Recursive checkonly QVT-R transformations with general *when* and *where* clauses via the modal mu calculus

Julian Bradfield and Perdita Stevens

School of Informatics
University of Edinburgh

Abstract. In earlier work we gave a game-based semantics for checkonly QVT-R transformations. We restricted *when* and *where* clauses to be conjunctions of relation invocations only, and like the OMG standard, we did not consider cases in which a relation might (directly or indirectly) invoke itself recursively. In this paper we show how to interpret checkonly QVT-R – or any future model transformation language structured similarly – in the modal mu calculus and use its well-understood model-checking game to lift these restrictions. The interpretation via fixpoints gives a principled argument for assigning semantics to recursive transformations. We demonstrate that a particular class of recursive transformations must be ruled out due to monotonicity considerations. We demonstrate and justify a corresponding extension to the rules of the QVT-R game.

1 Introduction

QVT-R is the OMG standard *bidirectional* model transformation language[6]. 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 no changes are required (checkonly mode) or as specifying exactly how one model should be modified so as to restore consistency that has been lost (enforce mode). This paper concerns checkonly mode, a thorough understanding of which 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.

QVT-R has several interesting features. In particular, the fundamental way in which a QVT-R transformation is structured, using a collection of so-called *relations* connected by *when* and *where* clauses is attractive as it appears to enable the transformation to be understood by the developer in a modular way. This transformation structuring mechanism might reasonably be applied in future bidirectional model transformation languages, so it is of interest even if QVT-R itself is not ultimately successful.

In earlier work [7] the second author provided a game-theoretical semantics for its use in "checkonly" mode, that is, as a logic for defining predicates on pairs of models. Given a QVT-R checkonly problem instance (a transformation, together with a tuple of models to check in a given direction), we defined a formal game between two players, Verifier and Refuter, such that Verifier had a winning strategy for the game if and only if the transformation should return *true* on the given tuple of models in the stated direction. We justified the correctness of the semantics defined in this way, by referring both to [6] and to the behaviour of the most faithful QVT-R tool, ModelMorf. In that work, we did not define which player would win an infinite play of the game. Instead, we placed a restriction on the permitted transformations such that all plays of the games in our semantics would be finite; we justified this by pointing out that the OMG semantics [6] implied nothing about what the semantics in the infinite play cases should be, but we remarked that it should be possible to do better "by intriguing analogy with the modal mu calculus". Intuitively the analogy is that the interplay of *when* and *where* clauses mixes induction with coinduction; the essential character of the mu calculus is that it does the same. In this paper, we make the analogy concrete; this allows us to give semantics to many recursive QVT-R transformations, and allows us to explain why considerations of monotonicity force other recursive transformations to remain forbidden. We also use mu calculus theory to prove that extra levels of nesting of *when* and *where* clauses provide genuine extra expressivity.

When defining the semantics of QVT-R via a translation to the mu calculus, it is natural also to permit more general *when* and *where* clauses than previous work has done. The translation is an aid to clear thought, only: having made it, we extend our earlier QVT-R game so that all the transformations we can translate can also be given semantics directly by this easy-to-understand game.

Both recursion and complex clauses are useful in practice, especially where metamodels contain loops of associations; indeed, both are used in the example in [6], even though it does not give semantics of recursion.

Related work Our earlier paper [7] discusses the field of previous work on semantics for checkonly QVT-R in full. As discussed there, very few authors have interested themselves in QVT-R *as a bidirectional language.* The majority approach is to study QVT-R transformations in enforce mode only, and furthermore with the restriction that the transformation function does not take a version of the target model, only source models. The target model produced depends only on the source model and the transformation. Recursive relations typically give rise to recursion (possibly with non-termination) in the target formalism, but this does not contribute to understanding recursion in checkonly QVT-R.

More relevantly, in [3] the authors aim to generate invariants in OCL, not in order to give a formal semantics for QVT-R but to support auxiliary analysis to increase confidence in a transformation's correctness. The paper includes an example of a complex recursive QVT-R relation (in Fig 6(a), relation `ChClass-Table` is given a *where* clause `Attribute-Column(c1,t) and ChClass-Table(c1,t)`). Unfortunately, as discussed in [7], key details of the

invariant generation are elided. Looking at the example, it appears that a recursive QVT-R relation will lead to a recursive OCL constraint. The problem is thereby moved into the OCL domain, where it is still problematic: [4] in fact forbids infinite recursion. [3] does not discuss this issue, and in particular, does not specify which QVT-R transformations can be translated without producing OCL whose meaning on the relevant models is undefined.

