An Algebraic Approach for Composing Model Transformations in QVT

Claudia Pons Roxana Giandini Gabriela Perez Gabriel Baum

LIFIA, Facultad de Informática, Universidad Nacional de La Plata
Buenos Aires, Argentina
[cpons, giandini, gperez, gbaum]@info.unlp.edu.ar

***Abstract**. The ability to orchestrate different transformations in a flexible and reliable manner is a major challenge in MDE. Most of the current work on model transformations seems essentially operational and executable in nature. Such incomplete view on the composition issue provides suitable answers to a wide range of practical needs. However it does not cover the entire problem's spectrum. In this article we describe how the algebraic theory of problems formalism is applied as a basis to build a mathematical foundation for the transformation composition problem embracing both dimensions (descriptive and operational). We report the implementation of a software tool supporting this algebraic formalization.*

1. Introduction

Model Driven Architecture (MDA) [1] [2] [3] and Model Driven Engineering (MDE) [4] propose a software development process in which the key notions are models and model transformations. In this process, software is build by constructing one or more models, and transforming these into other models. In turn these output models may be transformed into another set of models until finally the output consists of program code that can be executed. Ultimately, software is developed by triggering an intricate network of transformation executions.

Transformations can be specified or implemented using different tools and different languages and they can be manipulated as black-box entities. The ability to orchestrate different transformations in a flexible and reliable manner in order to produce the required output is a major challenge in MDE. There are various approaches for model transformation that offer forms of compositionality (see [5] for a survey), either based on internal or external composition of transformations For instance, the QVT standard [6] specifies a language in which one is able to express transformation definitions that consist of a number of mapping rules. The mapping rules may be combined by calling, or by using the reuse facilities: inheritance, merge and disjunction. Kleppe in [7] calls this the internal (or fine-grained) composition of transformations. On the other hand, the combination of transformations as black box entities is called the external (or coarse-grained) composition of transformations. 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.

Despite the fact that the QVT offers two modeling perspectives - it allows us to specify what the transformation does (declarative QVT) as well as how the transformation is accomplished (operational QVT) - most of the current work on model transformations seems essentially operational and executable in nature. Executable descriptions are necessary from the point of view of implementations; but from a conceptual point of view, transformations can also be viewed as descriptive models (see [8] for a deep analysis concerning this assertion) by stating only the properties a transformation has to fulfill and by omitting execution details. In particular, regarding the composition problem, most approaches are focused only on the operational aspects of the composition, neglecting its descriptive side. Such partial visualization of the composition problem is useful to offer a reasonable solution to a wide range of practical needs. However it does not cover the entire composition problem spectrum.

The aim of this article is to describe how the algebraic theory of problems can be applied as a basis to build a mathematical foundation for the transformation composition problem embracing both dimensions (descriptive and operational). The paper is structured as follows. Section 2 explains the basic formalism of the algebraic theory of problems. Section 3, describes a mathematical foundation for the transformation composition problem based on the theory; in addition, the main features of a software tool supporting such algebraic composition of transformations are presented. Section 4 contains a discussion on related work; finally section 6 presents the conclusions.

2. The basic formalism

In this section we will summarize the main concepts comprising the algebraic theory of problems [9] [11], which is based on intuitive ideas developed by Pólya [10]. In the last years the algebraic theory of problems has been used as foundation for calculus for program derivation, such as Fork algebras [12].

2.1. The concepts of 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 **D**x**R** 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 what Pólya calls the condition of the problem. The fourth element of the quadruple will be treated later. 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. Graphically, a problem can be visualized by using Cartesian coordinates, where **D** is a subset of the abscissas and **R** is a subset of the ordinates.

