Implementing QVT-R Bidirectional Model Transformations using Alloy

Nuno Macedo and Alcino Cunha

HASLAB — High Assurance Software Laboratory
INESC TEC & Universidade do Minho, Braga, Portugal
{nfmmacedo,alcino}@di.uminho.pt

Abstract. QVT Relations (QVT-R) is the standard language proposed by the OMG to specify bidirectional model transformations. Unfortunately, in part due to ambiguities and omissions in the original semantics, acceptance and development of effective tool support has been slow. Recently, the checking semantics of QVT-R has been clarified and formalized. In this paper we propose a QVT-R tool that complies to such semantics. Unlike any other existing tool, it also supports meta-models enriched with OCL constraints (thus avoiding returning ill-formed models), and proposes an alternative enforcement semantics that works according to the simple and predictable "principle of least change". The implementation is based on an embedding of both QVT-R transformations and UML class diagrams (annotated with OCL) in Alloy, a lightweight formal specification language with support for automatic model finding via SAT solving.

1 Introduction

Model-Driven Engineering (MDE) is an approach to software development that focuses on models as the primary development artifact. In MDE different models may capture different views of the same system (typically different models are used to specify structural and dynamic issues) or may be used at different levels of abstraction (code is obtained by refining platform-independent models to platform-specific ones). All these (possibly overlapping) models should be kept somehow consistent, and changes to one model should be propagated to all the others in a consistent manner. Ideally, specifications of transformations between models should be *bidirectional*, in the sense that a single artifact denotes transformations that can be used in both directions. Moreover, these transformations cannot just map a source to a target model and vice-versa: if some source information is discarded by the transformation, to propagate an update in the target back to a new consistent source access to the original source model is also required, so that discarded information can be recovered.

To support the MDE approach the Object Management Group (OMG) has launched the Model-Driven Architecture (MDA) initiative, which prescribed the usage of UML [16] and OCL [17] for the specification of (object oriented) models and constraints over them. To specify transformations between models, the OMG

proposed the Query/View/Transformation (QVT) standard [15]. While QVT provides three different languages for the specification of transformations, the most relevant to MDE is the *QVT Relations* (QVT-R) language, that allows the specification of a bidirectional transformation by defining a single declarative consistency relation between two (or more) meta-models. Given this specification the transformation can be run in two modes: *checkonly*, to test if two models are consistent according to the specified relation; or *enforce*, that given two models and an execution direction (picking one of them as the target) updates the target model in order to recover consistency. The standard prescribes a "check-before-enforce" semantics, that is, enforce mode cannot modify the target if the models happen to be already consistent according to checking semantics.

Effective tool support for QVT-R has been slow to emerge, which hinders the universal adoption of this standard. In part, this is due to the incomplete and ambiguous semantics defined in [15]. While the checking semantics has recently been clarified and formalized [19, 3, 9], the enforcement semantics still remains largely obscure and even incompatible with other OMG standards. Namely, it completely ignores possible OCL constraints over the meta-models, thus allowing updates that can lead to ill-formed target models. Likewise, none of the existing QVT-R model transformation tools supports such constraints, which makes them unusable in most realistic scenarios. Unfortunately, there are other problems that affect them. Some do not even comply to the standard syntax and support only a "QVT-like" language (including not providing both running modes as required by the standard). Others support only a subset of QVT-R that is not expressive enough to support truly non-bijective bidirectional transformations (for example, ignoring the original target model in the enforce mode). Some purposely disregard QVT-R intended semantics (including checking semantics) and implementing a new (still unclear and ambiguous) one. In most cases it is not clear if the supported checking semantics is equivalent to the one formalized in [19, 3, 9]. And finally, none clarify the problems and ambiguities in the standard concerning enforcement semantics, and none presents a simple enough alternative for this mode that makes its behavior predictable to the user.

In this paper, we propose a QVT-R bidirectional model transformation tool that addresses all these issues. Both the meta-models and transformation specifications may be annotated with OCL, and it supports a large subset of the standard QVT-R language, including execution of both modes independently as prescribed. The main restriction is that recursion must be non-circular (or well-founded), which is satisfied in most of the interesting case-studies. The checking semantics closely follows the one specified in the standard, being equivalent to the one formalized in [19, 3, 9]. Finally, instead of the ambiguous (and OCL incompatible) enforcement semantics proposed in the standard, our tool follows the clear and predictable *principle of least change* [13], and restores consistency by simply returning target models that are at a minimal distance from the original. In particular, the"check-before-enforce" policy required by QVT-R is trivially satisfied by this semantics. Our tool supports two different mechanisms to measure the distance between two models: the *graph edit distance* (GED) [21],

that just counts insertions and deletions of nodes and edges in the graph that corresponds to a model; and a variation where the user is allowed to parameterize which operations should count as valid edits, by attaching them to the meta-model and specifying their pre- and post-conditions in OCL.

