# Enforcing QVT-R with mu-calculus and games

Julian Bradfield and Perdita Stevens

School of Informatics
University of Edinburgh

**Abstract.** QVT-R is the standard Object Management Group bidirectional transformation language. In previous work, we gave a precise game-theoretic semantics for the checkonly semantics of QVT-R transformations, including the recursive invocation of relations which is allowed and used, but not defined, by the QVT standard. In this paper, we take up the problem of enforce semantics, where the standard attempts formality, but at crucial points lapses into English. We show that our previous semantics can be extended to enforce mode, giving a precise semantics taking the standard into account.

## 1 Introduction

QVT-R is the OMG standard *bidirectional* model transformation language [8]. It is bidirectional in the sense that, rather than simply permitting one model to be built from others, it permits changes to be propagated in any direction, something which seems to be essential in much real-world model-driven development. The same transformation can be read as specifying the circumstances under which models are consistent (checkonly mode) or as specifying exactly how one model should be modified so as to restore consistency that has been lost (enforce mode). This dual use of the same transformation text is beneficial in engineering terms; separate texts for checkonly and enforce transformations would be a maintenance nightmare. In earlier work [3, 12] we gave formal semantics for QVT-R in checkonly mode, including transformations in which a relation may recursively invoke itself; this feature is used even in the example in [8], and presents interesting complications which we tackled using the modal mu calculus. A thorough understanding of checkonly mode is prerequisite to understanding enforce mode, because of the requirement (hippocraticness) that running a transformation in enforce mode should not modify models which are already consistent.

In this paper, we go on to give a formal semantics for QVT-R in enforce mode. Unlike previous work, we do not restrict to the case where the target model is created afresh from the source model; we work with the general case in which there is an existing target model which must be taken into account when producing a new version. This, the typical case of bidirectional transformation arising in model-driven development, is more complex – even when the transformation is not recursive – because there will usually be many different target

models that are consistent with the given source model, and the preexisting target model influences which one is produced.

As an illustrative example, consider the following transformation (given in ModelMorf's QVT-R syntax), which operates on models that have two kinds of model elements `MEtop` and `MEchild`. Both kinds have names; `MEtop` elements can also have children which are of type `MEchild`. We represent such models in the obvious way as forests, and notate them using the names of elements, e.g. $\{N \to \{child1, child2, child2\}\}$ represents a model containing one element of type `MEtop` with `name` "N", having three children of type `MEchild` whose `name`s are "child1", "child2", "child2". Notice that there is nothing to prevent two model elements having the same `name` (no key declaration, for example).

```
transformation NonTopEnforce (m1 : TwoLayerMM1 ; m2 : TwoLayerMM2) {
top relation R {
  n : String;
  firstchild : TwoLayerMM1::MEchild;
  secondchild : TwoLayerMM2::MEchild;
  enforce domain m1 me1:MEtop {name = n, child = firstchild};
  enforce domain m2 me2:MEtop {name = n, child = secondchild};
  where { S(firstchild,secondchild); }}

relation S {
  n : String;
  enforce domain m1 me1:MEchild {name = n};
  enforce domain m2 me2:MEchild {name = n};}}
```

Models are consistent according to this transformation provided that two directional checks succeed. For every `MEtop` element `me1` in `m1`, there must be an `MEtop` element `me2` of the same `name` in `m2`, such that the `name` of every `child` of `me1` occurs as the `name` of a `child` of `me2`. The check in the `m2` direction is symmetric.

Running this transformation in checkonly mode in ModelMorf, consulting [8], yields no surprises. Running it in enforce mode lets us illustrate semantic choices that [8] has made and some odd behaviour of ModelMorf. Let `m1` be $\{N \to \{child1, child2, child3\}\}$ throughout. If `m2` is identical, of course it is not modified (except that ModelMorf, for some reason, rewrites all the xmi:ids): check-before-enforce ensures that, since the models are already consistent, no change is made. Next, let `m2` be the empty model, and enforce so that a new `m2` is created from `m1`. ModelMorf produces $\{N \to \{child1\}, \{N \to \{child2\}, \{N \to \{child3\}\}$ (not the copy of `m1` that some might intuitively expect). This is consistent with (our reading of) [8]; it arises because each valid binding to variables in `R` is checked independently. Thus for each binding to `me1`, (`n` and) `firstchild` for which no matching binding is found, new objects are created for `me2` and `secondchild` and the `child` property of `me2` is set to `secondchild`.

Next consider `m2` $= \{N \to \{child1, child2, cuckoo\}\}$. Interpreting this as the name of the third child being wrong, we would intuitively expect the least change modification to be made, i.e. for the name "cuckoo" to be changed to "child3". By the same argument as above, however, we see that ModelMorf is in

accord with [8] in actually returning $\{N \to \{child1, child2\}, \{N \to \{child3\}\}\}$. Rather than modify an existing child, a whole new binding is created to be the match. Here we have the first obvious use of the delete phase: the `child` named "cuckoo" has been removed. Why is this? We may argue: because if it had been left alone – if ModelMorf had returned $\{N \to \{child1, child2, cuckoo\}, \{N \to \{child3\}\}\}$ – the resulting version of `m2` would not have been consistent with `m1` *when checked in the direction of* `m1`. Even though any run of ModelMorf has a direction (in contrast to [8] in which a transformation being evaluated in checkonly mode involves checks in all directions, as explained in detail in [12]), ModelMorf does ensure that the result of an enforce transformation passes the checkonly transformation in all directions, not just in the same direction as the enforce.