We will say that a problem is *feasible* if and only if there is for each datum **d** of the set **I** of the interest instances at least an element **r** belonging to the results domain **R**, so that the pair **<d,r>** belongs to the condition **q**. That is to say, **q** must be defined for the whole set of the interest instances: *(feasibility condition)* $(\forall \mathbf{d})\,(\,\mathbf{d} \in \mathbf{I} \rightarrow (\exists r)\,(\,r \in \mathbf{R} \wedge \mathbf{q}(\mathbf{d}, r)\,)\,)$

A solution for the problem **P** must be a function of **D** in **R** which fulfills the condition **q** for the set of interest instances. But, also a solution should hold the property α, which is called the *admissibility context*. Such α property may be extensional, such as: continuous, derivable, increasing, calculable etc.; or intentional, such as: efficient, elegant, easy, etc. In particular, we are interested in solution being amenable to be calculated by an efficient computer algorithm (i.e. no more than polynomial complexity). Let us call α-*solution* to functions having such characteristics and let us call Ω_P to the set of all α-solutions of a problem **P**.

2.2. Operations on problems and operations on solutions

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. From the problem theory point of view, breaking means to state the given problem in terms of operations on composing problems, while the recombination is done by means of the corresponding operations on solutions. The operations on problems are those of the Fork Algebra [12] which is an algebra obtained by expanding Relation Algebra with a new operator called ∇ (*fork)*. The Relation Algebra [13] 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,$\neg$, $\fatsemi$,$\smile$,1), such that (P(V), $\cup$, $\cap$, $\varnothing$, V, $\neg$) is a Boolean algebra and (P(V), $\fatsemi$,1) is a monoid. The definitions of the algebra's operators are extended to problems and solutions in a quite straightforward way. In the next sections we will describe one of them in detail: the union operator.

Union of problems

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. That is to say:

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

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 set: 0=<$\varnothing$,$\varnothing$,$\varnothing$,$\varnothing$>.

α-solutions for the union of problems

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∪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? 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 an election 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
$(\delta_P \underline{\cup} \delta_Q) = \lambda d.$ `if` $\delta_P(d) = \perp$ `then` $\delta_Q(d)$ `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∪Q.

2.3. Declarative languages vs. imperative languages

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.

Declarative Languages for the description of problems

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.

It is desirable that the operations on problems of the algebraic theory of problems could be expressed in $\mathcal{L}_{\mathcal{D}}$. Let us suppose that $\mathcal{L}_{\mathcal{D}}$ is a specification language, e.g. a first order logic language. In logic, we have a way of combining statements to obtain the effect of the union of problems: the disjunction connective ∨.

Imperative languages for the description of solutions

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**] computed by the program whose restriction is defined by the precondition expressed in the **Impl** text.

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 algorithm which calculates an α-function.

We should also consider the lexicon of language $\mathcal{L}_{\mathcal{A}}$. $\mathcal{L}$et us suppose that $\mathcal{L}_{\mathcal{A}}$ is a usual programming language, such as Java. It is desirable that the operations on α-solutions 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 $\underset{\sim}{\circ}$ between α-solutions: the construction of the command composition ";".

3. The algebraic theory of problems as a foundation for model transformation languages

The QVT Language [6] 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 standard QVT language and the algebraic theory of problems.

3.1. Declarative QVT vs. imperative QVT

The declarative parts of QVT (named ***Relations Language***) allows for to creation of declarative specification of the relationships between MOF models. It supports complex object pattern matching. In the Declarative QVT a transformation defines how one set of models can be transformed into another. 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. The imperative expressions in QVT realize a compromise between some functional features found in OCL and the more traditional constructs that we found in general purpose languages like Java. 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.

Thus, when we refer to 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 defines the semantics of declarative expressions in terms of *problems*, while function υ gives the semantics of imperative constructions in terms of *solutions* (see a technical report at [24] containing the formal definition of both semantics functions). By having the semantics of the QVT language expressed in terms of the algebraic theory of problems we are able to formally verify whether a QVT implementation is correct with respect to its QVT specification or not. Such correctness condition is illustrated by the horizontal arrows in figure 1 and it is defined as follows,

Definition 3. (QVT correctness condition)
Let **Spec** be a transformation specification written in Declarative QVT, and let **Imp** be an implementation written in Operational QVT. **Imp** is a correct implementation for **Spec** if and only if the function $\delta_P=\upsilon$**[Impl]** is an α-solution for the problem **P=**μ**[Spec]**.

