A simple game-theoretic approach to checkonly QVT Relations

Perdita Stevens

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract. The QVT Relations (QVT-R) transformation language allows the definition of bidirectional model transformations, which are required in cases where a two (or more) models must be kept consistent in the face of changes to either. A QVT-R transformation can be used either in checkonly mode, to determine whether a target model is consistent with a given source model, or in enforce mode, to change the target model. Although the most obvious semantic issues in the QVT standard concern the restoration of consistency, in fact even checkonly mode is not completely straightforward; this mode is the focus of this paper. We need to consider the overall structure of the transformation as given by when and where clauses, and the role of trace classes. In the standard, the semantics of QVT-R are given both directly, and by means of a translation to QVT Core, a language which is intended to be simpler. In this paper, we argue that there are irreconcilable differences between the intended semantics of QVT-R and those of QVT Core, so that the translation cannot be helpful. Treating QVT-R directly, we propose a simple game-theoretic semantics. We demonstrate that consistent models may not possess a single trace model whose objects can be read as traceability links in either direction. We briefly discuss the effect of variations in the rules of the game, to elucidate some design choices available to the designers of the QVT-R language.

1 Introduction

Model-driven development (MDD) is widely agreed to be an important ingredient in the development of reliable, maintainable multi-platform software. The Object Management Group, OMG, is the industry's consensus-based standards body, so the standards it proposes for model-driven development are necessarily important. In the area of MDD, a key standard is Queries, Views and Transformations (QVT, [5]), a specification of three different languages for defining *transformations* between models, which may include defining a restricted *view* of a model which abstracts away from aspects of the model not relevant to a particular class of intended user. Rather disappointingly, however, the Queries, Views and Transformations languages have been slow to be adopted. Few tools are available for any of the languages: notably, it sometimes happens that even those tools which use "QVT" in their marketing literature do not actually provide any of the three QVT languages, but rather, provide a "QVT-like" language. In this paper we will consider QVT Relations (QVT-R), the language which best permits the high-level, declarative specification of bidirectional transformations. There have been two candidate implementations of this: Medini QVT[1] and ModelMorf[2]. ModelMorf is the more faithful to [5], but unfortunately development of it seems to have ceased while the tool was still in pre-beta.

Why has the uptake of QVT been so low? Optimistically, we may point to the fact that, while the QVT standard has been under development for a long time, it has only recently been standardised. However, the same applied to other OMG standards, most notably UML, and did not prevent their adoption before finalisation. Lack of support for important engineering activities like testing and debugging may also play a role, but this does not explain why there *do* exist several tools each of which uses its own transformation language other than the OMG standard ones, and case studies of successful use of these tools. Perhaps a contributory factor is that, whereas the UML standard was developed following years of widespread use of various somewhat similar modelling languages, the model transformation arena is still far more sparsely populated. Therefore, how to define, or recognise, a good model transformation language for use on a particular problem is less well understood. We consider that the difficulty developers have in understanding the semantics of QVT may play a role, and we develop a game-theoretic semantics which we hope may be more accessible.

In this paper, we only consider transformations in checkonly mode. That is, we are interested in the case where a QVT-R transformation is presented with two models, and the transformation engine must return true if the models are consistent according to the definition of consistency embodied in the transformation, or false otherwise. Perhaps surprisingly, it turns out that this already raises some interesting issues.

Related work This paper follows on from earlier work by the present author, [9], in which questions answered here, specifically the role of relation invocation in when and where clauses (relation definition applied to particular arguments), were left open. Discussion of the foundations of, and range of approaches to, bidirectionality, not specific to QVT, are presented in [8] and [7] respectively.

Greenyer and Kindler [3] presented at MODELS 2007 a discussion of the relationship between QVT Core and Triple Graph Grammars, together with an outline of a translation from QVT Core to TGGs. Romeikat and others [6] translated QVT-R transformations to QVT Operational. Garcia [1] formalised aspects of QVT-R in Alloy, permitting certain well-formedness errors to be detected.

Formal games have been widely used in computer science; the most relevant strand for this paper is surveyed in [10]. In modelling, the GUIDE tool [11] uses games to support design exploration and verification.

---

[1] http://projects.ikv.de/qvt/, version 1.6.0 current at time of writing

[2] http://www.tcs-trddc.com/ModelMorf/index.htm, but download page not available at time of writing

2 Background

QVT Relations A QVT-R transformation is structured as a number of relations, connected by referencing one another in when and where clauses. The idea is that an individual relation constrains a tuple of models in a rather simple, local, way, by matching patterns rooted at model elements of particular kinds. The power, and the complexity, of the transformation comes from the way in which relations are connected. A relation may also have a when clause and/or a where clause. In these clauses, other relations are invoked with particular roots for their own patterns to be matched. In this way, global constraints on the models being compared can be constructed from a web of local constraints. The allowed dependencies between the choices made of values for variables – in a typical implementation, the order in which these choices are made – are such that the when functions as a kind of pre-condition; the where clause imposes further constraint on the values chosen during the relation to which it is attached (it is, in a way, a post-condition).

