A Two-level Calculus for Composing Hybrid QVT Transformations

Claudia Pons [1,2] Roxana Giandini[1,3] Gabriela Pérez[1] Gabriel Baum[1]

[1] *LIFIA, University of La Plata, Buenos Aires, Argentina*
[2] *CONICET, Consejo Nacional de Investigaciones Científicas y Técnicas, Argentina*
[3] *Universidad Abierta Interamericana (UAI), Argentina*
[cpons, giandini, gperez, gbaum]@info.unlp.edu.ar

Abstract

The standard for model transformations QVT offers two dialects: Relations Language and Operational Mappings Language. Each one of these dialects can be used in isolation, resulting in purely declarative transformations or purely imperative transformation respectively; alternatively, both dialects can be combined resulting in a hybrid transformation approach. On the other hand, the availability of compositional approaches to produce complex transformations from smaller units is a major concern in the area of model transformations. Compositional approaches for pure QVT transformations are supported by a number of tools; however no composition technique exists that can consistently manage the hybrid approach. Such partial techniques provide suitable answers to most practical needs; but they do not cover the entire composition spectrum. The aim of this article is to describe a technique for composing model transformations embracing both dimensions - declarative and imperative - so that the hybrid approach can be smoothly supported. Additionally, we report the implementation of a software tool supporting such technique and we sketch its validation.

1. Introduction

In the context of MDD [1] [2] [3] model transformations can be specified and implemented using different tools and languages [4] [5]. The OMG standard for model transformations, QVT [6], is a language with a hybrid declarative/imperative nature. The declarative part of QVT (named Relations Language) allows the creation of declarative specification of the relationships between models. Besides, it supports complex object pattern matching. In addition to the declarative language there is an operational language (named Operational Mappings Language) for invoking imperative implementations of transformations. Various authors suggest using different model transformation approaches for the diverse transformation problems [5]. Declarative transformation approaches are appropriate to specify simple transformations and relations between source and target model elements, while imperative approaches are best suitable for implementing complex transformations that involve detailed model analysis. Each one of these QVT dialects can be used in isolation, resulting in purely declarative transformations or purely imperative transformations; alternatively, both dialects can be combined resulting in a hybrid transformation approach. Pure QVT transformations are supported by a number of tools; for example Medini QVT [7] is an execution engine for the Relations Language, while SmartQVT [8] is an execution engine for the Operational Mappings Language. However no engine exists that can execute the hybrid approach. Most MDD platforms will only provide one optimized execution engine onto which the transformations written in different languages are mapped. When implementing such an approach, the mapping of declarative and hybrid languages onto imperative languages, which are provided with execution engines, is the chosen heuristic [9].

On the other hand, model transformation languages have matured to a point where the focus of interest that initially was set on the expressiveness of languages and on their execution engines, is now moving to other properties such as scalability, maintainability and reusability of the transformation definitions themselves. In this context, the availability of compositional approaches to produce complex transformations from smaller units has become a major concern in the area of model transformations. Composition of model transformations allows us to

create smaller, maintainable and reusable model transformation definitions that can scale up to a larger model transformation. Definitely, the simplest method of composition is to chain several model transformations together by providing the output of one transformation as input for another transformation, however it is well known that software artifacts can be composed in many different ways, such as inheritance, merging, etc. [10].

Regarding the composition of model transformations we confront a dichotomy that is similar to the one observed in the field of execution engines for model transformations. That is to say, there are various approaches for model transformation that offer forms of compositionality either on the QVT Relations language or on the Operational Mappings Language; but there is not composition technique fully supporting the hybrid approach. Such partial composition techniques offer a reasonable solution to a wide range of practical needs. However they do not cover the entire composition spectrum. The aim of our work is to define a technique for composing model transformations embracing both dimensions (declarative and imperative), so that the hybrid approach is consistently supported. For sustaining this technique we make use of the algebraic theory of problems [11] [12] which is an algebraic formalism running on two levels: problems (descriptive entities) and solutions (operational entities), thus it suits the hybrid nature of QVT.

The paper is structured as follows. In section 2 we define the main concepts that comprise the algebraic theory of problems and their connection to QVT. In section 3 we use the algebraic theory of problem to build an integral technique for composing hybrid model transformations. In section 4 we present the main features of a software tool supporting the composition technique. Section 5 contains a discussion on related work; finally section 6 presents the conclusions and anticipated future works.

2. A two-level calculus for QVT

In this section we will summarize the main concepts comprising the algebraic theory of problems, which in the last years has been used as foundation for calculus for program derivation, such as Fork algebras [13]. We opted for applying the algebraic theory of problems due to its two-level formalism that makes it easier to understand the connections between declarative and imperative languages. On top of this formalism we are able to build a calculus for composing hybrid model transformations.

