A two-level formal semantics for the QVT language

Roxana Giandini, Claudia Pons, Gabriela Pérez

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

Abstract. Model Driven Engineering (MDE) proposes a software development process in which software is built 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. Model transformation is the MDE engine. The QVT Language is a concrete linguistic object for expressing model transformations with a hybrid declarative/imperative nature. The semantics of the QVT is informally defined in natural language or by its translation to a formal language, but these approaches are only focused on one dialect of QVT: Relations Language or Operational Mappings Language. They do not cover the hybrid nature of QVT. In this paper we provide a formal semantics for the entire QVT language, embracing and synchronizing its two dimensions: declarative and operational.

Keywords: Model Driven Engineering, Model Transformation, QVT Semantics

1. Introduction

Model Driven Architecture (MDA) [1] [2] and Model Driven Engineering (MDE) [3] [4] propose a software development process in which the key notions are models and model transformations. In this process, software is built 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.

Model transformation is the MDE engine, playing a relevant role; models are no longer mere contemplative entities and become productive entities. The MDE initiative covers a broad spectrum of research areas: modeling languages, definition of languages for model transformation, construction of support tools, etc. Currently, some of these aspects are well-based, and are being applied with some success; however, other aspects are still undergoing their definition process. In this context, it is necessary to make every effort to convert MDE and its concepts and related techniques into a coherent proposal, based on open standards, and supported by mature tools and techniques. The definition of model transformations requires the application of specific languages; these languages should have a formal base, and at least a metamodel describing their abstract syntax and they should be amenable to be automatically treated.

In MDA, model transformations have become a relevant issue by means of the forthcoming standard Query/Views/Transformations (QVT) [5]. The QVT Language

is a concrete linguistic object for expressing model transformations with a hybrid declarative/imperative nature. QVT includes an abstract syntax that is rigorously defined trough a metamodel, but its semantics, like the semantics of others languages inspired by QVT, such as: ATL[6], Kent[7] and TefKat [8], is only informally expressed by using natural language (see Section 7.10 and Annex B in [5]).

The aim of this article is to formally define a semantics for QVT, - considering both declarative level and imperative (or operational) level - applying the intuitive Theory of Problems [9] [10]. By having the semantics of this language defined in a precise form, we will be able to prove the fulfillment of certain properties and correctness conditions between its declarative and operational levels.

The organization of this paper is as follows: the theory of problems formalism is presented in Section 2. Section 3 shows the relationship between computational languages and the aforementioned formalism; in particular the connection between QVT and the theory of problems is analyzed. Section 4 presents a precise definition of the QVT semantics and enunciates its correctness condition. Finally, conclusions and related works are presented.

2. The basic formalism: the concept of problem and the concept of solution

In this section we will summarize the main concepts comprising the intuitive theory of problems. In the last years the algebraic theory of problems has been used as foundation for calculus for program derivation, such as Fork algebras [11].

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. 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 D will consist of UML classes and associations among others; 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.

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*) $(\forall d)\,(\, d \in I \rightarrow (\exists r)\,(\, r \in R \wedge q(d, r)\,)\,)$

Now, what does a solution for the problem P mean? The existential quantification in the feasibility condition means that for a given datum there is a result related with it

because of the condition¸ or what is the same, for a given datum d there are pairs <d,r> in the condition of the problem, and we must choose which one we are interested in. It seems impossible to generalize the choice of an arc of the condition containing a given datum in an effective method (in a reasonable sense) for obtaining problem solutions. For example, consider the derivation of Java code from UML class diagrams, a typical algorithm of the British Museum type, such as taking elements from the set of Java programs and verifying which ones implement the UML classes, does not seem the most satisfactory method to solve the problem of code derivation. Now then, having an effective procedure for building the result is very different from having to choose points in a relation. In our example of code derivation, using the British Museum algorithm is very different from knowing an algorithm to calculate the code from UML classes. So, a solution 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. In particular, we are interested in solutions 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.

