# A simple game-theoretic approach to checkonly QVT Relations

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

February 2011

## Abstract

The QVT Relations (QVT-R) transformation language allows the definition of bidirectional model transformations, which are required in cases where two (or more) models must be kept consistent in the face of changes to either or both. 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. A precise understanding of checkonly mode transformations is prerequisite to a precise understanding of enforce mode transformations, and this is the focus of this paper. In order to give semantics to checkonly QVT-R transformations, 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 no translation from QVT-R to QVT Core can be semantics-preserving, and hence no such translation can be helpful in defining the semantics of QVT-R. Treating QVT-R directly, we propose a simple game-theoretic semantics. We demonstrate its behaviour on examples and show how it can be used to prove an example result comparing two QVT-R transformations. 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.

Keywords: bidirectional model transformation, QVT Relations, QVT Core, games, semantics, consistency checking

## 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, [16]), 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 declarative specification of bidirectional transformations. There have been two main candidate implementations of this: Medini QVT[1] and ModelMorf[2]. ModelMorf is the more faithful to [16]. The landscape of QVT-R tools is discussed more fully in Section 8.1.

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, although work is proceeding on the latter, see for example [25]. This, however, 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. Especially, *bidirectional* languages are insufficiently understood. As we shall discuss in Section 8, almost all of the existing work on QVT-R focuses on the use of the language as a unidirectional model transformation language: although the same transformation text is taken to embody more than one unidirectional transformation, more than this is required for true bidirectionality, as we shall explain in Section 2. We consider that the difficulty developers have in understanding the semantics of QVT, especially its bidirectional aspects, may play a role, and we develop a game-theoretic semantics which we hope may be more understandable than the standard [16] itself and than any previous attempt at semantics.

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 or more 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. We hope in future to extend this approach to transformations in enforce mode, that is, which modify a model in order to restore consistency.

This is an extended version of a paper [22] presented at the International Conference on Model Transformations in June 2009. The main contributions of that paper were a new game-theoretic semantics for a version of QVT-R, and a demonstration that [16]'s translation from QVT-R to QVT Core is not semantics-preserving. As well as giving more formal details, this paper adds two new sections: Section 5, which applies the new semantics to a family of examples and compares the results with those from ModelMorf, and Section 6, which shows how the game semantics can be exploited to prove results about the (anti-)equivalence of checkonly QVT-R transformations. We also discuss the nature and role of trace objects in QVT-R more thoroughly, showing carefully that it is not possible, in general, for QVT-R trace objects to be correct independent of the direction of checking: that is, bidirectional trace objects need not exist. The papers follow on from earlier work by the present author, [21], 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 [20] and [19] respectively.

The structure of the paper is as follows. In Section 2 we give necessary background concerning the QVT-R language itself, logic, and games. Thinking about the meaning of a checkonly QVT-R transformation in logical terms enables us to pinpoint the difficulty with the semantics given in [16]. We can then use games, which have a long history of use in logic, to QVT-R in order to address this difficulty. The QVT-R background given here is brief, and assumes that the reader already has some acquaintance with

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

[2]`http://121.241.184.234:8000/ModelMorf/ModelMorf.htm`

the language, and will consult [16] as needed. Section 3 discusses translating QVT-R into QVT Core as a means of giving semantics to QVT-R and explains why, despite this being done in [16], it is unsuccessful. Section 4, the heart of the paper, gives our semantic game. Sections 5 and 6 have already been discussed. Section 7 discusses how variants of the game can be used to capture different possible semantics of checkonly QVT-R, should the community wish, thereby demonstrating the potential of the game approach to definining the semantics of model transformation languages in general. Section 8 discusses related work and (Section 8.1) QVT-R tools. Section 9 concludes.

## 2 Background

### 2.1 QVT Relations

The QVT-R transformation language allows the definition of bidirectional model transformations, which are required in cases where a two (or more[3]) models must be kept consistent in the face of changes to either or both. A transformation written in QVT-R can support scenarios in which one model is generated from another, for example, where an initial Platform Specific Model (PSM) is generated from a Platform Independent Model (PIM), or where an initial UML model is generated from an existing database schema. It can also support continued development of both. A QVT-R transformation embodies a notion of consistency: the choices made by the transformation writer specify the circumstances under which a given PIM will be considered to be consistent with a given PSM, or a given UML model consistent with a given schema. The same QVT-R transformation text also embodies instructions for restoring consistency in either direction. When changes have been made to one model, the other model can be automatically changed to be consistent with it again. Crucially, this does not involve simply regenerating the other model. The changes that must be made to restore consistency depend on both models. To give a simple illustrative example, suppose that some UML designers take the UML model generated from a database schema and change it by, say, adding a use case diagram. The resulting UML model may still be consistent with the database schema, if (as is plausible) the transformation does not constrain the UML model as to the existence or nature of a use case diagram. Thus these changes do not necessitate any change to the database schema. Now suppose that the database designers modify the schema in a way which does make the schema inconsistent with the UML model, perhaps by splitting a class into two. The QVT-R transformation can be used to restore consistency by modifying the UML model, perhaps by modifying the UML class diagram to show the newly split class. It is of course essential that the transformation does not, in the process, delete the use case diagram! That is, rerunning the generation process that produced the initial UML model is not appropriate. The transformation must, in general, take into account both the current state of the schema and the current state of the UML model, and it must produce a new UML model that takes both into account. (The initial generation then becomes a special case, where the current state of the UML model is that it is a null or empty model.)

When a QVT-R transformation is being used to determine whether a target model is consistent with a given source model, we say that it is being used in *checkonly mode*. When it is used to change a target model, we say that it is used in *enforce mode*. Naturally, immediately after enforcement, a checkonly transformation must return true. A less obvious, but important, decision made by the designers of the QVT-R language [16] is that it has "check before enforce" semantics, which we might vulgarly call "if it ain't broke, don't fix it". One consequence of this policy is that if two models are already consistent – the transformation in checkonly mode returns true – then the same transformation in enforce mode must not make any change at all. This policy is invaluable for model developers using transformations, who can be sure that their work will not be modified without good reason.

When a QVT-R transformation is run, it will mod-

[3]for clarity of exposition, we will assume just two models in this section: the semantics in this paper is for any number of models

ify at most one model, the target model if it is in enforce mode, or none if it is in checkonly mode. It has access, however, to both the source model(s) and the current version of the target model, and it will generally need to examine all of these models. In [21] it was explained that this means we can formalise a QVT-R transformation $S$ on two sets of models $M$ and $N$ as a triple:

- $S \subseteq M \times N$, the consistency relation, which holds of a pair of models – and we write $S(m, n)$ – if and only if the pair of models is deemed to be consistent;
- $\overrightarrow{S} : M \times N \longrightarrow N$, the enforce transformation producing a modified model from the set $N$
- $\overleftarrow{S} : M \times N \longrightarrow M$, the enforce transformation producing a modified model from the set $M$

satisfying certain constraints.

Although in general our enforcement functions $\overrightarrow{S}$ and $\overleftarrow{S}$ need both their arguments, this may not be so for a particular transformation if its consistency relation has some special property. For example, an important special case is when $N$ consists of *views* of $M$, so that $S(m, n)$ holds exactly when $n$ contains a specified subset of the information in $m$. In this case, given $m$ there is a unique $n$ such that $S(m, n)$, and the $N$ argument to $\overrightarrow{S}$ is redundant. In the even more special case that the consistency relation is bijective the transformation may be modelled by a pair of functions $\overrightarrow{S} : M \longrightarrow N$ and $\overleftarrow{S} : N \longrightarrow M$. Notice that what this means, informally, is that a model $m \in M$ contains exactly the same information as the unique consistent model in $N$, simply presented differently. We say that such a model transformation is *bijective*, since it incorporates a bijective consistency relation.

Such special cases undoubtedly arise and can be important. However, a model transformation language which *only* permitted the expression of bijective transformations would not be useful in the general MDD context, where it is usual for each model to incorporate some information not represented elsewhere (which may indeed be the reason for the model's existence). QVT-R is not restricted in this way: as stated, a QVT-R transformation has access to instances of all models, even though it modifies at most one of them.

This paper, focusing on checkonly transformations, formalises the consistency relation embodied in a transformation but not the enforcement functions. We should make explicit that, even though the same QVT-R transformation text embodies both the consistency relation and the enforcement functions, our limitation is contentful. There can indeed be different QVT-R transformations which embody the same consistency relation, but which differ in the instructions they give for restoring consistency when it is broken. For example, QVT-R provides the **key** keyword which makes no difference to the behaviour of the transformation in checkonly mode, but does make a difference to its behaviour in enforce mode. Since **key** is irrelevant to the semantics of checkonly transformations, however, we do not consider it further.

**Direction of transformation** The framework briefly described above 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 $n$. By contrast, an enforcement transformation has a direction, towards the target model, the one which will be modified if changes are necessary to restore consistency.

The standard [16] 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 $S$ if and only if $S$'s check evaluates to true in both directions. The directional executions of $S$ are, however, independent: only their Boolean results are conjoined. As we shall see in Section 3 this is different from how checking transformations work in QVT Core; as we shall see in Section 7.2, the difference is more important than one might at first expect.

Looking at the QVT-R tools, we find that 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 QVT, 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. This is one of several known respects in which Medini QVT departs from [16]; it is on Medini QVT's bug list, and is also mentioned by its developers in [13]. In itself it is superficial matter, because of QVT-R's "check then enforce" semantics. Given a QVT engine which was compliant with [16] 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. In fact, as we shall discuss further in Section 8.1, this is only one of the respects in which Medini QVT departs from [16].

**Structure of a QVT-R transformation** 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 an easy-to-understand, 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, in the sense that it is expected to be evaluated before the body of the relation to which it is attached; 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). However, referring to *when* and *where* clauses as pre- and post-conditions can be misleading. In ordinary programming, pre- and post-conditions help to *specify* behaviour which is actually defined separately: removing the pre- and post-conditions does not alter the behaviour of the system. In QVT-R, *when* and *where* conditions are essential parts of the language itself. Placing a *where* clause on a relation does not document a fact about the relation which is already true: it fundamentally alters the meaning of the relation.