The reader is referred to [5] for details: the relevant sections are Chapter 7 and Appendix B. A key point is that the truth of a relation is defined using a logical formula which states that *for every* legal assignment of values to certain variables, *there must exist* an assignment of values to certain other variables, such that a given condition is satisfied.

Logic In logical terms, this is expressed as a "for all–there exists" formula; more precisely, such a formula is called a Π_2 formula, provided that the formula which follows these two quantifiers is itself quantifier-free.

The difficulty in QVT-R is that actually, the truth of a complete transformation is expressed by a much more complex formula. Appendix B only expresses the truth of an individual relation, but this is defined in terms of the truth of the relations which may appear in its when and where clauses, so that, in fact, the number of alternations between universal and existential quantifiers (the length of a forall-thereexists-forall-thereexists... formula which would be equivalent to a whole QVT-R transformation evaluating to true) is unbounded. For example, consider the well-known example of transformation between UML class diagrams and RDBMS schemas, in which packages correspond to schemas, classes to tables and attributes to columns. Looking at [5] p197, we see that ClassToTable invokes relation AttributeToColumn in its where clause. The invocation gives explicit values for the root variables of the patterns in AttributeToColumn, but even though those are fixed, the usual rule applies as regards the rest of the valid bindings to be found in AttributeToColumn. Thus, for each valid binding of one pattern in ClassToTable (and of the when variables), there must exist a valid binding of the other pattern in ClassToTable, *such that* for each valid binding of the remaining variables of one pattern in AttributeToColumn (and of the when variables, except that in this case there are none), there exists a valid binding of the remaining variables of the other pattern in AttributeToColumn.[3] Note that, if there was more than one choice for the second binding in ClassToTable, it is entirely possible that it turns out that only one of these choices satisfies the rest of the condition, concerning the matching in AttributeToColumn: thus any evaluation, whether mental or by a tool, of ClassToTable has to be prepared either to consider both relations together, or to backtrack in the case that the first choice of binding made is not the best.

Therefore, while one might at first glance hope to be able to understand, and evaluate, the meaning of a QVT-R transformation by studying the relations individually, in fact, no such "local" evaluation is possible, because of the way the relations are connected.

Fortunately, similar situations arise throughout logic and computer science, and much work has been done on how to handle them. In particular, this is exactly the situation in which games have found to be a useful aid to developing intuition, as well as to formal reasoning.

Games There is a long history in logic of formulating the truth of a logical proposition as the existence of a winning strategy in a two-player game. For example, the formula $\forall x.\exists y.y > x$ (where x and y are integers, say) can be turned into a game between two players. The player who is responsible for picking a value for x is variously called ∀belard, Player I, Spoiler, Refuter, depending on the community defining the game, while the player responsible for picking a value for y is called ∃louise, Player II, Duplicator or Verifier. We will go with Refuter and Verifier. Refuter's aim is, naturally, to refute the formula, while Verifier's aim is to verify it. In this game, Refuter has to pick a value for x, then Verifier has to pick a value for y. Verifier then wins this play of the game if $y > x$, while Refuter wins this play otherwise. In fact, in this case, Verifier has a *winning strategy* for the game: that is, she has a way of winning the game in the face of whatever moves Refuter may choose. This is an example of a two-player game of perfect information (that is, both players can see everything about one another's moves).

Of course, it is entirely possible that for a particular value of x, there is more than one value of y which makes the formula true: that is, Verifier has more than one winning strategy. When a Π_2 formula is true, a Skolem function expresses a particular set of choices that constitute a winning strategy: given x, it returns the chosen y. Different Skolem functions may exist which justify the truth of the same formula. In the example above, one choice of Skolem function maps x to $x + 1$, another maps x to $x + 17$, another maps 1 to 23, 2 to 4, 3 also to 4, and so on. Clearly the trace model in QVT has something in common with a Skolem function.

Another family of examples comes from concurrency theory. Processes are modelled as labelled transition systems (LTSs), that is, an LTS is a set of states

[3] Actually, the version in [5] is a little more complicated than this: AttributeToColumn invokes further relations in its where clause, and it is those which require the binding of remaining variables: but the point is the same.