3. The QVT language and the theory of problems

Before presenting the relationship between QVT and the theory of problems, in this section we analyze, from a more general point of view, the relationship between computational languages and this theory, as follows:
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 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 $\mathbf{P} = \mu[\mathbf{Spec}]$ 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.

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**] 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.

Due to the hybrid nature of QVT that embraces two kinds of languages, to apply a two-level formalism for defining its semantics seems to be the right choice. There is a close connection between the central concepts of the formalism (problem and solution) and the two levels of QVT (declarative and imperative).

3.1 Declarative QVT vs. imperative QVT

The QVT Language [5] 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.

The declarative parts of QVT (named *Relations Language*) allows for the creation of declarative specification of the relationships between MOF [12] 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 [13] 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 an 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. Notice that the operational mappings language is executable [18]. Figure 1 shows the connection between the two QVT linguistic levels and the 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. In the following sections, we present the formal definition of both semantic functions.

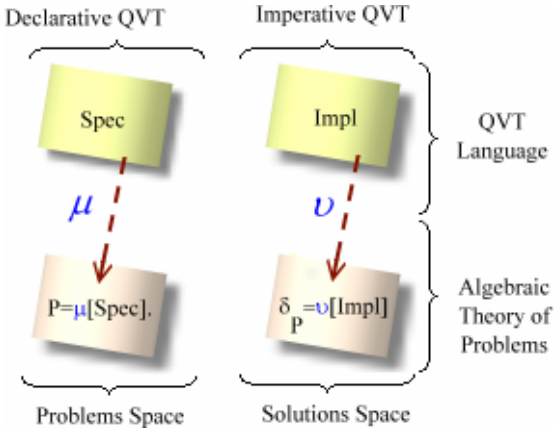

Figure 1. QVT language semantics in terms of the algebraic theory of problems.

4. A two-level semantics for QVT

In this section we will define the semantic functions on the QVT language. For the definition of these functions we choose to use the OCL specification language instead of some another first order-logic language, for being a standard language widely known by the modeling community; it is also sufficiently expressive and friendly to use.

4.1 Semantics for Relational QVT

The abstract syntax of the declarative level of QVT is specified by means of the Relational Transformation metamodel, that is shown (lightly simplified) in Figure 2. As we already said, a relational transformation T basically consists of relations, therefore the semantics of T will be defined in terms of the semantics of its component relations. Let's analyze the semantic function μ applied on a transformation T, written in Relational QVT. This function interprets the relational language in the problems space of the Theory of Problems, as follows:

μ: RelationalTransformation -> PROBLEM

Before defining the semantics of the whole transformation, let's see how the semantics of the Relations conforming the transformation are defined and then how to define the semantics for the whole Transformation.

μ: Relation -> PROBLEM

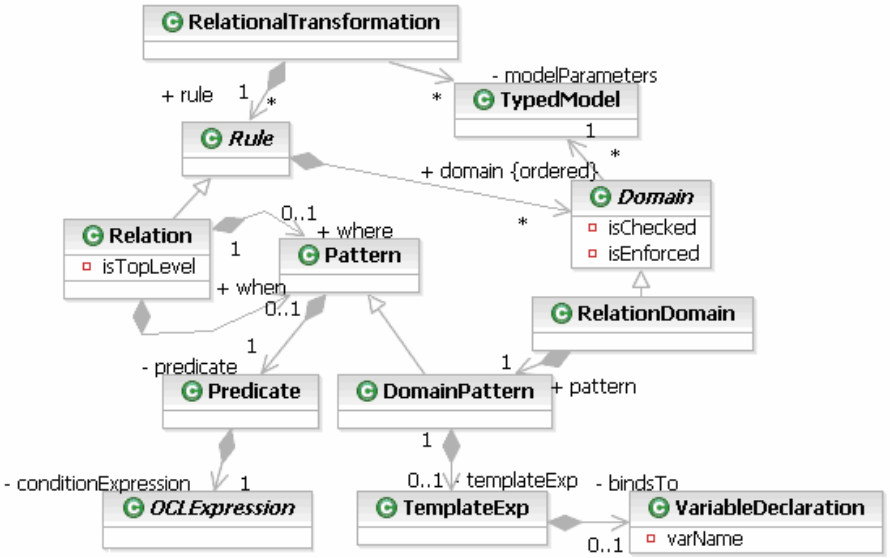

Figure 2. Relational Transformation metamodel.

