability support may not be a property of a transformation language. It may be provided by the transformation engine or the developer may take care of creating and using traces.

The second problem is how to apply the identified changes. One naive solution is to execute again the transformation on the modified model. However, for large models this may be time consuming, especially when there is a long chain of model refinements and compositions. A more efficient solution is to transform only those elements that are modified and to do only incremental changes at the target models.

It should be noted that this scenario does not necessarily call for bidirectional transformation programs. The two directions may be supported by two different transformation programs.

2.2 Data Translation Problems in Data Engineering Domain

Data translation, data mapping, and data integration are among the important subfields in data engineering. In this section we consider a real-life scenario that requires solving data translation problems. The scenario is generalized and it is shown that conceptually it exemplifies the classical data and schema translation problem.

The most important and challenging problem in data translation is the problem of *heterogeneity*. Data come from various sources, they are usually autonomous (controlled by different organizations) and distributed, structured according to different data models. To illustrate the complexity of the problem we give a list of some data formats used in practice: ER, Relational Model, Object-Relational Model, XML, SGML, comma-separated data, Excel sheets, Latex documents, Word documents, etc. Even relational data stored in systems coming from different vendors expose some differences.

Consider a scenario in which geographically distributed development teams work on a common product. Teams use different tools for bug tracking. One team uses Mantis, the second team uses Bugzilla, and the third team uses simple Excel spreadsheets to describe bugs. Teams are at different levels of maturity and may use different development processes. There is a need for exchanging information about bugs among the teams. However, every tool uses its own data format for bug description. Moreover, the conceptual models behind every tool used to describe bugs may also differ.

The scenario is illustrated in Fig. 4.

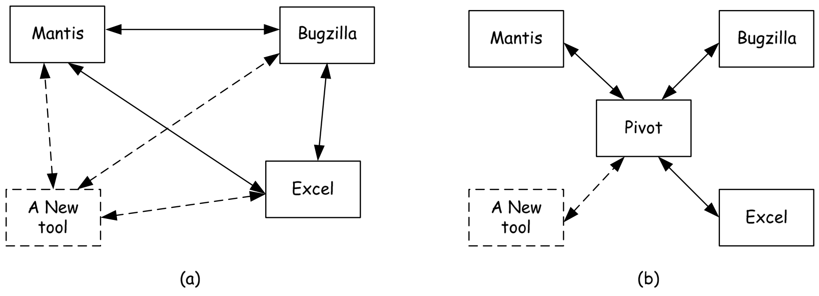

Fig. 4. Tool interoperability problem in bug management

Fig. 4a shows one possible way for interoperability in which there are bridges for every couple of tools. If a new team joins the project a potential new bug tracking system will be used. Then bridges must be built from the new tool to the existing tools. Fig. 4b shows a second way to handle the interoperability: a pivot model is defined that unifies the models used by the tools. Then a bridge is defined between the pivot model and every tool.

The scenario shown above may be generalized to the well known problem of schema and data translation [1]. It is illustrated in Fig. 5. We intentionally use terminology specific to the data engineering domain. We have three levels: *database*,