Finally, however, consider `m2` $= \{N \to \{child1, child2, child2\}\}$. In this case, the checkonly transformation run in the direction of `m1` will already succeed; the only problem is when running the checkonly transformation in the direction of `m2`, as there is no valid match for valid binding of `me1` to the unique `MEtop` and `firstchild` to the `MEchild` with `name` "child3". We would expect $\{N \to \{child1, child2, child2\}, \{N \to \{child3\}\}\}$ as the result of enforcement. In fact, however, ModelMorf unnecessarily deletes one of the "child2" `MEchild`s, giving exactly the same result as in the "cuckoo" case. As we shall discuss, the interpretation of element deletion in [8] is problematic and so it is not very surprising that ModelMorf sometimes gives odd results. We discuss this further in section 4.3.

We will return to these examples after presenting our semantics.

## 2 Related work

The first crucial feature of QVT-R enforce mode with which this paper is concerned is its true bidirectional nature. That is, the model which is computed as the result of a QVT-R transformation in enforce mode depends on two (or in general, more) models: the "source" model(s) and the existing "target" model. The second feature we emphasise is the possibility of recursion, in which one relation may invoke another, or even itself, on different or the same arguments. Both of these features are important to practical usability of QVT-R, as we argued more fully in [3] for recursion and in [11] for bidirectionality.

Most previous work on model transformations, and to our knowledge all previous work on QVT-R enforce mode, does not address either of these features. Rather, it formalises only the special case in which the original target model is empty, so that the QVT-R transformation simply produces the target model from the source model (we will call this "unidirectional QVT-R"); and it assumes that the *when-where* graph, the "call graph" of QVT-R relations, is acyclic ("acyclic QVT-R").

Triple graph grammars [10] do have this property of modifying an existing target model in the light of a source model. This works, roughly, by jointly parsing the input graph models, making use of (or constructing) a *correspondence*

*graph* that connects them, so that a hypothetical source model consistent with the current target model, together with a sequence of rules that would lead from the hypothetical to the current source model, is known. Then this sequence of rules is applied to the current target model to give the result target model. By placing careful restrictions, it is possible to ensure that the process succeeds and gives a uniquely defined result [7]. Greenyer and Kindler [6] gave a translation from QVT Core to Triple Graph Grammars (TGGs), and informally discuss trying to extend the approach to QVT-R, although they do not provide a semantics for QVT-R. Unfortunately, as discussed at length in [12], the claim in [8] that QVT-R can be translated in a semantics-preserving way to QVT Core is not sustainable (QVT Core is insufficiently expressive), so this approach does not lead to a semantics for QVT-R. The same problem may apply to [4] which a semantics for QVT-R enforce mode (with both unidirectional and acyclic restrictions) using coloured petri nets; it states that it is consistent with [8]'s incorrect translation to QVT Core. Another paper in this tradition is [14] which discusses using CPN theory as implemented in a model transformation framework called TROPIC to provide debugging facilities for QVT-R transformations. Again, the paper addresses only the unidirectional use of QVT-R.

Romeikat and others [9] translated QVT-R transformations, restricted to the acyclic case, to QVT Operational. The focus is on unidirectional use of QVT-R. The bidirectional case is discussed briefly, but not in detail; it seems to be considered only where all model elements are identified by key expressions. A different approach is to give semantics to QVT-R using algebraic specification, as exemplified by [1], which describes the MOMENT-QVT tool. This work, too, addressed only the unidirectional use of QVT-R. Recursive relation invocations are not discussed and do not seem to be allowed for.

The two extant tools addressing QVT-R are Medini and ModelMorf. As discussed in [12], Medini deliberately departs from [8]. ModelMorf is thus the most reliable (although not infallible) implementation and the one we compare our semantics with.

## 3 Preliminaries

### 3.1 QVT-R

A transformation $T$ is defined over a finite set of (usually two) *metamodels* (types for the input models) and, when executed in enforce mode, can be thought of as a function from tuples of models, each conforming to the appropriate metamodel, to an updated model (or failure). In any execution there is a *direction*, that is, a distinguished model which is being checked/enforced. The argument models are also known as *domains* and we will be discussing transformation execution in the direction of the $k$th domain. That is, the $k$th argument model is being checked/enforced against the others. See [12] for further discussion; here we assume some familiarity with QVT-R.

We use the notions of variables, values, typing, bindings and expressions. In QVT-R these matters are prescribed, building on the MOF metamodelling dis-

cipline and OCL. The available types are the metaclasses from any of the metamodels, together with a set of base types (defined in OCL) such as booleans, strings and integers, and collections. Values are instances of these. The expression language is an extension of OCL over the metamodels. QVT-R is a typed language, with some type inference expected.