2.1. The basic formalism: problem vs. solution

This theory considers two main concepts: problem and solution:

A *problem* for this theory is a quadruple P = < D,R,q,I> where D is the data domain and R is the result domain (both subsets of a fixed set U which will be called discourse universe), while q is a binary relation on DxR which is the specification of the problem, i.e. an element d of the data domain D and an element r of the result domain R are in the relation q if and only if r is an accepted result for d in that problem. In other words, q is the condition of the problem. For example, if we want to derive Java code from UML class diagrams, the data domain will consist of UML classes, the result domain will be the set of Java programs and the condition q will relate every UML class d belonging to D with some elements in R, each of which will be an acceptable Java implementation for the UML class d. We may be interested in deriving the code only for persistent classes, which is obviously a subset of the domain of all the UML classes. In this case, we will say that the subset made up from persistent classes is the set of interest instances of our problem¸ which is the fourth element, I, of our quadruple.

On the other hand, a *solution* for the problem P is a function of D in R which fulfills the condition q for the set of interest instances. But, also a solution should hold some property α, which is called the admissibility context. Such α property may be extensional, such as: derivable, calculable etc.; or intentional, such as: efficient, elegant, etc. Let us call α-solutions to functions having such characteristics and let us call Ω_P to the set of all α-solutions of a problem P.

2.2. Declarative dialects vs. imperative dialects in QVT

Problems as well as solutions are expressed by means of statements that are written in a given language which has its own syntax and semantics. Let us introduce some simple considerations about *declarative languages* and *imperative languages,* from the perspective of the algebraic theory of problems, pointing out the difference between syntactic and semantic aspects.

Problems are expressed by means of statements that are written in a declarative language $\mathcal{L}_{\mathcal{D}}$ which has its own syntax and a semantics given by a function μ. The role of this semantic function μ is to allow us to give a meaning to the statements of problems, associating each statement Spec, written in language $\mathcal{L}_{\mathcal{D}}$, to the problem P = μ[Spec] specified by the statement. Thus, we must distinguish statements from problems. A problem is an abstract and ideal mathematical object. On the other hand, a statement is a concrete linguistic

object, to the effect that its text consists of a group of symbols (or diagrams). The connection between them occurs by means of the semantic function μ which allows us to define problems from its statements.

On the other hand, solutions are expressed by means of programs, now written in a given algorithmic language $\mathcal{L}_{\mathcal{A}}$, which, besides its syntax, has semantics given by a function υ. The role of this function is, as in the case of μ for problems, to associate each (text of) program Impl, written in language $\mathcal{L}_{\mathcal{A}}$, to the α-function $\delta = \upsilon$ [Impl] denoted by Impl. As in the case of problems, it is important to distinguish programs from functions: a function is an abstract and ideal mathematical object, while a program is a concrete linguistic object (to the effect that it consists of a set of symbols or diagrams). The connection between both of them occurs by means of the semantic function υ. That is to say, a program is a description of an α-function.

The QVT Language is a concrete linguistic object for expressing model transformations with a hybrid declarative/imperative nature, so it allows developers to express *problems* as well as *solutions* in the domain of model transformations. In this section we will explain the connection between the QVT language and the algebraic theory of problems. The declarative part of QVT (named *Relations Language*) allows for the creation of declarative specification of the relationships between MOF models. It contains a set of relations, which are the basic units of transformation behavior specification in the relations language. A relation is defined by two or more relation domains that specify the model elements that are to be related, a *when* clause that specifies the conditions under which the relationship needs to hold, and a *where* clause that specifies the condition that must be satisfied by the model elements that are being related. In addition to the declarative language there is an operational language (named *Operational Mappings Language*) for invoking imperative implementations of transformations. This language provides OCL extensions with side effects that allow a more procedural style, and a concrete syntax that looks similar to the syntax of imperative programming languages. An operational transformation represents the definition of a unidirectional transformation that is expressed imperatively. It consists on a list of mapping operations which are operations that implement a mapping between one or more source model elements into one or more target model elements. A mapping operation is syntactically described by a signature, a guard (a *when* clause), a mapping body and a post condition (a *where* clause). A mapping operation is always a refinement of a relation which is the owner of the *when* and *where* clauses [6].

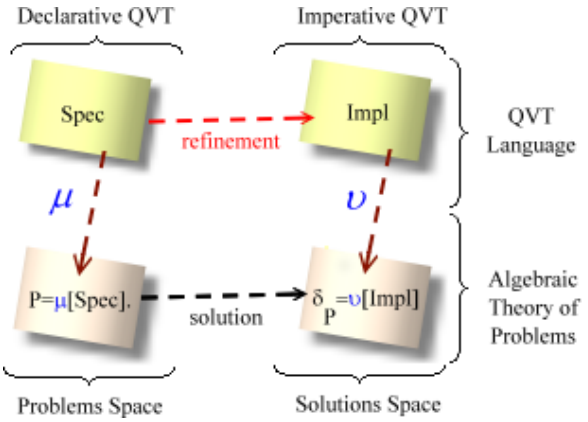

Figure 1: QVT semantics in terms of the algebraic theory of problems.

Thus, when we consider the QVT transformation language, we must bear in mind two different kinds of linguistic constructions: the statements of relations written in declarative QVT language, which denote problems in the terms of the algebraic theory of problems, and the descriptions of mappings written in operational QVT language, which denote solutions in the terms of the algebraic theory of problems. Figure 1 shows the connection between the two QVT linguistic levels and the algebraic theory of problems. Function μ associates declarative expressions with problems, while function υ connects imperative constructions with solutions (although having knowledge about the semantics definitions is not a requirement for understanding this paper, the reader is referred to [14] for reading the formal definition of both semantics functions).

