SoSyM manuscript No.
(will be inserted by the editor)

Bidirectional model transformations in QVT: semantic issues and open questions

Perdita Stevens[⋆]

School of Informatics, University of Edinburgh

December 9, 2009

Abstract We consider the OMG's Queries, Views and Transformations (QVT) standard as applied to the specification of bidirectional transformations between models. We discuss what is meant by bidirectional transformations, and the model-driven development scenarios in which they are needed. We analyse the fundamental requirements on tools which support such transformations, and discuss some semantic issues which arise. In particular, we show that any transformation language sufficient to the needs of model-driven development would have to be able to express non-bijective transformations. We argue that a considerable amount of basic research is needed before suitable tools will be fully realisable, and suggest directions for this future research.

Keywords: bidirectional model transformation, QVT, model-driven development, semantics

1 Introduction

The central idea of the OMG's Model Driven Architecture is that human intelligence should be used to develop models, not programs. The main intention behind this manifesto is not to privilege a graphical over a textual representation, but rather to press for the developer to be enabled to work at as high a level of abstraction as is feasible. Routine work should be, as far as possible, delegated to tools, and the human developer's intelligence should be used to do what tools cannot. To this end, it is envisaged that a single platform independent model (PIM) might be created and transformed, automatically, into various platform specific models (PSMs) by the systematic application of understanding concerning how applications are

⋆ Email: perdita@inf.ed.ac.uk. Fax: +44 131 651 1426

best implemented on each specific platform. The OMG's Queries, Views and Transformations (QVT) standard [13] defines languages in which such transformations can be written.

In this paper we will discuss *bidirectional* transformations, focusing on basic requirements which such transformations should satisfy.

This paper is an extended version of the conference paper [14] presented in MODELS'07. Compared with the conference version, the present paper adds a more extensive discussion of the requirements that should be placed on bidirectional transformations; an example to show that correctness and hippocraticness alone do not suffice; and a discussion of the argument against as well as for Undoability. We newly discuss the expressivity of QVT Relations for writing non-bijective transformations. We provide a fuller comparison of our proposed framework with the lens approach of Pierce et al. We have also expanded the explanation of several points that have proved to have been less clear than they could have been. Finally, we include proofs of all formal results.

The structure of the paper is as follows. In the remainder of this section, we motivate bidirectional transformation, and especially, the need for non-bijective bidirectional transformations; we then discuss related work. Section 2 briefly summarises the most relevant aspects of the QVT standard. Section 3 discusses key semantic issues that arise. Section 4 proposes and motivates a framework and a definition of "coherent transformation". Finally Section 5 concludes.

In order to justify the considerable cost of developing a model transformation, it should ideally be reused; perhaps a vendor might sell the same transformation to many customers. However, in practice a transformation will usually have to be adapted to the needs of a particular application. Similarly, whilst we might like to argue that only the PIM would ever need to be modified during development, with model transformation being treated like compilation, the transformed model never being directly edited, nevertheless in practice it will be necessary for developers to modify both the source and the target models of a transformation and propagate changes in both directions. The interesting albeit unfinished document [17] makes these and other points, emphasising especially that bidirectional transformations are a key user requirement on QVT, and that ease of use of the transformation language is another key requirement.

Even in circumstances where it is in principle possible to make every change in a single source model, and roll changes down to target models by reapplying unidirectional transformations, in practice this is not desirable for a number of reasons. A non-technical reason is that different developers are familiar with different models, and even different modelling languages. Developers are less likely to make mistakes if they change models they are comfortable with. A technical reason is that some changes are most simply expressed in the vocabulary, or with respect to the structure, of one model. For example, a single change to one model might correspond, semantically, to a family of related changes in the other.

Given the need for transformations to be applied in both directions, there are two possible approaches: write two separate transformations in any convenient language, one in each direction and ensure "by hand" that they are consistent, or use a language in which one expression can be read as a transformation in either direction. The second is very much preferable, because the checking required to ensure consistency of two separate transformations is hard, error-prone, and likely to cause a maintenance problem in which one direction is updated and the other not, leaving them inconsistent. The QVT Relations language (hereinafter QVT-R) takes this second approach: a transformation written in QVT-R is intended to be able to be read as a specification of a relation which should hold between two models, or as a transformation function in either direction.

1.1 Bidirectional versus bijective transformations

A point which is vital for the reader to understand is that bidirectional transformations need not be bijective. A transformation between metamodels M and N given by a relation R is *bijective* if for every model m conforming to M there exists exactly one model n conforming to N such that m and n are related by R, and vice versa (for every model n conforming to N there exists exactly one ...). This is an important special case because there is then no choice about what the transformation must do: given a source model, it must return the unique target model which is correctly related to the source. Ideally, the developer writing a bijective transformation does not have to concern herself with how models should be transformed: it should suffice to specify the relation, which will in fact be a bijective function. (In practice, depending on exactly how the relation is expressed, it might be far from trivial for a tool to extract the functions from the relation, however.) Modulo information encoded in the transformation itself, both source and target models contain exactly the same information; they just present it differently.

The classic example in the QVT standard [13] of transformation between a UML class diagram and a database schema is a case where both models contain almost (but not quite) the same information. Therefore it happens not to be a clear illustration of the fact that a language permitting only bijective transformations is inadequate to express all the transformations that MDD may require. More realistically we might express the requirement as the synchronisation of a full UML model, including dynamic diagrams, with a database schema, which makes it obvious that there will be many UML models which might be related to a given schema. More generally, bijective transformations are the exception rather than the rule: the fact that one model contains information not represented in the other is part of the reason for having separate models.

The QVT standard [13] is somewhat ambivalent about whether it intends all bidirectional QVT transformations to be bijective. On the one hand,

the requirements of MDD clearly imply that it should be possible to write non-bijective transformations (see also [17]): for example, in general, the development of a PSM will involve the addition, and preservation in the face of changes to the PIM, of information concerning design decisions on a particular platform. On the other hand, the standard does not explicitly point out that there may be a choice of consistent model to be made, or discuss the implications of this fact. Moreover, it is unfortunately easy to mis-read the QVT-R definition in such a way that it would be technically a consequence that all "valid" QVT-R transformations would be bijective, as we will show below. For the potential of MDD to be realised, the need for non-bijective transformations needs to be clearly understood, and the implications for language design addressed.

1.2 Related work

In the context of model transformations, almost all formal work on bidirectional transformations is based on graph grammars, especially triple graph grammars (TGGs) as introduced by Schürr (see, for example, [8]). Indeed, the definition of the QVT core language was clearly influenced by TGGs. Greenyer and Kindler in [5] (following Greenyer's master's thesis [4]) studies the relationship between QVT (chiefly QVT core) and TGGs, defining a translation from (a simplified subset of) QVT core into a version of TGGs that can be input into a TGG tool. More broadly, the field of model transformations using graph transformation is very active, with several groups working and tools implemented. We mention in particular [9,16]. Most recently, the paper [2] by Ehrig et al. addresses questions about the circumstances in which a set of TGG rules can indeed be used for forward and backward transformations which are information preserving in a certain technical sense. It is future work to relate our approach to TGGs.