To achieve this, we propose an embedding of both QVT-R transformations and UML class diagrams (annotated with OCL) in Alloy [11], a lightweight formal specification language with support for automatic model finding via SAT solving. Alloy is based on relational logic, which has been shown to be very effective to validate and verify object-oriented models. Its relation with the MDA has also been explored before. In particular, tools to translate UML class diagrams annotated with OCL to Alloy have been proposed [1, 6], on top of which we build our embedding. The proposed tool already proved effective in debugging existing transformations, namely helping us unveiling several errors in the well-known object-relational mapping that illustrates the QVT-R specification [15].

Section 2 introduces the QVT-R language, describes the standard checking semantics, presents some of the problems with the enforcement semantics, and proposes and formalizes a simpler alternative based one the *principle of least change*. Section 3 presents our embedding of UML class diagrams (annotated with OCL) and QVT-R transformations in Alloy. Finally, Section 4 analyzes some related work, while Section 5 draws conclusions and points to future work.

2 QVT Relations

In this section we introduce the basic concepts and the semantics of the QVT-R language. A more detailed presentation can be found in the standard [15].

2.1 Basic concepts

A QVT-R specification consists of a *transformation* T between a set of models that states under which conditions they are considered consistent. For the remainder of this paper, we will restrict ourselves to transformations between two meta-models for simplicity purposes, although most concepts could be generalized to n-directional transformations. From T, QVT-R requires the inference of three artifacts: a relation $\boldsymbol{T} \subseteq M \times N$ that tests if two models $m \in M$ and $n \in N$ are consistent and transformations $\overrightarrow{\boldsymbol{T}} : M \times N \to N$ and $\overleftarrow{\boldsymbol{T}} : M \times N \to M$ that propagate changes on a *source* model to a *target* model, restoring consistency between the two. Transformations can be executed in two modes: *checkonly* mode, where the models are simply checked for consistency, denoted as $\boldsymbol{T}\ (m, n)$; and *enforce* mode, where $\overrightarrow{\boldsymbol{T}}$ or $\overleftarrow{\boldsymbol{T}}$ is applied to inconsistent models in order to restore consistency, depending on which of the two models should be updated. Note that both transformations take as extra argument the original model: if we originally had consistent models $m \in M$ and $n \in N$, and m is updated to m', $\overrightarrow{\boldsymbol{T}}$ takes as input both m' and n to produce the new consistent n'. This way we are able to retrieve from n information discarded in the transformation. This formalization of QVT-R is inspired by the concept of *maintainer* [13], and was first proposed

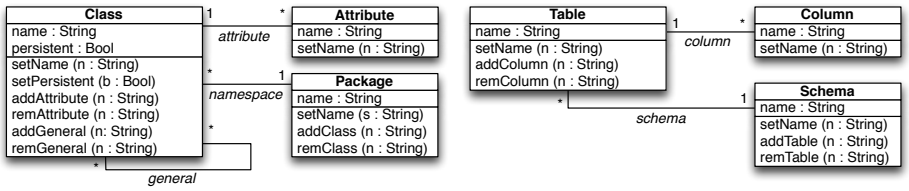

Fig. 1. Class diagrams of the `UML` and `RDBMS` meta-models.

in [18]. Naturally, when the transformations propagate an update the result is expected to be consistent. Formally, we say that the transformation is *correct* if:

$$\forall\ m \in M, n \in N : \boldsymbol{T}\ (m, \overrightarrow{\boldsymbol{T}}\ (m, n)) \wedge \boldsymbol{T}\ (\overleftarrow{\boldsymbol{T}}\ (m, n), n)$$

The transformations are also required to follow the "check-before-enforce" policy (also known as *hippocraticness* [18]), that can be formalized as follows:

$$\forall\ m \in M, n \in N : \boldsymbol{T}\ (m, n) \Rightarrow \overrightarrow{\boldsymbol{T}}\ (m, n) = n \wedge \overleftarrow{\boldsymbol{T}}\ (m, n) = m$$

A QVT-R transformation is defined by a set of *relations*. A relation consists of a *domain pattern* for each meta-model of the transformation, that defines which objects of the model it relates by pattern matching. It also may include *when* and *where* constraints, that act as a kind of pre- and post-conditions for the relation application, respectively. These constraints may contain arbitrary OCL expressions. The abstract syntax of a relation is the following:

```
[top] relation R { [variable declarations]
                   domain M a : A { π_M }
                   domain N b : B { π_N }
                   [when { ψ }] [where { φ }] }
```

In relation R, the domain pattern for meta-model M consists of a *domain variable* a and a template π_M for its properties, which candidate objects of type A must match. Likewise for the domain pattern π_N for meta-model N. To simplify the presentation, the above syntax restricts relations to have exactly one domain variable per meta-model. In a pattern template, an equality denotes an inclusion test if the multiplicity of the property is different from one. Templates can be complemented with arbitrary OCL constraints. Relations can optionally be marked as `top`, in which case they must hold for all objects of the specified class. Otherwise, they are only required to hold for particular objects when invoked in a `where` clause.