In this paper, as in the previous work, we do not depend on QVT-R's particular choices in these matters, but provide a framework applicable to any similar language. We assume given sets *Var* of typed variables, *Val* of values and *Expr* of typed expressions over variables. We write $fv(e)$ for the set of free variables in $e \in Expr$. *Constraint* is the subset of *Expr* consisting of expressions of type Boolean. A (partial) set of *bindings* $B$ for a set $V \subseteq Var$ of variables will be a (partial) function $B : V \rightharpoonup Val$ satisfying the typing discipline. We write $B' \succeq B$ when $dom(B') \supseteq dom(B)$ and $B'$ and $B$ agree on $dom(B)$. We assume given an evaluation partial function $eval : Expr \times Binding \rightharpoonup Val$ defined on any $(e, b)$ where $fv(e) \subseteq dom(b)$. Like [8] we will assume all transformations we consider are statically well-typed.

A transformation $T$ is structured as a finite set of *relations* $R_1 \ldots R_n$, one or more of which are designated as *top* relations. A QVT-R relation is not (just) a mathematical relation – it consists of: a unique name; for each domain a typed *domain variable* and a *pattern*; and optional *when* and *where* clauses. We allow *when* or *where* clauses to contain arbitrary boolean combinations of *relation invocations* and boolean constraints (from *Constraint*). A relation invocation consists of the name of a relation together with an ordered list of argument expressions. Evaluating these expressions yields values for the domain variables of the invoked relation. We write $rel(T)$ for the set of names of relations in $T$ and $top(T) \subseteq rel(T)$ for the names of relations designated top. A pattern is a set of typed variables together with a constraint ("domain-local constraint") over these variables and the domain variable. A variable may occur in more than one pattern, provided that its type is the same in all.

The set of all variables used (in QVT-R declarations can be implicit) in a relation $R$ will be denoted $vars(R)$. The subset of $vars(R)$ mentioned in the *when* clause of $R$ is denoted $whenvars(R)$. The subset mentioned in the domains other than the $k$th domain is denoted $nonkvars(R)$. The set containing the domain variables is denoted $domainvars(R)$. These subsets of $vars(R)$ may overlap.

### 3.2 Game/mu-calculus semantics for QVT-R

In [3], we gave our semantics both in terms of a game and in terms of a modal mu-calculus formula, the two presentations being equivalent. Although the game version is easier to understand, the logical version is more concise and easier to adapt; so for reasons of space, we here give details only in the logical form.

The meta-logic for our semantics is modal mu-calculus. We refer to [3] for a fuller explanation of the logic and its relation to the game. Here we recap briefly the key points. The structures for the logic are transition systems – i.e. edge-labelled graphs – and formulae are true or false at states (nodes) in the systems. The formula $[a]\,\phi$ is true at $s$ iff $\phi$ is true at every state reached from

$s$ by a single $a$-transition ('$a$-successor'); $\langle a \rangle \phi$ is true iff $\phi$ is true at some $a$-successor. The greatest and least fixpoints $\nu Z.\phi(Z)$ and $\mu Z.\phi(Z)$ are formally co-inductive and inductive definitions, but are best understood as allowing the specification of looping behaviour – infinite loops for greatest fixpoints, and finite (but unbounded) loops for least fixpoints. Establishing a formula corresponds to constructing a winning strategy in the game for Verifier where she chooses at $\vee$ and $\langle\rangle$, and Refuter chooses at $\wedge$ and []. See [2] for a detailed explanation of the relation between modal mu-calculus, parity automata, and parity games.

Our semantics translates a QVT-R checkonly transformation instance into a modal mu calculus model-checking instance. Again, we refer to [3] for a full explanation. The key points are that we build a transition system encoding all the non-logical information about the models and the transformation, and we build a formula encoding the purely logical aspects.

Apart from a distinguished initial node, nodes of the transition system we construct each consist of a pair $(R, B)$ where $R \in \mathit{rel}(T)$ and $B : \mathit{vars}(R) \rightharpoonup \mathit{Val}$ is a set of (well-typed, as always) bindings. In order to be able to handle cases where the same relation may be invoked more than once in the *when* or *where* clause of another relation, we begin by labelling each relation invocation in the static transformation text with a natural number, so that an invocation $R(e_1, \ldots, e_n)$ is replaced by $R^i(e_1, \ldots, e_n)$ for an $i$ unique within the transformation; invoking the relation at invocation $i$ will be modelled by a transition labelled $\mathrm{invoke}_i$. Figure 1 defines the LTS formally. Note that the direction parameter $k$ affects the meaning of *nonkvars*.

The boolean flag is needed to handle negation, and in particular the negation implicit in *when* clauses. When the flag is *true*, the players have their usual roles; when the flag is *false*, they swap turns, so that Verifier handles [] and so on.

The mu calculus formula does not represent the domain variables, the patterns or the arguments to the relation invocations: all that information is represented in the transition system, and the $\mathrm{invoke}_i$ transitions and modalities connect the LTS and formula appropriately. Figure 1 defines the translation process formally.

Note that *tr2* is used to translate *when* and *where* clauses, building an environment that maps relations to mu variables in the process. Relation invocations are translated using the environment if the relation has been seen before, and otherwise, using a new fixpoint.