In this context, it may seem foolhardy to write a paper which approaches semantic issues in bidirectional model transformations from first principles. However, there is currently a wide gap between what is desired for the success of MDD and what is known to be soundly supportable by graph transformations; the use of QVT-style bidirectional transformations has not spread fast, despite the early availability of a few tools, partly (we think) because of uncertainty among users over fundamental semantic issues; and moreover, there is a large body of quite different work from which we may hope to gain important insights. Here we give a few pointers. For a survey of the available approaches to bidirectional transformations (i.e. not focusing on QVT), and a discussion of research issues to be addressed in future, especially the engineering issues, see [15].

Benjamin Pierce and colleagues in the Harmony group have explored bidirectional transformations extensively in the context of trees [3], and more recently in the context of relational databases [1]. In their framework, a *lens*, or bidirectional transformation, is a pair of functions (a "get" function

and a "putback" function) which are required to satisfy certain properties to ensure their coherence. They define a number of primitive lenses, and combinators with which to build more complex lenses. Thus, they define a programming language operating on trees in which a program can be read either as a forwards transformation or as a backwards one. Coherence of the forward and backward readings of the program follows from properties of the primitives and combinators. Their framework is asymmetric, however: their forward transformation is always a function on the source model only, which, in conjunction with their coherence conditions, implies that the target model is always an abstraction of the source model: it contains less information. This has advantages and disadvantages. It is insufficiently flexible to serve as a framework for MDA-style model transformations in general, but the restriction permits certain constructs, especially composition, to work in a way which is not possible in the more general setting. We shall return to this work later in the paper.

Bidirectional programming languages have been developed in various areas, and a survey can be found in [3]. Notably Lambert Meertens' paper [10] addresses the question of developing "constraint maintainers" for use in user interface design, but his approach is far more general. His maintainers are essentially model transformations which, in terms we shall introduce below, are required to be *correct* and *hippocratic*, but not *undoable*. In [7], Kawanaka and Hosoya develop a bidirectional programming language for XML. In Tokyo, Masato Takeichi and colleagues Shin-Cheng Mu and Zhenjiang Hu have also worked extensively on an algebraic approach to bidirectional programming: see [11,12,6].

2 QVT

The OMG's Queries, Views and Transformations (QVT) standard [13] addresses a family of related problems which arise in tool-supported model driven development. Not all information which is modelled is relevant at any one time, so there is a need to be able to abstract out the useful information; and models need to be held in meaningful relationships with one another. Provided that we permit non-bijective transformations (required to support model views), transformations subsume views.

The QVT standard describes three languages for transformations: the Operational, Relations and Core languages. The Relations language is the most relevant here. In the Operational language, someone wishing to specify a bidirectional transformation would have to write a pair of transformations and make them consistent by hand, which we have already said is undesirable. QVT Core is a low level language into which the others can be translated; an example translation from QVT Relations to QVT Core is given in the standard. Since we are concerned with transformations as expressed by users, we will work with QVT Relations (QVT-R).

A QVT-R transformation embodies a definition of what it means for a source model to be correctly related to a target model, that is, it embod-

ies a consistency relation. QVT transformations are given a "check then enforce" semantics which means that a transformation must not modify a target model if it is already correctly related to the source model. This is pragmatically very important, to avoid confusing the user with apparently unmotivated changes. However, the combination of this rule with the possibility of a non-bijective consistency relation has important implications which tool developers and others need to appreciate.

For example, suppose that $R \subseteq M \times N$ is a non-bijective consistency relation. If the transformation in the target direction is to be given by the one-argument function $f : M \to N$ and that in the source direction is to be given by $g : N \to M$, then it is impossible for the transformation to respect the "check then enforce" semantics. This is easy to see: without loss of generality, suppose that $(m, n) \in R$ and $(m, n') \in R$, with $n \neq n'$. Running the transformation in the target direction should not modify either pair of models, since in both cases the target model is already correctly related to the source model. However, $f(m)$ cannot take both values n and n'. One might protest that the situation of running the transformation on models which are already consistent is artificial. However, the same problem will arise when applying a transformation to a pair of models which is inconsistent, if there is more than one way in which consistency could be restored.

A passage which the author originally read as indicating that QVT transformations should be able to be expressed as one-argument functions is on page 18 of [13]; it reads:

> In relations, the effect of propagating a change from a source model to a target model is semantically equivalent to executing the entire transformation afresh in the direction of the target model.

In fact[1] this is probably not what the passage intends. It concerns, instead, what should happen in the following situation: (m, n) is a consistent pair of models (perhaps held on different computers), and then m is modified to m_δ by making a change to it which a tool records as δ. Rather than "executing the entire transformation afresh" – which may involve expensive traversal of large models, and/or transporting a large model over a network – it is desirable for a tool to be able to propagate just the change, in effect applying a record of change δ to model n to get n_δ consistent with m_δ. The real force of the passage quoted is then that n_δ must be the same model[2] as would be obtained by directly synchronising m_δ with n.

This scenario too is trickier if the consistency relation is allowed to be non-bijective than if it is not. Let us illustrate this with an informal example, as follows. A UML model is considered to be consistent with a test-set iff the test-set contains at least one test for every class which *both* appears as an active class in a class diagram of the UML model *and* has a state diagram there. Change propagation can now only work properly for certain, carefully

[1] as pointed out by a referee to whom I am grateful

[2] or at least, the same up to some suitable notion of "semantic equivalence"

chosen, notions of what a "change" is. Suppose that, in a certain tool, one form that a record of change can take (parametrised on Z and C) is "added state Z to the state diagram for classifier C, creating such a diagram if it did not already exist". Now change records of this form cannot systematically be propagated without violating the "check then enforce" semantics.

To see this, consider two models m and m', both consistent with the same model n. They are the same except for one difference: in m, class C is an active class in the class diagram, but it has no state diagram; in m', class C appears as a non-active class, but it does have a state diagram. Since in neither m nor m' are *both* conditions for class C to need a test in n satisfied, n need not contain any test for C; let us suppose that it does not. Now suppose that a change record δ of the form just described is applied to each model. Although δ changes a state diagram in m', it makes no difference to m''s consistency with n: class C still does not need a test, because it still does not appear as an active class in the class diagram. Let m'_δ be the result of applying change δ to model m'. We have seen that m'_δ is still consistent with n. According to "check then enforce" semantics, therefore, propagating the change δ to n *must not* cause any change to n, because by "check then enforce", executing the entire transformation in the target direction on the already-consistent pair (m'_δ, n) would leave n unchanged. On the other hand, applying the very same change δ to model m will give a model m_δ which is inconsistent with n, because the class C now satisfies both criteria to need a test, but has none. Executing the entire transformation afresh on the inconsistent pair of models (m_δ, n) would cause a change to n, and therefore, propagating the change δ to n *must* cause a change to n. This is a contradiction.