None of the existing QVT-R tools have documented behaviour on recursive checkonly QVT-R.

2 Background

2.1 QVT-R

A transformation T is defined over a finite set of (usually two) *metamodels* (types for the input models) and, when executed in checkonly mode, can be thought of as a function from tuples of models, each conforming to the appropriate metamodel, to booleans. In any execution there is a *direction*, that is, a distinguished model which is being checked. The argument models are also known as *domains* and we will be discussing transformation execution in the direction of the kth domain. That is, the kth argument model is being checked for consistency with the others. See [7] for further discussion; here we assume some familiarity with QVT-R.

Let us discuss preliminary matters of variables, values, typing, bindings and expressions. In QVT-R these matters are prescribed, building on the MOF metamodelling discipline 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.

Our work will focus on the structural aspects of the transformation and will turn out to be independent of QVT-R's particular choices in these matters. We assume given sets *Var* of typed variables, *Val* of values and *Expr* of typed expressions over variables. We write $\mathit{fv}(e)$ for the set of free variables in $e \in \mathit{Expr}$. *Constraint* is the subset of *Expr* consisting of expressions of type Boolean. A (partial) set of *bindings* B for a set $V \subseteq \mathit{Var}$ of variables will be a (partial) function $B : V \rightharpoonup \mathit{Val}$ satisfying the typing discipline. We write $B' \succeq B$ when $\mathit{dom}(B') \supseteq \mathit{dom}(B)$ and B' and B agree on $\mathit{dom}(B)$. We assume given an evaluation partial function $\mathit{eval} : \mathit{Expr} \times \mathit{Binding} \rightharpoonup \mathit{Val}$ defined on any (e, b) where $\mathit{fv}(e) \subseteq \mathit{dom}(b)$. Like [6] 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. We will use the term relation since it is that used in QVT-R, but readers should note that a QVT-R relation is not (just) a mathematical relation. Instead, a relation consists of: a unique name; for each domain a typed *domain variable* and a *pattern*; and optional *when* and

where clauses (to be discussed shortly). 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 kth domain is denoted $nonkvars(R)$. The set containing the domain variables is denoted $domainvars(R)$. These subsets of $vars(R)$ may overlap.

For purposes of this paper a *when* or *where* clause may contain a boolean combination of *relation invocations* and boolean constraints (from *Constraint*). Each 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. The BNF (non-minimal, as it will be convenient to have all of *and*, *or* and *not*) for *where* clauses is:

$$\begin{array}{lcl} where(R) & := & S(e_1, \dots e_n) \qquad \text{where } S \in rel(T),\ e_i \in Expr \text{ and } fv(e_i) \subseteq vars(R) \\ & | & where(R) \quad \text{and} \quad where(R) \quad | \quad where(R) \quad \text{or} \quad where(R) \\ & | & \text{not} \quad where(R) \quad | \quad (where(R)) \\ & | & \phi \qquad \text{such that } \phi \in Constraint \text{ and } fv(\phi) \subseteq vars(R) \end{array}$$

and the BNF for *when* is the same, substituting *when* for *where*, and *whenvars* for *vars*. The use of *whenvars* in the definition of $when(R)$ does not constrain what can be written; $v \in vars(R)$ is in $whenvars(R)$ precisely if it is used in the *when* clause. QVT-R itself uses semi-colon (in some contexts, and comma in others) for "and", but this seems unnecessarily confusing when we also want to allow other boolean connectives.

Figure 1 reproduces the moves from the game theoretic semantics of QVT-R checkonly. We refer the reader to [7] for full discussion and examples. The game G_k is played in the direction of domain k; that is, model k is being checked with respect to the other model(s).

Apart from the distinguished Initial position, positions in the game are all of the form (P, R, B, i) where: P is a player (Verifier or Refuter), indicating which player is to move from the position; R is the name of a relation from the transformation, the one in which play is currently taking place; B is a set of bindings whose domain will be specified; and i is either 1 or 2, tracking whether only one or both players have moved in the current relation. Play proceeds by the player whose turn it is to move choosing a legal move. If no legal move is available to this player, play ends and the other player wins ("you win if your opponent can't go"). The transformation returns *true* if Verifier has a winning strategy, that is, she can win however Refuter plays.