As an example, we will define a simplified version of the classic object-relational mapping transformation that illustrates the QVT-R specification [15]. Although simplified, this version still exhibits some of the problems of the original version, which we will describe in the next section. Figure 1 depicts a simplified version of the object and relational meta-models, including possible edit operations. Figure 2 defines a transformation `Uml2Rdbms`, whose goal

```
transformation Uml2Rdbms
  (uml:UML,rdbms:RDBMS) {
  // PackageToSchema
  top relation P2S {
    n:String;
    domain UML p:Package {
      name=n };
    domain RDBMS s:Schema {
      name=n }; }
  // ClassToTable
  top relation C2T {
    n:String;
    domain UML c:Class {
      persistent=true,
      namespace=p:Package,
      name=n };
    domain RDBMS t:Table {
      schema=s:Schema,
      name=n };
    when { P2S(p,s); }
    where { A2C(c,t); } }
// AttributeToColumn
relation A2C {
  domain UML c:Class {};
  domain RDBMS t:Table {};
  where { PA2C(c,t);
          SA2C(c,t); } }
// PrimitiveAttributeToColumn
relation PA2C {
  n:String;
  domain UML c:Class {
    attribute=a:Attribute {
      name=n } };
  domain RDBMS t:Table {
    column=cl:Column {
      name=n } }; }
// SuperAttributeToColumn
relation SA2C {
  domain UML c:Class {
    general=g:Class {}};
  domain RDBMS t:Table {};
  where { A2C(g,t); } } }
```

Fig. 2. Simplified version of the `Uml2Rdbms` QVT-R transformation.

is to map every persistent class in a package to a table in a scheme with the same name. Each table should contain a column for each attribute (including inherited ones) of the corresponding class. A constraint of the `UML` metamodel that cannot be captured by class diagrams, neither QVT-R key constraints, is the requirement that the association `general` should be acyclic. One must resort to OCL to express it, for example by adding the invariant `context Class inv: not self.closure(general)->includes(self)`.

There are two top relations: `P2S` that maps each package to a schema with the same name, and `C2T` that maps each class to a table with the same name. To ensure that classes are only mapped to tables if they belong to related packages and schemas, the relation `C2T` invokes `P2S` (with concrete domain variables) in the `when` clause. For a concrete class `c` and table `t`, `C2T` also calls relation `A2C` in the `where` clause, that will be responsible to map the attributes of `c` to columns of `t`. `A2C` directly calls `PA2C`, that translates the attributes directly declared in `c` to columns on table `t`, and `SA2C`, that recursively calls `A2C` on the `generals` of `c`, so that inherited attributes are also translated to columns of `t`.

2.2 Checking semantics

QVT-R's checking semantics assesses if two models are consistent according to the specified transformation. Although the consistency test is by itself important, it is also an essential feature in enforce mode since it must "check-before-enforce".

The semantics of a relation differs whether it is invoked at the top-level or with concrete domain variables. The specified top-level semantics is directional. As such, from each relation R two consistency relations $R_{\blacktriangleright} : M \times N$ and $R_{\blacktriangleleft} : M \times N$ must be derived, to check if $m : M$ is R-consistent with $n : N$ and if $n : N$ is R-consistent with $m : M$, respectively. The former can be formalized as follows:

$$R_{\blacktriangleright}\ (m : M, n : N) \equiv \forall\ xs \mid \psi_{\triangleright} \wedge \pi_M \Rightarrow (\exists\ ys \mid \pi_N \wedge \phi_{\triangleright})$$
$$\textbf{where}\ xs = \mathsf{fv}(\psi \wedge \pi_M)\ \cup\ \{\, a : A \,\}, ys = (\mathsf{fv}(\pi_N \wedge \phi)\ \cup\ \{\, b : B \,\}) - xs$$

Here $\mathsf{fv}(e)$ retrieves the set of free variables from the expression e, so xs denotes the set of variables used in the `when` constraint and the source pattern, while ys is the set of variables used exclusively in the `where` constraint and in the target pattern. Given a formula ψ, $\psi_{\triangleright}$ denotes the same formula with all relation invocations replaced by the respective directional version. This semantics is rather straightforward: essentially, for every element $a : A$ that satisfies the `when` condition and matches the M domain pattern, there must exist an element $b : B$ that satisfies the `where` condition and matches the N domain pattern. The semantics in the opposite direction is dual. Two models are consistent according to a QVT-R transformation T if they are consistent for all top relations in both directions. Assuming that Top_T is the set of all top level relations we have:

$$\boldsymbol{T}\ (m : M, n : N) \equiv \forall\ R : \mathsf{Top}_T \mid R_{\blacktriangleright}\ (m, n) \wedge R_{\blacktriangleleft}\ (m, n)$$

The QVT-R standard [15] defines rather precisely the top-level semantics, but is omissive about the semantics of relations invoked with concrete domain variables. Recent works on the formalization of QVT-R check semantics [19, 3, 9] clarify that it is essentially the same as the top-level – still directional, but defined over specific meta-model classes by fixing the domain variables. As such, from each relation R with domain variables of type A and B, two consistency relations $R_{\triangleright} : A \times B$ and $R_{\triangleleft} : A \times B$ are inferred, to check if two concrete objects a and b are consistent:

$$R_{\triangleright}\ (a : A, b : B) \equiv \forall\ xs \mid \psi_{\triangleright} \wedge \pi_M \Rightarrow (\exists\ ys \mid \pi_N \wedge \phi_{\triangleright})$$
$$\textbf{where}\ xs = \mathsf{fv}(\psi \wedge \pi_M), ys = \mathsf{fv}(\pi_N \wedge \phi) - xs$$