The semantics domain, named ***PROBLEM,*** is a tuple <D, R, q, I> and is specified in OCL in the following way:
PROBLEM = TupleType (dom: Set(NamedElement), res: Set(NamedElement), q: Set (TupleType (d: NamedElement, cod: NamedElement)), i: Set(NamedElement))

Let r be a Relation:
μ (r).dom = r.dom().pattern.templateExp.bindTo.type.allInstances
The data domain of the relation is the set of all instances of the variable type defined in the domain[1].
μ (r).res = r.coDom().pattern.templateExp.bindTo.type.allInstances
The result domain of the relation is the set of all instances of the variable type defined in the coDomain,
where dom() and coDom() are query operations that respectively result in domain and codomain of a Relation. They are defined as follows:
Relation:: dom(): RelationDomain,
dom = self.domain->first()
Relation:: coDom(): RelationDomain,
coDom = self.domain->last()
μ (r).q = μ (r).dom -> product (μ (r).res) -> select (Tuple { d , cod } | r.where.predicate.conditionExpression [r.

[1] We think, for simplicity, that domain and codomain of every relation are composed by only one variable declaration, that is to say: r.dom ().pattern.templateExp.bindTo-> size () = 1 and r.coDom (). pattern.templateExp.bindTo-> size () = 1

dom().pattern.templateExp.bindTo.varName /d.name]
[r.coDom().pattern.templateExp.bindTo.varName/cod.name])
The relation condition q is the set of pairs of instances of domain and codomain respectively, that satisfy the where clause of the relation.
μ (r).i =μ (r).dom -> select (d | r.when.predicate.conditionExpression
[r.dom().pattern.templateExp.bindTo.varName / d.name])
The set of interest instances of the Relation is the set of instances of the variable defined in the domain that satisfy the when clause of the relation. Like in the case of q, the corresponding replacement of variable names was applied.

After having defined the semantics for Relations, we can define the semantics for RelationalTransformation: every component of the transformation semantics is the union of the corresponding components of every *topLevel* Relation in the Transformation, for example, the data domain of the transformation is the union of the data domain of all the top level relations that make up the transformation.

Let t be a RelationalTransformation:
μ (t).dom = t.rule -> select (r : Relation | r.isTopLevel) -> collect (r: Relation | μ(r).dom) -> flatten( ) -> asSet
μ (t).res = t.rule -> select (r : Relation | r.isTopLevel)-> collect (r: Relation | μ(r).res) ->
flatten( ) -> asSet
μ (t).q = t. rule -> select (r : Relation | r.isTopLevel)-> collect (r: Relation | μ(r).q) -> flatten( ) -> asSet
μ (t).i = t.rule -> select (r : Relation | r.isTopLevel) -> collect (r: Relation | μ(r).i) -> flatten( ) -> asSet

4.2 Semantics for Operational QVT

The operational level of QVT has an abstract syntax that is specified by means of metamodels. Figures 3, 4 and 5 show the principal metaclasses: OperationalTransformation, ImperativeOperation[2] and ImperativeExpression. In Operational QVT, the imperative expressions constitute an extension of OCL since they are subclasses of OCLExpression, but they can produce changes in the environment and in the store. Therefore, the Operational QVT semantics, which includes the imperative expressions semantics, was expressed following the notation used in the definition of the OCL semantics (i.e., semantic function I defined in Annex A of [13]) which is based on the Ph.D thesis of Richters [15].