Informally, each play begins by Refuter picking a top relation to challenge and bindings for variables from the domains other than the kth and for any variables that occur in the *when* clause (Row 1). Verifier may respond by finding matching bindings from model k (Row 2) or she may counter-challenge a *when*

invocation (Row 3), effectively claiming that Refuter's request for her to find matching bindings is unreasonable because this top relation is not required to hold at his chosen bindings. If she opts to provide matching bindings, Refuter will attempt to challenge a *where* invocation (Row 4). Thus play proceeds through the transformation until one player cannot move; e.g., if Verifier successfully provides matching bindings and there is no *where* clause, it is Refuter's turn but he has no legal move, so Verifier wins the play.

Position	Next position	Notes
Initial	$(\mathsf{Verif.}, R, B, 1)$	$R \in top(T)$; $dom(B) = nonkvars(R) \cup whenvars(R)$. B is required to satisfy domain-local constraints on all domains other than k.
$(P, R, B, 1)$	$(\overline{P}, R, B', 2)$	$B' \succeq B$ and $dom(B') = vars(R)$. B' is required to satisfy domain-local constraints on all domains.
$(P, R, B, 1)$	$(\overline{P}, S, C, 1)$	$S(e_1 \dots e_n)$ is any relation invocation from the *when* clause of R; $\forall v_i \in domainvars(S).C : v_i \mapsto eval(e_i, B)$; $dom(C) = domainvars(S) \cup nonkvars(S) \cup whenvars(S)$. C is required to satisfy domain-local constraints on all domains other than k.
$(P, R, B, 2)$	$(\overline{P}, S, D, 1)$	$S(e_1 \dots e_n)$ is any relation invocation from the *where* clause of R; $\forall v_i \in domainvars(S).D : v_i \mapsto eval(e_i, B)$; $dom(D) = domainvars(S) \cup nonkvars(S) \cup whenvars(S)$. D is required to satisfy domain-local constraints on all domains other than k.

Fig. 1. Summary of the legal positions and moves of the game G_k over T: note that the first element of the Position says who picks the next move, and that we write $\overline{P}$ for the player other than P, i.e. $\overline{\mathsf{Refuter}} = \mathsf{Verifier}$ and vice versa. Recall that bindings are always required to be well-typed.

2.2 Modal mu calculus

The modal mu calculus [5] is a long-established and well-understood logic for specifying properties of systems, expressed as labelled transition systems. Besides the usual boolean connectives, it provides

- modal operators: $[a]\,\phi$ is true of a state s if whenever $s \xrightarrow{a} t$, ϕ is true of state t, while $\langle a \rangle \phi$ is true of a state s if there exists $s \xrightarrow{a} t$ such that ϕ is true of state t
- greatest and least fixpoints $\nu Z.\phi(Z)$ and $\mu Z.\phi(Z)$, which are formally co-inductive and inductive definitions, but which are best understood as allowing the specification of looping behaviour – infinite loops for greatest fixpoints, and finite (but unbounded) loops for least fixpoints. The combination of both fixpoints with the modal operators allows the expression of complex behaviours such as fairness.

Its semantics is most easily explained as a game between two players, Verifier and Refuter. A position, in the game to establish whether $(i, A, S, \longrightarrow)$ satisfies ϕ, is (ψ, s) where ψ is a subformula of ϕ and $s \in S$. The initial position is (ϕ, i). The top connective of ψ determines which player moves; Verifier moves if it is $\vee$ (she chooses a disjunct), $\langle a \rangle$ (she chooses an a-transition) or a maximal fixpoint or its variable (she unwinds the definition). Dually, Refuter moves otherwise. A player wins if it is their opponent's turn and the opponent has no legal move, e.g. Refuter wins if the position is $(\langle a \rangle \psi, s)$ and there is no a-transition out of state s. In an infinite play, the winner is the owner of the outermost variable unwound infinitely often (i.e. Verifier if that is a maximal fixpoint variable, otherwise Refuter).