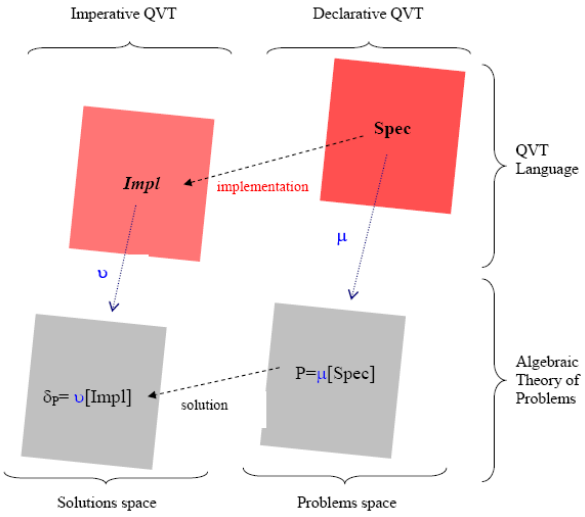

Figure 1: QVT language semantics in terms of the ATP.

Going one step forward, this formal definition of the QVT language in terms of the algebraic theory of problem provides a sound foundation to create an algebraic structure

supporting model transformation compositions. In this way, the QVT language will have the ability to express the problems to solve as well as their own decompositions as defined by the algebraic theory of problem. That is to say, the declarative QVT language would provide linguistic constructs to interpret the operations on problems (i.e.,$\cup$, $\fatsemi$, etc), while operational QVT language would provide the corresponding constructs regarding the operations on solutions (i.e., $\underline{\cup}$, $\underline{\fatsemi}$, etc.). For instance, we need an operation $\cup_{\mathbf{QVT}}$ to carry out the union of two declarative transformations, while on the other hand we need an operation $\underline{\cup}_{\mathbf{QVT}}$ to perform the union of two operational transformations. Table 1 shows the list of basic operations and their corresponding (expected) materializations into the QVT languages, which are described in the following sections.

Table 1. Materialization of the algebra of problems and the algebra of solutions into QVT

Operation	Problems algebra	Declarative QVT	Solutions algebra	Operational QVT
union	$\cup$	$\cup_{\mathbf{QVT}}$	$\underline{\cup}$	$\underline{\cup}_{\mathbf{QVT}}$
sequence	$\fatsemi$	$\fatsemi_{\mathbf{QVT}}$	$\underline{\fatsemi}$	$\underline{\fatsemi}_{\mathbf{QVT}}$
fork	∇	$\nabla_{\mathbf{QVT}}$	$\underline{\nabla}$	$\underline{\nabla}_{\mathbf{QVT}}$
intersection	$\cap$	$\cap_{\mathbf{QVT}}$	$\underline{\cap}$	$\underline{\cap}_{\mathbf{QVT}}$
converse	$\hookleftarrow$	$\hookleftarrow_{\mathbf{QVT}}$	$\underline{\hookleftarrow}$	$\underline{\hookleftarrow}_{\mathbf{QVT}}$
empty	$\varnothing$	$\varnothing_{\mathbf{QVT}}$	$\underline{\varnothing}$	$\underline{\varnothing}_{\mathbf{QVT}}$
identity	1	$1_{\mathbf{QVT}}$	$\underline{1}$	$\underline{1}_{\mathbf{QVT}}$
universal	V	$\underline{V}_{\mathbf{QVT}}$	$\underline{V}$	$\underline{V}_{\mathbf{QVT}}$

3.2. Expressing composition of problems in QVT

Almost no support is provided to combine declarations of transformations written in the QVT Declarative Language. To fill up this gap, we have formally specified an interpretation into QVT for each one of the operations on problems displayed in table 1. As illustration, in the next section we show the definition of the union of problems (i.e., $\cup_{\mathbf{QVT}}$).