As discussed in [3], the possibility of recursive relation invocations in *when* clauses leads to potential undefined results. We adopt the well-formedness requirement defined and justified in [3], that there must be an even number of negations and *when*s between two invocations of a relation.

## 4 Enforcement

The description of enforcement semantics in [8], as with its description of checkonly semantics, does not address how to treat relations that are called in the *when* and *where* clauses of other relations. The "formal" semantics in Annex B

**Input:** Transformation $T$ defined over metamodels $M_i$, models $m_i : M_i$, direction $k$.
**Output:** Labelled transition system $lts(T, m_i, k) = (Initial, A, S, \longrightarrow)$
**Nodes:**
$S = \{Initial\} \cup \{(R, B) : R \in rel(T), B : vars(R) \rightharpoonup Val\}$
**Labels:**
$A = \{\text{challenge}, \text{response}, \text{ext1}, \text{ext2}\} \cup \{\text{invoke}_i : i \in \mathbb{N}\}$
**Transitions:**

| | | |
|---|---|---|
| Initial | $\overset{\text{challenge}}{\longrightarrow}$ $(R, B)$ | if $R \in top(T)$ and $dom(B) = whenvars(R) \cup nonkvars(R)$ |
| $(R, B)$ | $\overset{\text{response}}{\longrightarrow}$ $(R, B')$ | if $dom(B) = whenvars(R) \cup nonkvars(R)$ and $B' \succeq B$ and $dom(B') = vars(R)$ |
| $(R, B)$ | $\overset{\text{ext1}}{\longrightarrow}$ $(R, B')$ | if $dom(B) = domainvars(R)$ and $B' \succeq B$ and $dom(B') = domainvars(R) \cup whenvars(R) \cup nonkvars(R)$ |
| $(R, B)$ | $\overset{\text{ext2}}{\longrightarrow}$ $(R, B')$ | if $dom(B) = domainvars(R) \cup whenvars(R) \cup nonkvars(R)$ and $B' \succeq B$ and $dom(B') = vars(R)$ |
| $(R, B)$ | $\overset{\text{invoke}_j}{\longrightarrow}$ $(S, B')$ | if $S$ is invoked at the invocation labelled $j$ in the *where* clause of $R$ with arguments $e_i$, $dom(B) = vars(R)$ and $dom(B') = domainvars(S)$ with $\forall i \in domainvars(S).B' : v_i \mapsto eval(e_i, B)$ |
| $(R, B)$ | $\overset{\text{invoke}_j}{\longrightarrow}$ $(S, B')$ | if $S$ is invoked at the invocation labelled $j$ in the *when* clause of $R$, with arguments $e_i$, $dom(B) \supseteq whenvars(R)$ and $dom(B') = domainvars(S)$ with $\forall i \in domainvars(S).B' : v_i \mapsto eval(e_i, B)$ |

**LTS definition**

**Input:** Transformation $T$. **Output:** $tr(T)$ given by:

| | | |
|---|---|---|
| $tr(T)$ | $= \bigwedge_{R_i \in top(T)} tr1(R_i)$ | |
| $tr1(R_i)$ | $= [\text{challenge}]\,(\langle\text{response}\rangle(tr2_\emptyset(where(R_i), true) \vee tr2_\emptyset(when(R_i), false))$ | |
| $tr2_E(\phi, true)$ | $= \phi$ | |
| $tr2_E(\phi, false)$ | $= \neg\phi$ | |
| $tr2_E(e \text{ and } e', true)$ | $= tr2_E(e, true) \wedge tr2_E(e', true)$ | |
| $tr2_E(e \text{ and } e', false)$ | $= tr2_E(e, false) \vee tr2_E(e', false)$ | |
| $tr2_E(e \text{ or } e', true)$ | $= tr2_E(e, true) \vee tr2_E(e', true)$ | |
| $tr2_E(e \text{ or } e', false)$ | $= tr2_E(e, false) \wedge tr2_E(e', false)$ | |
| $tr2_E(\text{not } e, b)$ | $= tr2_E(e, \neg b)$ | |
| $tr2_E(R^i(e_1 \ldots e_n), true)$ | $= \langle\text{invoke}_i\rangle E[R]$ | if $R \in \text{dom}E$ |
| $tr2_E(R^i(e_1 \ldots e_n), true)$ | $= \langle\text{invoke}_i\rangle \nu X.\,([\text{ext1}]\,(\langle\text{ext2}\rangle tr2_{E[R\mapsto X]}(where(R), true) \vee tr2_{E[R\mapsto X]}(when(R), false))$ | otherwise |
| $tr2_E(R^i(e_1 \ldots e_n), false)$ | $= [\text{invoke}_i]\,(\neg E[R])$ | if $R \in \text{dom}E$ |
| $tr2_E(R^i(e_1 \ldots e_n), false)$ | $= [\text{invoke}_i]\,\mu X.\,(\langle\text{ext1}\rangle\,([\text{ext2}]\,tr2_{E[R\mapsto \neg X]}(where(R), false) \wedge tr2_{E[R\mapsto \neg X]}(when(R), true))$ | otherwise |

**Mu calculus formula definition**

**Fig. 1.** Definition of the checkonly translation

uses a predicate logic formula, although it is actually understood as an imperative program. The only way to interpret this, is that relation invocations other than at top level are treated as pure predicates. Consequently, object creation or update happens only in top level relation calls. As we saw in the introduction, this leads to the creation of many new objects in top level bindings, where a smaller change could be achieved by recursively enforcing the lower level relation. In this paper we present only semantics following the approach of [8].