One may think of the difference between ν and μ in terms of defaulting to true or false. In a (formal) sense, a μ formula is one where every positive claim has to be demonstrated; whereas a ν formula holds unless there is a demonstrated reason why not. See [1] for further explanation and background.

3 Connecting QVT-R and modal mu calculus

We will translate a QVT-R checkonly transformation instance into a modal mu calculus model-checking instance. That is, given a QVT-R transformation T, a tuple of models $(m_1, \dots m_n)$ and a direction k, we shall build a mu calculus formula $tr(T)$ and an LTS $lts(T, m_1, \dots m_n, k)$ such that $(m_1, \dots m_n)$ is consistent in the direction of the kth domain according to T iff $lts(T, m_1, \dots m_n, k)$ satisfies $tr(T)$. Note that the LTS depends on the transformation as well as the models; this is because we choose to encode as much as possible in the LTS, leaving only the essential recursive structure to be encoded in the mu calculus formula. In particular, the LTS will capture the features of the model tuple that matter, ignoring the features that are irrelevant to this particular transformation.

Having defined our translation, we prove that this result holds for the restricted class of transformations covered by the QVT-R game. This validates the translation on the set of problem instances where a formal semantics already existed, which makes it prima facie reasonable to use the translation as the semantics of QVT-R on the full domain where it makes sense (which, as we shall see, includes many but not all transformations with recursive *when* and *where* clauses). We then propose an extension to the QVT-R game, such that the game semantics and the mu calculus translation semantics coincide everywhere. We then discuss the implications of doing so; what semantics does it assign to transformations with complex *when* and *where* clauses and/or recursive *when*/*where* structure? We will point out one decision point where two choices are possible, giving different semantics to the transformation language.

3.1 The transition system

Apart from a distinguished initial node, nodes of the LTS we construct each consist of a pair (R, B) where $R \in rel(T)$ and $B : vars(R) \rightharpoonup Val$ 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 invoke_i. Figure 2 defines the LTS formally. Note that the direction parameter k affects the meaning of *nonkvars*.

3.2 The mu calculus formula

Mu calculus model checking is generally done on a version of the syntax that does not include negation. The reason is that, if negation is permitted in the language, the negation can be pushed inwards until it meets the fixpoint variables using the duality rules such as $\neg [a] \phi \equiv \langle a \rangle \neg \phi$. A formula in the mu calculus with negation is only semantically meaningful if doing this process results in all negations vanishing (using the rule $\neg\neg X \equiv X$); otherwise, the fixpoints are undefined. (Technically, it is possible for a particular formula with non-vanishing negations to be semantically meaningful, but this cannot in general be determined from the syntax.)

As mentioned in Section 2.2, the semantics of a standard mu calculus formula can be defined using a two-player model-checking game. If negation is left in the language, it corresponds to the players swapping roles, just as happens in the QVT-R game on a *when* invocation. Rather than define a version of the mu calculus game involving such player swapping, we will translate a QVT-R transformation into a mu calculus formula without negation. Our translation function will carry a boolean argument to indicate whether roles have been swapped an odd (*false*) or even (*true*) number of times.

The mu calculus formula does not represent the domain variables, the patterns or the arguments to the relation invocations, so we ignore these in our translation process: all that information is represented in the transition system, already described, and the invoke_i transitions and modalities will connect the LTS and formula appropriately. Figure 2 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.

It is easy to check that for any environment E and relation R

Lemma 1.

$$tr2_E(R, \mathit{false}) = \neg tr2_E(R, \mathit{true})$$

□

3.3 Correctness of the translation w.r.t. the original QVT-R game

Let $M_k(T, m_1, \ldots, m_n)$ be the model-checking game played on $tr(T)$ and $lts(T, m_1, \ldots, m_n, k)$. We need to establish that, if we start with a QVT-R