3. The algebraic theory of problems as a foundation for composing hybrid model transformations

The *divide-for-conquer* paradigm is an essential strategy for the development of *programs* and in fact for the resolution of problems in general. It consists in breaking a problem into sub problems whose α-solutions are recombined in an α-solution for the original problem. Breaking means to state the given problem in terms of operations on simpler problems, while the recombination is done by means of the corresponding operations on solutions. The algebraic theory of problems defines two synchronized sets of operations: one set of operations on problems and one corresponding set of operations on solutions. Those operations are described in the following section.

3.1. The basic formalism: operations on problems and operations on solutions

The operations on problems are those of the Relation Algebra extended with a new operator called ∇ (*fork)*. The Relation Algebra [15] is an algebraic structure; a proper extension of the two-element Boolean algebra that is intended to capture the mathematical properties of binary relations. Mathematically, a Relation Algebra is a powerset algebra, $A= (P(V), \cup, \cap, \varnothing, V, \neg, ⨾, ↵, 1)$, such that $(P(V), \cup, \cap, \varnothing, V, \neg)$ is a Boolean algebra and $(P(V), ⨾, 1)$ is a monoid. The operations of the algebraic theory of problems are defined set-theoretically as follows:

union or disjoint:	$R \cup S = \{(x,y) \setminus xRy \vee xSy\}$
intersection or joint:	$R \cap S = \{(x,y) \setminus xRy \wedge xSy\}$
sequence:	$R⨾S=\{(x,y) \setminus \exists z.\ xRz \wedge zSy\}$
fork:	$R \nabla S=\{(x,y*z) \setminus xRy \wedge xSz\}$
complementation:	$\neg R = \{(x,y) \setminus xVy \wedge \neg xRy\}$
empty relation:	$\varnothing = \{\}$
universal relation:	$V = U \times U$
converse :	$R↵ = \{(x,y) \setminus yRx\}$
identity:	$.ı= \{(x,x) \setminus x \in U\}$

The definition of these operations is extended to problems and solutions in a quite straightforward way. In the next sections we describe one of them in detail: the union.

Definition 1. Let P and Q be problems; the problem $P \cup Q$ is defined as the problem whose components are the union of the components of P and Q, as follows,

$D_{P \cup Q} = D_P \cup D_Q$ and $R_{P \cup Q} = R_P \cup R_Q$ and $q_{P \cup Q} = q_P \cup q_Q$ and $I_{P \cup Q} = I_P \cup I_Q$

The involved problems might have different data domain, results and interest instances, although those sets do not need to be disjunctive. The union of problems is commutative, associative and idempotent. There exists the neutral element called 0, that is the problem made up from empty sets.

Let us analyze the union of problems from the point of view of its α-solutions. This is an important matter because if we decompose a problem S into two sub-problems P and Q, such that $S=P \cup Q$, and if we already know α–solutions δ_P and δ_Q for P and for Q respectively, then we would like to be able to calculate some α-solutions for S in terms of δ_P and δ_Q. What does the *theory of problems* say about the solution space of the union with respect to the solution space of the terms? For example, let P, Q and S be problems, such that $S=P \cup Q$. Let's take two α–solutions for these problems, $\delta_P \in \Omega_P$ and $\delta_Q \in \Omega_Q$. If we carry out the join of these solutions – regarding functions as sets of pairs - then the result is not a function because each element belonging to the set $I_P \cap I_Q$ will be attached to two results (one coming from δ_P and the other from δ_Q). Thus, the concept of union of problems cannot be extended to union of solutions in a straightforward way. We need to define a union operation to be applied to solutions. This operation $\delta_P \underline{\cup} \delta_Q$ is basically the join of δ_P and δ_Q together with a choice in the points where both are defined, that is to say:

Definition 2. Let δ_P and δ_Q be functions; the function $\delta_P \underline{\cup} \delta_Q$ is defined as

$(\delta_P \underline{\cup} \delta_Q) = \lambda d.\ \texttt{if}\ \delta_P(d) = \perp\ \texttt{then}\ \delta_Q(d)\ \texttt{else}\ \delta_P(d)$

Therefore, we have the following lemma:

Lemma 1 (union of problems and union of solutions): If δ_P is a solution for P and δ_Q is a solution for Q then $\delta_P \underline{\cup} \delta_Q$ is a solution for $P \cup Q$.

The lemma is illustrated in figure 2. The proof is straightforward.