The reader is referred to [16] 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.[4]

## 2.2 Logic

In logical terms, this is expressed as a "for all–there exists" formula; more precisely, such a formula is called a $\Pi_2$ 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

[4]Very similar definitions are found elsewhere in computer science, for example, in refinement calculus [2].

the truth of an individual relation, not the entire transformation. However, the truth of one relation is defined in terms of the truth of the relations which may appear in its *when* and *where* clauses. On substituting the Appendix B definition of each of those truth values, we see 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 [16] p14, 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.[5] Note that, if there was more than one choice for the second binding in ClassToTable, it is entirely possible for it to turn 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 use 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. This is an example of a two-player game of perfect information (that is, both players can see everything about one another's moves). 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. For example, she could decide always to obtain her value of $y$ by adding 42 to whatever value of $x$ is chosen by Refuter.

Formally speaking, we define a game as follows. Notice that our definition allows for the possibility of plays of a game being infinite, although it may happen that a particular game is defined in such a way that all plays are finite. We will use Player $P$, Player $\overline{P}$ to mean Verifier and Refuter in either order; that is, if Player $P$ is Verifier then Player $\overline{P}$ is Refuter, and vice versa.

**Definition 1.** *A game $G$ is $(Pos, Initial, moves, \lambda, W_{\mathcal{R}}, W_{\mathcal{V}})$ where:*

- *Pos is a set of positions. We use $u, v, \ldots$ for positions.*

[5] Actually, the version in [16] 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.

- $\lambda : Pos \to \{Verifier, Refuter\}$ *defines who moves from each position.*

- *Initial* $\in Pos$ *is the starting position: for purposes of this paper,* $\lambda(Initial) = Refuter$.

- *moves* $\subseteq Pos \times Pos$ *defines which moves are legal. A* play *is in the obvious way a finite or infinite sequence of positions, starting with Initial, where* $p_{j+1} \in moves(p_j)$ *for each* $j$. *We write* $p_{ij}$ *for* $p_i \dots p_j$.

- $W_{\mathcal{R}}, W_{\mathcal{V}} \subseteq Pos^\omega$ *such that* $W_{\mathcal{R}} \cup W_{\mathcal{V}} = Pos^\omega$ *are disjoint sets of infinite plays, and (for technical reasons)* $W_P$ *includes every infinite play* $p$ *such that there exists some* $i$ *such that for all* $k > i$, $\lambda(p_k) = \overline{P}$.

*Player* $P$ wins *a play* $p$ *if either* $p = p_{0n}$ *and* $\lambda(p_n) = \overline{P}$ *and* $moves(p_n) = \emptyset$ *(you win if your opponent can't go), or else* $p$ *is infinite and in* $W_P$.

Then a strategy for such a game is, informally, a set of instructions for one player, telling the player how to move in response to any (legal) move of the opponent. A strategy may be deterministic – it tells the player exactly how to move – or non-deterministic – it gives a set of possible moves. In general, the move prescribed by the strategy may depend on the entire play so far. A strategy in which the moves prescribed only depend on the current position (not on the way in which the current position was reached) is called memoryless or history-free. More formally:

**Definition 2.** *A (nondeterministic) strategy* $S$ *for player* $P$ *is a partial function from finite plays* $pu$ *with* $\lambda(u) = P$ *to sets of positions (singletons, for deterministic strategies), such that* $S(pu) \subseteq moves(u)$ *(that is, a strategy may only prescribe legal moves). A play* $q$ *follows* $S$ *if whenever* $p_{0n}$ *is a proper finite prefix of* $q$ *with* $\lambda(p_n) = P$ *then* $p_{n+1} \in S(p_{0n})$. *Thus an infinite play follows* $S$ *whenever every finite prefix of it does. It will be convenient to identify a strategy with the set of plays following the strategy and to write* $p \in S$ *for* $p$ *follows* $S$. $S$ *is a* complete *strategy for Player* $P$ *if whenever* $p_{0n} \in S$ *and* $\lambda(p_n) = P$ *then* $S(p_{0n}) \neq \emptyset$. *It is a* winning *strategy for* $P$ *if it is complete and every* $p \in S$ *is either finite and extensible or is won by* $P$. *It is* history-free *(or* memoryless*) if* $S(pu) = S(qu)$ *for any plays* $pu$ *and* $qu$ *with a common last position. A game is* determined *if one player has a winning strategy.*

All the games we need to consider are determined by standard game theory [15]: in fact, one player or the other will have a memoryless deterministic winning strategy. In this simpler situation, a strategy for Player $P$ will simply be a partial function $S$ from positions $u$ that have $\lambda(u) = P$ to positions, such that $S(u) \in moves(u)$, that is, the strategy only prescribes legal moves. We may conveniently describe $S$ by giving the set of maplets $\{u \mapsto S(u)\}$. For practical purposes, it will suffice to define the strategy at positions that are actually reachable by following the strategy (allowing, of course, for all possible choices by the opponent, Player $\overline{P}$).

Returning to our example of the logic game based on $\forall x.\exists y.y > x$, it is of course entirely possible that a player has more than one winning strategy. When a $\Pi_2$ 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: it expresses a way in which parts of one structure may be mapped to "corresponding" parts of another. We do not yet have a game for QVT transformations that would enable us to make this notion of correspondence precise, however: we will return to the issue in Section 7.2.

Another family of examples, which may get us closer, comes from concurrency theory. Processes are modelled as labelled transition systems (LTSs), that is, an LTS is a set of states $S$ including a distinguished start state $i \in S$, a set of labels $L$, and a ternary relation $\to \subseteq S \times L \times S$: we write $s \xrightarrow{a} t$ for $(s, a, t) \in \to$. 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 $\Pi_2$ formulae, there may be more than one simulation relation between a given pair of processes.)

A curious 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 [10]. As we shall see in Section 7.2, this is relevant.

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, [16] 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). [16] 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 – so simple that the direct semantics of QVT-R leaves no room for doubt about its intended behaviour. Translating the example into QVT Core should produce a semantically equivalent transformation, but we will show that it does not. In fact, the situation is worse than that: according to its intended semantics, QVT Core 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. For purposes of the present paper, indeed, it will be enough if the reader is convinced only that the translation into QVT Core involves some degree of semantic ambiguity: giving a language a semantics by translation into another language is useful only when both the translation and the target language are solid. This will suffice to justify our basing our formal semantics on the direct semantics given in [16] rather than on the translation into QVT Core. The reader who is happy to grant us that choice and is not interested in QVT Core may wish to skip the remainder of this section.

## 3.1 An example to translate from QVT-R to QVT Core

Consider an extremely simple MOF metamodel which we will call SimplestMM.[6] 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 shows the text of the transformation (we use ModelMorf syntax here).

Suppose that we use the QVT-R semantics to execute this transformation *in the direction of `m2`*. 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 [16] 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 Section 2 and [21]; 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). It turns out that non-bijectiveness is essential even when transforming models we might regard as equally detailed. 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 [21] for more discussion.

Now, taking [16] 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. A basic condition which we certainly expect is that the translation will take the same arguments as the original – it would not be acceptable if the translated transformation required extra information, such as a trace model specifying how model elements are to be linked. The specification of the translation is not so clear that mistakes are impossible (e.g., possibly the multiple importing of the same metamodel is unnec-

[6]XML files are available from the author's homepage, see `http://homepages.inf.ed.ac.uk/perdita/papers.html`