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	$\xrightarrow{\text{challenge}}$	(R, B)	if $R \in top(T)$ and $dom(B) = whenvars(R) \cup nonkvars(R)$
(R, B)	$\xrightarrow{\text{response}}$	(R, B')	if $dom(B) = whenvars(R) \cup nonkvars(R)$ and $B' \succeq B$ and $dom(B') = vars(R)$
(R, B)	$\xrightarrow{\text{ext1}}$	(R, B')	if $dom(B) = domainvars(R)$ and $B' \succeq B$ and $dom(B') = domainvars(R) \cup whenvars(R) \cup nonkvars(R)$
(R, B)	$\xrightarrow{\text{ext2}}$	(R, B')	if $dom(B) = domainvars(R) \cup whenvars(R) \cup nonkvars(R)$ and $B' \succeq B$ and $dom(B') = vars(R)$
(R, B)	$\xrightarrow{\text{invoke}_j}$	(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)	$\xrightarrow{\text{invoke}_j}$	(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. 2. Definition of the translation

transformation that conforms to the constraints accepted in [7], we have indeed achieved our aim of giving equivalent semantics. Therefore let T be a transformation in which the *when–where* graph is acyclic; no relation ever invokes itself, either directly or transitively. Suppose also that all *when* and *where* clauses in T consist of conjunctions of relation invocations only. We will call such a transformation *basic*.

Notice that in this restricted case no fixpoint variable actually occurs inside the body of the corresponding μ or ν, so that (a) there is no need for the translation to retain the environment (as it will never be used) and (b) all fixpoints in the translation can be discarded. That is, we may replace $\nu X.\, \phi$ and $\mu X.\, \phi$ by ϕ (which we can be sure does not contain X free) without changing the meaning of the formula. Thus the translation *tr* yields a mu calculus formula which is equivalent to a Hennessy–Milner Logic (HML) formula in which boxes and diamonds correspond directly to challenges and responses. As required, all plays are finite, and the only winning condition is "you win if it is your opponent's turn but they have no legal move".

Theorem 1. *If T is basic, then Verifier has a winning strategy for the model-checking game M_k iff she has one on the QVT-R game G_k.*

Proof. (Sketch) The game graphs are essentially isomorphic: every position where a player of G_k has a choice corresponds to a position where the same player of M_k has a choice, these are the only choices in M_k, and the available choices correspond in turn. We only have to say "essentially" because several consecutive positions in a play of M_k (beginning with one whose formula has an "invoke" modality as the top connective) can correspond to just one position in G_k. Every position in such a sequence, except the last, has exactly one legal move from it, however, so this is unimportant. Since there are no infinite plays, every play terminates when the player whose turn it is to move has no available legal moves; the same player will win a play in G_k and the corresponding play in M_k. □

3.4 Top relation challenges

The translation we have given is faithful to [6, 7] but readers may be wondering why we treated top relations so specially. Why is the initial challenge to a top relation so different from the invocation of a relation in a *when* or *where* clause, and why do we need two different pairs of labels in our transition system, challenge and response, and ext1 and ext2? The reason is that [6] is unequivocal that in the initial challenge to a top relation, the non-k domain variables ($\mathit{domainvars}(R) \cap \mathit{nonkvars}(R)$) are bound (chosen) at the same semantic point as the other variables in $\mathit{whenvars}(R) \cup \mathit{nonkvars}(R)$. By contrast when a relation is invoked from a *when* or *where* clause, the values of all the domain variables of the invoked relation are fixed (by the choices made for variables of the invoking relation) before values are chosen for any other non-k variables of the invoked relation. That is, in the initial challenge to a top relation, there never is a point at which the domain variables, but no others, have been bound (unless there are no others).

An alternative semantics, and one which might be considered preferable for a future language structured like QVT-R, would have Refuter challenge by picking a top relation and bindings for $domainvars(R) \cap nonkvars(R)$ only, and would then have Verifier respond by picking a binding for the kth domain variable. Then play would proceed just as though from a relation invocation with those bindings for the domain variables.

Our intuition that this might be preferable is based on the observation that a consistent pair of models would have a simpler notion of matching than in standard QVT-R. In this variant, if Verifier has a winning strategy, then given bindings for the non-k domain variables of a top relation (that is, an initial challenge by Refuter) there must be a binding for the kth domain variable (that is, a Verifier response) that matches; Verifier's choice at this initial stage must not depend on Refuter's choices of other bindings in the relation, so the matching is simpler and, perhaps, easier for a human developer to comprehend.