Although it may be tempting (and probably more intuitive) to define $R_{\blacktriangleright}$ in terms of $R_{\triangleright}$, that is $R_{\blacktriangleright}\ (m, n) \equiv \forall\ a : A \mid \exists\ b : B \mid R_{\triangleright}\ (a, b)$, this definition is not semantically equivalent to the one presented above, as already discussed in [3]. For instance, consider the semantics (in the direction of `UML`) of relation `PA2C` from the `Uml2Rdbms` transformation:

$$\texttt{PA2C}_{\blacktriangleleft}\ (m : \texttt{UML}, n : \texttt{RDBMS}) \equiv$$
$$\forall\ t : \texttt{Table}, cl : \texttt{Column}, n : \texttt{String} \mid cl \in t.\texttt{column} \wedge cl.\texttt{name} = n \Rightarrow$$
$$\exists\ c : \texttt{Class}, a : \texttt{Attribute} \mid a \in c.\texttt{attribute} \wedge a.\texttt{name} = n$$
$$\texttt{PA2C}_{\triangleleft}\ (c : \texttt{Class}, t : \texttt{Table}) \equiv$$
$$\forall\ cl : \texttt{Column}, n : \texttt{String} \mid cl \in t.\texttt{column} \wedge cl.\texttt{name} = n \Rightarrow$$
$$\exists\ a : \texttt{Attribute} \mid a \in c.\texttt{attribute} \wedge a.\texttt{name} = n$$

Consider a simple UML model where a Class a with an Attribute x extends a Class b with an Attribute y. Consider also a RDBMS with a Table a with Columns x and y. While $\texttt{PA2C}_{\blacktriangleleft}$ holds for this pair of instances, $\texttt{PA2C}_{\triangleleft}$ returns false for every pair of Class and Table.

Due to this asymmetry and the directionality of the semantics, the behavior of QVT-R transformations may not be the expected one. In particular, Uml2Rdbms as defined in the standard does not have a bidirectional semantics, in the sense that the only pairs of consistent and valid finite models are ones where all classes are non-persistent and there are no tables. To see why this happens, consider the relations A2C and SA2C when checked in the direction of Class. These relations call each other recursively, and their non top-level semantics is:

$$\texttt{A2C}_{\triangleleft}\ (c : \texttt{Class}, t : \texttt{Table}) \equiv \texttt{PA2C}_{\triangleleft}\ (c, t) \wedge \texttt{SA2C}_{\triangleleft}\ (c, t)$$
$$\texttt{SA2C}_{\triangleleft}\ (c : \texttt{Class}, t : \texttt{Table}) \equiv \exists\ g : \texttt{Class} \mid g \in c.\texttt{general} \wedge \texttt{A2C}_{\triangleleft}\ (g, t)$$

Assuming the transformation takes into account the OCL constraint requiring general to be acyclic, $\texttt{A2C}_{\triangleleft}\ (c, t)$ never holds in a finite model, since c will be required to have an infinite ascending chain of generals. This is due to the under-restrictive SA2C domain pattern in the RDBMS side (empty in this case), that requires every Table to have a matching Class with a general, which, due to recursion, is also required to have a general, and so on. This is but one of the problems that occur in the original specification of this transformation, and is another example of the ambiguities that prevail in the QVT standard [15]: while it requires consistency to be checked in both directions, the case-study used to illustrate it was clearly not developed with bidirectionality in mind. Note that checking consistency only in the direction of RDBMS does not suffice, since, for example, it will not prevent spurious tables to appear in the target schema.

Concerning recursion we can distinguish two situations: one is well-founded recursion, where the call graph of the transformation contains a loop, but in any evaluation it is traversed only finitely many times; another is cyclic (or infinite) recursion, where such a loop may actually be traversed infinitely many times (e.g., when a relation directly or indirectly calls itself with the same arguments). The semantics of well-founded recursion is not problematic, but the standard is omissive about what should happen when infinite recursion occurs. A possible interpretation is that it should not be allowed, although in general it is undecidable to detect it. Similarly to some QVT-R formalizations [19, 9], the embedding presented in this paper is not well-defined when infinite recursion occurs.

Recently, a formal semantics of QVT-R was proposed [3] that is well-defined even in presence of infinite recursion, by resorting to the modal mu calculus. To see why taking OCL constraints into account is fundamental, a transformation conforming to this semantics, but that ignores the requirement that general is acyclic, would consider a (ill-formed) UML model with a single persistent Class a that generalizes itself consistent with a RDBMS model with a Table a.