The semantics is defined for no-blackbox transformations, which start their execution by invoking an entryOperation. This operation rises the execution of the remaining imperative expressions. The EntryOperation is special with regard to the rest of the imperative operations. It has no parameters and in its body it has invocations that provoke the execution of other imperative operation or the execution of another transformation. That is to say, the rest of the operations will not be executed in isolation. In the following paragraphs, we define the necessary semantic domains and functions to define the semantic function υ for operational transformations:

[2] The Imperative Expressions metamodel, with adaptations, can be found in [14], chapter 3.

Semantic Domains

Let´s consider the following domains:

Bvalue, primitive domain for basic values (for example: bool, nat)
Oids, primitive domain for identifiers of storable values
Vars, primitive domain for names of variables
Evalue = Bvalue + Oids, domain for expressible values (i.e. basic values and identifiers)

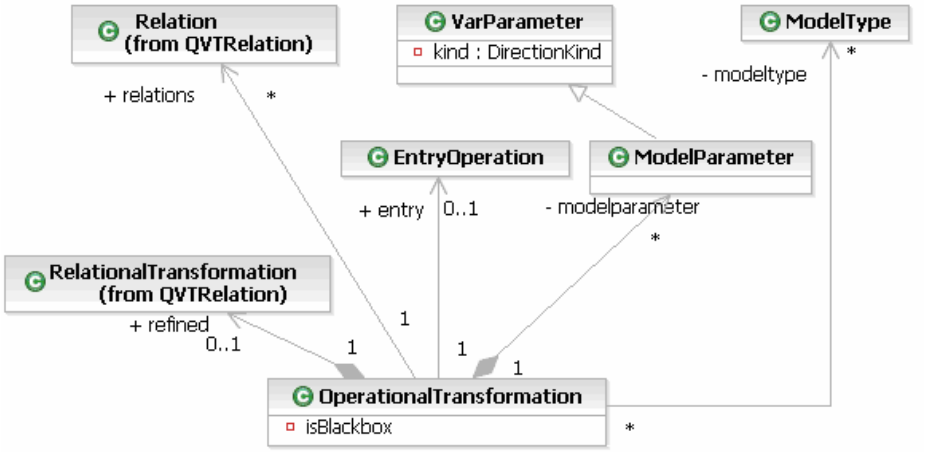

Figure 3. Operational Transformation metamodel.

Svalue, domain for storable values (objects)
Env = [Vars -> Evalue] domain for the environment of variables. An environment is a function that binds variable names with expressible values.
Store = [Oids -> Svalue] domain for the store of objects. A store is an injective function that binds object identifiers with objects.
Σ = (Env × Store) domain for states that represents the assignment of values to variables. Each state is represented by the environment of execution (***Env***) and the store (***Store***).
To evaluate free-side effects OCL expressions, we use the above mentioned standard semantics of OCL, named I, which is defined as follows:

I : OclExpression -> ***(Store × Env) -> Evalue***

That is to say, the function ***I*** considers the states represented by the pair Store × Env, in this order, while function ***υ***, that we define next, considers the states represented by the pair Env × Store.

The semantic function *υ*

The semantic function υ is applied to the language constructions (expressions) of model transformations. This semantics is defined as a function from an initial state to a final state. The function is defined on the metamodel hierarchic structure, as follows:

