# Engineering Bidirectional Transformations

Richard F. Paige, richard.paige@york.ac.uk

University of York, United Kingdom

**Abstract.** Bidirectional transformations, like software, or model and program transformations in general, need to be carefully engineered in order to provide guarantees about their correctness, completeness, acceptability and usability. This paper summarises a collection of lectures pertaining to engineering bidirectional transformations using Model-Driven Engineering techniques and technologies. It focuses on stages of a typical engineering lifecycle, starting with requirements and progressing to implementation and verification. It summarises Model-Driven Engineering approaches to capturing requirements, architectures and designs for bidirectional transformations, and suggests an approach for verification as well.

## 1 Introduction

This set of lectures presents a collection of techniques and tools that can be used for engineering bidirectional transformations (BX). The motivation for these lectures is our view that transformations in general are like other software systems: they are designed to be executed on a machine, are complicated (and in some cases even complex), and are difficult to build correctly. As such, like software, transformations should be engineered by following a rigorous process. The advantages of doing so are the same as for software, including:

- *Repeatability:* by following a process, we potentially make it easier for others to repeat our work, or to reduce the amount of effort required to build a similar system in the future.
- *Review and Scale:* by decomposing a large engineering problem into stages, we potentially make it easier to audit and validate the results of each stage, and to solve larger problems than we would be able to if we treated the problem monolithically.
- *Automation:* by following a process we have greater opportunities to automate parts of it, e.g., generation of code or documents.

BX are special kinds of transformations with, in our opinion, complicated execution semantics. As such, BX may especially benefit from following a repeatable, reviewable, scalable and automated process for their development.

### 1.1 BX as software

The assumption that we are making in the preceding is that BX are software systems. A software system is an executable artefact: given a specification of

software (e.g., in a programming language or suitable modelling language), its expected outputs can be produced by executing the specification on a suitable machine (e.g., a server, a virtual machine, a simulator). A BX is an executable artefact: assuming that the BX is expressed in a suitable programming language or modelling language (and we review some of the key state of the art in Section 2) then its expected outputs can be produced by executing the BX on a suitable machine.

Like software, BX must satisfy functional and non-functional requirements, can (and probably should) be designed, and can exhibit unacceptable behaviour – that is, BX can contain bugs or faults, which may lead to failures. As we will see, depending on the technologies used to represent and specify BX, different types of failures may arise (e.g., inconsistencies) and different techniques may be used to verify the BX to help ensure that faults are caught during engineering.

### 1.2 Scope

There are numerous techniques and approaches that can be used to build and engineering BX; in Section 2 we will consider some of these. However, the focus of this paper will be on Model-Driven Engineering (MDE) techniques, because of our experience with them. Many of the techniques that we present in later sections can be used both with and without MDE tools, and if there are particular aspects that depend specifically on MDE, we will point these out.

### 1.3 Background

Before we commence with the technical content of this paper, we provide some basic definitions and terminology, in order that the paper remain reasonably self-contained.

As mentioned, we are focusing on MDE techniques for engineering BX. The key concepts of MDE are as follows.

- MDE involves the semi-automated construction and manipulation of *models*, which are structured, machine-implemented specifications of phenomena of interest. Models are meant to be processable by automated tools, and capture static and dynamic characteristics of systems.
- Models in MDE are *structured*; this structure can be defined in a number of ways, including via *metamodels*, which are specifications of abstract syntax (you can think of a metamodel as a the definition of the syntax of a language). A model is said to *conform* to a metamodel. Related approaches to defining the structure of models include schemas (e.g., XML), type rules and constraints. Many of these approaches define structure using graphs or graph-like concepts. As such, models themselves are typically graphs. This is a key distinction between MDE and so-called *modelware* approaches to engineering, and grammar-based or *grammarware* approaches.