S including a distinguished start state $i \in S$, a set of labels L, and a ternary relation $\rightarrow \subseteq S \times L \times S$: we write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. The question of when two processes should be deemed to have consistent behaviour can be answered in many ways depending on context. One simple choice is *simulation*. A process $B = (S_B, i_B, L_B, \rightarrow_B)$ is said to simulate a process $A = (S_A, i_A, L_A, \rightarrow_A)$ if there exists a simulation relation $\mathcal{S} \subseteq S_A \times S_B$ containing (i_A, i_B). The condition for the relation to be a simulation relation is the following:

$$(s, t) \in \mathcal{S} \Rightarrow (\forall a, s' \ .(s \xrightarrow{a} s' \Rightarrow \exists t' \ . \ t \xrightarrow{a} t' \wedge (s', t') \in \mathcal{S}))$$

This can very easily be encoded as a game: starting at the start state of A, Refuter picks a transition. Verifier has to pick a transition from the start state of B which has the same label. We now have a new pair of states, the targets of the chosen transitions, and the process repeats: again, Refuter chooses a transition from A and Verifier has to match it. Play continues unless or until one player cannot go: either Refuter cannot choose a transition, because there are no transitions from his state, or Verifier cannot choose a transition because there is no transition from her state which matches the label on the transition chosen by Refuter. A player wins if the other player cannot move. If play continues for ever, Verifier wins. It is easy to show that in fact, Verifier has a winning strategy for this game exactly when there exists a simulation relation between the two processes; indeed, in a sense which can be made precise, a simulation relation *is* a winning strategy for Verifier. (As with the Skolem functions for Π_2 formulae, there may be more than one simulation relation between a given pair of processes.)

A curious and relevant fact about simulation is that even if B simulates A by simulation relation $\mathcal{S}$ *and* A simulates B by simulation relation $\mathcal{T}$, it does not follow that A simulates B by the reverse of $\mathcal{S}$, nor even that there must exist some relation which works as a simulation in both directions. This is the crucial difference between simulation equivalence and the stronger relation of bisimulation equivalence; see for example [2].

We will shortly define the semantics of QVT-R using a similar game, but first, we must consider an alternative approach.

3 The translation from QVT Relations to QVT Core

In an attempt to help readers and connect the several languages it defines, [5] defines the semantics of QVT Relations both directly, and by translation to QVT Core. Both specifications are informal (notwithstanding some minor use of logic e.g. in Appendix B). [5] does not specify what should happen in the case of conflicts between the two, nor does it explicitly argue for their consistency. Therefore any serious attempt to provide a formally-based semantics for QVT-R needs to take both methods into consideration.

In this section, we consider the translation, with the aid of a very simple example QVT-R transformation. We then argue that, not only is what we believe to be the intended translation of this transformation not semantically equivalent, but also, the intended semantics of QVT Core appear to be such that it simply cannot express semantics equivalent to those of our simple QVT-R example. That is, even if our reading of the translation is incorrect, the problem remains: *no* translation can correctly reproduce the semantics of QVT-R. If the reader is convinced by the argument, it follows that the translation of QVT-R to QVT Core cannot contribute to an understanding of QVT-R.

Consider an extremely simple MOF metamodel which we will call SimplestMM. It defines one metaclass, called ModelElement, which is an instance of MOF's Class. It defines nothing else at all, so models which conform to this metamodel are simply collections (possibly empty) of instances of ModelElement. (Of course, in the usual object-oriented fashion, there is no obstacle to having several instances of ModelElement which are indistinguishable except by their identities.) We will refer to three models which conform to SimplestMM, having zero, one and two ModelElements respectively. We will imaginatively call them Zero, One and Two. Indeed, models conforming to SimplestMM can be identified in this way with natural numbers: a natural number completely determines such a model, and vice versa.

Next, consider a very simple QVT-R transformation between two models each of which conforms to SimplestMM. Figure 1 show the text of the transformation (we use ModelMorf syntax here).

```
transformation Translation (m1 : SimplestMM ; m2 : SimplestMM)
{
  top relation R
  {
    checkonly domain m1 me1:ModelElement {};
    checkonly domain m2 me2:ModelElement {};
  }
}
```

Fig. 1. A very simple transformation

Suppose that we use the QVT-R semantics to execute this transformation *in the direction of* `m2` (we will return to the issue of directionality of checkonly transformations below, in Section 4). When executed in the direction of `m2`, it should return true if and only if, for every valid binding of `me1` there exists a valid binding of `me2`. There are no constraints beyond the type specification, so this is equivalent to: if model `m1` is non-empty, then model `m2` must also be non-empty. If model `m1` is empty, then there is no constraint on model `m2`. Thus, when invoked on the six possible pairs of models from Zero, One and Two, the transformation should return false on the pairs (One,Zero) and (Two,Zero), otherwise true. Conversely, if we check in the direction of `m1`, the transformation returns false if `m1` is empty and `m2` is not, otherwise true. Reassuringly, ModelMorf gives exactly these results.