The union of declarative transformations

For a better understanding of the union of transformation, we introduce the specification of two transformations T1 and T2, and the transformation T1$\cup_{\mathbf{QVT}}$T2 resulting from the application of the operation $\cup_{\mathbf{QVT}}$ on them. The following examples of transformations are based on Lano's Catalogue [15].

```
Transformation T1 (Uml:UML2.0,Java:JAVA)
{TopLevel Relation
PersistClass2ClassWithKey {
  checkonly domain Uml c: Class
  enforced domain Java jc: JavaClass
  when {c.isPersistent}
  where {c.name =jc.name and
   jc.ownedFields-> exists (jf: JavaField | jf.name = "primaryKey"
   and jf.type = "Integer") and jc. ownedMethods -> exists (jm|
   jm.name = "getPrimaryKey" and jm.returnType = "Integer")  }
}}
```

```
Transformation T2 (Uml:UML2.0,Java:JAVA)
{TopLevel Relation Class2Class {
  checkonly domain Uml c: Class
  enforced domain Java jc: JavaClass
  when { not c.isPersistent }
  where { c.name = jc.name and
   c. ownedAttribute -> forAll (p : Property |   ( jc.ownedFields->
   exists(jf:JavaField | p.name = jf.name) and   jc.ownedMethods ->
   exists  (jm1,  jm2  :  JavaMethod  |  Attr2Getter  (p,  jm1)  and
   Attr2Setter (p, jm2))  }
   }
Relation Attr2Getter{…}
Relation Attr2Setter{...}
Query firstToUpperCase(string: String) : String {…}
}
```

Both transformations T1 and T2 are applied between a UML model and a JAVA model. T1 defines a unique *top level* relation, called *PersistClassWithKey*. The data domain of this relation is the set of UML classes. The result domain of this relation is the set of Java classes. The *when* clause of this relation determines its set of *interest instances*, which is restricted to all the persistent classes. When we apply the transformation, for each UML class, a Java class with the same name will be generated. A new Integer identity attribute called *primaryKey* is introduced to the Java class. Also, a new operation called *getPrimaryKey* is added to allow us to retrieve this attribute.

Transformation T2 defines a unique *topLevel relation, Class2Class*, whose data domain is also the set of UML classes and whose set of interest instances is restricted to all the non persistent classes. The result domain is integrated by Java classes.

Class2Class invokes two other relations, *Attr2Getter* and *Attr2Setter*. When the relation *Class2Class* is applied, per each attribute defined in the original class, a getter method and a setter one are added in the resulting Java model. Besides these relations, the transformation includes a *query* which converts a given string in another one where the first character appears in capital letter.

The intuitive 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. Thus, *Attr2Getter* and *Attr2Setter,* which are non-*top-level,* will be part of the result. The same happens to the *query* defined in the second transformation. On the other hand, the union operation is applied between the *top level* relation *PersistClass2ClassWithKey* and *Class2Class*, in order to combine them.

The union of relations is defined as follows,

- the data domain of the result relation is the union of the data domains of both factors;
- the result domain of the result relation is the union of the result domains of both factors;
- 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 *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).

The result from applying the operation union on T1 and T2 is as follows:

```
Transformation T1∪QVTT2 (Uml:UML2.0, Java:JAVA)
```

```
{
TopLevel Relation PersistClass2ClassWithKey∪QVTClass2Class
{
    checkonly domain Uml c: Class
    enforced domain Java jc: JavaClass
    when { not c.isPersistent or c.isPersistent }
    where
    { (not c.isPersistent AND c.name = jc.name and
    c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields->
    exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods ->
    exists (jm1, jm2 : JavaMethod | Attr2Getter (p, jm1) and
    Attr2Setter (p, jm2))  )
    OR
    ( c.isPersistent AND c.name =jc.name and jc.ownedFields-> exists
    (jf: JavaField | jf.name = "primaryKey" and jf.type = "Integer")
    and jc. ownedMethods -> exists (jm: JavaMethod | jm.name =
    "getPrimaryKey" and jm.returnType = "Integer")}
}
Relation Attr2Getter{ … }
Relation Attr2Setter{ … }
Query firstToUpperCase(string: String):String {…}
}
```

Let us point out that the union of declarative transformations might introduce non-determinism in the resulting transformation. This situation takes place when the interest instances of T1 and T2 are not disjunctive (which is not the case of the previous example).