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

Figure 1: A very simple transformation

essary), 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;
}
```

An object of the trace class `TR` connects a model element in `m1` to a corresponding model element in `m2`.

To understand the effect of this QVT Core transformation, and in particular to understand what trace objects will be created and which model elements will be linked by them, we need to look carefully at the QVT Core definition (Chapter 9 of [16]). There are several important differences between the semantics of QVT Core and those of QVT-R and several separate problems of interpretation arise.

## 3.2 QVT Core's checking mode

Like QVT-R, QVT Core has two modes for transformation execution, which in this case are called checking and enforcement mode. From p119 of [16]:

> A transformation may be executed in one of two modes: checking mode or enforcement mode. In checking mode, a transformation execution checks whether the constraints hold between the candidate models and the trace model, resulting in reporting of errors when they do not. In enforcement mode, a transformation execution is in a particular direction, which is defined as the selection of one of the candidate models as the target model.

The obvious expectation is that the translation of a QVT-R *checkonly* transformation is to be a QVT Core *checking* transformation. However, the above passage suggests, although it does not quite imply, that a trace model will be an argument to a checking transformation, and, especially since trace models play no formal role in the QVT-R world, there is no reason why the user who wants a QVT-R checkonly transformation interpreted on a set of models should be able to provide a trace model. Indeed we have already remarked on the unacceptability of the translation requiring more information than the original transformation did. We might (and shall, after this discussion) get round that problem by supposing that an initially empty trace model is to be provided, which the checking transformation is to attempt to populate. But a literal reading of Chapter 9 of [16] says that a checking transformation is not permitted to modify the models – no exception is made for the trace model – or to create instances of `realize` variables – such as the trace object in the translated transformation above. According to this view, all a checking transformation can do is to take a set of candidate models and a given trace model relating them, and verify whether certain conditions hold. That is, in this interpretation the QVT Core checking transformation is doing something much less interesting than the QVT-R checkonly transformation. It is not checking whether the candidate models are consistent according to the transformation: it is merely checking whether such consistency is successfully demonstrated by the given trace model. Any incompleteness or incorrectness in the trace model given as argument to the checking transformation will result in the checking transformation reporting inconsistency – even if one of the candidate models could have been produced from another by the very same QVT Core transformation run in enforce mode. This interpretation would make it impossible to use QVT Core checking transformations as the target of translation from QVT-R checkonly transformations.

We next try the hypothesis that this QVT Core transformation, even though it is a translation of a checkonly QVT-R transformation, is in fact to be executed in QVT Core's enforce mode, in order to create the trace model (while not modifying the given candidate models). However, reading [16] with this in mind does not result in sense: the specification simply does not allow for the possibility that a transformation is run in enforce mode without one of the candidate models (i.e., not the trace model) being enforced. See [16] 9.10.1 for example.

It is difficult to see how to get out of even this initial difficulty in interpreting [16]'s translation from QVT-R to QVT Core. What we will do, because it seems to make more sense than any other interpretation, is to assume that a QVT Core checking transformation *is* permitted to create trace objects if required, and is expected to return a Boolean result and (in the case of success) a set of trace objects recording the mapping constructed.

This is actually a less serious problem than might appear, because regardless of how the trace model is constructed, once a supposedly correct trace model exists, the transformation run in checking mode with the original candidate models and that trace model should not return any errors. What matters for our purposes is the existence or non-existence of the correct trace model.

One difference between QVT-R and QVT Core which must be noted, although (until we consider bidirectional trace objects in Section 7.2) it will seem to be presentational more than semantic, is that a QVT Core checking transformation definitely does not have a direction: it can check more than one domain in one transformation execution. This is clearly implied by the passage already quoted, and again by the logical notation for checking on p124 which gives a single logical statement for the case that "domains L and R are given and both are checked". As we would expect, the statement amounts to the conjunction of the two directional checks that would be done if only one domain were checked. Indeed, QVT Core trace objects do not have a distinguished target domain, as can be seen from the definition of class `TR` above. These trace objects are *bidirectional* in the sense that they link bindings in both directions equally. The checking transformation takes a single trace model and (whether or not it also constructs its contents) uses it to check consistency in all directions in the same execution.

In evaluating our translated transformation, therefore, we need to check that the translated version returns True exactly where the original QVT-R version returned True in both directions. In our examples, that means that if it is a semantically correct translation it should return False when exactly one of the models it is checking is Zero, otherwise True. If we think of the transformation as constructing the trace model, it should fail if one of the candidate models is Zero, otherwise it should succeed.

## 3.3 Bijections between sets of valid bindings

A more obviously serious difference, independent of the above, is that, in QVT Core, unlike in QVT-R, a mapping sets up a bijection between sets of valid bindings. On p122 of [16] we are told:

> A mapping declares essentially that all bottom patterns should relate one-to-one. That is, for each valid binding of one of the bottom patterns there must be exactly one valid binding for each other bottom pattern in that mapping. This implies that each valid binding of a bottom pattern may only be part of one unique valid combination of valid bindings for each bottom pattern.

That is, unlike in QVT-R, a valid binding is to be considered used up once it has been used once. Note especially the final sentence: if we are in the common case of checking two domains, there are three bottom patterns, one for each domain and one "middle" pattern for the trace object that links the two domains. The text above insists that, given a valid binding for *any one* of these three places, valid bindings for both of the other two are determined. The same binding in one domain cannot be used twice in the same mapping, together with two different bindings in another domain.

Again, p123 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. The use of "the checked domain" may be confusing to the reader in the light of the fact that QVT Core checking transformations do not have a direction and check all checkable domains in one execution. However, in context it is clear that "the checked domain" is simply the one which is being checked at some given point in the transformation execution: where there are several checkable domains, the condition will be checked for each of them and the results conjoined before the transformation can return its result, as shown in logical notation on p124 of [16]. In our case of two checkable domains, the two texts both imply that a successful execution of the checking transformation produces a bijection between the set of valid bindings in one domain and the set of valid bindings in the other domain. The set of middle bindings, also in bijection with each of these two sets, could be regarded as the set of pairs which constitutes the bijection: each trace object produced by the mapping represents such a pair.

In our example, a valid binding in the domain `m1` is simply a choice of ModelElement to assign to `anM1element`, and similarly a valid binding in domain `m2` is simply a choice of `anM2element`.

## 3.4 Interpretations of the uniqueness of bindings

There is one area which is arguably still open to interpretation: in what set of valid bindings is there supposed to be exactly one choice? The set of all candidate valid bindings from the model being examined, or only the set of valid bindings actually recorded by the mapping? Since this has been the cause of some confusion in an earlier version of the paper, let us spell out the possibilities explicitly in a simple case where there are two domains with sets $M$ and $N$ of valid bindings, leaving out guards for simplicity. We will write $R(m, n)$ if linking $m$ and $n$ by the trace object represented by $(m, n)$ would be a valid combination: bindings $m$ and $n$ are compatible according to the transformation. $R \subseteq M \times N$ represents the space of all trace objects a tool could validly construct: returning a pair $(m, n)$ not in $R$ would be illegal irrespective of any other pairs returned. Now a tool may in fact, perhaps, construct not the whole of $R$ but a *trace* of $R$, a relation we will call

$$T \subseteq R \subseteq M \times N$$

Notice that this distinction between $T$ and $R$ is one we are introducing for the sake of discussion, not one that appears in [16]: the reader who thinks there should be no such distinction will just insist that $T = R$ in what follows.

Now, what is the force of the uniqueness requirement? There are three possibilities. We start with one which seems legalistically possible but which is clearly too weak to be useful.

1. $(\forall m \in M . \forall n \in N . \forall n' \in N$
   $(T(m,n) \wedge T(m,n') \Rightarrow n = n')) \wedge$
   $(\forall n \in N . \forall m \in M . \forall m' \in M$
   $(T(m,n) \wedge T(m',n) \Rightarrow m = m'))$

   That is, $T$ is a bijection on the subsets of $M$ and $N$ which actually occur in its pairs, but it need not be a bijection on the whole of $M$ and $N$; in fact, it need not even be non-empty.

   Since all our conditions have this two-part form, we will now write only the first and use "and vice versa" to represent the second.

2. $\forall m \in M . \exists ! \, n \in N \, . \, T(m,n)$ and vice versa

   That is, $T$ is a bijection on $M \times N$, although its super-relation $R$ need not be. A tool that constructed $T$ may have made some contentful choices about which bindings to link, from among a set of valid combinations.

   For the avoidance of doubt we may spell out the uniqueness symbol ! thus: $\forall m \in M . \exists n \in N \, . \, (T(m,n) \wedge \forall n' \in N . (T(m,n') \Rightarrow n = n'))$ and vice versa.

3. $\forall m \in M . \exists ! \, n \in N \, . \, R(m,n)$ and vice versa, which we could spell out just as above.

   That is, $R$ is a bijection on $M \times N$; this condition does not mention $T$, but since $T$ is a subrelation of a bijection it will certainly satisfy 1.

We reject 1. because it permits the tool to produce "successful" mappings which are empty despite the presence of valid bindings in the model. At least for the universal quantification, it seems clear that we want to quantify over all valid bindings in a model. For the same reason, condition 3. is only practical if the tool is required to return the whole of $R$, not some arbitrary subrelation. To consider the difference between 2. and the version of 3. with $T = R$, let us return to our examples.

## 3.5 Application to the example

As discussed in Section 3.3, it is unambiguously illegal for a mapping to contain two trace objects that both link to the same valid binding in one model. When the transformation checks the pair of models (Two,One), it cannot return two trace objects, each of which links a model element from Two to the unique model element in One.

The rejected possibility 1. above would correspond to permitting the mapping to return an empty set of trace objects, or more interestingly a single trace object, which links one of the model elements from Two to the unique model element in One. In each of these cases, bindings would be unique within the set of trace objects returned, but we have rejected this interpretation because under it, *any* QVT Core checking transformation would be able to succeed on *any* models.

According to 2., and to 3. with $T = R$, the transformation must return False on (Two,One). Two provides two valid bindings in the domain `m1`, that is, two choices of ModelElement to assign to `anM1element`; One provides only one valid binding in the domain `m2`; sets with different cardinalities cannot be placed in bijection, as both 2. and 3. require.

Next consider the transformation running on (Two,Two). Any pair of valid bindings, one from each model, is actually a valid combination, since our transformation imposes no restrictions. That is, $R$ is not a bijection (it is, rather, the whole of $M \times N$) so if we take interpretation 3. with $T = R$, the transformation must fail. On the other hand, there are subrelations of $R$ which are bijections on $M \times N$. Therefore if we take interpretation 2., in which the tool must construct a bijection $T$ which is a subrelation of $R$ – that is, must construct a set of trace objects which gives a bijection, from among a possibly large set of potential trace objects – the transformation must succeed. If we call the model elements in Two 1 and 2, the tool may return $\{(1,1),(2,2)\}$ or $\{(1,2),(2,1)\}$.

In conclusion, we can find no interpretation of QVT Core which would render the translated transformation semantically equivalent to the original. In all interpretations, the translated transformation will return False on (Two,One), whereas the original QVT-R transformation returned True.

Unfortunately, no implementation of QVT Core seems to be available. Therefore we cannot investigate what actual QVT Core tools do.

Could we write a QVT Core transformation which did have the same behaviour as our simple QVT-R transformation, perhaps by using more complicated bindings? 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).

Therefore, QVT Core cannot express the semantics of our QVT-R transformation, so no translation from QVT-R to QVT Core can be semantics-preserving. Let us leave QVT Core and use the direct semantics for QVT-R given in [16] as our basis.

## 4 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 QVT-R (with simplifications to be explained shortly), 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 [15] 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.[7]

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, following Definition 2, 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 finite set of models $\{m_1, \ldots, m_n\}$, where each $m_i$ conforms to a metamodel $M_i$, and a transformation definition given in a simplified version of QVT-R. Specifically, we say that *when* and *where* clauses are only allowed to contain (conjunctions of lists of) relation invocations, not arbitrary OCL. We do not consider extension or overriding of trans-

[7]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.

formations 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. This parameterisation provides a separation of concerns: it allows us to give a semantics of QVT-R without committing to a particular choice of metamodelling or constraint language semantics.

We will first define a game which corresponds to the evaluation of a QVT-R checkonly transformation in the direction of one of its typed models. If this model is the $k$th, $m_k$, we will call the game $G_k$. For ease of understanding we will explain the progress of the game informally first: Figure 2 defines the positions and the moves of the game more systematically. The player whose turn it is to move is encoded directly as the first element of the current position; Refuter moves from the initial position. 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. As we shall discuss in a moment, we forbid infinite plays, so this completes the elements needed for a formal game definition, as listed in Definition 1.

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 there may be more than one top relation in a given transformation: the fact that Refuter chooses which one to play in, so that in order to have a winning strategy Verifier must be able to defeat him regardless of his choice, corresponds to the requirement that all top relations hold. Notice also 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 to 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? (Notice that even if there are only finitely many possible positions, a play can still be infinite: it would, of course, have to revisit some position infinitely often, and this is exploited in practical work with infinite games. Recall the simulation game example from Section 2, which can have infinite plays even when both processes are finite state.) We may choose just to 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 (which we will refer to as the *when-where* graph), should be acyclic. There is probably[8] 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 [16] has nothing to say about this situation: it corresponds to infinite regress of its definitions. For now, we will forbid infinite plays. One way to achieve this is to declare any QVT-R transformation with a cyclic *when-where* graph to be ill-formed. However, note that even a transformation whose *when-where* graph does contains cycles will often not lead to infinite plays, because constraints on the metamodels will often ensure that following such a cycle can be done only finitely many times, e.g. because it entails moving up or down a metamodel hierarchy. Since this paper is not concerned with metamodelling semantics we will not investigate this further here.

## 4.1 Discussion of the treatment of *when* clauses

Most of the above game definition is immediate from [16], but the treatment of *when* clauses requires discussion. From Chapter 7, ([16], 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."

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 [16], whenever Refuter challenged her 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 pre-condition-like must be more subtle than this. In programming, giving a function a pre-condition makes it easier for the function to 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 sense in which this is pre-condition-like is that the facts about what packages correspond to what schemas are supposed to have been pre-computed: but the order of computation of facts is not something we need to concern ourselves with here, since we are not interested in efficiency but only in meaning.

In trying to settle whether we really mean "a schema" or "the schema" in the paragraph above, we refer again to Appendix B of [16]. 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 the authors of [16] have (between them)

[8]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 $\mu$ calculus model-checking

| Position | Next position | Notes |
|---|---|---|
| Initial | $(\mathsf{Verif.}, R, B, 1)$ | $R$ is any top relation; $B$ comprises valid bindings for all variables from domains other than $k$, and for any *when* variables. $B$ is required to satisfy domain-local constraints on all domains other than $k$. |
| $(P, R, B, 1)$ | $(\overline{P}, R, B', 2)$ | $B'$ comprises $B$ together with bindings for any remaining variables. $B'$ is required to satisfy domain-local constraints on all domains. |
| $(P, R, B, 1)$ | $(\overline{P}, S, C, 1)$ | $S$ 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 $S$. $C$ is required to satisfy domain-local constraints on all domains other than $k$. |
| $(P, R, B, 2)$ | $(\overline{P}, T, D, 1)$ | $T$ 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 $T$. $D$ is required to satisfy domain-local constraints on all domains other than $k$. |

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

two different interpretations of this, perhaps not 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. 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 bijective correspondence between valid bindings is incompatible with Appendix B. This makes the operational view untenable as far as we can see, whereas we shall show that we can follow the purely relational view successfully, remaining compatible with [16]. However, we will shortly (Section 7.4) consider a variant of the game which brings us closer to the operational view, at some cost.

We close this section with an easy lemma which reinforces the special interest of *when* clauses by showing that if they are absent from a transformation, then a Verifier winning strategy for the game has a particularly simple form.

**Lemma 1.** *Let $G_k$ be a game based on a QVT-R transformation which includes no when clauses, and* *for which Verifier has a winning strategy. Let $\sigma$ be a deterministic memoryless winning strategy for Verifier, which is minimal in the sense that $\sigma(u)$ is defined only where $u$ is a position, from which Verifier is to move, which may be reached in a play where Verifier follows $\sigma$. Then every maplet in $\sigma$ is of the form*

$$(\mathit{Verifier}, R, B, 1) \mapsto (\mathit{Refuter}, R, B', 2)$$

*for some relation $R$ and some sets $B$, $B'$ of bindings. That is, the only kind of thing the strategy does is to tell Verifier how to complete a given set of bindings within a given relation of the transformation (row 2 of Table 2): it never instructs Verifier to select a relation invocation from a when or where class (rows 3 or 4).*

*Proof.* Following the initial move, the position is (Verifier, $R, B, 1$) for some $R$ and $B$. Thereafter, since there are no *when* clauses in the transformation, neither player ever has a legal move of the form shown in row 3 of Table 2. All subsequent legal moves must be of the form shown in rows 2 and 4. These moves always swap Verifier for Refuter or vice versa, while at the same time swapping 1 for 2 and vice versa. Therefore every position from which Verifier is to move has a 1 as its final element, so Verifier's only legal moves come from Row 2 of Table 2 and have the form stated. □

## 5 Examples and comparison with QVT-R implementations

In this section we use a family of simple examples to illustrate the game-based semantics. We go on to compare the semantics it implements with what is implemented by ModelMorf.

### 5.1 A family of examples

We use a metamodel which defines just one metaclass, ABoolean, which defines just one boolean value. We denote using $T$, $F$, the simplest two models conforming to this metamodel, each of which defines one ABoolean with value true, false respectively. These models and the metamodel can be downloaded (in a form suitable for ModelMorf) from the author's web page[9]. For convenience we will refer to the single model element in each model as *tt*, *ff* respectively.

To illustrate the effects of *when* and *where* clauses, we will consider the very simple transformations shown in Figures 3 to 6, which all use the same two basic relation definitions but differ in how these are put together using *when* and *where* invocations.

### 5.2 The examples in our semantics

Running each of our example transformations, on each of the four possible pairs of models, in each of the two directions, yields 32 examples. In each case, the semantics must return true or false: true if the two models are considered consistent according to the transformation in the considered direction, false otherwise. In our game-based semantics, the result is, by definition, true if Verifier has a winning strategy for the game, otherwise false. That is, a demonstration that the semantics gives result true on a particular problem *is* a winning strategy for Verifier on the game; a demonstration that the semantics gives result false is a winning strategy for Refuter on the game.

First let us see how this works for PwhereQ run on the pair of models $(T, T)$ in the direction of `m2`. Unsurprisingly the result of this checkonly transformation, according to both our semantics and ModelMorf, is true, which we will demonstrate by exhibiting a winning strategy for Verifier.

Play begins, as always, at the Initial position from which Refuter is to move. Reading off from Figure 2, he has to choose a top relation, and there is only one, viz. SameValue. He also has to choose valid bindings from domains other than `m2`; that is, he must choose an ABoolean $s1$ and its value $i$. Again, he has only one option. There are no *when* variables so he is done; the only possible play so far is

$$\mathit{Initial}, (\mathsf{Verifier}, \text{SameValue}, \{s1 \mapsto \mathit{tt}, i \mapsto \text{true}\}, 1)$$

[9]via `http://homepages.inf.ed.ac.uk/perdita/papers.html`