QVT-R works this way because its semantics are specified using logical “for all–there exists” formulae, without reference to a trace model or any other means of enforcing a permanent binding of one model element to another, such that a model element might be considered “used up”. While [5] says that running a QVT-R transformation “implicitly” generates a trace model, the definition of the transformation does not rely upon its existence. It is simply assumed that an implementation will build a trace model, and use it, for example, to allow small changes to one model to be propagated to another without requiring all the computation involved in running a transformation to be redone. However, because the definition of QVT-R is independent of any trace model or its properties, there is no obstacle to the same model element being used more than once, which is why the transformation has the semantics discussed, rather than enforcing any more restrictive condition, such as that the two models have the same number of model elements. This helps to provide QVT-R the ability to express non-bijective transformations in the sense discussed in [9]; this ability in turn is essential to allow the expression of transformations between models which abstract away different things. The absolute requirement to be able to do this is most obvious when we consider a transformation between a fully-detailed model and an abstracted *view* onto it, where either the full model or the view may be updated (this is called the “view update problem” in databases). Even in transformations between models we might regard as equally detailed, though, it turns out that non-bijectiveness is essential. For example, in a realistic interpretation of a transformation between UML packages and RDBMS schemas, there are many schemas which are consistent with a given package, and many packages consistent with a given schema. See [9] for more discussion.

Now, taking [5] at face value, we expect to be able to translate this simple QVT-R transformation into a QVT Core transformation which has the same behaviour, and which, in particular, will return the same values when invoked on our simple models. The specification of the translation is not so clear that mistakes are impossible (e.g., possibly the multiple importing of the same metamodel is unnecessary), but this is what the author believes to be the intended translation:

```
module SimpleTransformation imports SimplestMM {
transformation Translation {
  m1 imports SimplestMM;
  m2 imports SimplestMM;
}

class TR {
  theM1element : ModelElement;
  theM2element : ModelElement;
}

map R in Translation {
check m1() {
  anM1element : ModelElement
```

```
}
check m2() {
  anM2element : ModelElement
}
where () {
  realize t:TR|
    t.theM1element = anM1element;
    t.theM2element = anM2element;
}
```

The effect of this QVT Core transformation is to construct for every model element in `m1` an object of the trace class `TR` which connects this model element to a corresponding model element in `m2`. However, [5] says several times that in QVT Core, valid bindings must be unique. For example, p133 says:

> There must be (exactly) one valid-binding of the bottom-middle pattern and (exactly) one valid binding of the bottom-domain pattern of a checked domain, for each valid combination of valid bindings of all bottom-domain-patterns of all domains not equal to the checked domain, and all these valid bindings must form a valid combination together with the valid bindings of all guard patterns of the mapping.

and this sentiment is then repeated in a logical notation. In executing the QVT Core version of our transformation on the models (Two,One), this condition would fail because, given the valid binding of the single ModelElement in One to variable `me2`, there would have to be two valid bindings to `me1`, one binding each of the ModelElements in Two. What is not so clear is whether this condition is intended to be satisfied if we run the example on (Two,Two): a literal reading would seen to suggest not, yet it seems impossible that QVT Core is intended to be unable to express the identity relation. The problem is *where* exactly the valid binding is supposed to be unique: in the model, or just in the mapping? That is, given a model element in m2, must there exist only one model element in m1 which could validly be linked to it, or is it, more plausibly, enough that there is only one model element which actually is linked to it by some trace object? Either way, though, (Two,One) will still fail.

Unfortunately no implementation of QVT Core seems to be available. Various sources refer to a pre-release of Compuware OptimalJ, but OptimalJ no longer exists. Therefore we cannot investigate what actual QVT Core tools do.

It is noteworthy, though, that this misapprehension that model elements, or at least patterns of them, must correspond one-to-one in order to make bidirectional transformations possible is pervasive: it appears even in the documentation for Medini QVT, which intends to be an implementation of QVT-R (see Medini QVT Guide, version 1.6, section QVT Relations Language, Bidirectionality).

Could we write a QVT Core transformation which did have the same behaviour as our simple QVT-R transformation? Unfortunately not. A moment's thought will show that the requirement that valid bindings correspond one-to-one (even if only in the constructed trace model) precludes any QVT Core transformation that could return true on both (One,Two) and (Two,One) but false on (One,Zero).