To prevent the above described problem in the Uml2Rdbms transformation, one could tag columns with the path to the particular general they originated from, and then refine the RDBMS domain pattern to prevent problematic recursive calls. A simpler alternative is to resort to the transitive closure operation

(recently added to OCL [17]), and just map at once all declared or inherited attributes of a given class to columns of the respective table. In this new version of `Uml2Rdbms` (that will be considered in the remainder of the paper), `A2C`, `PA2C` and `SA2C` are replaced just by the following alternative definition of `A2C`:

```
relation A2C { cn:String; a:Attribute; g:Class;
  domain UML c:Class {} { (c->closure(general)->includes(g) or g=c) and
                          g.attributes->includes(a) and a.name=cn };
  domain RDBMS t:Table { column=cl:Column { name=cn } }; }
```

The OCL constraint in the `UML` domain pattern acts as a pre-condition when applying the transformation in the direction of `RDBMS`, and as a post-condition in the other direction. As such, it could not be specified in the `when` clause, since it would act as (an undesired) pre-condition for both scenarios.

2.3 Enforcement semantics

Unlike the checking semantics, and as far as we know, no attempt has been made to completely formalize the enforcement semantics described in the standard [15]. Although it has many ambiguities and omissions, due to the reasons presented next, we believe that the intended semantics for this mode is quite undesirable. Instead, we propose an alternative that is easy to formalize, more flexible, and more predictable to the end-user.

In the QVT-R standard, update propagation is required to be deterministic. This is a desirable property, since it makes its behavior more predictable. However, to ensure determinism, every transformation is required to follow very stringent syntactic rules that reduce update translation to a trivial imperative procedure. Namely, it should be possible to order all constraints in a relation (except for the target domain pattern), such that the value of every free variable is fixed by a previous constraint. Although not clarified in the standard, this means that relations that are invoked in `when` and `where` constraints are either invoked with previously bound variables, or are required to also be deterministic, even if the intention was to only make update propagation deterministic. For example, in transformation `Uml2Rdbms`, update propagation in the direction of `RDBMS` will only be deterministic for relation `C2T` if at most one s is consistent with p according to relation `P2S` (note that s is still free in the `when` clause). In this particular example that happens to be true, but in general such determinism is undesirable since it forces relations to be one-to-one mappings, limiting the expressiveness of the language. Moreover, it defeats the purpose of a declarative transformation language, since one is forced to think in terms of imperative execution and write more verbose transformations. For example, our simpler version of `A2C` using transitive closure would not be allowed, since the value of g is not known a priori when enforcing consistency in the direction of `UML`.

Another problem is the predictability of update propagation. Being deterministic is just part of the story – it should be clear to the user why some particular element was chosen to be updated instead of another. The only mechanism proposed by QVT-R to control updatability are *keys*. For example, we could add

the command `key Table (name, schema);` to our running example to assert that tables are uniquely identified by the pair of properties `name` and `schema`. If an update is required on a table to restore consistency (for example, when an attribute is added to a class), such key is used to find a matching table. When found, an update is performed, otherwise a new table is created. This works well when all domains involved in relations have natural keys, which again points to have only one-to-one mappings, but fails if such keys do not exist. In those cases, the standard prescribes that update propagation should always be made by means of creation of new elements, even if sometimes a simple update to an existing element would suffice. Since creation requires defaults for mandatory (multiplicity one) properties, this would result in models with little resemblance with the original (which would basically be discarded).

Our alternative enforcement semantics is based on the *principle of least change*, first proposed in the context of *maintainers* [13], and that enforces predictability by requiring updates to be as small as possible. QVT-R "check-before-enforce" policy is just a particular case of this more general principle. Let $\Delta_M : M \times M \to \mathbb{N}$ be an operation that computes the update distance between elements of M. Then, the principle of least change states that the models returned by the transformations $\overrightarrow{\boldsymbol{T}}$ and $\overleftarrow{\boldsymbol{T}}$ are just the consistent models closest to the original. Formally, we have:

$$\forall\, m \in M, n, n' \in N : \boldsymbol{T}\ (m, n') \Rightarrow \Delta_N\ (\overrightarrow{\boldsymbol{T}}\ (m, n), n)\ \leqslant\ \Delta_N\ (n', n)$$
$$\forall\, m, n' \in M, n \in N : \boldsymbol{T}\ (m', n) \Rightarrow \Delta_M\ (\overleftarrow{\boldsymbol{T}}\ (m, n), m)\ \leqslant\ \Delta_M\ (m', m)$$

Assuming that the distance is only null when the model is unchanged (i.e., $\Delta\ (n, n') = 0 \equiv n = n'$), it is trivial to show that these properties reduce to hippocraticness when the models m and n are already consistent. Note, that this principle by itself does not ensure determinism, although it reduces substantially the set of possible results. If among the returned models the user further wishes to favor a particular subset, keys or OCL constraints can be added to the metamodel to guide the transformation engine. In the next section we will describe the implementation of the proposed semantics. We will also propose two different techniques to measure update distance between models. In one of them, the user is allowed to parameterize which operations should count as valid edits, thus providing an extra mechanism to achieve determinism if the user so desires.

3 Embedding QVT-R in Alloy

In this section we present our embedding of QVT-R in Alloy [11]. Due to space limitations some knowledge of Alloy will be assumed, although we believe most definitions will be clear from context.