```
transformation PwhereQ (m1 : BoolMM ; m2 : BoolMM)
{
  top relation SameValue
  {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=i};
    checkonly domain m2 s2:ABoolean {value=i};
    where {FirstIsTrue(s1,s2);}
  }

  relation FirstIsTrue
  {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=true};
    checkonly domain m2 s2:ABoolean {value=i};
  }
}
```

Figure 3: Transformation PwhereQ

```
transformation PwhenQ (m1 : BoolMM ; m2 : BoolMM)
{
  top relation SameValue
  {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=i};
    checkonly domain m2 s2:ABoolean {value=i};
    when {FirstIsTrue(s1,s2);}
  }

  relation FirstIsTrue
  {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=true};
    checkonly domain m2 s2:ABoolean {value=i};
  }
}
```

Figure 4: Transformation PwhenQ

```
transformation QwhereP (m1 : BoolMM ; m2 : BoolMM)
{
  top relation FirstIsTrue
  {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=true};
    checkonly domain m2 s2:ABoolean {value=i};
    where {SameValue(s1,s2);}
  }

  relation SameValue
  {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=i};
    checkonly domain m2 s2:ABoolean {value=i};
  }
}
```

Figure 5: Transformation QwhereP

```
transformation QwhenP (m1 : BoolMM ; m2 : BoolMM)
{
  top relation FirstIsTrue
  {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=true};
    checkonly domain m2 s2:ABoolean {value=i};
    when {SameValue(s1,s2);}
  }