That this would, indeed, give different semantics for the same QVT-R transformation is demonstrated by the following relation:

```
top relation R
  domain m1 v1:V1 {}
  domain m2 v2:V2 {}
  when { S(v1,v2) }
}
```

Suppose we use a transformation with this as its only top relation, on model *m1* in which there is some model element of type *V1*, and model *m2* in which there is no model element of type *V2*, in checkonly mode in direction *m2*. In the QVT-R semantics, this will return *true*. The reason is that Refuter will be unable to pick valid bindings for $nonkvars(R) \cup whenvars(R)$ since there is no valid binding for $v2 \in whenvars(R)$ (the top level "for all valid bindings..." statement will be vacuously true). In the alternative semantics, it would return *false*, since Refuter would initially challenge with any valid binding for *v1* and Verifier would be unable to match. It would be easy to modify everything in this paper to support this alternative semantics, if desired; in fact this would simplify the translation.

4 Extending the QVT-R game

Since not everyone will enjoy using a formal semantics of QVT-R in terms of mu calculus, we next extend the rules of the QVT-R game to match the translation. The extension to permit recursive transformations modifies only the winning conditions. To permit complex *when* and *where* clauses we need some new positions and moves.

4.1 Complex *when* and *where* clauses

Lines 3 and 4 in Figure 1, showing the moves that involve challenging a *when* or *where* clause, are removed and replaced by the moves shown in Figure 3.

Source position	Mover	Target position	Notes
$(P, R, B, 1)$	P	$\overline{P}$ to show *when*(R) under B	This simply indicates that player P is challenging the *when* clause of relation R, which is *when*(R), in the presence of bindings B.
$(P, R, B, 2)$	P	$\overline{P}$ to show *where*(R) under B	This simply indicates that player P is challenging the *where* clause of relation R, which is *where*(R), in the presence of bindings B.
P to show Ψ_1 and Ψ_2 under B	$\overline{P}$	P to show Ψ_i under B	$i = 1, 2$: the other player chooses which conjunct P should show
P to show Ψ_1 or Ψ_2 under B	P	P to show Ψ_i under B	$i = 1, 2$: this player chooses which disjunct to show
P to show not Ψ under B	–	$\overline{P}$ to show Ψ under B	there is exactly one legal move, so it does not matter which player chooses
P to show $S(e_1 \ldots e_n)$ under B	$\overline{P}$	$(P, S, C, 1)$	$\forall v_i \in domainvars(S).C : v_i \mapsto eval(e_i, B)$; $dom(C) = domainvars(S) \cup nonkvars(S) \cup whenvars(S)$. C is required to satisfy domain-local constraints on all domains other than k.
P to show ϕ under B	–	–	P wins the play immediately if $eval(\phi, B) = true$ and loses the play immediately otherwise.

Fig. 3. Extensions to the moves of G_k to permit complex *when* and *where* clauses

After a player (as before) chooses to challenge a clause, we enter a sub-play, with a different form of position, to determine which relation, if any, we move to and which way round the players will be then. The positions within the subplay are of the form "P to show Ψ under B" where Ψ is a subformula of the *when* or *where* clause (recall the BNF given earlier) and B (which remains unaltered within the subplay, but is needed at the end of the subplay) is the set of bindings in force at the point where the clause was challenged. Within the subplay, as is usual in logic games, one player chooses between conjuncts, the other between disjuncts, while negation corresponds to the players swapping roles. Notice that in the simple case where *when* and *where* clauses were simply conjunctions of relation invocations, all we have done is to split up what would have been a single move according to Line 3 or 4 of Figure 1 into a sequence of moves – all by the same player who would have chosen that single move – leading eventually to the same position that was the target in the original game.

4.2 Recursive transformations

Our translation can be applied to QVT-R transformations in which a relation does, directly or indirectly, invoke itself recursively. However, because the translation introduces negations, in certain cases it will result in an ill-formed mu calculus formula, as remarked earlier. We need a criterion that can be applied directly to the original QVT-R transformation which will ensure that the target mu calculus formula is well-formed. Fortunately this is easy.