4 Transformation direction

The reader who is familiar with [9] may have noticed an inconsistency between the treatment of bidirectional transformations in that paper and the way we described checkonly transformations above. The framework in [9] is based on a direction-free notion of consistency: a transformation between sets of models M and N specifies, for any pair $(m, n) \in M \times N$, whether or not m is consistent with n. In the above, however, our consistency check had a direction: checking `Translation` in the direction of `m2` is not the same as checking it in the direction of `m1` and indeed, can give different answers. When `Translation` is checked in the direction of `m1` on the pair of models (Zero, One), it returns true, since there are no model elements on the left to be matched. When the same transformation is checked on the same pair of models in the other direction, it returns false.

The standard [5] is slightly ambivalent about whether a checkonly QVT-R transformation has a direction. Compare p13, which talks about "checking two models for consistency" and implicitly contrasts execution for enforcement, which has a direction, with execution for checking, which implicitly does not, with the details of the QVT-R definition which clearly assume that checking has a direction. The resolution seems to be (p19, my emphasis): A transformation can be executed in "checkonly" mode. In this mode, the transformation simply checks whether the relations hold **in all directions**, and reports errors when they do not.

That is, the notion of consistency intended by the QVT-R standard is given by conjunction: `m1` is consistent with `m2` according to transformation R if and only if R's check evaluates to true in both directions.

In fact, ModelMorf requires a transformation execution to have a direction specified, even when it is checkonly: to find out what the final result of a checkonly transformation is, one has to manually run it in each direction and conjoin the results. Medini, by contrast, makes it impossible to run a transformation in checkonly mode: if you run a transformation in the direction of a domain which is marked enforce, there is no way to make the transformation engine return false if it finds that the models are inconsistent, rather than modifying the target model. These seems to be a misinterpretation of [5] and indeed is on the bug list. However, it is a superficial matter, because QVT-R is supposed to have "check then enforce" semantics: that is, it is not supposed to modify a model unless it is necessary to do so to enforce consistency. Therefore, given a QVT engine which was compliant with [5] except that it did not provide the ability to run transformations in checkonly mode, it would be easy to construct a fully compliant engine using a wrapper. The wrapper would save the target model, run the transformation, and compare the possibly modified target model with the original. If the target model had been modified, it would restore the original version and return false; otherwise, it would return true.

5 A game-theoretic semantics for checkonly QVT-R

Given a set of metamodels, a set of models conforming to the metamodels, a transformation written in a simplified version of QVT-R, and a direction for checking, we will define a formal game which explains the meaning of the transformation in the following sense. The game is played between Verifier and Refuter. Refuter's aim in the game is to refute the claim that the check should succeed; Verifier's aim is to verify that the check should succeed. The semantics of QVT is then defined by saying that the check returns true if and only if Verifier has a winning strategy for the game. If this is not the case, then (since by Martin's standard theorem on Borel determinacy [4] the game we will define will be determined, that is, one or other player will have a winning strategy) Refuter will have a winning strategy, and this corresponds to the check returning false.

This approach has several advantages. Most importantly, it separates out the specification of what the answer should be from the issue of how to calculate the answer efficiently. Calculating a winning strategy is often much harder (in both informal, and formal complexity, senses) than checking that a given strategy is in fact a winning strategy. Indeed, it can be useful to calculate a strategy using heuristics or other unsound or unproved methods, and then use a separate process to check that it is winning: this is the game equivalent of a common practice in formal proof, the separation between the simple process of proof checking and the arbitrarily hard process of proof finding. Nevertheless, although this paper does not address the issue of how winning strategies can be calculated efficiently, it is worth noting that formulating the problem in this way makes accessible a wealth of other work on efficient calculation of winning strategies to similar games.[4]

We may also hope to be able to use the game to explain the meaning of particular transformations, or of the QVT-R language in general, to developers or anyone else who needs to understand it: similar approaches have proven successful in teaching logic and concurrency theory.

Finally, a game-theoretic approach is a helpful framework in which to consider the implications of minor variations in decisions about what the meaning of a QVT-R transformation should be, since many such differences arise as minor variations in the rules of the game.

In order to specify a two-player game of perfect information, we need to provide definitions of the positions, the legal moves, the way to determine which player should move from a given position, and the circumstances under which each player shall win.

We fix a set of models, where each m_i conforms to a metamodel M_i, and a transformation definition given in a simplified version of QVT-R. Specifically, we consider that when and where clauses are only allowed to contain (conjunctions of

[4] For the most complex games we consider here, such work is collated in the PGSolver project, http://www.tcs.ifi.lmu.de/pgsolver/. If we insist that the graph of relations should be a DAG, as discussed later in this section, simpler automata-based techniques suffice.

lists of) relation invocations, not arbitrary OCL. We do not consider extension or overriding of transformations or relations. Further, our semantics is parametrised over a notion of pattern matching and relation-local constraint checking: in other words, we do not give semantics for these, but assume that an oracle is given to check the correctness, according to the relevant metamodel, of a player's allocation of values to variables, and local constraints such as identity of values between variables in different domains.