  relation SameValue
  {
    i : Boolean;
    checkonly domain m1 s1:ABoolean {value=i};
    checkonly domain m2 s2:ABoolean {value=i};
  }
}
```

Figure 6: Transformation QwhenP

What are the legal moves from this position? There is no *when* clause so the legal move must come from line 2 of Figure 2: Verifier has to extend the set of bindings to include a binding for $s2$. Notice that she does not choose a binding for $i$, even though it does appear in the `m2` domain clause in the current relation, because $i$ has already been bound by Refuter's choice. She has only one choice for $s2$, so there is only one legal move, and this must be the move that our strategy prescribes. Play continues from the new position

$$(\mathsf{Refuter}, \text{SameValue}, \{s1 \mapsto tt, s2 \mapsto tt, i \mapsto \text{true}\}, 2)$$

We see from the final line of Figure 2 that Refuter now needs to pick a relation invocation from the *where* clause of the current relation – that is, the invocation of FirstIsTrue. The set of bindings in the new position includes the already existing bindings for the root variables of patterns in FirstIsTrue, that is, the bindings $\{s1 \mapsto tt, s2 \mapsto tt\}$ are retained. Notice, however, that these are the only bindings that are retained: the binding of $i$ is discarded at this point. Now Refuter chooses bindings for any variables other than the root in domain `m1`, and for any *when* variables – but there are none, so we simply check that Refuter's trivial "choice" constitutes a valid binding – which it does, since the value in $tt$ is true – and we are done. The new position is

$$(\mathsf{Verifier}, \text{FirstIsTrue}, \{s1 \mapsto tt, s2 \mapsto tt\}, 1)$$

Verifier needs to add a binding for $i$; in our scenario this binding must be to true, although in fact it's unconstrained, giving new position

$$(\mathsf{Refuter}, \text{FirstIsTrue}, \{s1 \mapsto tt, s2 \mapsto tt, i \mapsto \text{true}\}, 2)$$

At this point, Refuter has no legal move, since there is no *where* clause. Therefore Verifier wins this play. Since at no stage did Refuter have any alternative choices that might have proved more successful for him, we can immediately say that Verifier's winning strategy is

$$\begin{aligned} S = \quad & \{(\mathsf{Verifier}, \text{SameValue}, \{s1 \mapsto tt, i \mapsto \text{true}\}, 1) \mapsto \\ & (\mathsf{Refuter}, \text{SameValue}, \{s1 \mapsto tt, s2 \mapsto tt, i \mapsto \text{true}\}, 2), \\ & (\mathsf{Verifier}, \text{FirstIsTrue}, \{s1 \mapsto tt, s2 \mapsto tt\}, 1) \mapsto \\ & (\mathsf{Refuter}, \text{FirstIsTrue}, \{s1 \mapsto tt, s2 \mapsto tt, i \mapsto \text{true}\}, 2)\} \end{aligned}$$

In general, of course, a tool might have to search for a winning strategy, and, whilst a depth-first exploration of play will certainly work for games with no infinite plays, much more efficient searches might be possible. The advantage of the game-based semantics, from the point of view of understandability, is that it does not matter how the strategy is found. Given a strategy, it is easy (both cognitively and computationally) to check that it is a winning strategy, which is what is required.

Next let us consider an example containing a *when* clause: PwhenQ run on (F,T) in the direction of `m2`. This proves even simpler. Play must begin with Refuter moving from Initial to $(\mathsf{Verifier}, \text{SameValue}, \{s1 \mapsto ff, s2 \mapsto tt, i \mapsto \text{false}\}, 1)$. From here, Verifier has no legal move. She cannot use line 2 of Figure 2, because (even though she has no values to bind) she would be required to ensure the satisfaction of the domain-local constraint that the value in $tt$ was false, which of course she cannot do. She cannot use line 3, challenging the *when* clause, because she would be required to ensure the satisfaction of the domain-local constraint that the value in $ff$ was true, which again she cannot do. Therefore, since it is Verifier's turn to move and she cannot do so, Refuter wins the play. Refuter has a winning strategy which consists simply of moving from Initial to the only possible position.

Now let us go on to summarise the results on the whole family of examples. Figure 7 shows the results when the transformations are checked in the direction of `m1`. Figure 8 shows the results when the transformations are checked in the direction of `m2`.

## 5.3 Comparison with ModelMorf

Running the set of examples discussed on this section in ModelMorf yields agreement in 31 of the 32 cases.

| $\leftarrow$ | (T,T) | (T,F) | (F,T) | (F,F) |
|---|---|---|---|---|
| PwhereQ | $\mathcal{V}$ | $\mathcal{R}$ | $\mathcal{R}$ | $\mathcal{R}$ |
| PwhenQ | $\mathcal{V}$ | $\mathcal{R}$ | $\mathcal{V}$ | $\mathcal{V}$ |
| QwhereP | $\mathcal{V}$ | $\mathcal{R}$ | $\mathcal{R}$ | $\mathcal{R}$ |
| QwhenP | $\mathcal{V}$ | $\mathcal{V}$ | $\mathcal{V}$ | $\mathcal{R}$ |

Figure 7: Results in the direction of m1. $\mathcal{V}$ indicates that Verifier has a winning strategy, that is, that the result of the transformation is True; $\mathcal{R}$ similarly that the result is False.

| $\rightarrow$ | (T,T) | (T,F) | (F,T) | (F,F) |
|---|---|---|---|---|
| PwhereQ | $\mathcal{V}$ | $\mathcal{R}$ | $\mathcal{R}$ | $\mathcal{V}$ |
| PwhenQ | $\mathcal{V}$ | $\mathcal{R}$ | **R** | $\mathcal{V}$ |
| QwhereP | $\mathcal{V}$ | $\mathcal{R}$ | $\mathcal{V}$ | $\mathcal{V}$ |
| QwhenP | $\mathcal{V}$ | $\mathcal{V}$ | $\mathcal{V}$ | $\mathcal{V}$ |

Figure 8: Results in the direction of m2

The exception is PwhenQ run on (F,T) in the direction of `m2`, where our semantics gives false in contrast to ModelMorf's true. On the author's reporting this to Sreedhar Reddy (one of those at TATA who has been closely involved with ModelMorf and who is also an author of [16]) he confirmed that false is the correct answer[17], so that this was a bug in ModelMorf.

We have already demonstrated the application of the game-based semantics in this case (see above). Expanding the definitions in [16] shows that the result should be true iff

`FirstIsTrue(s1,s2)` $\Rightarrow$ `s1.value = s2.value`

To evaluate this we need to know what `FirstIsTrue(s1,s2)` means. Unfortunately, [16] does not discuss how to evaluate such an expression – it only defines relations without parameters. That definition is of the form "for all valid bindings... there exists a valid binding such that... conditions". A sensible interpretation of the parameterised case would therefore seem to be that for all valid *completions* of `s1` to a valid binding (recalling that a valid binding has to satisfy domain-local constraints), there should exist a valid completion of `s2` to a valid binding, such that the conditions hold on those bindings. In other words, specifying values for certain variables restricts the scope of the bindings that need to be considered in the definition; some decisions have already been made. This seems consistent with the intention of [16].

In this case, there is no way to complete `s1` to a valid binding, since the definition of FirstIsTrue insists that the value in `s1` should be true, which it is not. Therefore, the universally quantified formula is vacuously true, so the implication

`FirstIsTrue(s1,s2)` $\Rightarrow$ `s1.value = s2.value`

is false, explaining why the final result of the checking transformation is false, and consistent with the answer produced by our game-based semantics.

It is encouraging that, on all the cases we have explored with the exception on the one case which has been confirmed to be a bug in ModelMorf, our semantics gives the same answers as ModelMorf. This suggests that our semantics has resolved ambiguities in [16] in a way compatible with the way the authors of QVT intended (since several of the same people are involved in both the document and the tool). Thus the game-based semantics may be useful as a way of explaining the intended meaning of QVT-R transformations, and perhaps of exploring further possibilities such as debugging tools, without needing to argue for a different meaning of transformations.

## 6 Duality

An intriguing aspect of the QVT-R language is that it seems that *when* and *where* clauses are in a certain sense dual. As far as we are aware, however, there are no proven results on this subject in the literature. In this section we show how the game-based semantics can help to access provably correct statements along these lines, and we give an example.

Inspection of the definition of the moves of the game $G_k$ as shown in Figure 2 shows that it is only in moving from Initial that we need to specify a player (Verifier or Refuter) by name: in every other kind of move, we simply swap or preserve the player, without needing to know whether we started with a Verifier or a Refuter position. (We may also note that the Initial move is also the only place in the game definition where it matters whether a relation is or is not designated a top relation.) Moreover, since our game definition and strategies are memoryless, it makes sense to talk about a winning strategy from a given position, not only from the Initial position. Immediate from this is the following observation:

**Lemma 2.** *Fix a game $G_k$. Let $P$ be either Verifier or Refuter, and let $\overline{P}$ be the other player. Independently, let $A$ be either Verifier or Refuter, and let $\overline{A}$ be the other player, and let $i$ be either 1 or 2. Then $P$ has a winning strategy starting from position $(A, Q, B, i)$ iff $\overline{P}$ has a winning strategy starting from position $(\overline{A}, Q, B, i)$.*

*Proof.* The same strategy will work in both cases. More precisely, suppose we are given a winning strategy for $P$ from $(A, Q, B, i)$, that is, a partial map $S$ from positions to positions which satisfies the conditions to be a winning strategy from $(A, Q, B, i)$. Construct a new partial map $\overline{S}$ from positions to position by replacing $A$ by $\overline{A}$ and vice versa wherever they occur. By duality of the game rules, $\overline{S}$ is a winning strategy for $\overline{P}$ from $(\overline{A}, Q, B, i)$. □

This lemma can be used to compose what we know about different parts of a game graph. Here is an example deliberately constructed to have a simple proof, intended to serve merely as an example of the proof technique:

**Proposition 1.** *Fix a set $m_i$ of models. Consider two transformations, $T$ and $T'$, which differ only in their definitions of one relation, their unique top relation, which is not invoked by any other relation; let $T$'s unique top relation be $P$ while $T'$'s is $P'$. In $P$, there is a when clause that simply invokes relation $Q$ with arguments $s_i$, and there is no where clause. In $P'$, there is a where clause that simply invokes $Q$ with the same arguments, and there is no when clause. Moreover, $P$, $P'$ and $Q$ satisfy the following conditions with respect to the models $m_i$:*

1. *in each of relations $P$, $P'$, there is a unique choice of valid bindings for the variables in domains other than $m_k$ (satisfying the domain-local constraints of domains other than $m_k$) and for the arguments to $Q$, and these bindings assign the same values to the arguments of $Q$;*
2. *in $P$ there is* no *valid binding of the variables in domain $m_k$ that, together with the unique choice of valid binding for the other variables, also satisfies the domain-local constraints on $m_k$;*
3. *in $P'$ there* is *a valid binding of the variables in domain $m_k$ that, together with the unique choice of valid binding for the other variables, also satisfies the domain-local constraints on $m_k$;*
4. *in $Q$ there is a unique choice of valid bindings for the variables in domains other than $m_k$ (satisfying the domain-local constraints of domains other than $m_k$).*

*Then the checkonly transformation $T$ run on the set $m_i$ of models in the direction of $m_k$ returns true iff the transformation $T'$ run on the same set of models in the same direction returns false.*

*Proof.* The effect of the properties insisted on is to ensure that, from Initial, play in the $T$ game in can only proceed to $(\mathsf{Refuter}, Q, B', 1)$, where $B'$ is the unique possible set of bindings, and similarly play in the $T'$ game can only proceed to $(\mathsf{Verifier}, Q, B', 1)$. From this point, Lemma 2 gives the result, since the reachable portions of the game graph are indistinguishable from those points. □

Notice that although we have imposed very stringent conditions on the relations $P$, $P'$, $Q$, here, it is permitted that $Q$ invoke other relations that can be arbitrarily complex. For a concrete example, take $T$ to be PwhenQ from Figure 4, run in direction `m2`, the models to be $(F, T)$, and $T'$ to be a variant that replaces the *when* clause by a *where* clause and imposes no constraints in the top relation.