3.1 UML class diagrams annotated with OCL

The models upon which our transformations are defined consist of UML class diagrams annotated with OCL constraints. Some translations have been proposed

to embed such models in Alloy, namely [1, 6]. We will base our embedding on the translation proposed in [6], since, unlike other proposals, it covers an expressive OCL subset that includes closure and operation specification via pre- and pos-conditions. Here, we will just briefly present this translation.

Classes and associations (including attributes) can be directly translated to signatures and relations in Alloy. Likewise for the inheritance relationship, that Alloy also supports. Since Alloy instances are built from immutable atoms, we resort to the well-known *local state idiom* [11] to capture updates to a given model. This means that a special signature will be introduced to represent each meta-model, whose atoms will denote different models (or evolutions of a given model). To each relation (representing an association or an attribute) an extra column of this type is added, to allow its value to change in different models. We also extend the translation proposed in [6] to allow classes to have different elements in different models: for each class a special binary relation with the same name will capture the objects of that class that belong to each model. Boolean attributes are encoded similarly: a binary relation captures which objects have the attribute set to `true` in each model. For example, the `Class` class of our `UML` meta-model is translated to the following signature declaration.

```
sig Class { class : set UML, attribute : Attribute -> UML,
            general : Class -> UML, namespace : Package -> UML,
            name : String -> UML, persistent : set UML }
```

The binary relation `class` captures the `Class` objects that exist in each `UML` model. The remaining relations model the respective `Class` associations and attributes. With the relational composition operator we can access the values of these relations for a given `UML` model `m`. For example, `general.m` is a relation that maps each `Class` to its `general` in model `m`, and `persistent.m` is the set of `Class`es that have the attribute `persistent` set to `true` in that model.

Constraints must also be generated to ensure the correct multiplicities, and that relations only relate elements in the same model (inclusion dependencies). For example, fact `all m:UML | namespace.m in class.m -> one package.m` is generated to capture the cardinality constraints of association `namespace`, and to force it, for each `UML` model `m`, to be a subset of the cartesian product between `class.m` and `package.m` (respectively, the sets of `Class`es and `Package`s of model `m`). OCL invariants are also automatically translated to Alloy facts, resulting in universal quantifications over the given type. For example, the OCL invariant stating that `general` is acyclic is translated to Alloy as `all m:UML, self:class.m | self not in self.^(general.m)`, where `^(general.m)` is the transitive closure of relation `general` projected over `m`.

3.2 QVT-R transformations

For each relation R we declare two Alloy predicates to specify $R_{\triangleright}$ and $R_{\triangleleft}$. Besides the respective domains elements, these are also parameterized by the models they are being applied to. Since in Alloy predicates cannot call each

other recursively, predicates $R_{\triangleright}$ and $R_{\triangleleft}$ are defined in terms of auxiliary relations specified by comprehension. Top relations $R_{\blacktriangleright}$ and $R_{\blacktriangleleft}$ are also specified by predicates, which are only parameterized by the models. The definition of all these predicates follows closely the formalization of Section 2.2. For example, $\texttt{C2T}_{\blacktriangleright}$ is specified as follows:

```
pred When_C2T_RDBMS [m:UML, n:RDBMS, p:Package, s:Schema] {
  P2S_RDBMS[m,n,p,s] }
pred Pattern_C2T_UML [m:UML, c:Class, n:String, p:Package] {
  n in c.name.m && c in persistent.m && p in c.namespace.m }
...
pred Top_C2T_RDBMS [m:UML,n:RDBMS] {
 all c:class.m, n:String, p:package.m, s:schema.n |
  When_C2T_RDBMS[m,n,p,s] && Pattern_C2T_UML[m,c,n,p] =>
  some t:table.n |
    Pattern_C2T_RDBMS[n,t,n,s] && Where_A2C_RDBMS[m,n,c,t] }
```

Predicates are used to specify the `when` and `where` clauses, and the domain patterns of each relation. Note that predicate `P2S_RDBMS` is the predicate specifying $\texttt{P2S}_{\triangleright}$. Note also how, in the specification of $\texttt{C2T}_{\blacktriangleright}$, quantifications are restricted to range over the respective models.

The checking semantics of the transformation is a predicate that checks all top relations in both directions. In our running example we have:

```
pred Uml2Rdbms [m:UML,n:RDBMS]{ Top_P2S_RDBMS[m,n] && Top_P2S_UML[m,n]
                             && Top_C2T_RDBMS[m,n] && Top_C2T_UML[m,n] }
```

Regarding enforcement semantics, as described in Section 2.3, we implement the principle of least change, which requires the measurement of the update distance between two models. We propose two different mechanisms for measuring such distance. The first one is the *graph edit distance* (GED) [21], which counts the distance between two graphs as the number of node and edge insertions and deletions needed to obtain one from the other. Note that an Alloy instance is isomorphic to a labelled graph whose nodes are the atoms, and whose edges are tuples in relations. With this mechanism, $\Delta_{\texttt{UML}}$ can be computed as follows:

```
fun Delta_UML [m,m':UML] : Int {
  (#((class.m - class.m') + (class.m' - class.m))).plus[
  (#((name.m - name.m') + (name.m' - name.m))).plus[...]] }
```

Assuming `m'` represents an updated version of `m`, this function sums up, for every signature and relation, the size of their symmetric difference in both models. To avoid Alloy's standard wrap around semantics for integers, model finding is executed with option `Forbid Overflow` [14].