We will first define a game G_k which corresponds to the evaluation of a QVT-R checkonly transformation in the direction of one of its typed models, m_k. For ease of understanding we will explain the progress of the game informally first: Figure 2 defines the moves of the game more systematically. At every stage, if it is a player's turn to move, but that player has no legal moves available, then the other player wins.

To begin a play of game G_k, Refuter picks a top relation (call it R) and valid bindings for all patterns except that from m_k, and for any when variables (that is, variables which occur as arguments in relation invocations in the when clause of R). Notice that he is required to pick values which do indeed constitute valid bindings and satisfy relation-local constraints, as confirmed by the oracle mentioned earlier. Play moves to a position which we will notate $(\text{Verifier}, R, B, 1)$, indicating that Verifier is to move, that the relation in play is R, that bindings in set B have been fixed, and that only one of the players has yet played a part in this relation.

Verifier may now have a choice.

1. She may pick a valid binding for the as-yet-unbound variables from the m_k domain (if any), such that the relation-local constraints such as identity of values of particular variables are satisfied according the oracle. Let the complete set of bindings, including those chosen by both players, be B'. (If there are no more variables to bind, Verifier may still pick this and $B' = B$.) In this case, play moves to a position which we will notate $(\text{Refuter}, R, B', 2)$ indicating that Refuter is to move, that the relation in play is still R, that the bindings in set B' have been fixed, and that both players have now played their part in this relation.
2. Or, she can challenge one of the relation invocations in the when clause (if there are any), say S (whose arguments, note, have already been bound by Refuter). Then play moves to S, and before finishing her turn, she must pick valid bindings for all patterns of S except that from m_k, and for any when variables of S. Say that this gives a set of bindings C, in which the bindings of the root variables of all domains are those from B, and bindings of the other variables are those just chosen by Verifier. The new position is $(\text{Refuter}, S, C, 1)$.

If Verifier chose 2., play proceeds just as it did from $(\text{Verifier}, R, B, 1)$ *except that, notice, the roles of the players have been reversed.* It is now for Refuter to choose one of the two options above, in the new relation S.

If Verifier chose 1., Refuter's only option is to challenge one of the relation invocations in the where clause, say T (whose arguments, note, are bound). (If there are none, he has no valid move, and Verifier wins this play.) Then play moves to T, and, before finishing his turn, Refuter must pick valid bindings for all patterns of T except that from m_k, and for any when variables of T. Say that this gives a set of bindings D, in which the bindings of the root variables of all domains are those from B', and bindings of the other variables are those just chosen by Refuter. The new position is (Verifier, $T, D, 1$). Play now continues just as above.

The final thing we have to settle is what happens if play never reaches a position where one of the players has no legal moves available: who wins an infinite play? We could just forbid this to happen, e.g., by insisting as a condition on QVT-R transformations that the graph in which nodes are relations and there is an edge from R to S if R invokes S in a where or when clause, should be acyclic. There is probably[5] a reasonable alternative that achieves sensible behaviour by allowing the winner of an infinite play to be determined by whether the outermost clause which is visited infinitely often is a where clause or a when clause: but this requires further investigation. Note that [5] has nothing to say about this situation: it corresponds to infinite regress of its definitions.

Position	Next position	Notes
Initial	(Verif., $R, B, 1$)	R is any top relation; B comprises valid bindings for all variables from domains other than k, and for any when variables.
$(P, R, B, 1)$	$(\overline{P}, R, B', 2)$	B' comprises B together with bindings for any remaining variables.
$(P, R, B, 1)$	$(\overline{P}, S, C, 1)$	S is any relation invocation from the when clause of R; C comprises B's bindings for the root variables of patterns in S, together with valid bindings for all variables from domains other than k in S, and for any when variables of S.
$(P, R, B, 2)$	$(\overline{P}, T, D, 1)$	T is any relation invocation from the where clause of R; D comprises B's bindings for the root variables of patterns in T, together with valid bindings for all variables from domains other than k in T, and for any when variables of T.

Fig. 2. Summary of the legal moves of the game G_k: 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. $\overline{\text{Refuter}}$ = Verifier and vice versa. Recall that bindings are always required to satisfy relevant metamodel and relation-local constraints.

5.1 Discussion of the treatment of when clauses

Most of the above game definition is immediate from [5], but the treatment of when clauses requires discussion. From Chapter 7, ([5], p14): "The when clause specifies the conditions under which the relationship needs to hold, so the relation ClassToTable needs to hold only when the PackageToSchema relation holds between the package containing the class and the schema containing the table."