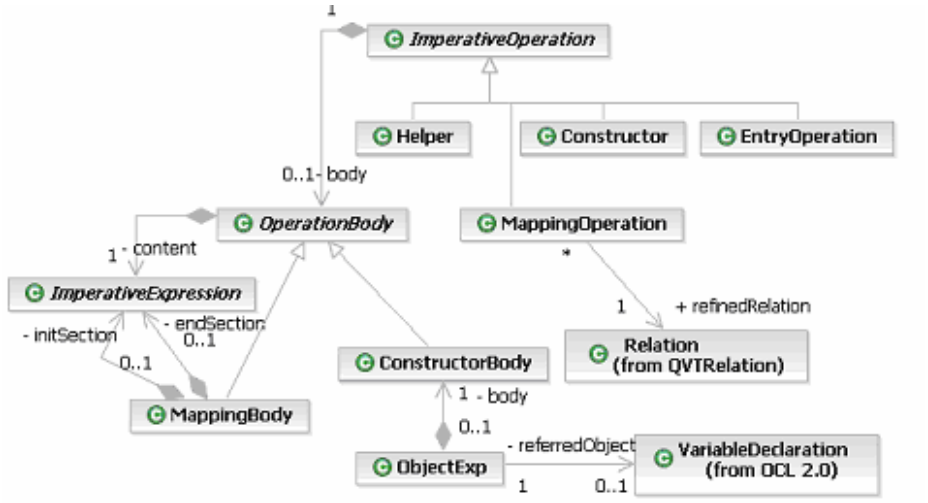

Figure 4. Imperative Operation metamodel.

υ : OperationalTransformation –> ***(Env × Store) -> (Env × Store)***
υ : ImperativeOperation –> ***(Env × Store) -> (Env × Store)***
υ : OperationBody –> ***(Env × Store) -> (Env × Store)***
υ : ImperativeExpression –> ***(Env × Store) -> (Env × Store)***

Let δ: ***Env***, σ: ***Store****:*
υ (ot: OperationalTransformation) (δ,σ) = υ (ot.entry) (δ,σ)
The semantics of an OperationalTransformation construction is the result of applying υ to the "*main*" operation (an EntryOperation) of the transformation, obtained by the *entry* association.
υ (entry: EntryOperation) (δ,σ) = υ (entry**.** body) (δ,σ)
The semantics of a EntryOperation is the result of applying υ to the operation body, obtained by the *body* association.
υ (body: OperationBody) (δ,σ) = υ (body.content) (δ,σ)
The semantics of an operation body is the result of applying υ to the operation body content, obtained by the *content* association. The body content corresponds to one instance of the ImperativeExpression hierarchy. Consequently, from this point of view, υ is the semantic function applied to the ImperativeExpression. In general, the execution of ImperativeExpressions produces changes in the environment (Env), not in the Store, except when objects are created, for example, by the *ObjectExp* expression.
Function υ is applied to ImperativeExpression as follows:
υ (e: NullExp) (δ, σ) = (δ, σ)
The semantics of the NullExp expression is a function that does not produce modifications nor in the Env, nor in the Store.
υ (e: SeqExp) (δ, σ) = υ (e.e2) (υ (e.e1) (δ, σ))

The semantics of the SeqExp expression is the result of firstly apply υ to the first component expression of *e* (*e.e1*), and soon, in that modified state, to apply υ to the second component of *e*, (*e.e2*).

υ (e:AssigExp) (δ,σ) = (δ [e.left / ***I*** (e.value) (σ, δ)], σ)

The semantics of the AssigExp expression is a function that only modifies the environment (Env), binding the left part of *e* with the result of evaluating by the function *I*, the *e.value* expression.

υ (e:IfExp) (δ, σ) = υ (e.thenExpression) (δ, σ), ***I*** (e.condition) (σ, δ) = tt
= υ (e.elseExpression) (δ, σ), ***I*** (e.condition) (σ, δ) = ff

The IfExp expression works like the IfExp of the original OCL. The difference is that in this case thenExpression and elseExpression belong to the ImperativeExpression hierarchy.

υ (e:WhileExp) (δ, σ) = (δ, σ) , ***I*** (e.condition) (σ, δ) = ff
= υ (e) (υ (e.body) (δ, σ)), ***I*** (e.condition) (σ, δ) = tt

The WhileExp expression represents the conditional iteration, with recursive definition. It depends of the *e.condition* evaluation, by the function ***I***.