Very informally, we may say that this result captures the observation that the transformation "$P$ where $Q$" is equivalent to the negation of "$\overline{P}$ when $Q$" where $P$ and $\overline{P}$ are opposites in a suitable sense such as the one imposed by the conditions above. Of course many variants on this result are possible: we have presented a particularly simple case for purposes of exposition. For example, it is not necessary to insist that there should be a unique set of valid bindings in each place where we did so, provided that care is taken to insist that choices and the players who choose them match up appropriately. Nevertheless, the need to take care over these aspects intuitively explains why no really general duality result seems to hold. We cannot offer any very general result for translating transformations using *when* into equivalent transformations using *where* instead, independent of the models to which the transformation is to be applied. More practical experience with QVT-R will be required to see what what examples might actually be interesting, for purposes of efficient implementation or otherwise.

# 7 Variants of the game

One of the advantages of the game-based approach to defining semantics is that it provides an intuitive means of examining the design decisions which have been made in choosing one semantics over another. In this section, we examine some alternatives.

## 7.1 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 [16]. 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. In order to explain this, we need a digression on trace objects in QVT-R and how they relate to the game-based semantics.

## 7.2 Trace objects and the game-based semantics

Because QVT-R, as already stated, does not depend on the definition of any trace objects, the way in which trace objects are generated when a QVT-R transformation is executed is not prescribed in [16]. A reasonable initial assumption that a trace object for QVT-R is similar to the trace objects which are described in detail for QVT Core, so that there is one trace class per relation in the transformation, with attributes corresponding to the variables in each domain pattern of the relation. Since, unlike checking transformations in QVT Core, even QVT-R checkonly transformations have a direction, we might wonder whether a QVT-R trace object should record which domain is the target domain, or whether this is unnecessary. In this subsection we will investigate this question and the connection between trace objects and Verifier winning strategies.

Consider a checkonly transformation $S$ run on a set of models $\{m_i\}$ in the direction of $m_k$. Let the directional game based on this transformation and these models be $G_k$ as usual.

Suppose a QVT-R transformation engine finds that the transformation succeeds and returns a trace object whose attribute values link a collection of valid bindings for the various domains of a particular relation $R$. What can we infer by looking at the valid bindings that are linked by the trace object? Surely, in some suitable sense, the valid bindings are known to "match" according to $S$, but the QVT standard does not say precisely what one should be able to rely on, given the existence of such a trace object. Obviously the bindings recorded in the trace object must be type correct and must satisfy any $R$-local constraints such as equalities between them; but more than this, we expect them to match in the sense that proceeding through the transformation, e.g. by invoking a relation from $R$'s *where* clause, and following the transformation wherever it may lead, will not expose inconsistencies, breaking the match between the linked valid bindings. This leads naturally to:

**Definition 3.** *Let $t$ be a trace object for relation $R$ in transformation $S$ as above, and let $B'$ be the com-*

*plete set of bindings recorded in $t$. Then $t$ is* correct *for $S$ in direction $k$ if the recorded set of bindings $B'$ satisfies all the type constraints and domain-local constraints in $R$, and further, Verifier has a winning strategy in $G_k$ starting from $(Refuter, R, B', 2)$.*

(Recall the notion of a winning strategy starting from a given position from Section 6.)

Verifier can safely use such a correct trace object to guide her play, as follows: if she is challenged in relation $R$ with some of the bindings from $B'$, she can safely read off the remaining bindings from $B'$ and use them in her response. Formally if play has reached any (legal) position $(\text{Verifier}, R, B, 1)$ where $B \subseteq B'$, she may legally move to $(\text{Refuter}, R, B', 2)$, and she will have a winning strategy from that position. (This trace object, of course, does not tell her what that winning strategy is.)

We chose to use $B'$ as the name for the binding set here for consistency with the next definition: any Verifier winning strategy for $G_k$ gives rise to a set of trace objects:

**Definition 4.** *Let $\sigma$ be any winning strategy for Verifier on the game $G_k$ defined from transformation $S$ on models $m_i$. Then* the set of trace objects arising from $\sigma$ *is as follows. For each maplet in $\sigma$ that defines a move of the form shown in Row 2 of Table 2, that is, which is*

$$(\mathit{Verifier}, R, B, 1) \mapsto (\mathit{Refuter}, R, B', 2)$$

*for some relation $R$ and sets of valid bindings $B$, $B'$, we define a trace object in the trace class for $R$, whose attributes are given values according to $B'$.*

**Lemma 3.** *All the trace objects arising from a winning strategy are correct.*

*Proof.* Because a winning strategy must prescribe only legal moves, we have immediately that the recorded set of bindings $B'$ satisfies all the type constraints and domain-local constraints in the relation. Moreover, because the strategy is winning, Verifier must a fortiori have a winning strategy from every position that her strategy tells her to go to. So Verifier has a winning strategy from $(\text{Refuter}, R, B', 2)$ where $B'$ is the set of bindings taken from a trace object for $R$ arising from a winning strategy. $\square$

Notice that not all of the information in the winning strategy has been recorded in the trace objects. Trace objects can tell Verifier how to choose matching bindings when she needs to do so, and in practice this is probably the most valuable part of the transformation engine's work to save. Any maplets in the strategy that prescribed moves from Rows 3 or 4 of Table 2, however, have been discarded. The trace objects do not tell Verifier under what circumstances to challenge a relation from a *when* or *where* clause. Thus a set of trace objects is not in general a complete witness to the success of a checkonly transformation: even given a complete set of trace objects, some computation is required to use them to reconstruct why the transformation succeeded. Let us formalise the notion of a complete set of trace objects.

**Definition 5.** *Let $S$ be a checkonly transformation run on models $\{m_i\}$ in direction $k$, and let $G_k$ be the corresponding directional game. Let $\mathcal{T}$ be a set of correct trace objects. Then $\mathcal{T}$ is* complete *if there exists some Verifier winning strategy $\sigma$ for $G_k$ such that $\mathcal{T}$ is the set of trace objects arising from $\sigma$.*

In particular, *if no top relation in $S$ has a when clause*, then a complete set of trace objects must include a trace object for each combination of valid bindings of domains other than $k$ of top relations, and this trace object will give "matching" bindings of the domain $k$ variables. This is because Refuter's initial move might challenge with any such combination of valid bindings, so any winning strategy must be able to meet each such challenge. In general, Verifier's response to an initial challenge can use a move from Row 2 (match the bindings) or a move from Row 3 (challenge a relation from the *when* clause); if we rule out *when* clauses, only the first remains, and these moves give rise to trace objects.

Note that there is no requirement for such linkage to be bijective: several trace objects for the same or different relations may link to the same model element in $m_k$.

Note in passing that it is in principle possible for a transformation engine to determine that a winning strategy for Verifier exists without actually calculating one. Therefore it is not inevitable that evaluating a checkonly transformation on a set of models (and returning true or false) involves generating a set of trace objects.

So far, there is an inherent direction to our trace objects: from tuples of model elements in models other than $m_k$, towards model elements in $m_k$. Next let us consider whether it is really necessary for QVT-R trace objects to have (whether implicitly or explicitly recorded) a direction, that is, a distinguished target domain. We saw in Section 3 that QVT Core trace objects do not. Suppose we have a checkonly QVT-R transformation and two models which are consistent in both directions. Verifier has a winning strategy in each direction, so we know how to construct a complete set of trace objects for each direction. Of course, Verifier might have many different winning strategies for the game in each direction, giving rise to different sets of trace objects. Therefore, we cannot expect that if we have an arbitrary pair of Verifier winning strategies, one in each direction, they will give rise to the same set of trace objects. However, might we be able to arrange, by careful design of our transformation engine that searches for winning strategies, that the same set of trace objects would arise from both strategies? The trace objects in such a set would be bidirectional: they could safely be read in either direction, since each trace object is correct in both directions, having arisen both from a winning strategy in one direction and from a winning strategy in the other direction. It is not unreasonable to hope that by picking "compatible" winning strategies in the two directions this might be possible. A pair of winning strategies for the two directional games $G_1$ and $G_2$ is, of course, the same thing as a winning strategy for the non-directional game $G$, so we are asking whether a single set of trace objects can serve for both halves of a winning strategy for $G$.

We will answer the question in the negative using an example derived from one in [10]. To do this, we need to demonstrate a QVT-R transformation and a pair of models, such that the transformation returns true in both directions, and yet where *no* pair of winning strategies for the directional game can give rise to the same set of trace objects in each direction. We will do this by showing that, in one direction, any winning strategy must give rise to a particular trace object, while in the other direction, no winning strategy can give rise to that trace object. Thus, if we wish to have trace objects corresponding to success of the transformation in both directions, we will have to keep two separate sets: we cannot have a single complete set of bidirectional trace objects.

Figure 9 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 [16], 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 there must be a trace object that takes `a1` to `xa`, etc.; we must have a trace object that links `a1` to something and there is nothing else it can do. That is, *any* winning strategy for Verifier in the `m1` direction must give rise to a trace object which links `a1` to `xa`, because when Refuter challenges in ContainersMatch by binding `a1` to `c2`,

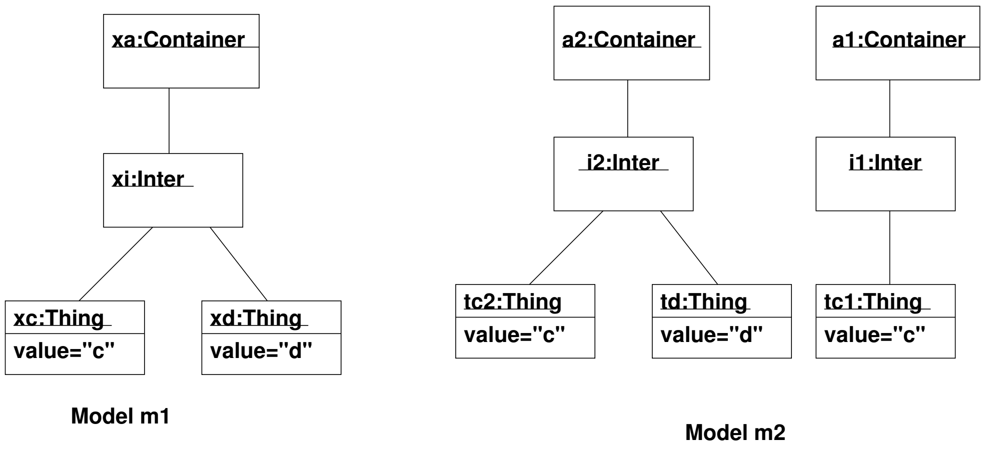

Figure 9: m1 and m2 are (two-way) consistent according to QVT-R transformation Sim, but no set of bidirectional trace objects can link them

Verifier has no other legal move than to respond by binding `xa` to `c1`.

Yet in the `m2` direction, a trace object which took `xa` to `a1` would be incorrect. If, when Refuter challenged in ContainersMatch by binding `xa` to `c1`, Verifier responded according to this trace object, by binding `a1` to `c2`, she would lose: Refuter could drive play to `xd` which she cannot match.

To put this another way, the position

$(\text{Refuter}, \text{ContainersMatch},$

$\{\texttt{c1} \mapsto \texttt{xa}, \texttt{c2} \mapsto \texttt{a1}, \texttt{inter1} \mapsto \texttt{xi}, \texttt{inter2} \mapsto \texttt{i1}\}, 2)$

is a legal and reachable position in both $G_1$ and $G_2$. In $G_1$, Verifier has a winning strategy from this position, and in fact any winning strategy for this game must tell her to reach this position. In $G_2$, however, Refuter has a winning strategy from this position.

It might perhaps be objected that in using sets of trace objects that are complete, we have been too stringent. However, if we assume that "a set of trace objects" must include at least matches for valid combinations of bindings in domains other than $k$ in top relations, as discussed above, this is enough to make this counterexample work. This seems like a minimal condition on a set of trace objects for us to consider that it captured the correctness of a transformation. For example, we certainly do not want to count, as a set of bidirectional trace objects capturing the correctness of a transformation, the set that might result from the following "cheating" procedure: a transformation engine just generated a set of trace objects in each direction, and then returned the intersection of the two sets (whether or not it was empty, or in any sense complete for the transformation).