- Models are typically specified alongside a set of *constraints* that capture well-formedness rules that cannot normally be captured with a metamodel. For example, a metamodel might be used to express that a model may include containers, and that containers may be nested (e.g., packages in Java or UML). But a metamodel – which captures abstract syntax – will not normally express that containers have unique names; this can be expressed by a separate constraint, which is normally packaged up with the metamodel or models. If a model conforms to a metamodel, it must also normally be checked against any constraints, in order to establish that it is well formed.
- Standard technologies exist for capturing models, metamodels and constraints in the MDE world. The standard technology used for metamodelling is Ecore (a part of the Eclipse Modelling Framework (EMF)). For constraints, engineers typically use the Object Constraint Language (OCL), which also has an official Eclipse implementation. There are other languages and technologies available as well (e.g., typed graphs, MetaDepth, XMI) for metamodelling and for expressing constraints.
- Models by themselves typically encapsulate business value, but are also meant to be processed by automated tools. These tools implement a variety of operations applicable to models, including the aforementioned transformations, but also comparisons, merging, migration, matching and others.

Transformations are a key operation in MDE, and have been the subject of widespread study (e.g., see the proceedings of the long-running conference on model transformation). Numerous classifications and surveys have been published on transformations in general, and BX in the specific. Four common categories of transformations in MDE are:

- Unidirectional transformations, from a source model to a target model. Such transformations are usually implemented in terms of metamodels, and are typically used when the source and target metamodel are linguistically similar, e.g., between different dialects of UML, or from an object-oriented model to a relational database model. Unidirectional transformations typically are written in one of three styles: purely declarative, operational, and hybrid (i.e., a mixture of operational and declaration parts). In our experience, many complicated transformations are very difficult to express in a purely declarative style, and as such hybrid transformation languages (such as ATL and ETL) tend to see the most use in practice.
- Update-in-place transformations, which specify modifications made to one model. Update-in-place transformations can be specified using languages suitable for unidirectional transformations, or specialist languages.
- Model-to-text (sometimes called model-to-grammar) transformations, where the source/input to the transformation is a model, but the output is no longer a model, e.g., free-form text or text conforming to a grammar. Model-to-text transformations are used in order to step outside of the modelware technical space and move to the grammarware technical space.
- Bidirectional transformations.

Transformations (and other operations on models) have side-effects. This includes purely declarative transformations. The side-effect in question is the production of *traceability* information, i.e., so-called *trace-links*, which relate source and target model elements. Trace-links can be generated automatically by transformation tools (such as Epsilon or ATL) and they can be stored for later audit and analysis. Trace-links are important in the context of transformations and BX as they provide (a) the basis for verification and validation of transformations; and (b) the connection to the theory behind BX, specifically delta lenses (in particular, delta lenses are a sound theory for trace models, encoded in an algebraic form).

### 1.4 Structure of paper

The paper starts with a brief review of the state-of-the-art in engineering BX with MDE, focusing firstly on BX scenarios of use in MDE, followed by an overview of MDE languages, tools and techniques for supporting BX. The remainder of the paper considers different aspects of a BX engineering lifecycle, starting with an overview of techniques for requirements engineering for BX, focusing on requirements specification and requirements analysis. We then move to an overview of techniques for architecture and design of BX, including a small selection of relevant design patterns. Finally, we briefly consider one approach for verification of BX, which applies to a specific approach to BX implementation and design. The paper concludes with a discussion on future challenges and perspectives on engineering of BX.

## 2 State of the Art

This section addresses some of the important state of the art in MDE approaches to BX, focusing on three specific elements: important BX *scenarios* that have been identified in the literature; important *languages* that have been influential in research in BX – in this case, we focus on QVT; and important *tools* that implement aspects of BX and that are based on MDE technology. We do not consider non-MDE approaches to BX in this brief review, and we also exclude TGG approaches because these are covered in detail by other lectures.

### 2.1 BX Scenarios

A number of recurring scenarios of use for BX have appeared in the MDE literature. Many of the MDE tools and languages that we discuss in the sequel have been designed to address these scenarios.