Formalizing the union of declarative transformations

In this section we illustrate the formal definition of the operations for performing the union of transformations. The definition is given by *pre* and *post* conditions expressed in OCL, in the context of the *Transformation* and *Relation* metaclasses of the QVT metamodel, as follows (the complete definition can be read in [24]),

Definition 4. *(Union of declarative transformations)*

```
Context Transformation:: ∪QVT(t2: Transformation): Transformation
Post:
 --The name of the result transformation is the concatenation of
 --the names of both factors with the word '∪QVT' intercalated.
 result.name = self.name.concat ('∪QVT').concat (t2.name)
 --The set of typed models which specifies the types of models
 -- that may participate in the transf. is the union of both sets.
 result.modelParameter=
 self.modelParameter ->union(t2.modelParameter)
 --The set of non-top-level relations of the result is the union
 -- of the set of non-top-level relations of each factor.
 result.rule->select(not isTopLevel)=
     (self.rule ->union (t2.rule)) ->select (not isTopLevel)
  --the union is applied on top level relations (self.rule ->union
 (t2.rule))->select(isTopLevel) ->forAll(r1,r2|
 result.rule->select(isTopLevel)->includes(r1 ∪QVT r2)  )

Context Relation:: ∪QVT (r2: Relation) : Relation
Post: ….
```

3.3. Expressing composition of solutions in QVT

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. In the operational QVT 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.

However, those mechanisms do not provide a clean black box instrument to perform the composition of transformations due to the fact that we still need to write the code for the compound transformation instead of just applying some composition operation. Let us see the details in the following sections.

The union of operational transformations

Let us suppose we have two operational transformation $Impl_{T1}$ and $Impl_{T2}$ implementing declarative transformations T1 and T2 respectively, where T1 and T2 are the transformations specified in the previous section. We would like to be able to calculate an implementation **Impl** for the composite transformation $T1 \cup_{QVT} T2$ in terms of $Impl_{T1}$ and $Impl_{T2}$. That is to say, we need linguistics constructs to combine $Impl_{T1}$ and $Impl_{T2}$ complying with the semantics of the union of solutions established by Lemma 1.

Such combination can be 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. It can be expressed in the following way:

```
transformation Impl(in uml:UML2.0,out java:JAVA)
access Impl_T1(), Impl_T2();
main()
{ var returncode := (new Impl_T1(uml,java)).transform();
  if (returncode.failed())then
        (new Impl_T2(uml,java)).transform()endif }
```

Instead of using the `if-then` construct, it would be desirable to rely on a higher level mechanism to perform the union of operational transformations. Let us call $\cup_{QVT}$ to such operator, that is to say: $Impl = Impl_{T1} \cup_{QVT} Impl_{T2}$

Formalizing the union of operational transformations

The formal definition of the operation $\cup_{QVT}$ is given by *pre* and *post* conditions expressed in OCL, in the context of the *OperationalTransformation* metaclass of the QVT metamodel, as follows (only a part is displayed due to space limitations),

Definition 5. *(Union of imperative transformations)*

```
Context OperationalTransformation:: ∪QVT
  (t2:OperationalTransformation):OperationalTransformation
Post:
--The name of the result is the concatenation of the
-- names of both factors with the word '∪QVT' intercalated
      result.name = self.name.concat ('∪QVT').concat (t2.name)
--The result is black box iff any factor is black box.
      result.isBlackbox = self.isBlackbox or t2.isBlackbox
--The result is abstract iff any factor is abstract.
      result.isAbstract = self.isAbstract or t2.isAbstract
--the parameters results from the union of the parameters of
-- each factor.
result.modelParameter=self.modelParameter->union(t2.modelParameter)
--The relational transformation being refined by the result
--operational transformation is the union of the refined
--transformations of each factor.
result.refined = self.refined ∪QVT t2.refined
--The body of the result consists of an if-then expression
--that carries out the coarse-grained composition of both
--transformations …
```

3.4. α-solutions for the composition of transformations

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 "$\underline{\cup}_{QVT}$", we are able to prove the following lemma, which is illustrated in the top level of figure 2.