Definition 1. *A recursion path in a QVT-R transformation is a finite sequence, whose elements may be relation names, "when", "where" or "not", such that:*

1. *the first and last elements of the sequence are the same relation name*
2. *any subsequence $R \ldots S$, where R and S are relation names and no intervening element is a relation name, corresponds to S being invoked from a* when *or* where *clause of R in the obvious way. That is, the intervening elements can only be:*
   - *"when" followed by some number $i \geq 0$ of "not"s, if S is invoked in R's* when *clause and the invocation is under i negations; or*
   - *"where" followed by some number $i \geq 0$ of "not"s, if S is invoked in R's* where *clause and the invocation is under i negations.*

Definition 2. *A QVT-R transformation is recursion-well-formed if on every recursion path the number of "not"s plus the number of "when"s is even.*

Since every not, every when, and nothing else, causes the boolean flag in the translation function to be flipped, the recursion-well-formed QVT-R transformations are precisely those that result in well-formed mu formulae.

Having decided which transformations that may lead to infinite plays to permit, we need to specify which player will win which infinite plays. In an infinite play, one or more relation names must occur infinitely often in positions of the play, that is, as the second element of a 4-tuple like those in Figure 1. Of these, let R be the one that occurs earliest in the play *not counting the positions before the first* when/where *invocation* (because the initial challenge to a top relation is different, as discussed in Section 3.4). Look at any 4-tuple involving R (after the first invocation). If the first element is Verifier and the last is 1, or the first element is Refuter and the last is 2 (i.e. the players are "the usual way round"), then Verifier wins this play; otherwise Refuter wins. We will get a consistent answer regardless of which position we examine, because otherwise the transformation would not have been recursion-well-formed, i.e., would have been excluded on monotonicity grounds.

Theorem 2. *The QVT-R game as modified in this section is consistent with the translation semantics.*

Proof. (Sketch) Again, the games map one-to-one onto the standard model-checking games for the mu-calculus formulae of the translation.

Remark: we could have assigned the infinite plays exactly oppositely; this would correspond to swapping μ and ν in the translation. If we did both, we would still get Theorems 1,2. This is a choice for the language designer.

5 Examples and consequences

Fig. 4. Metamodel M and model m for examples

Consider a transformation on models conforming to the metamodel shown in Figure 4, having as its only relation the following:

```
top relation R {
  domain m1 e1:Element {}
  domain m2 e2:Element {}
  where {(e1.next is not null and e2.next is not null)
         and R(e1.next,e2.next)}
}
```

Let us play the extended game in the direction of *m2*. Refuter picks an element to bind to `e1`. Verifier must match by finding an element `e2`. Refuter will challenge the *where* clause, so the new position is "Verifier to show `(e1.next is not null and e2.next is not null) and R(e1.next,e2.next)` under B" where B records the bindings to `e1` and `e2` that the players have just made. (`(e1.next is not null and e2.next is not null)` $\in$ *Constraint*, abbreviated ϕ.) Because the top level connective of the formula in the new position is `and`, Refuter chooses a conjunct, giving new position either $p =$"Verifier to show ϕ under B" or "Verifier to show `R(e1.next,e2.next)` under B". In the first case, Verifier wins the play unless, in fact, `e1.next` or `e2.next` was null. Thus in choosing bindings for `e1` and `e2` we see that it is in Refuter's interest to choose an `e1` with no `next` if there is one – in that case he has a winning strategy – and in Verifier's interest to avoid such a choice for `e2`. In fact, Refuter can win by eventually driving play to position p (with some bindings B) iff *either* there is some Element `e` in *m1* with `e.next == null` (in which case, he may as well choose it immediately) *or* there is no loop in the `next` graph of *m2*, i.e. every element `e` eventually leads, by following `next` links, to some element `e'` with `e'.next == null`. What should happen, however, if Refuter never has the chance to drive play to a position p, because every element `e` from *m1* has non-null `next` and there is some loop in *m2* that Verifier can use to match? (Or, indeed, if he could, but does not choose to?) Refuter can repeatedly choose the "Verifier to show `R(e1.next,e2.next)` under B" position, and play will continue for ever. We consider it natural that Verifier should win such a play, and under our extended rules this is what happens; e.g. position $(\mathsf{Refuter}, R, B, 2)$ recurs.