υ (e:ImperativeCallExp) (δ, σ) =
υ (e.referredOperation.body) (δ [self / ***I*** (e.source) (σ, δ)] [p_i / a_i], σ)

The ImperativeCallExp expression (see figure 6) works like the OperationCallExp expression. The difference is that in this case the invoking operation belongs to the ImperativeExpression hierarchy. In the environment (Env), the pseudo-variable *self* binds to the result of evaluating the receive expression (e.*source*) and where:

[p_i / a_i] is the substitution function of every formal parameter named **p_i** for its corresponding real argument. The function ***I*** must be applied to every argument named **a_i,** to obtain its value.

The index **i** is such as:

i = 1.. (e.referredOperation.parameter-> size());

p_i =e.referredOperation.parameter -> at (i).name; a_i = ***I*** (e. arguments -> at(i)) (σ, δ)

Particularly, the invocation to a MappingOperation is defined by a MappingCallExp, subclass of ImperativeCallExp (see figure 6). Its structure differs from the rest of the imperative operations: it can have initSection, middleSection and endSection. On the other hand, TransformationCallExp is also a ImperativeCallExp subclass and is used to explicitly invoke another transformation from the body of an ImperativeOperation.

The class ImperativeCallExp is used for the invocation of the remaining ImperativeOperation.

υ (e:MappingCallExp) (δ, σ) = υ (e.referredOperation.body.endSection)
(υ (e.referredOperation.body.middleSection)
(υ (e.referredOperation.body.initSection) (δ [self / ***I*** (e.source) (σ, δ)] [p_i / a_i], σ)))

The semantics of the MappingCallExp expression is the result to incrementally apply υ to each body section (from the initial to the ended ones) of the referred operation.

υ (e:TransformationCallExp) (δ, σ) = υ (e.referredTransformation) (δ, σ)

The semantics of the TransformationCallExp expression is the result to apply υ to the referred transformation.

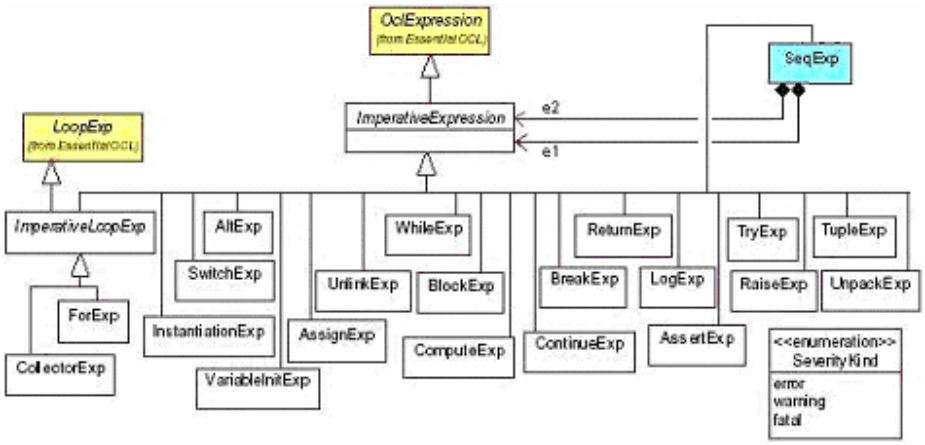

Figure 5. Imperative Expression metamodel.

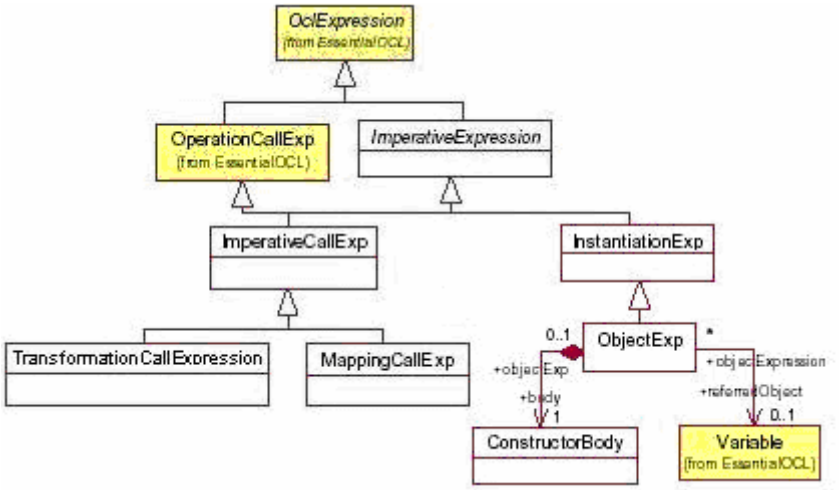

Figure 6. Imperative Expression for invocation and object instantiation.