The [8, Annex B] enforcement specification breaks into two steps: first, create (or modify) any objects in the target required to satisfy the transformation; secondly, delete certain objects in the target not required to exist by the transformation. We use the same phases.

### 4.1 Extending the QVT game/logic for enforcement

Determining that two models are consistent, in our semantics, amounts to finding a winning strategy for Verifier, or alternatively establishing the truth of a formula expressed in mu-calculus. To play an enforcement game, we need to give Verifier additional moves: if she is unable to win the checkonly game at a certain point, she has the option to change the model and try again. In mu-calculus terms, this amounts to adding disjunctions at appropriate places in the formula, with formulae involving a model-changing transition.

The model-changing is encoded thus: the states of the transition system in Fig. 1 are extended to be of the form $(Initial, M)$ or $(R, B, M)$ so that they carry the (entire) target model $M$ as part of the state, and the transitions defined there leave it untouched. For technical reasons, the models $M$ also include a 'modification record' for each model element, saying whether a given property has been changed.

### 4.2 The object creation/update phase

The first extension to our previous semantics is to force the re-start of checking/enforcement after a model update. While Annex B does not discuss this, it is obvious that after a model update, all top relations may need to be checked again. (Of course a tool might optimise.) Therefore we wrap the entire top level formula in a least fixpoint, so that the first line of Fig. 1 'formula definition' is changed to

$$tr(T) = \mu W. \bigwedge_{R_i \in top(T)} tr1(R_i)$$

where the variable $W$ will appear later in the translation, and the fixpoint has to be minimal because for enforcement to succeed, only a finite number of updates can be done.

Per Annex B, object creation (or update) occurs if, after source and *when* bindings are chosen, there is no binding to the target variables that satisfies the domain pattern. In our game, this occurs at the point following a challenge transition taken by Refuter. If Verifier is unable to choose bindings that let her

win, or to win by challenging the *when* clause, she has the possibility to update the model, so line 2 of Fig. 1 'formula definition' becomes

$$\begin{aligned} tr1(R_i) = [\text{challenge}]\,(&\langle\text{response}\rangle(tr2_\emptyset(where(R_i), true)) \\ &\vee tr2_\emptyset(when(R_i), false) \\ &\vee \langle\text{update}\rangle W) \end{aligned}$$

and the update transitions are defined by

$$(R, B, M) \xrightarrow{\text{update}} (\text{Initial}, M') \quad \text{if (*)}$$

where the side-condition property (*) must capture when changing the model to $M'$ is legal. (*) depends on whether the $k$th domain variable, say `me:ME`, is already in $dom(B)$ (which will be the case if $\texttt{me} \in whenvars(R)$), and whether a key constraint is specified for `ME`. If $\texttt{me} \notin dom(B)$ and `ME` has no key constraint, then $M'$ is a model formed from $M$ by creating a new object, say $o$, of type `ME`, with properties set according to the $k$th domain pattern of $R$ with bindings from $B$.

However, the domain pattern does not (usually) specify all properties of $o$ and, for enforcement to succeed, properties that are not specified may nevertheless matter, as they may have to take values which will cause a *where* clause to succeed – note that the actual point in the transformation at which the values of these properties of $o$ matter could be arbitrarily many invocations away from $R$. [8] does not specify how such properties are to be set, but a useful tool must find correct values as often as possible (not "if they exist", because it is clear that the problem in general is noncomputable, given a constraint language as powerful as OCL). In our semantics, we model an update transition for *every* legal choice of the properties. A choice is legal if it obeys the metamodel and domain pattern (including domain-local constraints). Note that our transition system already contained infinite branching because of the potentially infinite choices for bindings; we will shortly discuss how a transformation engine could ensure determinacy by searching systematically for a winning strategy (and thus always finding the same one, even if there are many).

If a new object $o$ cannot be created because `me` was already bound (say to $o'$) in $B$, then (*) must permit no update transition unless there is a key specification for `ME`, because only then will it be legal to modify properties of $o'$.

Now consider the case where there is a key specification for `ME` such that the bindings in $B$ determine an object (say $o'$) in $M$ (regardless of whether this is because $B$ includes a binding of `me` to $o'$ itself, or because it includes bindings for key properties that determine $o'$). Then (*) is adjusted to make $M'$ the result of modifying properties of $o'$, that have not already been marked 'modified' to a different value, in any way which is valid according to the metamodel, domain pattern, and domain local constraint as before, and setting the modification flag on the object's modified properties. (The reason for the modification flag is that enforcement is required to fail if an object is modified in inconsistent ways.)