This problem lies, of course, in the form of the change record that this hypothetical tool permitted. We could envisage alternatives, such as recording in the change whether or not the classifier concerned already had a state diagram, so that the change propagation mechanism could use that information to act accordingly. More generally, this kind of problem could always be avoided by adding more information to the change records to be propagated – ultimately, the "change record" could include the entire new version of the model, in which case it is clear that this kind of problem cannot arise. However, it is not obvious to a tool designer reading [13] that they will need to choose the form of their change records changes so carefully, and it is not clear to this author how to give a general rule by which to determine whether a proposed change record is safe.

3 Semantic issues

The previous section made the general point that it is important to realise that there are far-reaching implications of allowing non-bijective bidirectional transformations, and that it would be better if the QVT standard made this more explicit. In this section we raise a variety of other issues

which we consider to need further study: they are settled neither by the QVT standard, nor as far as we know by existing related work.

3.1 What exactly it makes sense to write

The QVT-R language is designed to be easy for someone familiar with related OMG standards such as OCL to learn and use; this has clearly been a higher design goal than ensuring that only safe transformations can be written. There are several places (when and where clauses, among others) where arbitrary OCL expressions are permitted, even though only certain expressions make sense as part of a bidirectional transformation. For example, a transformation may in one direction give an attribute a value using an non-invertible expression.[3] Specifying exactly what language subset is permitted, however, is likely to run quickly into a familiar problem: that any reasonably easy-to-define language subset which is provably safe will also exclude many safe expressions of the full language. It may well be preferable to be permissive, and rely on users not to choose to write things that don't make sense. They will, however, require a clear understanding of what it means for a transformation to "make sense". In Section 4 we propose first steps in this direction and give simple postulates which, we argue, any bidirectional model transformation should obey.

3.2 Determining validity of a transformation

Let us suppose that the reader and the developer accept that model transformations will be written in an expressive, unsafe language, but that they also accept that the transformations written should obey certain postulates. (These might be the ones proposed in this paper, or others; the general point is the same.) Since we lack a transformation language in which any transformation is guaranteed to satisfy the postulates, this has to be verified, formally or informally, on a case-by-case basis. How can developers become confident that their transformations do indeed obey these postulates? Ideally, the language and the postulates should be so clearly understandable that the developers can be confident in their intuition: tool support is no substitute for this kind of clarity, since however easy it is to correct an error once it has been made, it is better not to make the error in the first place. However, it is also desirable that a tool should be able to check, given the text of a transformation and the metamodels to which it applies, that it obeys the postulates. That is, this transformation should be able to be verified statically at the time of writing it, as opposed to failing when it is

[3] Note that permitting non-bijective transformations does not make this unproblematic: since transformations are to be deterministic, where there are several relationally possible choices of value the language needs to provide a way to specify which should be chosen.

applied to arguments which expose a problem. To what extent this can be done – for the postulates discussed in this paper or any others – is an open question.

A major danger with bidirectional transformations is that one direction of the transformation may be a seldom used but very important "safety net". This is why we would like to have a systematic, even if incomplete, way to verify that a given transformation is correct (including, for example, that it obeys agreed "sanity" postulates). For the most frequently used direction of a bidirectional transformation, the ordinary process of trying to use the transformation might suffice; the other direction, however, might not be exercised enough to find its bugs. It will be unfortunate if the user only finds out that their transformation cannot be executed in the less usual direction long after the transformation has been written, in circumstances where the reverse transformation is really needed...

3.3 Composition of relations in QVT: when *and* where *clauses*

Most of this paper takes a high level view of transformations, in which a whole transformation text specifies a relation and a pair of transformational functions. We have not yet considered the details of how simpler relations are combined and built up into transformations in QVT. This is interesting, however, and not least because it gives another justification for considering non-bijective transformations. A QVT relational transformation has an overall structure something like this:

```
transformation ... {
top relation R {
  domain a...
  domain b...
  when {...}
  where {...}
}
top relation S ...
relation ...
relation ...
...
}
```

In order to understand when and where clauses, note that [13] uses two different notions of a relation holding. Let us explain this in terms of the running example from [13]. The reader does not need to be familiar with the example: it suffices to know that it concerns a bidirectional transformation `umlRdbms` between UML models and relational database schemas, and includes a top relation `ClassToTable`, which explains how classes in a UML model are related to tables in a database schema.

At the top level, a relation holds of a pair of models – checking will return TRUE – if they are consistent. For example, the top relation `ClassToTable`

will hold of UML model `m` and RDBMS model `s` if, as far as the classes and tables go, there is no inconsistency. If models `m` and `s` are consistent according to relation `ClassToTable`, we will write $\texttt{ClassToTable}^{+}\texttt{(m,s)}$.

The + is intended to distinguish this notion from the following: the top-level consistency between `m` and `s` is *demonstrated* by matching individual classes in `m` to individual tables in `s`. That is, the text of the transformation `ClassToTable` explains under what circumstances an individual table `t` can be considered to correspond to an individual class `c`. (More formally, it is the valid bindings of domain variables in the text of `ClassToTable` which are related, but that is not important here.) This correspondence is a relation which relates the set of classes to the set of tables, and the relation is defined by the text of `ClassToTable`. We will write `ClassToTable(c,t)` if `c` corresponds to `t`.

Note that the relation on models $\texttt{ClassToTable}^{+}$ is a lifted version of the relation on bindings `ClassToTable`: a UML model is related to an RDBMS model by $\texttt{ClassToTable}^{+}$ iff for every class there is a table related to it by `ClassToTable` and vice versa.

The when and where clauses can contain arbitrary OCL, but are typically expected to contain (if anything) statements about relations satisfied by variables of the domain patterns. Thus in fact, the relation `R` holds if for every valid match of the first domain, there exists a valid match of the second domain *such that the where clause holds*. The when clause "specifies the conditions under which the relationship needs to hold". The example used in the standard is the relation `ClassToTable` with domains binding `c:Class` (and hence `p:Package` etc.) and `t:Table` (and hence `s:Schema` etc.), the when clause being `PackageToSchema(p,s)` and the where clause being `AttributeToColumn(c,t)`.

Now, what does this mean in relational terms, and specifically, what is the difference between the when clause and the where clause, both of which appear at first sight to impose extra conditions on valid matches of bindings, thus forming an intersection of relations? Unfortunately, this is not straightforward to express relationally. Operationally, the idea is that the variables in the when clause "are already bound" "at the time this relation is invoked". Roughly speaking, when a relation `ClassToTable` has domain patterns with variables including `p : Package` and `s : Schema`, and a when clause which states `PackageToSchema(p,s)`, the QVT engine is supposed to have already processed the `PackageToSchema` relation (if not, it will postpone consideration of the `ClassToTable` relation). The matchings calculated for `PackageToSchema` provide bindings for variables `p` and `s` in `ClassToTable`. Evaluation of `ClassToTable` now proceeds, looking for compatible valid bindings of all the other variables.