[5] by thinking from first principles about cases in which a play goes through a when (rsp. where) clause infinitely often, but only finitely often through where (rsp. when) clauses; or by intriguing analogy with μ calculus model-checking

The naive way to interpret this would have been to say that both Refuter and Verifier choose their values, and then, if it turns out that the when clause is not satisfied given their choices, Verifier wins this play. This interpretation is not useful, however, as it often gives Verifier a way to construct a winning strategy which does not tell us anything interesting about the relationship between the models. When challenged by Refuter to pick a value for her domain, all she would need to do would be to pick a binding such that the when clause was not satisfied. In the case discussed by [5], whenever Refuter challenged with a class, she would reply with any table from a schema not corresponding to the package of his class, the when clause would not be satisfied, and she would win.

So the sense in which a when clause is a precondition must be more subtle than this. In programming, giving a function a precondition makes it easier for the function satisfy its specification, but here the idea is rather to restrict Verifier's choices: if Refuter chooses a class C in package P, Verifier is bound to reply not with any table, but specifically with a table T which is in a/the schema that corresponds to package P. The intuition behind allowing Verifier to counter-challenge the when clause is that Refuter may "unfairly" challenge Verifier to match the class from a/the "wrong" schema.

In trying to settle whether we really mean "a schema" or "the schema" in the paragraph above, we refer again to Appendix B of [5]. The problem is that this is not a complete definition. E.g., in order to use it to interpret ClassToTable, we already need to be able to determine whether, for given values of a package p and schema s, the when clause `when` $\{$ `PackageToSchema (p,s)` $\}$ holds. Informally it seems that people who write about QVT have two different interpretations of this, perhaps not always realising that they are different:

1. the purely relational: the pair `(p,s)` is any member of the relation expressed in `PackageToSchema`, *when it is interpreted using the very same text which we are now trying to interpret*
2. the operational: the program which is checking the transformation is assumed to have looked at `PackageToSchema` already and chosen a schema to correspond to package `p` (recording that choice using a trace object). According to this view, we only have to consider `(p,s)` if `s` is the very schema which was chosen on this run of the checking program.

To see the difference, imagine that there are two schemas, `s1` and `s2`, either of which could be chosen as a match for `p` in `PackageToSchema`. In the first interpretation, both possibilities have to be checked when `ClassToTable` is interpreted; in the second, only whichever one was actually used.

In our main game definition, we have taken the purely relational view, since we can do so while remaining compatible with the definitions in [5], whereas as we have seen in the SimplestMM example – which, recall, had no when or where clauses and whose semantics were therefore defined unambiguously by Appendix B – the idea that there should be a one-to-one correspondence between valid bindings is incompatible with Appendix B; but we will shortly consider a variant of the game which brings us closer to the latter view.

5.2 Variants of the game

Non-directional variant Let G be the variant of G_k in which, instead of a direction being defined as part of the game definition, Refuter is allowed to choose a direction ("once and for all") at the beginning of the play. Clearly, Verifier has a winning strategy for G if and only if she has a winning strategy for every G_k. This is the way of constructing a non-directional consistency definition from directional checks that is specified in [5]. However, note that it is not automatic that there should be any simple relationship between the various winning strategies; hence, there may not be any usable multi-directional trace relationship between the bindings in different models. Let us explain using an example derived from one in [2].

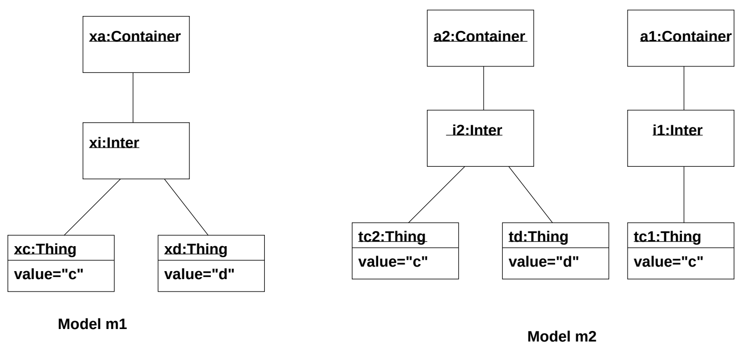

Fig. 3. m1 and m2 are (two-way) consistent according to QVT-R transformation Sim, but no set of bi-directional trace objects can link them