Corresponding moves are added to the game presentation, allowing Verifier a move modifying the target model if she cannot win by either providing bindings from the current model or challenging the when clause. (Details are elided

as, although simple, they do become long-winded: if Verifier tries to choose an update move when in fact she could have won by one of the other move types – that is, if she makes an unnecessary change to the model – we need to let Refuter win by demonstrating that her update was unnecessary. This corresponds to the inverting of Boolean flags in the short circuit evaluation version of the mu formula below.)

A logical formula has no order of evaluation built in, whereas the imperative interpretation of Annex B does. It is possible to impose an order of evaluation externally upon the model-checker; it is also possible to modify the formula to simulate it. This makes no difference to whether our enforcement formula succeeds, but it does affect the model that results from evaluating whether enforcement succeeds. If our formula is to do updates only when needed, we can simulate short-circuit evaluation by modifying the new formula thus:

$$\begin{aligned} tr1(R_i) = [\text{challenge}]\,((&\langle\text{response}\rangle(tr2_\emptyset(\mathit{where}(R_i), \mathit{true}))) \\ &\vee tr2_\emptyset(\mathit{when}(R_i), \mathit{false}) \\ &\vee(([\text{response}]\; tr2_\emptyset(\mathit{where}(R_i), \mathit{false})) \\ &\quad \wedge tr2_\emptyset(\mathit{when}(R_i), \mathit{true}) \wedge \langle\text{update}\rangle W)) \end{aligned}$$

so that Verifier can only successfully choose the update branch if the other branches fail. (Here we use [3, Lemma 1].)

There is one issue that is best dealt with by the evaluation/model-checking procedure, rather than in the formula or game. There is, in general, nothing to stop Verifier from making unnecessary updates, by choosing an update transition that does not in fact satisfy the *where* clause; she will then re-start, and update again. With sufficient additional book-keeping in the model and formula, this could be avoided; however, it is simpler to invoke the notions of 'canonical tableau' or 'optimal progress measures' from the theory of mu-calculus model-checking, so that the model-checker constructs the strategy with the smallest number of updates. (So-called bottom-up model-checkers do this automatically; top-down model-checkers do not. See [2].)

At this point, our semantics matches that of Annex B after the create phase. Showing that the formula is true amounts to constructing a winning strategy for Verifier in the game, and the constructed model is extracted by examining the update transitions in the winning strategy (or, in practice, by examining the trace of updates taken during construction).

Enforcement may involve creating and choosing properties for many new objects. It is also possible that an object created by one update may be used (e.g. with modification) by a subsequent update. It is therefore possible in general that the constructed model depends on the order of transition choices in modalities, which in turn depends on the order in which source bindings are checked – in our terms, on the order in which Refuter makes choices when challenging. While this is also the case in Annex B, it is desirable that an enforcement algorithm should be deterministic, which requires fixing the order of all choices. It is not feasible or sensible to encode this into the logic; rather it is appropriately done by specifying how the mu-calculus model-checking algorithm proceeds, or equivalently how the

construction of a winning strategy proceeds. While all real model-checkers make such choices, they do not normally expose them; if the result of enforcement is to be unique, the choices must be explicit. For example, we might specify that formulae are checked left to right, and that when choosing a transition, the possible transitions are ordered according to their internal representations and the lowest in the order is taken.

### 4.3 The deletion phase

Here we need to be cautious because a literal reading of [8] (sections 7.10.2 and Annex B) gives behaviour that is clearly undesirable and contradicts ModelMorf's behaviour. [8] specifies that certain elements of the target model, that constitute valid bindings of domain $k$ variables, are deleted if they are not 'required to exist' by the relation. For example, if model m1 contains an `E1type` with name 'foo', and m2 contains no `E2type` called 'foo', then enforcing the relation

```
top relation Zap {
  n : String;
  domain m1 e1:E1type { name=n };
  domain m2 e2:E2type { name=n };
  when { n = 'foo'; } }
```

on `m2` would, in addition to creating a new `E2type` called 'foo', delete all `E2type` elements in the old `m2`, because they are not required to exist by the relation. It is hard to believe that this is the desired result, and indeed ModelMorf does not do this (it sensibly creates a new 'foo' `E2type`, leaving other `E2type`s alone).

To see what is probably intended, consider the same relation without the *when* clause, which ought to embody a more stringent consistency check:

```
top relation Matchname {
  n : String;
  domain m1 e1:E1type { name=n };
  domain m2 e2:E2type { name=n };}
```

Bidirectionally, this says every `E1type` element in one domain has a (not necessarily unique) matching-named element in the other. Suppose we enforce on `m2`, and there is an element `e` called `'foo'` in `m2` with no match in `m1`. We expect it to be deleted. [8] will delete it because it is 'not required'. However, the real reason for deleting is surely that it fails the relation *in the direction* `m1`; it is not that `e` is not required, it is that its *absence* is required for `Matchname` to check in the `m1` direction (as we are not allowed to create an element in `m1`).

Deciding that we should delete objects whose absence is required, we can implement the deletion phase more easily: for each checked (source) domain $j$ (e.g. m1 above), we set up the transition system and formula for checking in the direction of the $j$th domain, and then modify the formula exactly as before, but replacing update by delete, and add transitions

$$(R, B, M) \xrightarrow{\text{delete}} (\text{Initial}, M') \quad \text{if } (**)$$

where property (**) is: $M'$ is the model formed from $M$ by deleting the object bound in $B$ to the top level domain variable of the $k$th domain (which is now a source domain for the checkonly formula). In the typical case of two domains, we only need run the delete step once; where there are $k > 2$ checked domains and we are enforcing on the $k$th, we must run the deletion step for each of the domains $j = 1, \ldots, k-1$.