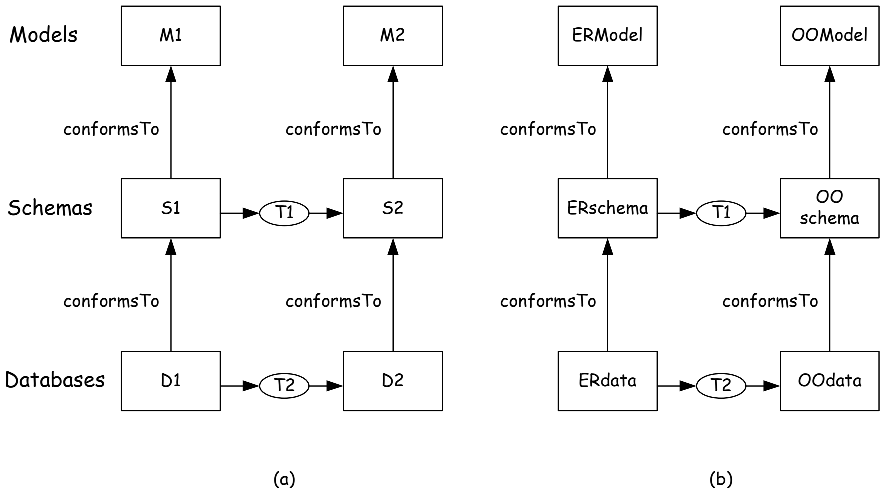

Fig. 5. Schema and data translation problem in data engineering

schema, and *model*. Databases conform to schemas and schemas conform to models. This three-level organization corresponds to the three levels of *model*, *metamodel*, and *metametamodel*.

The schema and data translation problem is formulated as follows. Given two models *M1* and *M2*, a source schema *S1* conforming to *M1*, and a source database *D1* conforming to *S1*, find a translation *T1* that generates a target schema *S2* conforming to the model *M2*, and a translation *T2* that translates the database *D1* to a database *D2* conforming to *S2*. Fig. 5a diagrammatically shows the problem and Fig. 5b gives a concrete example. An interesting question is if it is possible to automatically derive *T2* from *T1*.

The main observation on this problem is that it may involve a large degree of heterogeneity. We also have two possibilities for translations between a pair of models: *lossless* and *lossy* transformations. This depends on the level of compatibility between the schemas/models. In data translation we are interested in preserving the information as much as possible across models and schemas. This requirement is known as *preservation of information capacity* [7, 8].

3 QVT Requirements

After the presentation of two problem domains and the requirements they pose to model transformation systems we present the QVT standard proposed by OMG.

The requirements for the QVT language are described in the formal QVT Request for Proposals (QVT RFP) [13] issued by OMG. Here we briefly summarize the requirements without repeating them in full. QVT requirements are divided into *mandatory* and *optional* requirements.

Mandatory requirements:

- **Query language:** Proposals shall define a language for querying models;
- **Transformation language for MOF models:** Proposals shall define a language for transformation definitions. Definitions describe relationships between source and target MOF metamodels;
- **QVT abstract syntax in MOF:** The abstract syntax of the QVT languages shall be described as MOF 2.0 metamodel;
- **Declarative language:** The transformation definition language shall be declarative in order to apply incremental updates done on the source model immediately to the target model;
- **MOF 2.0 model instances:** All the mechanisms defined by proposals shall operate on models instances of MOF 2.0 metamodels;

Optional requirements:

- **Bidirectional transformations:** Proposals may support transformation definitions that can be executed in two directions (either through a symmetric definition or through a couple of definitions);
- **Traceability between source and target models:** Proposals may support traceability between source and target model elements after transformation execution;
- **Reusable transformations:** Proposals may support mechanisms for reuse of transformation definitions;
- **In-place updates:** Proposals may support execution of transformations where the source and target models are the same;

It should be noted that not all the requirements are listed here. For example, the requirement for view definition is skipped since it is not implemented in the proposed standard.

We also give the definitions of the three concepts that are used in the name of the QVT language (Query, View, and Transformation) as defined by OMG documents.

Query: A query is an expression that is evaluated over a model. The result of a query is one or more instances of types defined in the source model, or defined by the query language.

View: A view is a model which is completely derived from another model (the base model). There is a 'live' connection between the view and the base model.

Transformation: A model transformation is a process of automatic generation of a target model from a source model, according to a transformation definition.

An analysis of the requirements shows that main attention is paid to the alignment of QVT to the rest of the OMG standards, most notably MOF2.0. On the base of the mandatory requirements we may infer the following operational context of the QVT language (Fig. 6).