We have sketched the operational view of what happens in one example, but an open problem is to give a clean compositional account of even the relation (let alone the transformation) defined by a whole QVT transformation. Making this precise would involve a full definition of R^{+} taking account of when and where clauses, and an account of the relationship between

properties of R and properties of R^+. As an example of the complications introduced by dependencies between relations, suppose that there are two ways of matching pairs of valid bindings (skolem functions) for one relation, one of which leads to a compatible matching for a later-considered relation and one of which does not. If a QVT engine picks "the wrong" matching for the first relation considered, is it permitted to return the result that the models are inconsistent, even though a different choice by the tool would have given a different result? Surely not: but then there is a danger that the tool will need to do global constraint solving or arbitrarily deep backtracking to ensure that it is not missing a solution. Not only is this inefficient, but it will be very hard for the human user to understand. Now, looking at the examples in [13], it seems clear that this kind of problem is not supposed to arise, because when clauses are used in very restricted circumstances. However, it is an open question what can be permitted, and we can expect to encounter the usual problem of balancing expressivity against safety.

For a simple example of "spatial" composition of relations where we *can* lift good properties of simple relations to good properties of a more complex relation, see the next section.

3.4 Sequential composition of transformations

We have discussed the ways in which relations are composed in QVT to make up transformations. A different issue is the sequential composition of whole transformations.

Throughout this paper, we envisage a bidirectional QVT tool which does not retain information between uses: it simply expects to be given a pair of models, a transformation, and a command telling it in which direction to apply the transformation and whether to check or enforce. If the tool is allowed to retain trace information – the correspondence graph in TGG terms – between executions, the composition problem becomes more tractable. This, however, is a severe pragmatic limitation: for example, it does not extend well to situations in which models are being used and modified in multiple tools, not only the trace-aware transformation tool. See [15] for further discussion.

Model-driven development envisages chains of transformations, some of which may be purchased, others written in-house. For example, a scenario envisaged as essential by [17] is that an organisation purchases a PIM-to-PSM transformation, but needs to modify it to meet their specific requirements, e.g. by post-composing with a relatively simple transformation which expresses how this organisation's particular needs differ from what is implemented in the purchased transformation.

Thus we naturally expect to be able to define a transformation to be the sequential composition of two other transformations, and then treat the composition as a first-class transformation in its own right. In this case, the pair of models given to the tool will be the source of the first transformation

and the target of the second: the tool will not necessarily receive a version of the intermediate model, the one which acts as target of the first transformation and source of the second. In order to define a general way to compose transformations, we need to suppose that we are given transformations R from M to N and S from N to P and show how to construct a composed transformation $T = R; S$, giving the relational and functional parts of the composed relation in terms of the parts of the constituent relations.

We will return to this issue in the next section, after introducing appropriate notation.

4 Requirements for bidirectional model transformations

In this section we discuss bidirectional model transformations which are not necessarily bijective, and discuss under what circumstances these will make sense. We will give postulates which are clearly satisfied by bijective transformations, but also by certain non-bijective transformations.

4.1 Basic requirements

First let us set some basic notation and terminology. Much of this work concerns sets of models, and in model-driven development a metamodel is often used as a convenient way to specify a set of models (those that conform to the metamodel). Since that is the audience we principally address, we shall use the term "metamodel" wherever a model-driven developer would expect to see it, and will make some a few remarks that do refer to the content of a metamodel. However, in reading the technical work the reader may consider "metamodel" to be synonymous with "set of models": any other way of specifying a set of models would work as well. We will use capital letters such as R, S, T for the relations which transformations are supposed to ensure. That is, if M and N are metamodels (sets of models) to be related by a model transformation, the relation $R \subseteq M \times N$ holds of a pair of models – and we write $R(m, n)$ – if and only if the pair of models is deemed to be consistent. As well as this consistency relation, the transformation developer will also need, somehow, to specify the two directional transformations:

$$\overrightarrow{R} : M \times N \longrightarrow N$$

$$\overleftarrow{R} : M \times N \longrightarrow M$$

The idea is that $\overrightarrow{R}$ looks at a pair of models (m, n) and works out how to modify n so as to enforce the relation R: it returns the modified version. Similarly, $\overleftarrow{R}$ propagates changes in the opposite direction.

In practical terms, what we expect is that the programmer writes a single text (or set of diagrams) defining the transformation in QVT-R (or indeed,

in another appropriate language). This same text can be read in three ways: as a definition of a relation which must hold between pairs of models; as a "forward" transformation which explains how to modify the right-hand model so as to make it relate to the left-hand model; as a "backward" transformation which explains how to modify the left-hand model so as to make it relate to the right-hand model. By slight abuse of notation, we will use capital letters R, S etc. to refer to the whole transformation, including both transformational functions as well as the relation itself, when no confusion can result.

Our notation already incorporates some assumptions, or rather assertions, which need justification.

First, and most importantly, that the behaviour of a transformation should be deterministic, so that modelling it by a mathematical function is appropriate. The same transformation, given the same pair of models, should always return the same proposed modification. This is a strong condition: it proscribes, for example, transformation texts being interpreted differently by different tools. An alternative approach, which we reject, would have been to permit a tool to modify the target model by turning it into *any* model which is related to the source model by the relation encoded in the transformation. There are several good reasons to reject that approach. Most crucially, the model transformation does not take place in isolation but in the presence of the rest of the development process. Even though certain aspects of one model may be irrelevant to users of the other – so that the transformation will deliberately abstract away those aspects – this does not imply that the abstracted away aspects are not important to other users! Usually, it will be unacceptable for a tool to "invent information" in any way, e.g. by making the choice of which related model to choose. The developer needs full control of what the transformation does. Even in rare cases where certain aspects of the transformation's behaviour (say, the choice of name for a newly created model element) might be thought of as unimportant, we claim that determinism is necessary in order to ensure, first, that developers will find tool behaviour predictable, and second, that organisations will not be unacceptably "locked in" to the tool they first use. Experience shows that even when a choice is arbitrary, people find it important that the way the arbitrary choice is made be consistent. One example of this is the finding that, even though the spatial layout of UML diagrams does not (generally) carry semantic information, it is important for UML tools to preserve the information.