This simple definition for distance assumes a fixed repertoire of edit operations which may not be desirable. In particular, there is no control over the "cost" of complex operations. For example, changing the name of a class will have a cost of 2, since it requires deleting the current name edge and inserting a new one, while adding a new attribute to a class will cost 3, since it requires creating a new attribute, setting its name, and adding it to the class. One may wish

both these operations to be atomic edits and have the same unitary cost. Also, one may wish to allow only particular edits in order to control non-determinism.

As such, we propose an alternative measure, where the user is allowed to specify in the meta-model which edit operations that are allowed for each class. We require them to be specified using pre- and post-conditions defined in (a subset of) OCL, to be automatically converted to Alloy using the translation procedure defined in [6]. Essentially, each operation will originate an Alloy predicate that checks if it holds between given pre- and post-models. For example, Figure 1 defines the interface of possible edit operations for our running example.

Given the specifications of operations, we constrain models to form an ordering, where each step corresponds to the application of an edit operation.

```
open util/ordering[UML]
pred setName [p:Package, n:String, m,m':UML] { ... }
pred addClass [p:Package, n:String, m,m':UML] { ... }
...
fact { all m:UML, m':m.next | {
         some p:package.m, n:String | setName[p,n,m,m'] or
         some p:package.m, n:String | addClass[p,n,m,m'] or ... } }
```

In this case, $\Delta_{\texttt{UML}}$ will be the number of models (intermediate steps) required to achieve a consistent target, which, as we will see next, will be determined by the scope of the signature denoting the respective meta-model.

3.3 Executing the semantics

Executing the transformation in checkonly mode is fairly simple: we just need to check the consistency predicate for a pair of concrete models. To represent a concrete model, we use singleton signatures to denote specific objects and facts to fix the interpretation of relations. For example, a `UML` model `M` with two classes `A` and `B` with no attributes in a single package `P`, where `A` is persistent and extends the non-persistent `B`, can be specified as follows:

```
one sig M extends UML {}
one sig P extends Package {}
one sig A,B extends Class {}
fact { class.M = A + B && package.M = P && namespace.M = A->P + B->P &&
       general.M = A->B && no attribute.M && persistent.M = A && ... }
```

To check if `UML` model `M` is consistent with `RDBMS` model `N` the command **check** `{ Uml2Rdbms[M,N] }` is issued, with the scope of each signature being set to the number of elements of the respective class in each of the two models. Regarding enforce mode with GED minimization, in order to determine a new `UML` model `M'` consistent with `RDBMS` model `N`, with original model `M`, the command **run** `{ Uml2Rdbms[M',N] && Delta_UML[M,M']=`Δ `}` is issued with increasing values Δ (starting at 0). In this case, the scope of each signature is set to the number of elements of the respective class plus Δ, to allow complete freedom in the choice of edit operations. The calculation and increment of both Δ and the scope is performed automatically by our tool.

To execute the enforce mode with user-specified edit operations the command `run { Uml2Rdbms[M',N] && M=first && M'=last }` is issued with increasing scopes Δ (plus one) for signature `UML`. The original and target models are constrained to be the first and last in the model ordering, respectively. Determining the scope for the remaining signatures is not straightforward in this case, since edits can be arbitrary operations. For the moment we are using a rough approximation, that assumes creation of new objects to be specified via existential quantification: for every increment of Δ, the scope of a signature is increased by the maximum number of such quantifications over all edit operations.

The user is required to specify an upper-bound for Δ that limits the search for consistent targets. If several consistent models are found at the minimum distance our tool warns the user and allows him to see the different alternatives. If the user then desires to reduce such non-determinism, he can, for example, add extra OCL constraints to the meta-model or narrow the set of allowed edit operations to target a specific class of models.

4 Related Work

Regarding tools support for QVT-R transformations, Medini and ModelMorf are the main existing functional tools. Medini [10] is an Eclipse plugin for a subset of the QVT-R language. Although popular, its (unknown) semantics admittedly disregards the semantics from the QVT standard (it does not have a checkonly mode for instance). ModelMorf [20] allegedly follows the QVT standard closely (although once again the concrete semantics are unknown), since its development team was involved in the specification of the standard. However, the development of the tool seems to have stopped. None of these tools has support for OCL constraints on the meta-models. Other prototype tools have been proposed but once again the implemented semantics are not completely clear. Moment-QVT [2] is an Eclipse plug-in for the execution of QVT-R transformations by resorting to the Maude rewriting system; [12] proposes the embedding of QVT-R in Colored Petri Nets; [8] discusses the possible implementation of QVT-R transformations in TGGs. All these tools support only unidirectional transformations, in the sense that they ignore the original target model. As such, they are not able to retrieve information not present in the source, leading to the generation of completely new models every time the transformation is applied. Once again, none supports OCL constraints on the meta-model.