Figure 3 illustrates two models which conform to the obvious metamodel MM: a model may include multiple Containers, each of which references one Inter, each of which may reference multiple Things, each of which has a value. The following QVT-R transformation evaluates to true on the models shown, in both directions (both according to [5], and according to ModelMorf). Indeed, Verifier has a winning strategy for G: the only interesting choice she has to make is in G_2, where she has to be sure to reply with a2 (and i2), not a1 (and i1), if Refuter challenges in ContainersMatch by binding xa to c1 (and xi to inter1).

```
transformation Sim (m1 : MM ; m2 : MM)
{
  top relation ContainersMatch
  {
    inter1,inter2 : MM::Inter;
    checkonly domain m1 c1:Container {inter = inter1};
    checkonly domain m2 c2:Container {inter = inter2};
    where {IntersMatch (inter1,inter2);}
  }
```

```
  relation IntersMatch
  {
    thing1,thing2 : MM::Thing;
    checkonly domain m1 i1:Inter {thing = thing1};
    checkonly domain m2 i2:Inter {thing = thing2};
    where {ThingsMatch (thing1,thing2);}
  }

  relation ThingsMatch
  {
    s : String;
    checkonly domain m1 thing1:Thing {value = s};
    checkonly domain m2 thing2:Thing {value = s};
  }
}
```

Now, in the m1 direction the constructed trace will take a1 to xa, etc.; there is nothing else it can do. Yet in the m2 direction, a trace object which took xa to a1 would be erroneous. Thus *there can be no single set of trace objects whose links can be read in either direction, which could capture the correctness of this QVT-R transformation.*

Model-switching variant Let G' be the variant of G in which, instead of the first player to move in a new relation being constrained to pick a valid binding everywhere except in the once-and-for-all designated target model m_k, the player is permitted to pick valid bindings for all but any one domain, making a new choice of which domain to leave out every time. This is a different way to define a non-directional variant of the game. The modification to the game rules is analogous to the difference, in concurrency theory, between a game which defines bisimulation equivalence and that which defines simulation equivalence. This might well have better properties as regards the existence of a sensible multi-directional trace model. This requires further investigation. Certainly in the example above, it will be Refuter who has a winning strategy for G': he will first challenge in m2 with a1, and later switch to m1 where he leads play to the "d" which cannot be matched starting from a1 in m2.

Trace-based variant Let G^T be the variant of G in which, as play proceeds, we build a global auxiliary structure which records, for each relation, what choices of valid binding have been made by the players (for example, "Package P was matched with schema S"). It is an error if subsequent moves in a play try to choose differently (and we might consider a multi-directional subvariant in which *either* matching P with S' *or* matching S with P' was an error, along with uni-directional subvariants in which only one of those would be an error). The player to complete such an erroneous binding would immediately lose. Otherwise, play would be exactly as in G_k, *except that* it loops: if Refuter cannot go, he can "restart", choose a new top relation and play again, but the old auxiliary structure is retained. If play passes through infinitely many restarts, Verifier wins.

This game would impose one-to-one constraints on valid bindings, and construct well-defined trace objects, at the expense of having a semantics incompatible with [5] and having curtailed expressivity.

6 Conclusions

We have presented a game-theoretic semantics of QVT-R checkonly transformations, based on the direct semantics in [5]; we justified our choice to ignore the translation to QVT Core by pointing out a fundamental incompatibility between the two languages. We have briefly discussed variants of the game, demonstrating in the process that bi-directional trace objects may not exist.

References

1. Miguel Garcia. Formalization of QVT-Relations: OCL-based Static Semantics and Alloy-based Validation. In *Proceedings of the Second Workshop on MDSD Today*, pages 21–30, October 2008.
2. R.J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.
3. Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems, MODELS 2007*, volume 4735 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2007.
4. Donald A. Martin. Borel determinacy. *Annals of Mathematics. Second series*, 102(2):363–371, 1975.
5. OMG. MOF2.0 query/view/transformation (QVT) version 1.0. OMG document formal/2008-04-03, 2008. available from www.omg.org.
6. Raphael Romeikat, Stephan Roser, Pascal Müllender, and Bernhard Bauer. Translation of QVT relations into QVT operational mappings. In *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, pages 137–151, Berlin, Heidelberg, 2008. Springer-Verlag.
7. Perdita Stevens. A landscape of bidirectional model transformations. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer, 2008.
8. Perdita Stevens. Towards an algebraic theory of bidirectional transformations. In *Proceedings of the International Conference on Graph Transformations, ICGT'08*, volume 5214 of *Lecture Notes in Computer Science*, pages 1–17. Springer, September 2008. invited paper.
9. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Journal of Software and Systems Modeling (SoSyM)*, 2009. to appear.
10. Colin Stirling. Bisimulation, model checking and other games. In *Notes for Mathfit Instructural Meeting on Games and Computation*, 1997. Available from http://homepages.inf.ed.ac.uk/cps/mathfit.ps.
11. Jennifer Tenzer and Perdita Stevens. GUIDE: Games with UML for interactive design exploration. *Journal of Knowledge Based Systems*, 20(7), October 2007.