If this formula evaluates to true, the resulting model is read off from the winning strategy as before.

### 4.4 Putting it together

Enforcement now amounts to first evaluating the creation formula; if it is true, read off the model and evaluate the deletion formula. If that is true, read off the model. If either stage fails, then enforcement fails, either because there is no way to restore consistency, or because the simple-minded update strategy is not powerful enough to do so.

There is one further check needed: the resulting model must be checked (in checkonly mode) *ab initio*, as it is possible that inconsistency in the transformation arises from the combination of creation and deletion – for example, an object might be required to exist by checking in direction $k$, but required not to exist by checking in the direction $k'$.

This procedure *can* be coded up to produce a single giant transition system and formula, but the process is not enlightening, and does not simplify correctness.

## 5 Example

Consider the example `NonTopEnforce` with $\texttt{m1} = \{N \to \{child1, child2, child3\}\}$, from Section 1, enforcing against an empty model `m2`. We demonstrate a "best" winning strategy (canonical tableau) for Verifier: as discussed this avoids unnessary updates. Refuter will challenge in `R` by binding $\{n \mapsto$ 'N', $firstchild \mapsto c1\}$ where c1 is one of the three children, say the one with `name` 'child1'. (In mu calculus terms, we pass through $[challenge]$ and along a challenge transition to a game position with this binding.) Verifier will be unable to find matching bindings and there is no *when* clause so she will update the model. (In mu calculus terms, the first two disjuncts of the short-circuiting formula are false so model checking proceeds to the $\langle update \rangle$ disjunct.) There are no key expressions and `me2` is not bound, so Verifier creates a new `MEtop` element and sets its properties according to the domain pattern: that is, its `name` will be 'N'. She must also create a `MEchild` element to bind to `secondchild` but is not constrained as to its properties: all of the infinitely many choices are legal game moves and correspond to infinitely many update transitions each to a different modified model. Playing a best strategy, however, she creates an `MEchild` with name 'child1'. Play now restarts at the initial position with the modified model. If Refuter were to make the same challenge this time, he would lose because

Verifier now has bindings with which to match. If, instead, he challenges with one of the other children, say with binding $\{n \mapsto$ 'N', $firstchild \mapsto c2\}$ where $c2$'s `name` is 'child2', exactly the same thing will happen. Although Verifier could this time choose matching bindings, she could not *win* by doing so (in mu calculus terms, although $\langle \text{response} \rangle \mathtt{tt}$ is true, $\langle \text{response} \rangle (tr2_{\emptyset}(where(R_i), true))$ is false) and so she will choose to create a new `MEtop` element, again with name 'N', and a new `MEchild` to be the value of its `child` property and to bind to `secondchild`. Repeating once more we find that the result of the create phase of the game (playing with a best strategy so that no junk has been created) is $\{N \to \{child1\}, \{N \to \{child2\}, \{N \to \{child3\}\}$. (In mu calculus terms we have unwound the fixpoint $W$ three times, once for each update; no fewer unwindings would lead to success.) The checkonly game in direction `m1` is won by Verifier already so no deletions are required.

## 6 Properties of transformations

In [11] we formalised properties that, we argued, should hold of bidirectional transformations; other work in this direction includes [5, 13]. Now that we have formal semantics for QVT-R in both checkonly and enforce mode, it makes sense to ask whether it has properties of interest. Recall that a bidirectional transformation $R : M \leftrightarrow N$ can be modelled by a triple $R \subseteq M \times N$ (slight abuse of notation), $\overrightarrow{R} : M \times N \to N$, $\overleftarrow{R} : M \times N \to M$. In QVT-R, $R(m, n)$ should be true iff $R$ returns true when run in checkonly mode (either direction) on models $m$ and $n$, while $\overrightarrow{R}(m, n)$ returns $n'$ if $R$ run in enforce mode with source $m$ and target $n$, i.e. in the direction of $n$, modifies $n$ to $n'$. Because enforcement is not guaranteed to succeed in our setting (whether because of inconsistency or uncomputability), we must modify the framework to make $\overrightarrow{R}$ and $\overleftarrow{R}$ partial functions, which may return $\perp$.

Relatively uncontroversial properties are (partial) *correctness* and *hippocraticness*. We prove that these both hold of every well-defined QVT-R transformation, interpreted according to our checkonly and enforce semantics.

**Theorem 1.** *Given our semantics, QVT-R is (partially) correct: that is, for any well-defined transformation $R$ and models $m$ and $n$, if $\overrightarrow{R}(m, n) \neq \perp$, then $R(m, \overrightarrow{R}(m, n))$, and dually.*

*Proof.* By construction, our enforcement semantics ensures that any transformation execution finishes with a full checkonly evaluation, and fails if this is not satisfied.