Our second assertion is that the behaviour of a transformation may reasonably depend on the current value of the target model which will be replaced, as well as on the source model. This follows from our argument in Section 1 that restricting bidirectional transformations to be bijective is too restrictive. Of course, the fact that we choose a formalism which permits the behaviour of a transformation to depend on both arguments does not force it to do so. In the special case of a bijective transformation, the result of $\overrightarrow{R}$ may be independent of its second argument, and the result

of $\overleftarrow{R}$ independent of its first argument. Another important special case is when the transformation in one direction uses only one of its arguments, while the reverse transformation uses both. Pierce et al.'s lenses fall into this category, and we will discuss how they fit into this framework below.

A technical point is that we require transformations to be total, in the sense that $\overrightarrow{R}$ and $\overleftarrow{R}$ are total functions, defined on the whole of $M \times N$. We may want to define, for a metamodel M, a distinguished "content-free" model ϵ_M to be used as a dummy argument e.g. in the case that a target model is created afresh from a source model. Note that since the model containing no model elements might not conform to M, ϵ_M might not literally be empty.

Correctness Our notation is chosen to suggest that the job of $\overrightarrow{R}$ and $\overleftarrow{R}$ is to enforce the relation R, and our first postulates state that they actually do this. We will say that a transformation T is *correct* if

$$\forall m \in M\ \forall n \in N \qquad T(m, \overrightarrow{T}(m,n))$$

$$\forall m \in M\ \forall n \in N \qquad T(\overleftarrow{T}(m,n), n)$$

These postulates clearly have to be satisfied by any QVT-like transformation.

Hippocraticness, or "check-then-enforce" The QVT standard very clearly states that a QVT transformation must not modify either of a pair of models if they are already in the specified relation. That is, even if models n_1 and n_2 are both properly related to m by R, it is not acceptable for $\overrightarrow{R}$, given pair (m, n_1), to return n_2. Formally, we say that a transformation is *hippocratic*[4] if for all $m \in M$ and $n \in N$, we have

$$T(m,n) \implies \overrightarrow{T}(m,n) = n$$

$$T(m,n) \implies \overleftarrow{T}(m,n) = m$$

These postulates imply that if the relation T is *not* bijective, then (at least one of) the transformations must look at both arguments. As a consequence, applying a transformation to a source model in the presence of an existing target model will not in general be equivalent to applying it in the presence of an empty target model.

This is as expected. Imagine, for example, that we generate an initial PSM automatically from a PIM. Subsequently, work is done on the PSM that is irrelevant to the PIM (that is, does not make the modified PSM inconsistent with the original PIM). Next, a modification is made to the PIM, and the transformation is applied to roll the changes through to the

[4] First, do no harm. Hippocrates, 450-355BC

PSM and restore consistency. The PSM developers would be very unhappy if the transformation took no account of the work they had done in the meantime. (Of course, depending on the change to the PIM, it may or may not have been invalidated: this is part of the challenge of writing such transformations.)

Are these requirements enough? Correctness and hippocraticness are the only two high-level requirements clearly placed on QVT transformations in the specification [13] (although, of course, that specification also defines the language, so providing high-level requirements for it is an optional extra). Are they sufficient requirements on a bidirectional model transformation language to ensure sane behaviour?

Here is a pathological example to show that it may be worth investigating further conditions. We give it in natural language to illustrate that it is a general issue, independent of the particular languages used.

Let M be a set of models in which a model consists of a set of elements representing composers, each having a `name` and `dates`.

Let N be a set of models in which a model consists of a set of elements, also representing composers, each having `name` and `nationality`.

Let R be the obvious consistency relation: m and n are consistent iff the same names occur in each.

Let $\overrightarrow{R}(m, n)$ have the following behaviour. If $R(m, n)$ then return n (i.e., make no modification, so as to be hippocratic). Otherwise, create n' by:

1. deleting any objects in n whose names do not occur as names of objects in m;
2. identifying any names that occur in m but not n, and adding new objects with those names;
3. set the nationality field of all objects in n' (not just the new ones) to "Lithuanian".

Similarly, let $\overleftarrow{R}(m, n)$ return m if m and n are already consistent, and otherwise, return a model with the correct set of names but with all dates set to some arbitrary value.

This transformation is correct, because the consistency relation does not mention the dates or nationality fields. It is hippocratic by construction. However, it is intuitively deeply unsatisfactory. Somehow we expect something more fine-grained than hippocraticness: we expect that applying a transformation will not modify any information unless it is somehow necessary to do so to restore consistency. How can we capture this intuition? This is harder than it seems, because although it is easy to come up with individual transformations that do not seem sensible, it is generally also possible to construct scenarios in which technically similar behaviour would be justifiable. For example, we would not want to insist that a transformation must never modify some collection of elements of a model if modifying a smaller collection would also restore consistency.

Notice that the QVT specification states ([13],) "the semantics of check-before-enforce ensures that target model elements that satisfy the relations

are not touched." However, to make this statement precise we would need to have a precise notion of what it means for a model element, as opposed to the model it is part of, to satisfy a relation. This brings us back to the discussion in Section 3.3.

4.2 Further requirements

In the conference paper [14], we proposed Undoability as part of the definition of coherent transformation, mentioning but not making explicit the counterargument, that this postulate might be too strong. It still seems important enough to be part of the main coherence definition, but let us now give both sides of the argument.

Undoability Our final pair of postulates is motivated by thinking about the following scenario. The developer, beginning with a consistent pair of models m (the source) and n (the target, perhaps produced by a model transformation), makes a modification to the source model, producing m', and propagates it using the model transformation tool (so that target model n is replaced by $\overrightarrow{T}(m', n)$). Immediately, without making any other changes to either model, our developer realises that the modification was a mistake. She reverts the modified model to the original version m, and propagates the change. The developer reasonably expects that the effect of the modification has been completely undone: just as the modified model has been returned to its original state m, so has the target model been returned to its original state n.

Formally, we will say that transformation T is *undoable* if for all $m, m' \in M$ and $n, n' \in N$, we have

$$T(m,n) \implies \overrightarrow{T}(m, \overrightarrow{T}(m', n)) = n$$

$$T(m,n) \implies \overleftarrow{T}(\overleftarrow{T}(m, n'), n) = m$$

Note that our pathological example in Section 4.1 above is not undoable: the result of making a change and then immediately undoing it will be to replace the dates or nationality values with the arbitrary choice given in the transformation.

The trouble with undoability as a general requirement on transformations is that it is too strong. Consider a transformation between models comprising sets of `Composer`s each with a `name` and `dates`, and models comprising sets of `Compositor`s each with a `nome` and `nacionalidade`. In the natural way, suppose that two models are supposed to be consistent if they have the same set of `name`s (`nome`s: we are using a mixture of English and Portuguese just to make it clearer which model is referred to in what follows.)