In conclusion *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.*

## 7.3 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: it gives a tighter connection between models, while still permitting non-bijective transformations. 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. Formally, looking at the positions and moves in Figure 2, we would simply need to modify the positions of the form $(....,1)$, by adding an additional integer element $k$ specifying in which domain the challenge is to be answered, that is, which domain may not have all its bindings chosen yet. The legal moves that result in such positions would have to specify that the player making the move has to choose which domain that shall be.

Having made this modification to the game, what is the effect semantically? A winning strategy for Verifier in the game $G'$ can still be regarded as determining a set of trace objects, as before. In this sense bidirectional trace objects will exist in $G'$. However, the price may be that Verifier too seldom *has* a winning strategy: this corresponds to the observation that for many practical purposes in concurrency, bisimulation equivalence proves to be too strong an equivalence. 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`.

### 7.4 Trace-based variant

Let $G_k^T$ be the variant of $G_k$ 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. 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 [16] and having curtailed expressivity. Several variants are possible: for example, if $P$ has been matched by $S$, 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.

## 8 Related work: research and tools

We will first describe several strands of related research papers; in Section 8.1 we will describe tool support for QVT.

We will not attempt to survey the field of bidirectional model transformations in general, let alone bidirectional programming. A recent multidisciplinary report on the state of the art is [5]; the present author's [19] also discusses the approaches most closely related to model transformation. Particularly interesting is the work of the Harmony group such as [8], which supports non-bijective bidirectional transformations within a carefully specified, compositional and tool-supported framework.

Turning to the related work that formalises QVT-R semantics, two frequently-occurring features are notable.

First, 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. As discussed in Section 2, when used bidirectionally, such an approach works only for those bidirectional transformations which are actually bijective. We cannot use the "trick" discussed in Section 2.1 of constructing an interpretation of checkonly transformations by saying that a transformation returns True exactly if it would not have modified the target model, because such transformations do not even look at the target model. We will call the approach the unidirectional approach to QVT-R. Of course QVT-R can be used as a unidirectional transformation language; it might be chosen, for example, because a declarative language was desired, rather than for its bidirectional support. This paper's concern, however, is to investigate more general bidirectional transformations than just the bijective ones, even at the cost of restricting the scope of the work in other ways, especially restricting (for now) to checkonly transformations.

Second, there is remarkably little discussion of how semantics have been validated. The development of complicated translations into complex target languages such as Coloured Petri Nets is error-prone, and besides, one might expect the developer of such a translation to come across issues with the interpretation of [16] such as those discussed here. In doing the work for this paper, it proved invaluable to investigate what the evolving semantics did on many different QVT-R transformations, some of which have been presented here. By contrast, most of the papers we shall discuss mention only one example of a QVT-R transformation, generally some simplified version of the UML to RDBMS transformation from [16]. This makes it hard to investigate possible disagreements between them. Perhaps it is also why noone before the present author appears to have found the contradiction between the direct semantics of QVT-R given in [16] and the translation to QVT Core.

Greenyer and Kindler [12]'s main contribution is a translation from QVT Core to Triple Graph Grammars (TGGs). TGGs are a reasonably close semantic match for QVT Core, although there are some problems which have to be handled, such as that TGGs have a single correspondence graph in place of QVT Core's separate trace class for each relation. The QVT Core semantics given seems to agree with the conclusions of Section 3 about how to interpret QVT Core's "uniqueness" requirement. The paper also contains an interesting informal discussion of some issues that arise when trying to extend the approach to QVT-R, although it does not provide a semantics for QVT-R. There is, however, no discussion of the implications of the fact that QVT-R checkonly translations have a direction. Given the inherent symmetry of the TGG correspondence graph, we conjecture that this approach, if completed, would probably fail to handle the example of our Section 7.2 naturally.

Romeikat and others [18] translated QVT-R transformations to QVT Operational. Their focus is on unidirectional use of QVT-R, in which a source model is transformed into a target model without reference to an earlier version of the target model. Update transformations are briefly discussed, and the authors correctly point out that it is not sufficient to rely simply on `key` declarations, and mention some possible approaches. The implementation does not cover more than the unidirectional approach: the authors write "The checkonly mode of Relations is not supported. For each binding of the root variable of the source domain, only one binding in the target domain is allowed."

Garcia [9] formalised aspects of QVT-R in Alloy, permitting certain well-formedness errors to be detected, but not providing a semantics of QVT-R.

Several authors have worked on giving QVT-R semantics in terms of coloured Petri nets (CPNs). These have a token game which is, at first sight, an attractive match for the flow of control through a transformation; relations are translated into transitions of the net and places represent metaclasses. Although fairly popular for modelling concurrent systems, CPNs are relatively complex and their use in this work becomes much more so when they are adjusted to deal with issues such as the absence of negative conditions for firing transitions.

de Lara and Guerra in [6], for example, propose to translate QVT-R into coloured Petri nets, and thus provide it with a semantics in a way which can take advantage of existing tools and techniques for CPN. Again, their focus is unidirectional; the paper describes an idea of how to handle check before enforce semantics, which the authors conflate with the use of keys, but not enough information is given to allow the reader to infer what their approach would do on examples other than the one given. They do not discuss the validation of their translation into CPN, which is apparently based on the QVT-R to QVT Core translation. This work does benefit from the existence of a mature CPN tool, which is combined to good effect with the Medini QVT parser, but the implementation does not include their idea for CBE semantics. The authors point out that the use of CPN tools enables them to investigate questions traditional in the Petri-net based modelling community, given in Petri net terms, such as whether places are bounded and whether transformations are confluent (deterministic). This enables them to identify restrictions that they need to impose on the class of transitions handled; as usual, permitting arbitrary OCL in transformations, as technically allowed in QVT-R, is not sensible for bidirectional transformations.

Another paper in this tradition is [26] 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.

A different approach is to giving semantics to QVT-R is to use algebraic specification, as exemplified by [3], which describes the MOMENT-QVT tool which will be discussed in the next subsection. This work, too, addressed only the unidirectional use of QVT-R.

In this paper we have, by considering checkonly transformations, focused on the use of QVT-R to check consistency between models, where a specific relevant notion of consistency is embodied in each transformation. Although we have chosen to focus on checkonly transformations more because understanding their semantics is a pre-requisite to understanding the semantics of the QVT-R language in general, than because of any special interest in consistency checking per se, we should mention other work in this area. For example, Egyed in [7] discusses how to check for common types on inconsistency in UML models specifically, with a focus on efficiency. A recent thorough survey of approaches to consistency management in UML models is [14]. In fact, this paper proposes using QVT-R to capture the different notions of consistency used by various authors. It does not address the semantics of QVT-R itself; instead, the specifications actually used are written in a language called lQVT-Maude ("like-QVT Maude") which incorporates certain features of QVT.

On a slightly different note, [4] discusses how to deduce OCL constraints from a QVT-R (or Triple Graph Grammar) specification, as an aid to increasing confidence in the correctness of the transformation. The approach is promising; undoubtedly, we will need connections between model transformations and simpler, limited specifications of them such as these constraints. For each relation in a QVT-R specification certain key constraints can be derived, having the "for all-there exists" form we would expect, and these should correspond naturally to statements that Verifier has a winning strategy from a particular reachable game position. However, a weakness of the work is a lack of precision in the definition of the supporting constraints that say when a relation is "enabled". For top relations, we are just told (Definition 9 of [4]) that these "are derived following the same procedure described for TGG rules"; however, it is not clear to the present author how to adjust that procedure for QVT. For non-top relations, where the question is more interesting, the paper appears to lack any definition (it is used in Definition 10 but not defined there). There is an example (the usual UML to RDBMS) but this is not enough to enable a closer comparison.

There is a vast literature on the application of games to various logic-related problems in theoretical computer science: a good survey is [11]. The most strand most relevant to this paper is surveyed in [23].

A connection which might be interesting to explore further is that with games for refinement, such as [2]. In this work a system is viewed as a collection of agents, obeying contracts. Properties of the system and refinement possibilities are investigated by partitioning the agents into a set of friendly ("angelic") agents and a complementary set of unfriendly ("demonic") agents, and investigating under what circumstances the angelic group has a winning strategy for a game which is defined so as to capture the achievement of a goal of interest. As with the present work, the game presentation supports an intuition understanding of the situation while being fully rigorous. On the other hand, the structuring of the systems involved and the nature of behaviour appears to be quite different from what we have in model transformations, so this would require investigation.

In modelling, the GUIDE tool [24] uses games to support design exploration and verification. The game itself can be modified as a design model and its specification co-evolve.

## 8.1 QVT tools

There are two main tools that aim to support QVT Relations. These are ModelMorf from TATA Consultancy, and Medini QVT from IKV++.

ModelMorf is a command-line tool that takes models, metamodels and QVT transformations as file arguments. It is used by TATA consultancy and is made available for academic use[10]. Although TATA uses the tool internally, it currently has no plans to make the tool available commercially [1]. TATA, in the persons of Sreedhar Reddy and R. Venkatesh, has contributed to the development of the QVT standard, and it is therefore not surprising to find that ModelMorf conforms closely to the published specification. It is therefore the main tool to which we refer in this work.