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

Proof:
from the hypothesis that δ_{T1} is an implementation for T_1 and δ_{T2} is an implementation for T_2 and by using the formal semantics of the QVT language, we build the following chain of deductions (which are illustrated in figure 2):

[1]. $\upsilon(\delta_{T1})$ is a solution for $\mu(T1)$ (by definition 3, given that δ_{T1} is an impl. for T1)
[2]. $\upsilon(\delta_{T2})$ is a solution for $\mu(T2)$ (by definition 3, given that δ_{T2} is an impl. for T2)
[3]. $\upsilon(\delta_{T1}) \underline{\cup} \upsilon(\delta_{T2})$ is a solution for $\mu(T1) \cup \mu(T2)$ (by lemma 1)
[4]. $\upsilon(\delta_{T1}) \underline{\cup} \upsilon(\delta_{T2}) = \upsilon(\delta_{T1} \underline{\cup}_{QVT} \delta_{T2})$ (by correctness of function υ)
[5]. $\mu(T1) \cup \mu(T2) = \mu(T1 \cup_{QVT} T2)$ (by correctness of function μ)
[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 an implementation for $T_1 \cup_{QVT} T_2$ (by definition 3).

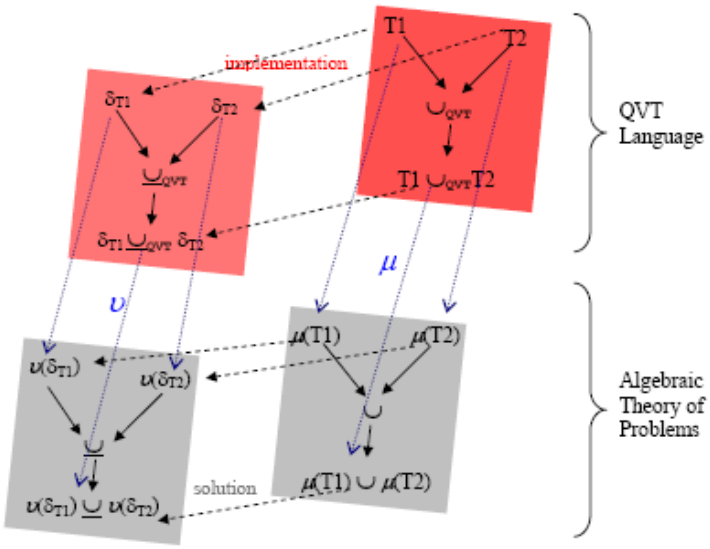

Figure 2. Union of problems and union of solutions in QVT.

Similar results can be proven for the QVT materialization of each operation of the algebraic theory of problems.

3.5. The composition calculator

To assist the transformation development activities we are developing a software tool that allows developers to edit and store atomic QVT transformations. Then, they can build more complex transformations by applying the composition operations presented in this paper. The resulting transformations are automatically 'calculated' by the tool. The tool was built as an extension of ePlatero[25] that is an open source plug-in for Eclipse running on top of the EMF metamodelling framework.

Figure 3 shows the most relevant screenshots. The first screen exhibits the selection, from a transformation repository, of the transformations to be composed; the second and third screens show the application of an algebraic operation on the selected transformations. On the bottom panel we can see a preview of the composition result. Left screen displays the composition on the declarative level while right screen displays the same composition but on the operational level.