The operational context is based on the three-level MOF metamodeling architecture. The QVT abstract syntax is defined as a metamodel (*QVT*). QVT transformations are models conforming to the QVT metamodel. Fig. 6 shows an example transformation *Tab*. It is based on the input and output metamodels *MMa* and *MMb*. In general, QVT allows more than one input and output models and their

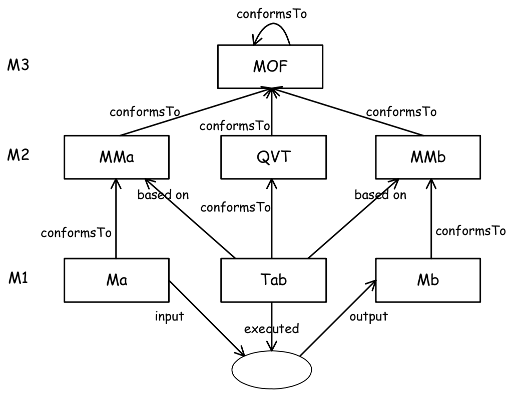

Fig. 6. QVT operational context

corresponding metamodels to be used. For simplicity, we show only single input and output models/metamodels. QVT transformations are executed by taking input models (*Ma*) and producing output models (*Mb*).

The optional requirements correspond to some well-known software quality properties. The RFP does not give any domain analysis and in-depth coverage of possible scenarios in which QVT will be used.

4 QVT Languages

According to Fig. 6, the abstract syntax of QVT is defined as a MOF 2.0 metamodel. This metamodel defines three sublanguages for transforming models. They rely on OCL 2.0 as navigation and query language for models. Creation of views on models is not addressed in the proposal.

4.1 QVT Architecture

QVT languages are arranged in a layered architecture shown in Fig.7. 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.

Relations language provides capabilities for specifying transformations as a set of relations among models. Core language is a declarative language that is simpler than the Relations language. 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).

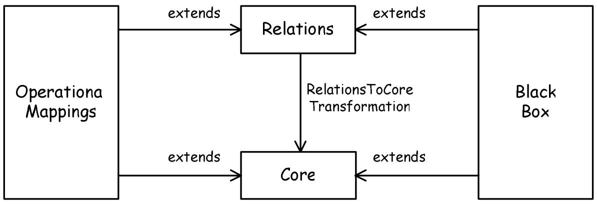

Fig. 7. Layered architecture of QVT languages

Operational Mappings language extends the Relations language with imperative constructs and OCL constructs with side effects. The syntax of Operational Mappings language provides constructs commonly found in imperative languages (loops, conditions, etc.). The QVT specification indicates a relation between Operational Mappings and Core. However, such a relation cannot be identified after inspecting the metamodels of these languages.

Black Box mechanism allows plugging-in and executing external code during transformation execution. This mechanism allows complex algorithms to be implemented 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 their functionality is arbitrary and is not controlled by the transformation engine.

Fig. 7 does not suggest any particular implementation of a 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.

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.

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

Table 1. QVT conformance points for tools

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

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.

It should be noted that the QVT specification does not define the term "QVT compliant transformation language". This term tends to be more and more used. However, its meaning is not clear. It is an attractive possibility to attach a standard label to an existing transformation language. The specification gives us only the possibility to claim compliance for tools and not for languages.

4.2 Relations Language

Transformations written in the Relations language consists of declarations of relations among metaelements. Relations are based on an arbitrary number of domains. When a relation is specified no execution direction is assumed. When a transformation is executed an execution direction is chosen. This opens the possibility to specify bidirectional transformations if their logic permits so. The following transformation scenarios are supported by the Relations language:

- **Check-only:** transformation execution checks if given models satisfy the relations specified in the transformation definition. No new models/model elements are created and no changes are made to the existing models. The answer is *yes* or *no* depending if the relations hold;
- **Unidirectional transformation:** the transformation is executed in a given direction. The target model is created according to the relations in the transformation definition. After the transformation execution, the input and output models satisfy the relations in the transformation definition;
- **Model synchronization:** the transformation engine checks if the relations in a transformation definition hold for a given set of models. If a relation is not satisfied the engine makes changes in the models in order to satisfy the relation. This may lead to creation of new elements, deletion, and update of existing elements. This scenario is motivated by the need for handling model updates in an incremental fashion;
- **In-place update:** in this scenario there is only one model that may be changed according to the specified relations;