1. *Round-trip engineering*, i.e., generating code from models, modifying the code by hand, and then regenerating the models to reflect changes made in the code. A BX approach would, conceptually, aim to apply the principle of least change and minimise the number of modifications necessary to the

original model, instead of regenerating the entire model after each change. Research in MDE related to incremental transformation is also addressing this scenario.
2. *Collaborative modelling*, wherein multiple stakeholders are editing the same model simultaneously. In practice, what often happens is that each stakeholder has a local copy (or view) of the source model, and their changes are reflected back on the master/source copy at specified points of time.
3. *Synchronisation,* e.g., synchronising documents and code, like assurance cases and source code. This is related to round-trip engineering but synchronisation can involve model management operations other than transformations.
4. *Reflection,* for example, reflecting the results of some kind of analysis in a source model. A concrete instance of this was investigated in the MADES project where a UML MARTE model was transformed into a variety of formal models (UPPAAL, TRIO) to support analysis, and some of the results of the analysis were reflected in the MARTE models. This is an interesting example of a BX as the backwards transformation is generating a view of the target model which needs to be synchronised with the source model.

### 2.2 Standard MDE languages for BX: QVT

While there are numerous tools and approaches, based on MDE technology (like Eclipse EMF) for supporting BX, most of these approaches are strongly influenced by a significant standardised language for transformation: the OMG's Query, View and Transformations (QVT) standard [1]. QVT is a family of languages that were first envisaged in 2002 upon issue of an OMG request for proposals to support aspects of the OMG's Model-Driven Architecture standard. A number of replies were received, and the first version was submitted and approved in 2005. The most recent version, QVT 1.3, was released in June 2016.

QVT, as mentioned, is a family of languages. These languages are meant to support transformation and querying of MOF models; transformations and queries in turn can be used to generate views. The basic architecture of QVT is illustrated in Figure 2.2. The QVT architecture builds on other OMG languages, particularly MOF but also the Object Constraint Language (OCL), from which QVT acquires its expression and collection manipulation facilities.

The Relations language provides mechanisms for the declarative specification of the relationships between MOF models. It in turn supports complex object pattern matching, and implicitly creates trace classes and their instances to record what occurred during the execution of a transformation. Assertions can also be made; for instance, relations can assert that other relations also hold between particular model elements matched by their patterns. As illustrated in the figure, the intention is that Relations specifications can be translated in to the QVT Core language, along with a set of trace models, which in total provide a formal semantics for QVT Relations.

QVT Core, by contrast, is a small yet expressive language that only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. It is intended to be semantically equivalent to

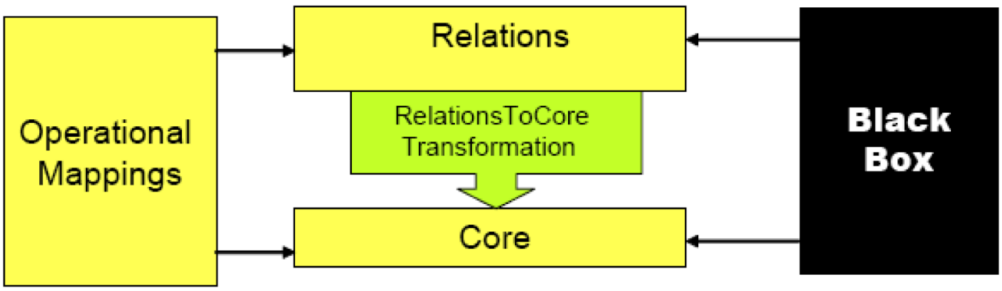


**Fig. 1.** QVT architecture from [**?**]

QVT Relations, but equivalent QVT Relations programs are liable to be more concise than the QVT Core programs.

The Operational Mappings (sometimes called QVT Operations, or QVT-o) is an operational model transformation language that extends Relations with imperative constructs. Of all the QVT languages, it is QVT-o that has received the most use and attention.