Going back to our initial scenario, suppose the change the developer made was to *delete* some information from m to get m'. (For example, perhaps she deleted a `Composer` with `name` "Sibelius" and `dates` "1865-1957".) When she applied the transformation to propagate the change, any "corresponding information" from n was deleted, yielding n'. (A `Compositor` with `nome` "Sibelius" was deleted.) But also deleted was any information which was "stuck" to that information in n, even if it wasn't represented in m. (The `nacionalidade` "Finnish", was deleted along with the rest of the `Compositor`.) So when the developer reverted to m, and propagated the change back, the transformation can restore the information which is contained in m (for example, it can recreate a `Compositor` with `nome` "Sibelius"), but it may not be able to recreate all the information that had been deleted ("Finnish" has been lost). This lost information may have to be replaced with "default values", so that the final model is not exactly the same as at the beginning.

This seems to be a fundamental difficulty with bidirectional transformations. The above counter-argument is quite convincing: to forbid such non-undoable transformations altogether will be a severe restriction. On the other hand, a model transformation which did not allow one's changes to be undone in this way would be quite confusing. It may be that a pragmatic solution to deal with the scenario we began with would be implemented at the level of the interactive tool, which might retain extra information during a modelling session to provide editor-style "undo" capabilities separate from the model transformations.

Whether or not we wish to impose undoability, it is useful to have a term for transformations that satisfy all the conditions we have so far considered.

Definition 1 *Let R be a transformation between metamodels M and N, consisting of a relation $R \subseteq M \times N$ and transformation functions $\overrightarrow{R} : M \times N \longrightarrow N$ and $\overleftarrow{R} : M \times N \longrightarrow M$. Then R is a* coherent transformation *if it is correct, hippocratic and undoable.*

4.3 Examples and consequences

Having presented a framework for bidirectional transformations and argued for a set of postulates that they should obey, let us explore the consequences of our choices. First we state two reassuring trivialities:

Lemma 1 *Let M be any metamodel. Then the trivial transformation, given by:*

- $R(m, n)$ *if and only if* $m = n$
- $\overrightarrow{R}(m, n) = m$
- $\overleftarrow{R}(m, n) = n$

is a coherent transformation.

Proof Straight from the definitions.

Lemma 2 *Let M and N be any metamodels. Then the universal transformation, given by:*

- $R(m, n)$ *always*
- $\overrightarrow{R}(m, n) = n$
- $\overleftarrow{R}(m, n) = m$

is a coherent transformation.

Proof Straight from the definitions.

Note that the latter lemma already proves that our postulates permit bidirectional transformations which are not bijective. We would of course expect that any bijective transformation is coherent, and so it is:

Lemma 3 *Let M and N be any metamodels. Then any bijective transformation, given by:*

- $R(m, n)$ *if and only if* $n = r(m)$
- $\overrightarrow{R}(m, n) = r(m)$
- $\overleftarrow{R}(m, n) = r^{-1}(n)$

where $r : M \longrightarrow N$ is a bijective function, is a coherent transformation.

Proof Straight from the definitions.

4.4 Relationship with lenses

The relationship between our framework and that of [3] is close. Let us briefly recapitulate that approach. (Warning to those already familiar with that work: we are slightly modifying the notation, for ease of comparison. We are also omitting the create function: it plays essentially the same role as our content-free model element, and would complicate the discussion without adding anything.)

Given a "concrete" set C and an "abstract" set A, a *lens* l comprises:

$$l.\text{get} : C \longrightarrow A$$
$$l.\text{put} : C \times A \longrightarrow C$$

satisfying the following laws for every $c, c' \in C$ and $a \in A$:

GETPUT: $l.\text{put}(c, l.\text{get}(c)) = c$
PUTGET: $l.\text{get}(l.\text{put}(c, a)) = a$

A key concern of the lens work is to provide a way to build up complex lenses from simple ones; in order to give a semantics to the language, they have to consider partial lenses. However, the top-level lens eventually constructed is always expected to be total, and here we need only consider total lenses.

The key difference between this set-up and ours is that [3] assumes a concrete domain and a strict abstraction: that is, there is an inherent asymmetry.

Lemma 4 *A lens l with concrete domain C and abstract domain A gives rise to a transformation R between C and A which is correct and hippocratic, as follows:*

- *$R(c,a)$ iff $l.get(c) = a$*
- *$\overrightarrow{R}(c,a) = l.get(c)$*
- *$\overleftarrow{R}(c,a) = l.put(c,a)$*

Proof First we need to show, using the lens laws, that R as defined here is correct. Given $c \in C$ and $a \in A$, simply expand the definitions:

1. $R(c, \overrightarrow{R}(c,a))$ iff $l.\text{get}(c) = \overrightarrow{R}(c,a)$ iff $l.\text{get}(c) = l.\text{get}(c)$, that is, trivially.
2. $R(\overleftarrow{R}(c,a), a)$ iff $l.\text{get}(\overleftarrow{R}(c,a)) = a$ iff $l.\text{get}(l.\text{put}(c,a)) = a$ which is true by PUTGET.

Next, to show that R is hippocratic, suppose that $R(c,a)$ holds.

1. We need to show that $\overrightarrow{R}(c,a) = a$. Expanding the definition, $\overrightarrow{R}(c,a) = l.\text{get}(c) = a$ by assumption.
2. Finally we must show that $\overleftarrow{R}(c,a) = c$. Expanding the definition, $\overleftarrow{R}(c,a) = l.\text{put}(c,a)$. By assumption this is $l.\text{put}(c, l.\text{get}(c))$ which is c by GETPUT.

Of course, a correct and hippocratic transformation does not allow us to define an exactly corresponding lens, because of the asymmetry in the lens framework. However, we can recover analogues of GETPUT and PUTGET:

Lemma 5 *Let M and N be any metamodels, and let R be a correct and hippocratic (but not necessarily undoable) transformation. Then for any $m \in M$, $n \in N$:*

- *$\overleftarrow{R}(m, \overrightarrow{R}(m,n)) = m$*
- *$\overrightarrow{R}(\overleftarrow{R}(m,n), n) = n$*

Proof By correctness, $R(m, \overrightarrow{R}(m,n))$. So by hippocraticness, $\overleftarrow{R}(m, \overrightarrow{R}(m,n)) = m$ as required. The other case is symmetric.

A *very well behaved* lens is one which, in addition to satisfying the lens laws GETPUT and PUTGET, satisfies

PUTPUT: $l.\text{put}((l.\text{put}(c,b), a) = l.\text{put}(c,a)$

Lemma 6 *Any* very well behaved *lens l can be regarded as a coherent transformation, using the same definition as in Lemma 4.*

Proof We need to show that, given the lens laws including PUTPUT, the transformation will be undoable. Suppose that $R(c, a)$, that is, that $l.\mathrm{get}(c) = a$.