υ (e:ObjectExp) (δ, σ) = (δ [e.referredObject.varName / id], σ [id / obj])
An ObjectExp expression (see figure 5) is used for instantiation or creation of new objects. ObjectExp has a body and references to a VariableDeclaration (*referredObject*), which has varName and type.

Here, the function next_id returns an identifier not used in the store, be id = next_id (σ). The identifier is bound to the referred variable by the expression, in the environment (δ). The new instance, named obj, binds to the new identifier in the store. Obj is an instance conforming the referred variable type, that is to say:
e.referredObject.type.allInstances -> includes (obj) and satisfices *e.body*, which conforms a ConstructorBody and defines the instance initialization.

4.3 QVT correctness in its two levels

By having the semantics of the QVT language -in its two levels- expressed in terms of the 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 7 and it is defined as follows,

Definition 1. (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]**.

5. Conclusions and related works

The QVT Language is a concrete linguistic object for expressing model transformations with a hybrid declarative/imperative nature. The QVT language, like other MDE concepts, lacks a formal semantics. This drawback hinders the rigorous and automatic treatment of the transformations expressed in such language.

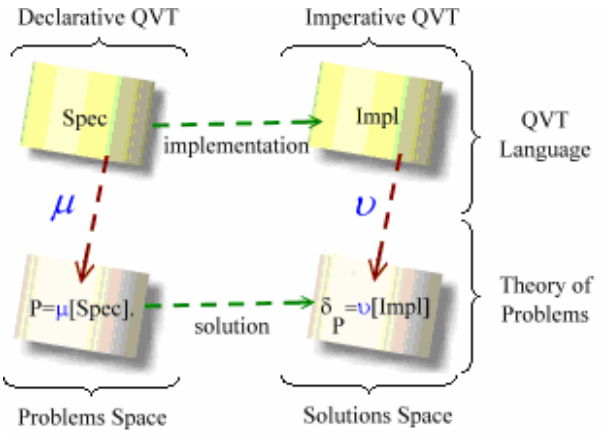

Figure 7. QVT correctness condition.

In this paper we have formally defined a semantics for QVT- considering the declarative level as well as the operational level - applying the intuitive Theory of Problems. As we have already mentioned, the hybrid nature of QVT has a close connection with the central concepts of this formalism, that is to say, we identify the declarative level with the concept of "problem" and the imperative level with the concept of "solution". For this reason the selection of the theory of problems as semantics domain seems to be a reasonably acceptable choice. Regarding the correctness of our semantics definition, we are only able to corroborate it intuitively. But we are not capable to perform a more deep analysis due to the absence of other formal semantics for QVT.

On top of this formalization we are able to enunciate correctness conditions between both levels of expression. This proposal constitutes a contribution to the maturity of the QVT language. By having the semantics of the QVT language expressed in terms of the theory of problems we are able to formally specify other mechanisms of model transformation languages, for example composition of hybrid transformations. In this direction we have implemented an algebra for composing model transformations that is supported by this formalization [20].

With respect to works that are related to our approach, the proposal presented in [16] develops an algebraic semantics for the MOF metamodeling framework, formalizing notions not yet clear in the MOF standard: *metamodel*, *model* and *conformance* of a model to its metamodel. By using the Maude language, this formal semantics is furthermore *executable*, and can be used to perform useful formal analyses. The authors plan to use this proposal as the kernel of a model management tool suite that provides support for QVT and graph-based model transformations within the EMF (Eclipse Modeling Framework).