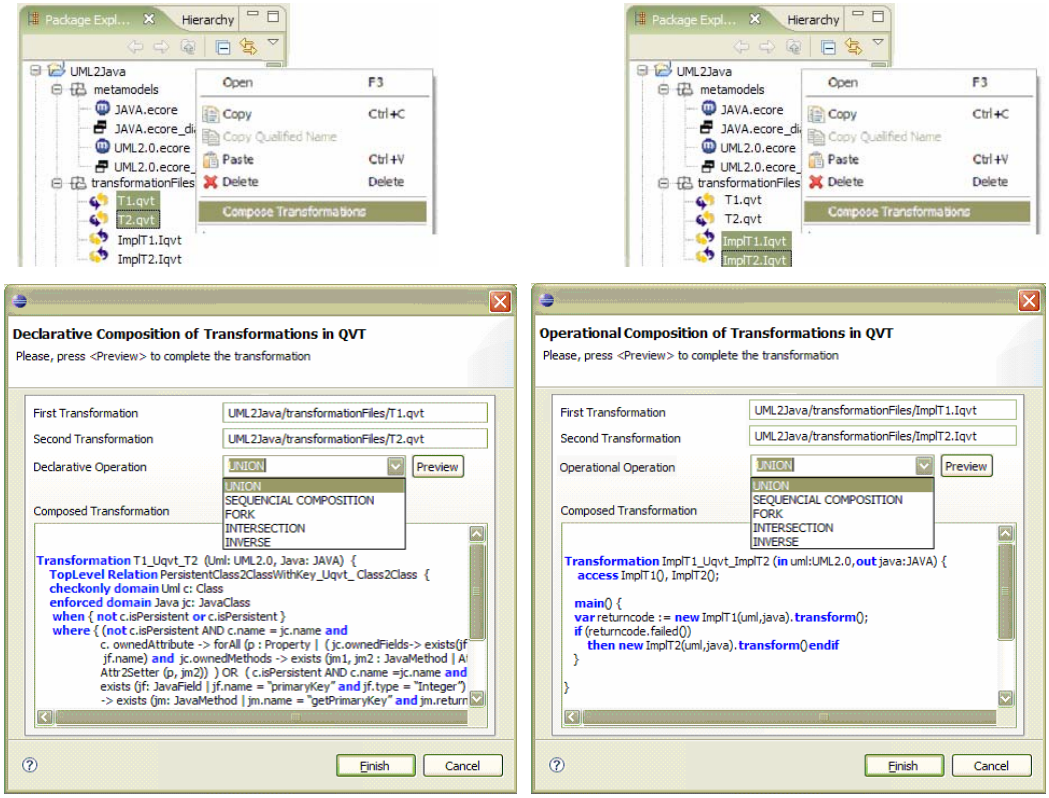

Figure 3. Composition calculator in Eclipse

4. Related work

Anneke Kleppe in [7] 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 described in [7] are the Model Bus approach [16] which tackles composition in the manner of OMG's CORBA and the UMLAUT transformation toolkit [17] 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.

On the other hand, Bézivin and colleagues in [18] analyze three model composition frameworks: the Atlas Model Weaver [19], the Glue Generator Tool (GGT)[20] and the Epsilon Merging Language (EML) [21]. From them, they derive a core set of common definitions regarding model composition, and define a set of requirements for model composition frameworks, in terms of language and tool support. 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.

All these approaches 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 transformation composition problem embracing declarative as well as operational dimensions.

5. Conclusion

In this article we described how the algebraic theory of problems is applied as a basis to build a mathematical foundation for the transformation composition problem in QVT, embracing both levels (descriptive and operational). Having the composition machinery accurately synchronized at both QVT levels would allow us to fully exploit the divide-and-conquer paradigm in the development of 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.

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 (apparently) corresponding operations on the other level is unclear and ambiguous.

To overcome this drawback, we formally specified a QVT interpretation for each one of the operations on problems and the operations on solutions as defined by the algebraic theory of problem. This mathematical characterization of the transformation composition problem provides a clear picture of the composition machinery at both QVT levels. It can be used as a foundation to improve the QVT specification towards the construction of simpler and more powerful transformation composition machinery, which will serve as a solid ground for the construction of model transformation languages and tools.

Regarding the expressiveness of this algebra of transformations, it was proved in [12] 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 [12]. 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 algebra 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 algebra of transformations.

References

[1] Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publ. Co., Inc., Boston, USA, 2003.