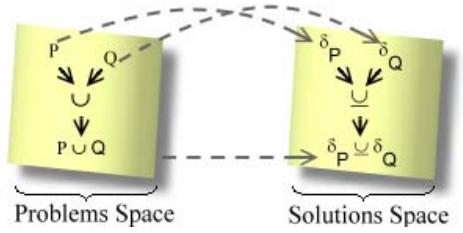

Figure 2. Union of problems and union of solutions

3.2. Applying the theory of problems for composing QVT transformations

We have stated before that problems are expressed by means of statements that are written in a declarative language $\mathcal{L}_{\mathcal{D}}$ which has its own syntax and semantics. Now, we wonder what the relation among the linguistic $\mathcal{L}_{\mathcal{D}}$, constructions and the operations of the algebraic theory of problems is. In order to answer this question, let us suppose that $\mathcal{L}_{\mathcal{D}}$ is a specification language, e.g. a first order logic language. It is desirable that the operations on problems of the algebraic theory of problems could be expressed in $\mathcal{L}_{\mathcal{D}}$. In logic, we have a way of combining statements to obtain the effect of the union of problems: the disjunction connective ∨.

On the other hand, α-solutions are expressed by means of programs, now written in a given imperative language $\mathcal{L}_{\mathcal{A}}$,. Let us suppose that $\mathcal{L}_{\mathcal{A}}$ is a usual programming language, such as Java. It is desirable

that the operations on α-solutions could be expressed in $\mathcal{L}_{\mathcal{A}}$. For example, in Java we have a way of combining programs in order to obtain the effect of the operation $\overset{\circ}{;}$ between α-solutions: the construction of the command composition ";".

Now, due to the fact that QVT is a hybrid language with a declarative/imperative nature, it is desirable that the operations on problems as well as the operations on solutions could be expressed in the QVT itself. That is to say, the QVT language would have the ability to express the decompositions of problem as well as the recombination of their solutions. More specifically, its declarative dialect will provide linguistic constructs to interpret the operations on problems (i.e.,$\cup$, $\overset{\circ}{;}$, etc), while its operational dialect will provide the corresponding constructs for the operations on solutions (i.e., $\underline{\cup}$, $\underline{\overset{\circ}{;}}$, etc.). For instance, we will count with an operation $\cup_{\mathbf{QVT}}$ to carry out the union of two declarative transformations, while on the other hand we will count with an operation $\underline{\cup}_{\mathbf{QVT}}$ to perform the union of two operational transformations. In the following sections we define these operations in more detail.

3.3. Expressing composition of declarative transformations

In this section we introduce the interpretation into QVT of some of the operations on problems displayed in table 1 (the remaining definitions can be read in [16]). For simplicity reasons (and adhering to the notation used in the Section '7.10 Detailed Semantics' of [6]) relational transformations are viewed as having the following abstract structure,

```
Transformation Ti
{
TopLevel Relation Ri1
   {
   Var <Ri1_variable_set>
   Domain:
     <typed_model_1_for_Ri1>
   <domain_1_variable_set_for_Ri1>
       {<domain_1_pattern_for_Ri1>}
   Domain:
     <typed_model_2_for_Ri1>
   <domain_2_variable_set_for_Ri1>
       {<domain_2_pattern_for_Ri1>}
   [when <when_condition_for_Ri1>]
   [where <where_condition_for_Ri1>]
   }
Relation Ri2 {...}
...
Relation Rin_i {...}
}
```

With the following properties:

• `Rij` denotes the Relation number j into the Transformation `Ti`, for j=1..n_i. For simplicity we assume that transformations have only one top level relation.

• `<Rij_variable_set>` is the set of variables occurring in the relation `Rij`.

• `<domain_k_variable_set_for_Rij>` is the set of variables occurring in domain k of the relation `Rij`, for k = 1..2. We restrict the transformation to have exactly 2 domains, also the property `isCheckable` is always true on the first domain while the property `isEnforceable` is true on the second domain (Those restrictions can be loosened later on).

• The term `<domain_k_pattern_for_Rij>` refers to the constraints implied by the pattern of domain k (see [6]).

We will use this abstract structure to informally describe our calculus. For those readers interested in the formal definition of these operations, the work in [16] contains their formal specification given by *pre* and *post* conditions expressed in OCL, in the context of metaclasses of the QVT metamodel.

Example: Union of declarative transformations

The union of transformations is useful for combining different transformation scenarios (i.e., splitting a problem by case analysis). For example, we can join a transformation T1 - that transforms persistent UML classes to Java code - with a transformation T2 - that transforms non-persistent UML classes to Java code - in order to obtain a transformation that is able to produce a result for any UML class (persistent or not). To perform the union, it is required for both transformations to have the same typed models and domain variables. On the other hand, when the scenarios are not disjunctive (i.e., the sets of interest instances of T1 and T2 are not disjunctive) the composition will introduce non-determinism in the resulting transformation. The union of declarative transformations is defined in accordance with the union of problems (see definition 1) in the following way,

Definition 3. Let T1 and T2 be transformations, thus the compound transformation $\mathtt{T1}\cup_{\mathtt{QVT}}\mathtt{T2}$ is:

```
Transformation T1∪QVT T2
{
TopLevel Relation R11∪QVTR21
   {
   Var <R11_variable_set>
   Domain:
    <typed_model_1_for_R11>
   <domain_1_variable_set_for_R11>
   {<domain_1_pattern_for_R11> OR
   <domain_1_pattern_for_R21>}
```

```
   Domain:
    <typed_model_2_for_R11>
   <domain_2_variable_set_for_R11>
   {<domain_2_pattern_for_R11> OR
   <domain_2_pattern_for_R21>}
   when { when_condition_for_R11  OR
   when_condition_for_R21 }
   where { (when_condition_for_R11 AND
   where_condition_for_R11)OR
   (when_condition_for_R21 AND
   where_condition_for_R21)
          }
   }
Relation R12 {...}
...
Relation R1n1 {...}
Relation R22 {...}
...
Relation R2n2 {...}
}
```