A technique that follows an approach similar to ours is the JTL tool [5], although it does not support QVT-R, but rather a restricted QVT-like language. Like ours, JTL generates models by resorting to a solver (the DLV solver), which is able to retrieve some information from the original target. However, it is not clear how the solver chooses which information to retrieve or how the new model is generated. It also forces the totality of the transformation, returning inconsistent models in case there is no consistent one.

Regarding the validation of QVT-R transformations two approaches have been proposed that also rely on solvers. In [7] the authors use Alloy to verify the

correctness of QVT-R specifications, in order to guarantee the well-formedness of the output and avoid run-time errors. In [4] OCL invariants of the shape "forall-there-exist" are inferred from QVT-R transformations (much like the checking semantics), that allow the validation of QVT-R specifications under a set of properties. It supports OCL constraints in the meta-model and recursive calls are translated to recursive OCL specifications. However, both these approaches are not focused on enforce mode and its semantics, and do not analyze the behavior of the transformation for concrete input models. Using our embedding we can do so, and also support the validation of similar properties, like checking if a transformation is injective or that all consistent models are well-formed.

5 Conclusions and Future Work

This paper proposed a QVT-R bidirectional model transformation tool, supporting both the standard checking semantics and a clear and precise enforcement semantics based on the principle of least change. It also supports meta-models annotated with OCL constraints and specification of allowed edit operations, which allows its applicability to non-trivial domains and provides a fine-grained control over non-determinism. The implementation is based on an embedding in Alloy, taking advantage of its model finding abilities. Although we only described the support for bidirectional transformations, our embedding can trivially be generalized to the multi-directional scenario, where updates on multiple models are propagated to a designated target, another feature not currently offered by any existing QVT-R tool.

Being solver-based, the main drawback of the proposed tool is performance. Improving it is the main goal of our future work: we intend to explore incremental solving techniques to speed-up the execution of successive commands with increasing scope, and to define mechanisms to infer which parts of target model can be fixed *a priori* in order to speed-up solving. However, even in its present status the tool is already fully functional and can be used to perform transformations of medium-sized models. In particular, it already proved effective in debugging existing transformations, namely helping us unveiling several errors in the well-known object-relational mapping that illustrates QVT-R specification.

Acknowledgements

The authors would also like to thank the anonymous reviewers for the valuable comments and suggestions. This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by national funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020532. The first author is also sponsored by FCT grant SFRH/BD/69585/2010.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9, 69–86 (2010)
2. Boronat, A., Carsí, J., Ramos, I.: Algebraic specification of a model transformation engine. In: FASE 2006. LNCS, vol. 3922, pp. 262–277. Springer (2006)
3. Bradfield, J., Stevens, P.: Recursive checkonly QVT-R transformations with general when and where clauses via the modal mu calculus. In: FASE 2012. LNCS, vol. 7212, pp. 194–208. Springer (2012)
4. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2012)
5. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer (2010)
6. Cunha, A., Garis, A., Riesco, D.: Translating between Alloy specifications and UML class diagrams annotated with OCL (2012), available at `http://www.di.uminho.pt/~mac/Publications/AlloyMDA.pdf`
7. Garcia, M.: Formalization of QVT-Relations: OCL-based static semantics and Alloy-based validation. In: MDSD Today 2008. pp. 21–30. Shaker Verlag (2008)
8. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. Software and System Modeling 9(1), 21–46 (2010)
9. Guerra, E., de Lara, J.: An algebraic semantics for QVT-relations check-only transformations. Fundam. Inform. 114(1), 73–101 (2012)
10. ikv++ technologies ag: medini QVT, available at `http://projects.ikv.de/qvt/`
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, London, England, revised edn. (2012)
12. de Lara, J., Guerra, E.: Formal support for QVT-Relations with Coloured Petri Nets. In: MoDELS 2009. LNCS, vol. 5795, pp. 256–270. Springer (2009)
13. Meertens, L.: Designing constraint maintainers for user interaction (1998), manuscript available at `http://www.kestrel.edu/home/people/meertens`
14. Milicevic, A., Jackson, D.: Preventing arithmetic overflows in Alloy. In: ABZ 2012. LNCS, vol. 7316, pp. 108–121. Springer (2012)
15. OMG: MOF 2.0 Query/View/Transformation specification (QVT), version 1.1 (January 2011), available at `http://www.omg.org/spec/QVT/1.1/`
16. OMG: OMG Unified Modeling Language (UML), version 2.4.1 (August 2011), available at `http://www.omg.org/spec/UML/2.4.1/`
17. OMG: OMG Object Constraint Language (OCL), version 2.3.1 (January 2012), available at `http://www.omg.org/spec/OCL/2.3.1/`
18. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and System Modeling 9(1), 7–20 (2010)
19. Stevens, P.: A simple game-theoretic approach to checkonly QVT relations. Software and System Modeling (2011), `http://dx.doi.org/10.1007/s10270-011-0198-8`
20. Tata Research Development and Design Centre: ModelMorf, available at `http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`
21. Voigt, K.: Structural Graph-based Metamodel Matching. Ph.D. thesis, University of Desden (2011)