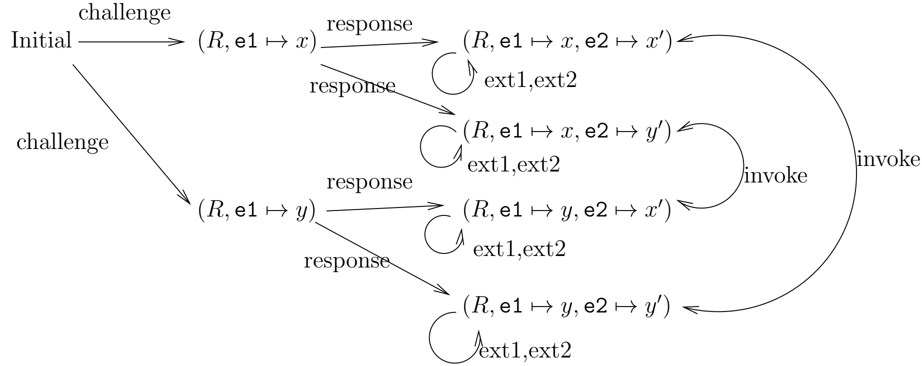

Fig. 5. Labelled transition system for example

Next we demonstrate how this example works under the translation. The translation of the transformation is

$$[\text{challenge}]\,\langle\text{response}\rangle(\phi \wedge \langle\text{invoke}\rangle\nu X.\ [\text{ext1}]\,\langle\text{ext2}\rangle(\phi \wedge X))$$

whose formal semantics corresponds closely to the above description. Specifically, if models *m1* and *m2* are both taken to be copies of m from Figure 4 (distinguished by *m2* having x', y'), the LTS is that shown in Figure 5. Any play of the model-checking game leads to one of the four right-hand LTS nodes, and then as the fixed point is repeatedly unrolled, loops between that node and the one connected to it by an invoke transition. Since our translation used a maximal fixpoint, unrolling the fixed point infinitely often is allowed and Verifier wins any play, so she has a winning strategy and our semantics says that the transformation returns *true*.

5.1 Expressiveness

In principle, a QVT-R transformation can have arbitrarily deep nesting of *when* and *where* clauses. A natural question is whether this actually adds expressivity, or whether every transformation could actually be re-expressed using at most n nestings, for some n. The corresponding question for the modal mu calculus is whether the alternation hierarchy is strict, which it is (see ([1] for details). That is, in the modal mu calculus, allowing more (semantic) nesting always does allow the expression of more properties. However, thus far we only have a translation from QVT-R to mu calculus; it could be that the image of this translation was a subset of mu calculus in which the alternation hierarchy collapsed. In fact, constructing a suitable family of examples enables us to show (see proof in Appendix of [2]):

Theorem 3. *There is no n such that every QVT-R transformation is equivalent to one with* when *and* where *clauses nested to a depth less than n.*

Clearly we inherit upper-bound complexity results also from the mu calculus; however, the complexity of mu calculus model checking is a long-open problem. It is known to be in the class NP ∩ co-NP but is not known to be in P. The problem instance size is the size of the model checking game graph; the running time of well-understood algorithms involves an exponent which depends on the alternation depth of the mu calculus formula. This is of mostly theoretical interest, however, since in practice alternation depths are typically small.

6 Conclusion

We have given a semantics to recursive checkonly QVT-R transformations with complex *when* and *where* clauses by first translating the checking problem into a modal mu calculus model checking problem, and then using this to discover a corresponding change to the rules of our earlier defined QVT-R game. Thus we end up with a semantics which is simultaneously formal and intuitive, requiring no formal training beyond the ability to follow the rules of a game. Our semantics can be instantiated with any desired metamodelling and expression languages, not just MOF and OCL.

Acknowledgements

We thank the referees for their constructive suggestions, including some that could not be implemented 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. 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.
2. Julian Bradfield and Perdita Stevens. Recursive checkonly QVT-R transformations with general *when* and *where* clauses via the modal mu calculus. Technical Report EDI–INF–RR–1410, University of Edinburgh, 2012. Includes Appendix.
3. Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
4. Object Management Group. Object constraint language, version 2.0, formal/2006-05-01, May 2006.
5. D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
6. OMG. MOF2.0 query/view/transformation (QVT) version 1.1. OMG document formal/2009-12-05, 2009. available from www.omg.org.
7. Perdita Stevens. A simple game-theoretic approach to checkonly QVT Relations. *Journal of Software and Systems Modeling (SoSyM)*, 2011. Published online, 16 March 2011. DOI 10.1007/s10270-011-0198-8.