The idea behind the union of two transformations T1 and T2 is that the result transformation contains the union of the relations defined in the factors. Relations that are non-top level appear in the resulting transformation without modification, while top level relations have to be combined. The union of relations is defined as follows,

- The name of the result relation is the concatenation of the names of both factors with the tag '$\cup_{QVT}$' intercalated;
- Given that both relations have the same typed models and domain variables, the resulting relation has also the same typed models and domain variables.
- The set of interest instances of the resulting relation is the union of both sets of interest instances (this is achieved by building the disjunction between the domain patterns of each factor and of the *when* clause of each factor);
- The problem condition of the resulting relation is the union of both conditions (this is achieved by building the disjunction between the two conjunctions formed by the *when* and *where* of each factor).

3.4. Expressing composition of operational transformations

The operational composition machinery of QVT is defined at two levels: coarse-grained composition is the capability to combine several complete (and eventually black-box) transformations, whereas fine-grained composition is the capability to combine partial transformations (such as mapping operations). For the purposes of our work we will only consider the coarse-grained composition facilities. Composition of coarse-grained transformation is an essential feature in large transformations. To achieve such composition the QVT language allows us to invoke transformations explicitly. An invocation of a transformation implies two steps: firstly the transformation is instantiated using the **new**() operator, and then the transformation is explicitly invoked through the **transform**() operation. Such invocation mechanism is combined with the usage of access and extension reuse mechanisms. An *access* behaves as a traditional package import, whereas *extends* combines package import and class inheritance paradigm.

The QVT operational formalism provides a set of elementary programmatic constructs to express external chaining of transformations. It offers the possibility to write loops, if-then-else controls, to pass parameters to the transformations and the possibility to retrieve the output of a transformation and to pass as input to the consequent transformation. However, those mechanisms do not provide a clean black box instrument to perform the composition of transformations due to the fact that in QVT we need to write the code for the compound transformation instead of just applying higher level composition operations [17] as we propose in this work. Besides, the current composition mechanisms on imperative transformations do not guarantee any consistency with the corresponding declarative side of the transformation (i.e. the Relations that the Mappings are refining). Let us see the details in the following sections.

Example: The union of operational transformations

Let us suppose we have two imperative transformation I1 and I2 refining declarative transformations T1 and T2 respectively, where T1 and T2 are the transformations specified in the section 3.3. We would like to be able to calculate an imperative transformation I for the compound declarative transformation T1$\cup_{QVT}$ T2 in terms of I1 and I2. That is to say, we need a linguistic construct to combine I1 and I2 in accordance with the semantics of the union of solutions stated in definition 2. Let us call $\cup_{QVT}$ to such operator that is defined as follows,

Definition 4. Let I1 and I2 be imperative transformations, thus the compound transformation I1$\cup_{QVT}$I2 is defined as,

```
transformation I1∪QVTI2 (in:
I1_input_parameter_set,
                                out:
I1_output_parameter_set)
 access I1(), I2();
 main()
 {
```

```
 var returncode :=  new
I1(I1_input_parameter_set,
I1_output_parameter_set).transform();
 if (returncode.failed()) then
   new I2(I2_input_parameter_set,
    I2_output_parameter_set).transform()
 endif
 }
```

In order to be compatible both transformations should have the same parameter set. The combination is carried out by the use of the `if-then` construct that allows us to perform the choice of the adequate transformation to be applied depending on the properties of the source element of the transformation. This composition mechanism is similar to the disjunction of mapping operations provided by QVT, but this one is applied on transformations instead of being applied on mappings.

3.5. Synchronizing the composition of transformations at both levels

Let us show an example of the benefit we could obtain by having the composition machinery accurately defined and synchronized at both QVT levels.

Given that the declarative QVT language has been extended in the previous section, with a linguistic construct called "$\cup_{QVT}$" to interpret the union of problems, and given that the operational QVT language has been extended with a corresponding construct regarding the union of solutions, called "$\underline{\cup}_{QVT}$", now we are able to prove the following lemma, which is illustrated in the top level of figure 3.

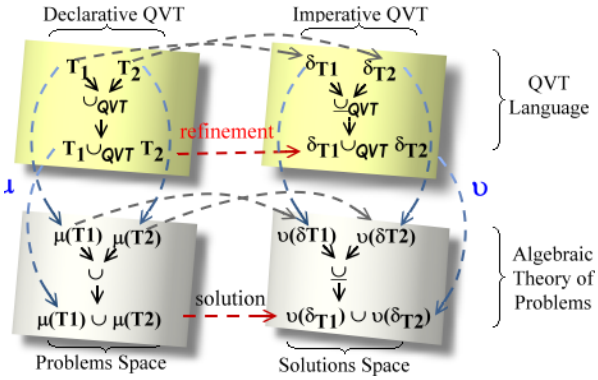

Figure 3. Union of problems and solutions in QVT.