Medini QVT, due to its greater visibility on the Web, Eclipse integration, debugging facilities and other developer-friendly features, has attracted rather more attention than ModelMorf. Unfortunately for us, although understandably, it has not taken faithfulness to the QVT specification as a major design goal, and its semantics are very significantly divergent from both the QVT specification and ModelMorf. For example, as explained in IKV++'s slides [13], Medini QVT deliberately uses deletion differently from either [16] or ModelMorf.

Medini QVT also significantly restricts the class of bidirectional transformations on which it expects to behave reasonably, compared with [16] and ModelMorf. A representative quotation from its documentation[11] is:

> However to achieve meaningful bidirectionality, the relations of a transformation must be left and right unique !!. To explain this with our example, consider a model containing only one package with name MyPackage. If umlToRdbms is executed in direction rdbms thereby creating a schema, then executing it back in the uml direction should map to exactly the same MyPackage, irrespective of whether it was executed in the context of traces or not !!.

In mathematics, a relation $T \subseteq A \times B$ is left unique if for every $a \in A$ there is exactly one $b \in B$ such that $T(a, b)$ holds; thus, a relation is left and right unique exactly when it is a bijection. In this case, the bijection is presumably between valid bindings of domain patterns. Because relations occur in the contexts of other relations – QVT-R relations are not the same as mathematical relations – there is still some ambiguity. For example, if in the UML to RDBMS example, there are two tables with the same name as a given class, but only one of them is in the schema of the class's package – or alternatively, only one of them will permit AttributeToColumn to be satisfied – does this violate the condition or not? Most likely, in terms of our games, the intended restriction is that in moves of the form shown in Row 2 of Table 2, Verifier should only ever have one valid move available to her, so that her strategy will not in fact need to record what choice she should make. Although this is reminiscent of the discussion in Section 3 of the restriction on how valid bindings are linked in QVT Core, there is, as discussed in that section, no corresponding restriction in [16] for QVT-R; imposing one restricts the range of transformations that can be expressed. As a trivial example, we saw in Section 3, this restriction makes it impossible to express the transformation in Figure 1. Realistic examples are also easy to imagine. For example, suppose a testing tool offers the facility to synchronise a collection of objects representing unit tests with a module diagram of the system under test. Suppose each test object records the name of the module it tests as the value of an attribute. Suppose the consistency condition is that there is to be at least one test for every module, and that all module names occurring in test objects should actually be modules of the system. Unless we modified the test objects' metamodel, which might be impossible or undesirable, the QVT-R relation we wrote would not be bijective: in the direction of the test objects, it would match a module with any test for that mod-

[10] from `http://121.241.184.234:8000/ModelMorf/ModelMorf.htm`

[11] Medini QVT Guide, available as online help from the RCP version of Medini QVT, version 1.6, section QVT Relations Language, Bidirectionality

ule, and there could be many. (However, requiring each relation in a transformation to be bijective does *not*, we should state clearly, force the transformation as a whole to be bijective.)

A further observation is that when a QVT-R transformation is run in Medini QVT, the order in which the relations are written in the transformation file is significant. Reordering relations sometimes causes Medini QVT to give different results when the transformation is run. (This is anecdotal: I have observed the effect and understand that others have too, but I am not sure precisely why, or under what circumstances, Medini QVT behaves differently with different relation orderings.) In [16], however, order of relations is not semantically significant (indeed, there is an equivalent graphical form for transformations in which it may not even be possible to see what order a transformation writer used), and in ModelMorf I have not observed order to matter.

Besides these semantic considerations, Medini QVT does not provide a "checkonly" mode for its transformations: if two models are not consistent according to the transformation, it is not possible to cause Medini QVT to say so without modifying the target model[12] In summary, despite its relative popularity and the likelihood that it is useful in practice, Medini QVT is best regarded as a tool which does not implement the OMG language QVT-R, but implements an alternative semantics for the same syntax.

MOMENT-QVT was a research prototype tool, based on an algebraic specification approach using Maude, described in [3]. It provided partial support for QVT-R, but this did not include support for checkonly transformations. Like so much of the research mentioned on QVT-R, it only supported enforce transformations in which a target model is produced from some source models without reference to a previous version of the target model; that is, it is in effect a unidirectional approach.

As this discussion shows, the current landscape of QVT-R tools is not such as to encourage a belief that the language has a bright future. ModelMorf, the only tool which is currently available and faithful to the OMG standard, is neither open source nor commercially marketed. Perhaps this is not surprising, given the issues that still exist concerning the semantics of the language, particularly when used bidirectionally. The fact that the Medini QVT transformation engine is available under an open source licence means that it would be possible to create and distribute a semantically modified version, however, even if IKV++ prefers to retain their own semantics.

There does not seem ever to have been a serious implementation of the QVT Core language. Various sources refer to a pre-release of Compuware OptimalJ, but OptimalJ no longer exists.

The QVT Operations language is not relevant to this paper, so we will not discuss its tool support.

## 9 Conclusions

We have presented a game-theoretic semantics of a subset of QVT-R checkonly transformations, based on the direct semantics in [16]. This semantics, although well short of a complete semantics for QVT-R, nevertheless goes beyond the state of the art, by carefully specifying the meaning of checkonly transformations which need not be bijective nor consist of bijective relations. This generality is important in MDD, in which the actual target model depends on decisions made by the people concerned with that model, and not just on the source model. As we have discussed, as soon as a bidirectional transformation admits more than one possible model which is consistent with a given model, it becomes unacceptable to model it simply as a pair (or set) of functions, each taking one or more source models and producing a fresh target model without taking into account the pre-transformation state of the target model. It is essential that, as in [16], the execution of a transformation involves examination of all relevant models, even though only one may be modified.

Formalising this is challenging, but any formalisation will necessarily embody a formal definition of consistency: after the application of an enforcement transformation, the models must be consistent according to the transformation (in the terminology of

[12]At the time or writing this is Ticket 12 against the engine, dated 2008-03-25.

[21], the transformation must be *correct*), and an enforce transformation will make no changes if and only if the models are consistent according to the transformation (the transformation must be *hippocratic*). Understanding checkonly transformations, therefore, is prerequisite to understanding general transformations, and this is the main contribution of this paper.

We justified our choice to ignore the translation to QVT Core by pointing out a fundamental incompatibility between the two languages. We have demonstrated that QVT-R can express a transformation, whose meaning is unambiguously defined by the direct semantics of QVT-R, yet which cannot be expressed in QVT Core. This remains true even when we consider as wide as possible a range of interpretations of the QVT Core semantics; in the course of our unsuccessful attempt to reconcile the two languages, we discussed three different interpretations of QVT Core's uniqueness requirements for bindings. Therefore no translation from QVT-R to QVT Core can be semantics-preserving. In particular, the translation given in [16] is not semantics-preserving and should not be used as part of the definition of QVT-R.

We have compared the semantics for checkonly QVT-R defined here with that embodied in ModelMorf, and showed that the two are consistent; we have discussed the fact that Medini QVT uses a different, non-standard semantics. We have briefly discussed variants of the game, demonstrating in the process that bidirectional trace objects may not exist. This provides an interesting contrast between QVT-R and triple graph grammars, with their single bidirectional correspondence graph.

Although we have focused specifically on QVT-R, this is also, as far as we are aware, the first time that a game approach has been used to give semantics to a model transformation language. We have demonstrated that this is feasible and that it permits reasoning about properties of the language.

**Acknowledgements** I would like to thank the referees, and the developers of the QVT-R tools mentioned for answering my various enquiries.

## References

[1] TATA Consultancy Services Asha Rajbhoj. Personal communication, 25th May 2010.

[2] Ralph-Johan Back and Joakim von Wright. Contracts, games, and refinement. *Information and Computation*, 156(1-2):25 – 45, 2000.

[3] Artur Boronat, José A. Carsí, and Isidro Ramos. Algebraic specification of a model transformation engine. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2006.

[4] 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.

[5] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. grace meeting notes, state of the art, and outlook. In Richard F. Paige, editor, *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer, 2009.

[6] Juan de Lara and Esther Guerra. Formal support for qvt-relations with coloured petri nets. In Andy Schürr and Bran Selic, editors, *Proceedings of MODELS'09*, volume 5795 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2009.

[7] Alexander Egyed. Instant consistency checking for the uml. In *In: Proceeding of the 28th In-*

*ternational Conference on Software Engineering*, pages 381–390, 2006.

[8] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007.

[9] 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.

[10] 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.

[11] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, LNCS **2500**. Springer, 2002.

[12] 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.

[13] Jörg Kiegeland and Hajo Eichler. medini QVT workshop. `http://projects.ikv.de/qvt/downloads/22`, February 2008. Slides presented by ikv++ technologies ag at the QVT Workshop in Enschede, Telematica Instituut.

[14] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of uml model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

[15] Donald A. Martin. Borel determinacy. *Annals of Mathematics. Second series*, 102(2):363–371, 1975.

[16] OMG. MOF2.0 query/view/transformation (QVT) version 1.1. OMG document formal/2009-12-05, 2009. available from www.omg.org.

[17] Sreedhar Reddy. Personal communication, 26th November 2009.

[18] 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.

[19] 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.

[20] 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.

[21] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. *Journal of Software and Systems Modeling (SoSyM)*, 2009. to appear.

[22] Perdita Stevens. A simple game-theoretic approach to checkonly QVT Relations. In *Proceedings of the International Conference on Model Transformations, ICMT'09*, Lecture Notes in Computer Science. Springer, June 2009.

[23] 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.

[24] Jennifer Tenzer and Perdita Stevens. GUIDE: Games with UML for interactive design exploration. *Journal of Knowledge Based Systems*, 20(7), October 2007.

[25] Manuel Wimmer, Gerti Kappel, Johannes Schönböck, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. A petri net based debugging environment for qvt relations. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 3–14. IEEE Computer Society, 2009.

[26] Manuel Wimmer, Angelika Kusel, Johannes Schoenboeck, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reviving qvt relations: Model-based debugging using colored petri nets. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 727–732, Berlin, Heidelberg, 2009. Springer-Verlag.