**Theorem 2.** *Given our semantics, QVT-R is hippocratic: that is, for any well-defined transformation $R$ and models $m$ and $n$, $R(m, n) \Rightarrow \overrightarrow{R}(m, n) = n$ and dually.*

*Proof.* Our enforcement semantics is the evaluation of a formula which (via the simulation of short-circuit evaluation) only proceeds into a branch containing

model-changing transitions if the models do not already satisfy the checkonly formula (proved correct in [3]).

Undoability, the third property discussed in [11], is more problematic. As is now well-understood, although theoretically desirable because it gives good algebraic properties, it is in practice too strong. It is straightforward to construct an example in which both our semantics and ModelMorf fail undoability: deleting, and then recreating, a piece of information on one side results in the loss of anything on the other side that was "stuck" to the deleted and recreated information. Thus we cannot expect QVT-R transformations to be undoable.

## 7 Conclusions and future work

By giving formal semantics to QVT-R enforce mode, we have clarified issues, particularly with object deletion, in the standard, and we have brought this hard problem into the much studied and well understood domain of model-checking. Our semantics relies on model-checking algorithms that can compute the canonical tableau or best winning strategy; in the case of finite models, this is routine, and in the case of infinite models there is an extensive body of work on algorithms for well-behaved families of infinite models. Our semantics, like our earlier checkonly semantics [3], has two equivalent formulations. We translate an enforce problem into a modified mu calculus model checking problem – the model checking process computes the changes needed to a model, and these changes are then verified by model checking. This presentation is convenient for proofs, because it enables us to exploit properties of mu calculus. The alternative, equivalent formulation, in terms of simple two-player games, is more convenient for direct use.

We hope that our semantics work may help to inform designers of future bidirectional languages. One lesson, we suggest, is that while the syntax of QVT-R is appealing in the intuitiveness of individual relations, the way in which QVT-R connects relations is probably not optimal.

In future, aiming to define a QVT-R-like language with semantics that better support MDD, we will consider semantic variations, including those (such as the bisimulation-like game [12], and reducing the special treatment of top relations [3]) that we considered in previous work on checkonly but for space reasons have not explored here. Variations specific to enforce mode include specifying deterministic update transitions, making use of information gathered in previous steps of the evaluation. More fundamentally we will explore allowing updates in recursive relation invocations, corresponding to on-the-fly enforcement of lower level relations. The present semantics, like [8], ensures that each change to the target model, considered locally, is necessary, and also that the minimum number of distinct updates is done. However, as we have seen, updates can be larger than necessary. Understanding what precise senses of minimal change are desirable, achievable and supportable by tools is a challenging and interesting problem.

**Acknowledgements**

We thank the referees for their constructive suggestions, including some that could not be addressed in this version for space reasons. The first author is partly supported by UK EPSRC grant EP/G012962/1 'Solving Parity Games and Mu-Calculi'.

## References

1. Artur Boronat, José A. Carsí, and Isidro Ramos. Algebraic specification of a model transformation engine. In *Proc. FASE 2006*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.
2. J. C. Bradfield and C. Stirling. Modal mu-calculi. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, volume 3, pages 721–756. Elsevier, 2007.
3. Julian C. Bradfield and Perdita Stevens. Recursive checkonly QVT-R transformations with general when and where clauses via the modal mu calculus. In *Proc. FASE 2012*, volume 7212 of *LNCS*, pages 194–208. Springer, 2012.
4. Juan de Lara and Esther Guerra. Formal support for QVT-relations with coloured petri nets. In Andy Schürr and Bran Selic, editors, *Proc. MODELS'09*, volume 5795 of *LNCS*, pages 256–270. Springer, 2009.
5. Zinovy Diskin. Algebraic models for bidirectional model synchronization. In *Proc. MODELS 2008*, volume 5301 of *LNCS*, pages 21–36. Springer, 2008.
6. Joel Greenyer and Ekkart Kindler. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.
7. Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of model synchronization based on triple graph grammars. In *Proc. MODELS 2011*, volume 6981 of *LNCS*, pages 668–682. Springer, 2011.
8. OMG. MOF2.0 query/view/transformation (QVT) version 1.1. OMG document formal/2009-12-05, 2009. available from www.omg.org.
9. Raphael Romeikat, Stephan Roser, Pascal Müllender, and Bernhard Bauer. Translation of QVT relations into QVT operational mappings. In *Proc. ICMT '08*, pages 137–151, Berlin, Heidelberg, 2008. Springer-Verlag.
10. Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *Proc. ICGT*, volume 5214 of *LNCS*, pages 411–425. Springer, 2008.
11. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Journal of Software and Systems Modeling (SoSyM)*, 9(1):7–20, 2010.
12. Perdita Stevens. A simple game-theoretic approach to checkonly QVT Relations. *Journal of Software and Systems Modeling (SoSyM)*, 2011. Published online, 16 March 2011.
13. Perdita Stevens. Observations relating to the equivalences induced on model sets by bidirectional transformations. *EC-EASST*, 049, 2012.
14. Manuel Wimmer, Angelika Kusel, Johannes Schoenboeck, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reviving QVT relations: Model-based debugging using colored petri nets. In *Proc. MODELS '09*, pages 727–732. Springer, 2009.