Lemma 3 (union of declarative transformations and their refinements):
Let T_1 and T_2 be declarative transformations, If δ_{T1} is a refinement for T_1 and δ_{T2} is a refinement for T_2 then $\delta_{T1} \underline{\cup}_{QVT} \delta_{T2}$ is a refinement for $T_1 \cup_{QVT} T_2$.

Proof: The proof (which is illustrated in figure 3) is straightforward by applying the definition 1 and the fact that υ and μ are homomorphism. From the hypothesis that δ_{T1} is a refinement for T_1 and δ_{T2} is a refinement for T_2 and by using the formal semantics of the QVT language [14], we build the following chain of deductions (which is illustrated in figure 3):

[1]. $\upsilon(\delta_{T1})$ is a solution for $\mu(T1)$ (by def. 1, given that δ_{T1} is a refinement for T1)
[2]. $\upsilon(\delta_{T2})$ is a solution for $\mu(T2)$ (by def. 1, given that δ_{T2} is a refinement for T2)
[3]. $\upsilon(\delta_{T1}) \underline{\cup} \upsilon(\delta_{T2})$ is a solution for $\mu(T1) \cup \mu(T2)$ (by lemma 1)
[4]. $\upsilon(\delta_{T1}) \underline{\cup} \upsilon(\delta_{T2}) = \upsilon(\delta_{T1} \underline{\cup}_{QVT} \delta_{T2})$ (function υ is homomorphism)
[5]. $\mu(T1) \cup \mu(T2) = \mu(T1 \cup_{QVT} T2)$ (function μ is homomorphism)
[6]. $\upsilon(\delta_{T1} \underline{\cup}_{QVT} \delta_{T2})$ is a solution for $\mu(T1 \cup_{QVT} T2)$ (replacing [4] and [5] in [3])
[7]. Then, $\delta_{T1} \underline{\cup}_{QVT} \delta_{T2}$ is a refinement for $T_1 \cup_{QVT} T_2$ (by def. 1).

Similar results have been proved for the QVT materialization of each algebraic operation.

4. The composition calculator

We have developed a software tool that allows developers to edit and store atomic QVT transformations using both QVT dialects. Then, they can build more complex transformations by applying the composition operations presented in this paper. The tool - that can be downloaded from [18] - was built as an open source plug-in for Eclipse running on top of the EMF metamodeling framework.

The inputs of the tool are text files containing transformations written in a hybrid way; that is to say, each transformation consists of a declaration (written in Relational QVT) and its implementation (written in Operational QVT). In first place, the text files are parsed and the instances of the QVT metamodel, corresponding to the parsed transformations, are created. The operations - that we implemented in the metaclass *DeclarativeTransformation* (and *OperationalTransformation*) of the QVT metamodel - are applied on these metamodel instances, producing a new instance of the metaclass *DeclarativeTransformation* (and *OperationalTransformation*). Finally, the resulting instance is automatically converted back to plain text format. To perform the parsing we used open source tools in particular, we use the parser defined by MediniQVT [7] for the declarative language and the parser of smartQVT [8] for the operational language.

Figure 4 shows the most relevant screenshot. The screen exhibits the selection, from a transformation repository, of the transformations to be composed and the preview of the composition result.

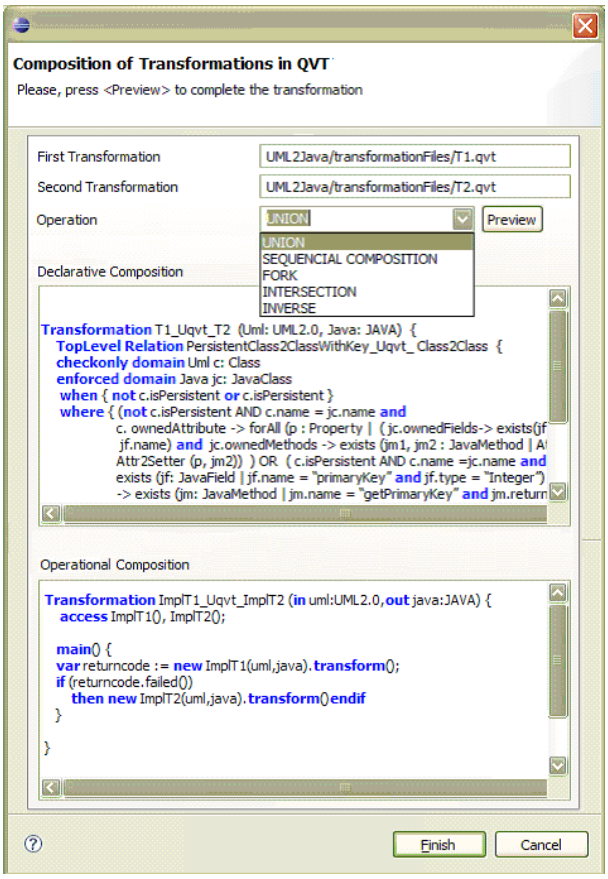

Figure 4. Composition calculator in Eclipse

In order to corroborate that the operations of the calculus are well-defined we designed a set of test cases. Each test consists in running the declarative transformations using the MediniQVT and the operational transformations using the SmartQVT. At first we performed the composition "by hand". On the other hand, we composed the transformations using the calculus and we run the resulting transformation. Finally we carried out a comparison between the resulting models. Several tests of this kind were run for each compound transformation. No difference was observed between the results obtained "by hand" and the results obtained automatically. The conclusion is that the operations are well-defined and implemented with respect to their expected semantics. The detailed description of some examples can be read in [18].