1. For any $b \in C$, we must show that $\overrightarrow{R}(c, \overrightarrow{R}(b, a)) = a$. Expanding the definition, (note that the inner use of $\overrightarrow{R}$ is ignored because of the lens asymmetry) the LHS is just $l.\mathrm{get}(c)$, which is a by definition.
2. For any $b \in A$, we must show that $\overleftarrow{R}(\overleftarrow{R}(c, b), a) = c$. The LHS is $l.\mathrm{put}(l.\mathrm{put}(c, b), a)$, which is $l.\mathrm{put}(c, a)$ by PUTPUT, which is $l.\mathrm{put}(c, l.\mathrm{get}(c))$ by assumption, which is c by GETPUT.

4.5 Composition of metamodels

Let us say that a metamodel M is the *direct product* of metamodels M_1 and M_2 if any model m conforming to M can be written in exactly one way as a pair of a model m_1 conforming to M_1 and a model m_2 conforming to M_2, and conversely, any such pair conforms to M. For example, perhaps M_1 and M_2 comprise disjoint sets of metaclasses, with no relationships or constraints between the two sets. (This is admittedly an artificially constraining scenario: we will discuss relaxations in a moment.)

Now suppose that we have coherent transformations R on $M_1 \times N_1$ and S on $M_2 \times N_2$. We can construct a transformation which we will call $R \oplus S$ on $M \times N$ pointwise as follows:

- $(R \oplus S)(m_1 \oplus m_2, n_1 \oplus n_2)$ if and only if $R(m_1, n_1)$ and $S(m_2, n_2)$
- $\overrightarrow{(R \oplus S)}(m_1 \oplus m_2, n_1 \oplus n_2) = (\overrightarrow{R}(m_1, n_1)) \oplus (\overrightarrow{S}(m_2, n_2))$
- $\overleftarrow{(R \oplus S)}(m_1 \oplus m_2, n_1 \oplus n_2) = (\overleftarrow{R}(m_1, n_1)) \oplus (\overleftarrow{S}(m_2, n_2))$

Then

Lemma 7 *If R and S are coherent transformations, $R \oplus S$ is also a coherent transformation.*

Proof We show the first of each pair of postulates: the others are symmetric.

Correctness: we need to show that $(R \oplus S)(m_1 \oplus m_2, \overrightarrow{(R \oplus S)}(m_1 \oplus m_2, n_1 \oplus n_2))$. Expanding the definition of $\overrightarrow{(R \oplus S)}$, this is equivalent to $(R \oplus S)(m_1 \oplus m_2, (\overrightarrow{R}(m_1, n_1)) \oplus (\overrightarrow{S}(m_2, n_2))$ which by definition of the consistency relation is true iff $R(m_1, \overrightarrow{R}(m_1, n_1))$ and $S(m_2, \overrightarrow{S}(m_2, n_2))$; these are true by correctness of R and S respectively.

Hippocraticness: we need to show that if $(R \oplus S)(m_1 \oplus m_2, n_1 \oplus n_2)$, that is, if $R(m_1, n_1)$ and $S(m_2, n_2)$, then $\overrightarrow{(R \oplus S)}(m_1 \oplus m_2, n_1 \oplus n_2) = n_1 \oplus n_2$. The LHS is $(\overrightarrow{R}(m_1, n_1)) \oplus (\overrightarrow{S}(m_2, n_2))$ which is $n_1 \oplus n_2$ by hippocraticness of R and S.

Undoability: we need to show that if $(R \oplus S)(m_1 \oplus m_2, n_1 \oplus n_2)$, that is, if $R(m_1, n_1)$ and $S(m_2, n_2)$, then for any $m_1' \oplus m_2' \in M$,

$$\overrightarrow{(R \oplus S)}((m_1 \oplus m_2, \overrightarrow{(R \oplus S)}(m_1' \oplus m_2', n_1 \oplus n_2)) = n_1 \oplus n_2$$

Once again, all we have to do is mechanically rewrite the definition.

Notice that the proof of each postulate involved only the corresponding postulate on R and S, and was completely routine. This captures the intuition that transformations on parts of models which are completely independent ought to be able to be combined without difficulty. One would expect to be able to extend this result to cover carefully-defined simple dependencies between the metamodel parts, perhaps sufficient to justify, for example, applying a transformation defined only for class diagrams to a complete UML model, rolling the resulting changes to the class diagram through to the rest of the model. Even here, though, the issues are not entirely trivial.

4.6 Sequential composition revisited

The relation part of the sequential composition of transformations must be given by the usual mathematical composition of relations: $(R; S)(m, p)$ if and only if *there exists some* n such that $R(m, n)$ and $S(n, p)$. Mathematically this is a fine definition, but we already see the core of the problem: a tool has no obvious way to find a relevant n. What about the associated transformations? For example, $\overrightarrow{T}$ may be given models m and p such that there does not exist any n such that $R(m, n)$ and $S(n, p)$. It is required to calculate an update of p; that is, to find a new model p' such that such an intermediate model does exist, and in general the choice of intermediate model will depend on both m and p. However, $\overrightarrow{R}$ "does not understand" p, etc., so there does not appear to be any way to do this in general.

We may consider two special cases in which it is possible to define composition of transformations.

1. If R and S are bijective transformations, then the intermediate model is unique, and is found by applying $\overrightarrow{R}$ just to the first argument m. Composition of transformations in this case is just the usual composition of invertible functions.
2. More interestingly, the Harmony group considers transformations in which $\overrightarrow{R}$ is a function of the source model only, even though $\overleftarrow{R}$ still uses both source and target model. Here $\overrightarrow{R; S}$ must clearly be defined to be $\overrightarrow{R}; \overrightarrow{S}$, and we can define $\overleftarrow{R; S}$ using a trick: use the function $\overrightarrow{R}$ to bring the source model forward into the middle in order to use it to push the changes back. Formally (and translating into our notation)

$$\overleftarrow{R; S}(m, n) = \overleftarrow{R}(m, \overleftarrow{S}(\overrightarrow{R}(m), n))$$

5 Writing non-bijective QVT transformations

Supposing, as argued above, that the QVT standard is not intended to restrict transformation writers to writing bijective transformations. How can they write non-bijective transformations?

In this paper we do not provide a formal semantics of QVT, and therefore we cannot provide mathematically proven results about QVT's expressiveness. We can, however, observe some features.

First, notice that any information in a model which is not mentioned in a QVT-R transformation is not constrained by it. This provides an important, but rather uninteresting, source of non-bijectiveness. For example, if composers with names and dates are matched with composers with names and nationality by name alone, then, given a model with dates, any model with the right names and any set of nationalities is consistent with it, so the consistency relation is not bijective.

A more interesting indication of non-bijectiveness – and hence expressiveness – in the language is the existence of choices that a transformation writer can make which do not affect the consistency relation, but only the method by which consistency is restored. That is, we can look for ways to define two transformations, R_1 and R_2, which define the same consistency relation, but whose forwards and backwards consistency-restoration functions have different behaviour.