Every relation contains 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. The code snippet below gives an example relation.

```
1.  relation AttributeToColumn {
2.     checkonly domain uml c:Class {};
3.     enforce domain rdbms t:Table {};
4.     primitive domain prefix:String;
5.
6.     where {
7.       PrimitiveAttributeToColumn(c, t, prefix);
8.       ComplexAttributeToColumn(c, t, prefix);
9.       SuperAttributeToColumn(c, t, prefix);
10.    }
11. }
```

In a hypothetical transformation that transforms UML class models to relational schemas there is a relation between UML attributes and columns of relational tables. The relation *AttributeToColumn* specifies this. It consists of three domains: *uml* (line 2), *rdbms* (line 3), and one primitive domain that allows passing strings to the relation in the form of a parameter (line 4). In order to hold, the relation must satisfy the object patterns in the domains and to have the condition in the *where* clause (lines 6-10) evaluated to true. The *where* clause illustrates the possibility for invoking one relation from another one.

The keywords *checkonly* and *enforce* play an important role for the semantics of the transformation. Checkonly indicates that the domain elements (in this case UML classes) cannot be changed (i.e. they are read-only) by the transformation execution. Enforce indicates that the engine should change the elements of the domain to ensure the relation. On the basis of the concrete transformation scenario these keywords have different effect on the domains. For example, if a unidirectional transformation is executed from classes to tables then the *uml* domain will be used for matching and the *rdbms* domain will be created. In this scenario the meaning of *enforce* is creation of new elements. If two models already exist and the transformation is executed to synchronize them, changes are allowed only in the enforced domains.

4.3 Core Language

Core language is a declarative language that is simpler than the Relations language. Transformation definitions written in it tend to be longer than the equivalent definitions written in Relations language. Traceability links are treated as ordinary model elements. The developer is responsible for explicitly creating and using the links. Both languages support the same set of transformation scenarios. The rationale behind Core is to support bidirectional incremental transformations. An ideal execution engine for Core should be event-based: every modification in one model is immediately handled and the required modifications in the other models are performed. The following is a snippet taken from a Core transformation specification.

```
map attributeColumns in umlRdbms {
  check enforce rdbms (t:Table) {
        realize c:Column|
        c.owner := t;
        c.key->size()=0;
        c.foreignKey->size()=0;
  }
  where (c2t:ClassToTable| c2t.table=t;){
        realize a2c:AttributeToColumn|
        a2c.column := c;
        c2t.fromAttribute.leafs->include(a2c);
        default a2c.owner := c2t;
  }
  map{  check enforce rdbms (ct:String) {c.type := ct;}
        where (p2n:PrimitiveToName){
             a2c.type := p2n;
             p2n.typeName := ct;
        }
  }
  map {..............................................................................}
```

A transformation in Core is a set of *mappings*. Mappings roughly correspond to relations in the Relations language. Mappings can be nested. The concepts of *enforced* and *check* domains are also available.

4.4 Operational Mappings

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. The syntax of Operational Mappings language provides constructs commonly found in imperative languages (loops, conditions, etc.). Transformations are always *unidirectional.*

```
1. transformation SimpleUML2FlattenSimpleUML(in source : SimpleUML,
2.                                           out target : SimpleUML);
3. main() {}
4. ........................
5. ...helpers..............
6. ...mapping operations....
7. mapping Class::leafClass2Class(in model : Model) : Class
8. when {not model.allInstances(Generalization)->exists(g | g.general
9.                                                          = self)}
10. {name:= self.name;
11.  abstract:= self.abstract;
12.  attributes:= self.derivedAttributes()->
13.                    map property2property(self);
14. }
```

A transformation in Operational Mappings always has an entry point from which the transformation execution starts. This is the mapping called *main* (line 3). From *main* other mappings may be invoked. The body of the transformation definition contains mappings and helper operations. An example of a mapping is called *leafClass2Class* (lines 7-14). This mapping creates an UML class from every UML class that satisfies the guarding condition specified in the *when* clause (lines 8-9). The properties of the created class are assigned with values in the body of the mapping (lines 10-13). It is possible to invoke other mappings from the body of the current one (the keyword *map* in line 13). In that way the execution order among the mappings is imperatively specified.