5. Related works

Anneke Kleppe in [19] describes an open environment for model transformations in which users may combine the available tools that implement transformations and apply them to models in various languages. Transformation tools can be combined by using three commonly known combinatorial operators: sequence, parallel and choice. Other approaches that are closely related to the work of Kleppe are the Model Bus approach [20] which tackles composition in the manner of OMG's CORBA and the UMLAUT transformation toolkit [21] that is build with the intension to provide the model designer with a freedom of choice with regard to combinations of transformations to be executed by providing a transformation library and a pluggable architecture. Jon Oldevik in [22] introduces a modeling framework for compound transformations, based on a hierarchy of transformation types, some of which represent simple atomic transformations, while others represent complex transformations. Perdita Stevens in [23] discuss semantic issues and open questions about model transformations in QVT. She considers sequential composition of transformations and a kind of Cartesian product. The paper presents interesting reflections about the problem of composition of transformations, but giving an integral description is out of its scope. Bert Vanhooff et al. in [24] propose a model-based approach to reuse and compose sub-transformations in a technology independent fashion. This is accomplished by developing a unified representation of transformations and facilitating detailed transformation specifications. However, only transformation chaining is considered. On the other hand Dennis Wagelaar in [25] presents an approach, called module superimposition, for internal composition of model transformations written in a rule-based model transformation language. The composition approach is defined on ATL and on the QVT Relations language. Module superimposition works at the granularity of transformation rules in ATL and relations in QVT Relations. To our knowledge this is the only one approach to compositionality on the Relations Language, but it does not support any synchronization with the Operational Mappings Language.

All the approaches described above are focused on the operational aspects of the composition machinery, offering interesting and useful solutions to a wide range of practical needs. However they do not cover the entire composition spectrum. This is the main difference with respect to our approach, which provides a holistic foundation for the composition of QVT model transformations, embracing declarative as well as operational levels in a synchronized way.

On the other hand, GGs (Graph Grammar) [26] are another model transformation technology. Although QVT and GGs have many concepts in common (the reader is referred to [27] to read a deep discussion about this claim), the existing composition mechanisms of GGs are not applicable to QVT in a straightforward way due to the fact that QVT is mainly text-based.

6. Conclusion and future work

QVT has full operational support to allow expressing arbitrary logic for composing operational transformations. However, those mechanisms do not provide a clean black box instrument to perform the composition. The developer is forced to write code using imperative programming language constructs instead of just applying high level composition operators. On the other hand, QVT provides scarce support to combine declarative transformations. Besides, the connection between composition operations in one level and corresponding operations on the other level is unclear and ambiguous. To overcome this drawback, in this article we propose a calculus for composing transformations in QVT, embracing both levels (descriptive and operational). Having the composition machinery accurately synchronized at both QVT levels allows us to fully exploit the divide-and-conquer paradigm in the development of QVT model transformations. That is to say, we are able to break a declarative transformation into sub transformations whose α-solutions might be recombined in an α-solution for the original transformation. Breaking a declarative transformation means to state the given transformation in terms of operations on composing declarative transformations, while the recombination to get the α-solution is done by means of the corresponding operations on the α-solution of each component.

This characterization provides clear composition machinery synchronized at both QVT levels. Additionally, by having the semantics of the QVT language expressed in terms of a formalism we are able to verify whether a QVT imperative statement is a correct refinement of a declarative statement or not.

Regarding the expressiveness of this calculus of transformations, it was proved in [13] that first-order theories can be interpreted as equational theories in fork algebras. Moreover, the algebraic specification can be obtained algorithmically from the first-order specification by using a computable mapping. Consequently, a wide class of problems (at least those that can be described in first-order logics) can be specified in the calculus of fork algebras. Due to the facts that this calculus of transformations is an instance of fork algebra and that QVT is a first-order language, it is trivial to prove that any model transformation that can be specified in QVT can also be specified in the calculus of transformations. However, to fully exploit the expressive power of the calculus it is necessary to count with design strategies allowing us to break up a transformation into smaller parts which can be re-combined by the application of operators to recompose the original transformation. In programming in general, examples of design strategies include case analysis, trivialization, divide and conquer, backtracking, and many more. The development of such design strategies is an expected future work.

Finally, with respect to our selection of the basic formalism, apart form the theory of problems in the area of formal methods we find several approaches supporting the systematic construction of formal specifications and their stepwise refinement into programs, such as the Refinement Calculus [28], the Specware System [29], or the Partch's approach [30]. Any of these formalisms could have been used as a foundation for our calculus, conducting to similar results.