QVT-R's main feature that lets transformation writers do this is: `key`. The definition is as expected: "a definition of which properties of a MOF class, in combination, can uniquely identify an instance of that class". (QVT, unlike MOF, permits a key to consist of several properties.) Within a transformation, the role of a key is to allow the transformation writer to choose whether, when an element is found in the target model that partially matches one in the source model, it should be modified, or a new object created.

For example, Figure 1 shows a QVT-R transformation which matches English and Portuguese representations of composers if they contain identical information, keying by name. Let us compare this transformation with the one that results from deleting the key declarations.

Consider a situation in which the model `english` contains a `Composer` with `name` "Britten" and `nationality` "English", while the model `portuguese` contains a `Compositor` with `nome` "Britten" and `nacionalidade` "British". If executed in the direction of `portuguese`, the transformation without key will leave the existing object unaltered and will also create a new `Compositor` with `nome` "Britten" and `nacionalidade` "English". The transformation with key will instead modify the existing `Compositor`, changing the `nacionalidade` field from "British" to "English". These two QVT-R transformations, therefore – with and without the key declarations – embody the same consistency relation, but provide different ways to restore consistency.

Fig. 1 QVT-R transformation illustrating use of `key`

```
transformation Translation (english : ConcreteMM ; portuguese : AbstractMM)
{
key ConcreteMM::Composer{name};
key AbstractMM::Compositor{nome};

top relation R
{
n1,n2:String;
enforce domain english c:Composer {name=n1, nationality=n2};
enforce domain portuguese d:Compositor {nome=n1, nacionalidade=n2};
}
}
```

This capability is important and easy to use, but limited. It provides no means to compute information: it only involves the copying and deletion of information.

The basic idea of connecting the notion of a key uniquely identifying a model element, with the notion of specifying whether a model element should be modified or deleted, is an interesting one, but not without disadvantages. Supposing that there is a key which is regarded as such by the model developers (whether or not actually recorded as a key in the metamodel), this conflation makes sense: we need some mechanism to prevent the model transformation breaking the key property by creating a new model element with the same key value as an old one. However, the transformation writer might want to have control over whether an element should be modified or deleted, even when there is no usable key. For example, in our composers example, the transformation writer might reasonably want to prevent the transformation from creating duplicate elements with the same name, without preventing the modeller from doing so. In fact, a situation in which the transformation specifies a key which is not specified as a key in the corresponding MOF model is dangerous: the QVT specification makes clear that a key declared in QVT must actually be a key, but a transformation engine will have no way to enforce this. If a model which is input to the transformation does not actually satisfy the key property declared in the transformation, the results are undefined: presumably it should be a run-time error.

The only other mechanism within QVT-R that is intended for choosing how to resolve inconsistencies is the use of black-box operations (see [13] p18). These are restricted to being used on primitive or simple object domains; that is, at the lowest level of granularity. They are to be invoked only when the relation fails to hold (notice that this again raises the question of determining this at a local level, and the connection again to a need for/assumption of local hippocraticness), and they must guarantee that the relation will hold after their invocation (correctness, locally). There is no

mention of any kind of consistency between forwards and backwards applications of black-box transformations: this is left entirely to the transformation writer.

What is not covered by this discussion, and would require investigation in the context of a formal semantics, is whether there are less obvious ways in which the transformation writer has a choice about how to restore consistency. If we defined a restricted version of QVT, without the use of key or blackbox operations, would it still ever be possible to write two transformations that had the same consistency relation, but differed in their restoration of consistency? For example, could there be a way to make different use of `when` and `where` clauses, giving two transformations which defined the same relation, but in which, as a side-effect of how the transformation is executed, would restore consistency differently? If such ways exist, we will also need to discuss whether this is good or bad. It might be that certain choices are natural consequences of different ways to write transformations; on the other hand, transformation writers would presumably be unhappy to be told that they had to restructure their transformations in order to change decisions.

We would also like to know other things about the level of expressiveness that QVT-R has. For example, suppose that a developer is used to writing model transformations in QVT-Operational, but has discovered that she has a need to be able to write bidirectional transformations. In making the switch to QVT-R, will expressiveness be lost? Or is it the case that for any QVT-Op transformation, there is a corresponding QVT-R transformation which has the same behaviour in the forward direction, but adds the backward direction as needed? We conjecture that the latter is too optimistic, but further work is needed.

6 Conclusion

We have explored some fundamental issues which arise when we consider relationally defined transformations between models which are bidirectional and not necessarily bijective. We have motivated our work from the current QVT standard, and some of the issues we raise are specific to it, but most are more general. We have suggested a framework and a set of postulates which ensure that bidirectional transformations will behave reasonably for some definition of "reasonable", and explored some consequences of our choice. Future work includes relating our framework to triple graph grammars, and further exploration of the relation with bidirectional programming.

Acknowledgements The author has benefited from discussions on this subject with too many people to list, and from helpful comments from several referees.

References

1. Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.
2. Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information preserving bidirectional model transformations. In *In proceedings of Fundamental Approaches to Software Engineering (FASE 2007)*, number 4422 in LNCS, pages 72–86. Springer, March/April 2007.
3. 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.
4. Joel Greenyer. A study of technologies for model transformation: Reconciling TGGs with QVT. Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, July 2006.
5. Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2007.
6. Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *In Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'04)*, pages 178–189, 2004.
7. Shinya Kawanaka and Haruo Hosoya. biXid: a bidirectional transformation language for XML. In *In Proceedings of the International Conference on Functional Programming, ICFP'06*, pages 201–214, 2006.
8. A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In R. Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Amsterdam, 2006. Elsevier Science Publ.
9. Alexander Königs. Model transformation with triple graph grammars. In *In proceedings, Workshop on Model Transformations in Practice*, September 2005.
10. Lambert Meertens. Designing constraint maintainers for user interaction. Unpublished manuscript, available from http://www.kestrel.edu/home/people/meertens/, June 1998.
11. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *In Proceedings of Programming Languages and Systems: Second Asian Symposium, APLAS'04*, pages 2–20, 2004.
12. Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *In Proceedings of Mathematics of Program Construction (MPC'04)*, pages 289–313, 2004.
13. OMG. MOF2.0 query/view/transformation (QVT) adopted specification. OMG document ptc/05-11-01, 2005. available from www.omg.org.
14. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proceedings of 10th International Conference on Model*

Driven Engineering Languages and Systems, MODELS 2007, Nashville, USA, September 30 - October 5, 2007, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.

15. Perdita Stevens. A landscape of bidirectional model transformations. In *Generative and transformational techniques in software engineering II*, number 5235 in Lecture Notes in Computer Science, pages 408–424. Springer, July 2007.
16. Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovsky, Ulrike Prange, Daniel Varro, and Szilvia Varro-Gyapay. Model transformation by graph transformation: A comparative study. In *Workshop on Model Transformations in Practice*, September 2005.
17. Steven Witkop. MDA users' requirements for QVT transformations. OMG document 05-02-04, 2005. available from www.omg.org.