4.5 Discussion

In section 2 we outlined several transformation scenarios. We observe a diversity of transformation problems that may require different transformation techniques. A logical question is if it is possible to handle these scenarios by a single transformation language in a satisfactory way. The answer is probably no. This is implicitly supported by the fact that QVT is not a single language. It is a suite of three languages that covers both the imperative and declarative paradigm, and addresses several transformation scenarios. Here we discuss briefly every scenario and how it can be handled by the QVT languages.

Regarding the semantics preservation property of model refinement, the QVT specification and the RFP do not require support for checking this. It is not clear yet what type of reasoning may be performed over QVT programs. We expect that a meaningful reasoning would require a limited version of the languages.

Model composition may be regarded as a transformation from at least two input models to a composed model. From that point of view, QVT supports model composition in general. There are proposals for model composition languages [3] specialized in model composition only.

Performing incremental bidirectional transformations is one of the scenarios in QVT Relations. It is somehow unclear how this scenario is implemented in the current engines. The approach suggested in the specification is to execute the transformation afresh by performing the required pattern matching and to execute only the required changes in the models. More experience is needed to judge if this approach provides satisfactory performance results.

QVT specification does not address data translation problems. Historically, the language is proposed as a solution to software development-related problems. The need for information capacity preservation is not analyzed. Due to the alignment of QVT to the OMG standards we may claim that from the data format point of view QVT transformations operate on XMI data. QVT is applicable in data engineering if suitable translators from and to XMI are available.

We may speculate about the need for domain-specific transformation languages adapted to a specific problem. From that point of view OMG proposes QVT as a general purpose transformation language similar to the role that XSLT plays in the XML domain. Some of the scenarios described in section 2 may require a specialized and eventually less expressive transformation language.

5 QVT Tools

Current tool support for the QVT languages is in its infancy. This is due to several reasons. First, the specification is not officially finalized and still unstable. Second, providing a mature tool requires time and efforts. Most tools do not support all the features of the languages. Once a tool is made available, the feedback from the user community is crucial. Practically all the current tools are dealing with bug fixes and are gaining experience from real life usage. Regardless the stability of the language specification many pragmatics issues are involved ranging from syntax-highlighting and visual syntax editors to the availability of comfortable debug facilities. All these make the current description of the tool support valid for a limited period of time. In this section we report on the tools available at the time of the writing of this paper.

Table 2 summarizes the currently available QVT tools. It is followed by more information on every tool.

Table 2. Tool support per QVT language

QVT Tools per Language	
Core	• A commercial add-on to OptimalJ
Relations	• IKV++ medini QVT • Tata Consultancy ModelMorf • MOMENT-QVT • Eclipse M2M Relations2ATLVM
Operational Mappings	• Borland Together Architect 2006 • SmartQVT • Eclipse M2M OM2ATLVM

Core Language

The Core language is supported by an add-on to the commercial tool OptimalJ provided by Compuware. However, OptimalJ is now in maintenance phase and its future development is questionable. It is expected that an open source implementation of a Core engine may be provided.

Relations Language

Relations currently enjoys the largest tool support. The *medini QVT* [5] developed by IKV++ is an Eclipse based interpreter with syntax highlighting editor, code completion, and debugging facilities. It is available as a part of a commercial suite and as a free downloadable distribution for non-commercial purposes.