Acknowledgments. This work has been sponsored by Microsoft under the LACCIR RFP 2008 Research Founding Initiative.

References

1. Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, USA. (2003)

2. Object Management Group, MDA Guide, v1.0.1, omg/03-06-01 (2003)

3. Stahl, T. and Völter, M. Model-Driven Software Development. John Wiley & Sons, Ltd. (2006)

4. Czarnecki K. and Helsen S. Feature-based survey of model transformation approaches. IBM System Journal, Vol. 45, No 3, (2006)

5. Jouault, F. and I. Kurtev. On the interoperability of model-to-model transformation languages. Science of Computer Programming, 68(3):114{137. (2007)

6. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification ptc/07-07-07. OMG. (2007)

7. Medini QVT. ikv++ technologies ag. http://www.ikv.de. Accessed in December 2007.

8. SmartQVT. An open source model transformation tool implementing the MOF 2.0 QVT-Operational language. http://smartqvt.elibel.tm.fr/. Accessed in March 2008. (2008)

9.Romeikat, Raphael, Roser, Stephan, Müllender, Pascal and Bauer, Bernhard. Translation of QVT Relations into QVT Operational Mappings. Procs. of ICMT2008 – Int. Conf. on Model Transformation. Switzerland. (2008)

10. Achermann, Franz and Nierstrasz, Oscar. A calculus for reasoning about software composition. Theoretical Computer Science ISSN 0304-3975 vol. 331, no 2-3. (2005)

11. Haeberer, A.M. and Baum, G. and Veloso, P.A.S. On an Algebraic Theory of Problems and Software Development. Pont. Universidad Catolica Research Report MCC 2/87. Rio de Janeiro (1987) (available in http://sol.info.unlp.edu.ar/eclipse/problemTheory.pdf)

12. Veloso, P. Outline of a mathematical theory of general problems. Philosophia Naturalis; vol. 2 No. 1, (1984)

13. Frias, M. and Veloso, P. and Baum, G. Fork Algebras: past, present and future. Journal on Relational Methods in Computer Science. Vol.1, pp.181-216. (2004)

14. Giandini, R., Pons, C. and Perez, G. A two-level formal semantics for the QVT language. In XII Iberoamerican Conference on Software Engineering (CIbSE). Colombia. April 2009. http://sol.info.unlp.edu.ar/eclipse/QVTsemantics.pdf

15. Maddux, Roger D. Relation Algebras, vol. 150 in Studies in Logic and the Foundations of Mathematics. Elsevier. (2006)

16. Giandini, Roxana. Ph.D thesis. University of La Plata. Buenos Aires, Argentina. (2008)

17. Belaunde, Mariano. Transformation composition in QVT. First European Workshop on Composition of Model Transformations at ECMDA 2006. (2006)

18. Composition Calculator. Downloads from http://www.lifia.info.unlp.edu.ar/eclipse/pages/compositorCalculator.htm (2008)

19. Kleppe, Anneke. MCC: A Model Transformation Environment. ECMDA-FA 2006, LNCS 4066, pp. 173 – 187, Spain, June 2006. (2006)

20. Blanc,X., Gervais, M., Lamari, M. and Sriplakich, P.. Towards an integrated transformation environment (ITE) for model driven development (MDD). In Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics, USA, July 2004. (2004)

21. Jézéquel, Jean-Marc and Ho, W. and Le Guennec, A. and Pennaneach, F. UMLAUT: an extendible UML transformation framework. In Proc. of the 14th IEEE Int. Conf. on Automated Software Engineering. (1999)

22. Oldevik, J. Transformation Composition Modeling Framework. DAIS 2005. LNCS 3543. (2005)

23. Stevens, Perdita. Bidirectional Model Transformations in QVT. In Proceedings of MoDELS 2007 Conference. Nashville, USA. LNCS 4735. Springer. (2007)

24. Vanhooff, Bert; Dhouha, Ayed et al. UniTI: A Unified Transformation Infrastructure. In Proceedings of MoDELS 2007 Conference, USA. Lecture Notes in Computer Science, number 4735. Springer. (2007)

25. Wagelaar, Dennis. Composition Techniques for Rule-based Model Transformation Languages. Procs. of ICMT2008 – Int. Conference on Model Transformation. Zurich, Switzerland. July 2008. (2008)

26. Ehrig, H. Engels, G. Kreowski, H.-J. and Rozenberg G. (eds.). Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools. World Scientific. (1999)

27. Greenyer, J. and Kindler, E. Reconciling TGGs with QVT. In Proceedings of MoDELS 2007 Conference. Nashville, USA. LNCS number 4735. Springer. (2007)

28. Back, R and von Wright, J. Refinement Calculus: A Systematic Introduction, Graduate texts in Computer Science, Springer Verlag.(1998)

29. Srinivas, Yellamraju V. and Jüllig, Richard. Specware: Formal support for composing software. In Proceedings of the Conference on Mathematics of Program Construction, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422. (1995)

30. Partsch, H. Specification and Transformation of Programs. Texts and Monographs in CS. Springer Verlag (1990)