[2] Mellor, Stephen J. and Scott, Kendall and Uhl, Axel and Weise Dirk. MDA Distilled, Principles of Model_Driven Architecture. Addison-Wesley, 2004.

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

[4] Favre Jean-Marie, Estublier Jacky, Blay Mireille. Beyond MDA : Model Driven Engineering (L'Ingénierie Dirigée par les Modèles : au-délà du MDA) Edition Hezmes-Lavoisier..(2006).

[5] Czarnecki K. and Helsen S. Feature-based survey of model transformation approaches. IBM System Journal, Vol. 45, No 3, 2006.

[6] MOF QVT final adopted specification. Technical Report ptc/05-11-01, OMG, 2005.

[7] Kleppe, Anneke. MCC: A Model Transformation Environment. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 173 – 187, Spain, June 2006.

[8] Bézivin, J., Büttner, F., Gogolla, M., Jouault,F., Kurtev, I., Lindow,A. Model Transformations? Transformation Models!. MODELS 2006 Int. Conf proceedings. LNCS vol. 4199 (2006)

[9] Haeberer, A.M. and Baum, G. and Veloso, P.A.S. On an Algebraic Theory of Problems and Software Develop.. Pont. Universidad.Catolica Research Rep. MCC 2/87 R. de Janeiro (1887).

[10] Pólya, G., How to Solve it: a new aspect of the mathematical method, Princeton University Press, Princeton, 1945 (2nd. ed..1956, repr. 1971)

[11] Veloso, P.A.S. Outline of a mathematical theory of general problems. Philosophia Naturalis; vol. 2/4 No. 1,. (1984)

[12] Frias, M. and Veloso, P.A.S and Baum, G. Fork Algebras: past, present and future. Journal on Relational Methods in Computer Science. Vol.1, 2004, pp.181-216.

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

[14] Belaunde, Mariano. Transformation composition in QVT. First European Workshop on Composition of Model Transformations at ECMDA 2006.

[15] Lano K., Catalogue of Model Transformation. 2006, http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf

[16] Blanc,X., Gervais, M., Lamari, M. and Sriplakich, P.. Towards an integrated transformation environment (ITE) for MDD. In Proceedings of the 8th SCI'2004, USA, July 2004.

[17] Jézéquel, Jean-Marc and Ho, Wai-Ming and Le Guennec, Alain and Pennaneach, Franccois. UMLAUT: an extendible UML transformation framework. In Proc. of the 14th IEEE International Conference on Automated Software Engineering, 1999.

[18] Bézivin, J. Bouzitouna, S. Del Fabro, M. D. Gervais, M. P. Jouault1, F. Kolovos, D. Kurtev I. et Paige R. F. "A Canonical Scheme for Model Composition". Proc. of the European Conference on MDA. LNCS, Springer-Verlag, June 2006.

[19] Atlas Model Weaver Project Web Page. http://www.eclipse.org/gmt/amw/, 2005.

[20] Bouzitouna, M. P. Gervais and X. Blanc, Model Reuse in MDA, Proceedings of the International Conference on Software Engineering Research and Practice (SERP'05), USA, June 2005.

[21] Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. The Epsilon Object Language (EOL). In Proc. Model Driven Architecture Foundations and Applications: Second European Conference, ECMDA-FA, volume 4066 of LNCS, June 2006.

[22] Oldevik, J. Transformation Composition Modeling Framework. DAIS 2005. Lecture Notes in Computer Science 3543, pp. 108-114. (2005).

[23] Pons, C. and Garcia, D. An OCL-based Technique for Specifying and Verifying Refinement Transformations in MDE. In 9th MoDELS International Conference. LNCS 4199. (2006).

[24] Giandini, R., Pons, C. and Perez, G. The QVT Semantics in terms of Problems and Solutions. Technical Report. http://sol.info.unlp.edu.ar/eclipse/QVTsemantics.pdf. Uploaded: July 2007.

[25] ePlatero Home page. http://sol.info.unlp.edu.ar/eclipse.