One of the original contributors to QVT that proposed the Relations language is Tata Consultancy. They provide a Java-based engine known as *ModelMorf* [9]. Currently ModelMorf is a command line tool. The web site indicates the plan to provide a commercial tool for Relations that implements both textual and visual syntax.

MOMENT-QVT [10] is an MDE project that is based on the term rewriting formalism MAUDE. It plans to provide implementation of OCL and QVT Relations.

Operational Mappings Language

Borland provides both an interpreter and a compiler to Java for one of the earlier QVT OM specifications. It is a part of Borland Together Architect 2006 for Eclipse. 15 days trial is available for download.

SmartQVT [18] is an open source Eclipse-based compiler for QVT Operational Mappings provided by France Telecom, the original initiator of QVT OM.

Both Together Architect and SmartQVT provide a front-end for Operational Mappings that can be used to parse transformation programs and obtain a model conforming to the QVT abstract syntax.

Eclipse M2M Project
M2M [4] is an open source project under Eclipse that aims at providing implementations for QVT and ATL [6]. M2M consists of three components: Procedural QVT (Operational Mappings), Declarative QVT (Relations and Core), and ATL. The committers in this project are: INRIA, Borland, and Compuware. The ATL Virtual Machine is adopted as a basic infrastructure for the project. Compilers from QVT OM and Relations to ATL VM code are under development. This effort is led by Obeo under the umbrella of the ATL industrialization project [1].

6 Conclusions

In this paper we presented QVT – the OMG standard language for model transformations in MDA. QVT is closely integrated with the existing suite of OMG standards, most notably with MOF 2.0 and OCL 2.0.

We believe that the standardization of QVT is a step in the right direction. A software standard has a high chance to attract the attention of a larger user community. This should open the possibility to gain experience with the model transformation technology in real life industrial projects. There are also risks, however. A standard lacking formal ground (as the current QVT specification), not supported by tools with industrial quality may compromise the whole idea behind model transformations. This should encourage the communities working on various transformation technologies to stress the importance of transformation problems in current software engineering practices and to promote alternatives to QVT.

References

1. ATL Pro web site, `http://www.atl-pro.com/`
2. Atzeni, P., Cappellari, P., Bernstein, P.A.: Model-Independent Schema and Data Translation. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 368–385. Springer, Heidelberg (2006)
3. Bézivin, J., Bouzitouna, S., Del Fabro, M.D., Gervais, M., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.: A Canonical Scheme for Model Composition. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 346–360. Springer, Heidelberg (2006)
4. Eclipse M2M Project, `http://www.eclipse.org/m2m/`
5. Medini QVT, `http://www.ikv.de`
6. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
7. Miller, R., Ioannidis, Y., Ramakrishnan, R.: The Use of Information Capacity in Schema Integration and Translation. In: Agrawal, R., Baker, S. (eds.) VLDB 1993, pp. 120–133. Morgan Kaufmann, San Francisco (2003)

8. Miller, R., Ioannidis, Y., Ramakrishnan, R.: Schema equivalence in heterogeneous systems: bridging theory and practice. Inf. Syst. 19(1), 3–31 (1994)
9. ModelMorf: A model transformer, `http://www.tcs-trddc.com/ModelMorf/`
10. MOMENT Project, `http://moment.dsic.upv.es/`
11. OMG/XMI XML Model Interchange (XMI) OMG document ad/98-10-05 (1998)
12. OMG. OMG Unified Modeling Language Specification v. 1.4. OMG document (2001)
13. OMG. MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10 (2002)
14. OMG. Meta Object Facility (MOF) Specification. OMG document formal/02-04-03 (2002)
15. OMG. MDA Guide version 1.0.1. OMG document omg/2003-06-01 (2003)
16. OMG. Object Constraint Language (OCL), OMG document ptc/03-10-14 (2003)
17. OMG. MOF QVT Final Adopted Specification. OMG document ptc/05-11-01 (2005)
18. SmartQVT Project, `http://smartqvt.elibel.tm.fr/`