In [17] the authors present a model transformation mechanism that is embodied by the *ModelGen* operator. *ModelGen* has been algebraically specified in Maude allowing developers to use formal tools to reason about transformation features, such as termination and confluence. The authors indicate how the *ModelGen* operator provides support for the QVT Relations language in the MOMENT (Model manageMENT) Framework.

In summary, the Semantics of the QVT is semi formally defined using a combination of natural language and mathematic notation as we can see in [5] or it is implicitly defined by its translation to a formal language, for example, the above mentioned translation of QVT Relations language to Maude. On the other hand, the implementation of execution engines for QVT- such as SmartQVT [18] for operational transformations and ModelMorf [19] for relational transformations – can be regarded as operational definitions of the QVT semantics

In general, all these works are only focused on one dialect of QVT: Relations Language or Operational Mappings Language. However, they do not cover the hybrid nature of QVT. This is the main difference with respect to our approach, which is expected to provide a formal semantics for the entire QVT language, embracing and synchronizing its two dimensions: declarative and operational. Finally, Poernomo in [22] develops an approach, not particularly for QVT, defining model transformation specifications as types and provably correct transformations as inhabitants of specification types.

As part of future work we intend to extend our approach, considering other semantics issues proposed by Stevens in [21], such as which transformations has sense to give them semantics and then, to specify the result of applying such semantics.

References

1. Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 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. 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, ISBN 2-7462-1213-7. (2006).
5. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification ptc/08-04-03. OMG, 2008.
6. Jouault F., Kurtev I. *Transforming Models with ATL*. Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005
7. Akehurst D., Howells W., McDonald-Maier K. Kent. *Model Transformation Language.* Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005
8. Lawley M., Steel J. *Practical Declarative Model Transformation with TefKat*. Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005
9. 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).
10. Veloso, P.A.S. Outline of a mathematical theory of general problems. Philosophia Naturalis; vol. 2/4 No. 1, (1984)
11. 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.
12. Meta Object Facility (MOF) 2.0. Final adopted Specification ptc/06-01-01, OMG, 2006.
13. OMG. The Object Constraint Language Specification–Version 2.0, for UML 2.0, http://www.omg.org, May 2006.
14. Giandini, Roxana. Ph.D thesis. University of La Plata. Buenos Aires, Argentina. February 2008.
15. Richters, M. A Precise Approach to Validating UML Models and OCL Constraints. Ph.D. thesis, Universit¨at Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.
16. Boronat, A. and Meseguer, J.: An Algebraic Semantics for MOF. In: Fiadeiro J. and Inverardi P. (Eds.): FASE and ETAPS 2008, Budapest, Hungary. LNCS, vol. 4961, pp.377–391Springer, April 2008.
17. Boronat, A, Cars´ý J., Ramos, I.: Algebraic specification of a model transformation engine. In: Baresi, Heckel,(eds.) FASE 2006 and ETAPS 2006. LNCS, vol. 3922, pp. 262–277. Springer, Heidelberg (2006)
18. smartQVT. An open source model transformation tool implementing the MOF 2.0 QVT-Operational language. http://smartqvt.elibel.tm.fr/. (2008)
19. ModelMorf. A Model Transformer for the MOF 2.0 QVT-Relations language. http://www.tcs-trddc.com/ModelMorf/. (2008)
20. Claudia Pons, Roxana Giandini, Gabriela Perez, Gabriel Baum. An Algebraic Approach for Composing Model Transformations in QVT. ATEM-4th International Workshop on Software Language Engineering at the 10th International Conference MoDELS 2007. Nashvillle, TN, US. October 2007.
21. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. Proc. of MoDELS 2007, Lecture Notes in Computer Science, 4735, pp. 1-15. Springer, October 2007.
22. Iman Poernomo. Proofs-as-Model-Transformations. Proc. of ICMT 2008, Lecture Notes in Computer Science, 5063, pp. 214-228. Springer, July 2008.