The abstract syntax of the Relations language is illustrated in Figure 2.2.

The abstract syntax can be interpreted as follows: a QVT Relations program contains a set of rules which are relations. Relations are made up of a patterns, and are applied to a set of typed model parameters. In particular, these relations can be interpreted in forward and backwards directions – that is, Relations is a BX by design.

An example of the concrete syntax of Relations is shown in Listing 1.1. This example gives a relation that is used to map persistent classes in an object oriented program to a table. The example includes three parts: a domain (a set of patterns which defines the variables and constraints that model elements bound to those variables must satisfy – i.e., the bindings for the relation); the *when* clause (the conditions under which the relation must hold); and the *where* clause (the condition that must be satisfied by all model elements participating in the relation). The interpretation of when-clauses in the Eclipse QVT implementation is that these are preconditions, and where-clause are postconditions. Both of these clauses may contain arbitrary OCL expressions.

**Listing 1.1.** An example of QVT Relations

```
relation ClassToTable /* map persistent class to table */
{
  domain uml c:Class {
       namespace = p:Package {},
       kind = 'Persistent',
       name = cn
  }
  domain rdbms t: Table {
```

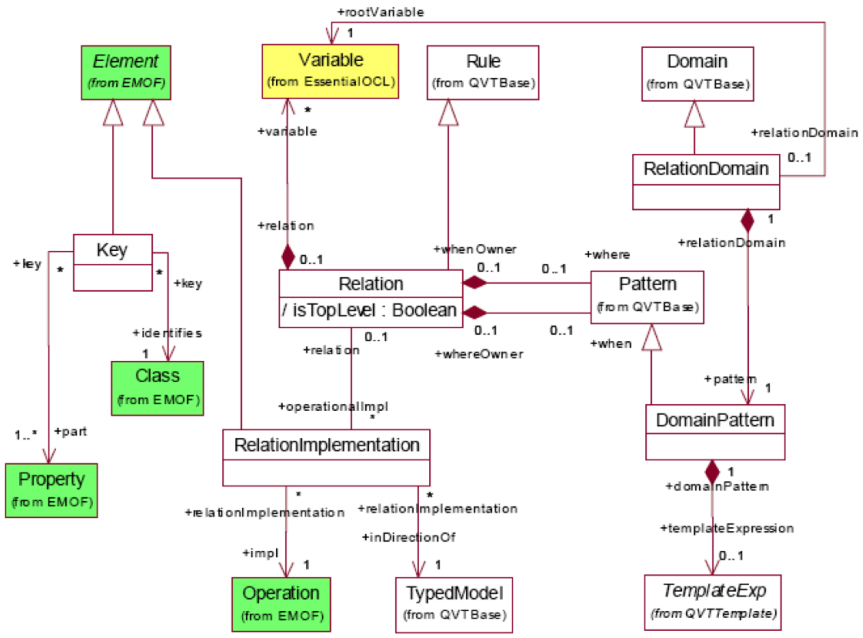


**Fig. 2.** QVT Relations abstract syntax

```
        schema = s:Schema {},
        name = cn,
        column = cl:Column {
          name = cn + '_tid',
          type = 'NUMBER'},
          primaryKey = k:PrimaryKey {
            name = cn + '_pk',
            column = cl }
    }
    when {
      PackageToSchema(p,s);
    }
    where {
       AttributeToColumn(c,t);
     }
}
```

In this particular example, the domain clauses establish which model elements in a UML and a RDBMS model are of interest (they satisfy the predicate part

of the domain clauses), and the when and where clauses are defined elsewhere by other relations.

The BX capabilities of QVT can also be illustrated by an example from QVT Core. Figure 2.2 illustrates an example of a single mapping rule in QVT Core. This is a *checking* example, which is used to check that particular patterns are satisfied by models. Once again, this is an example involving relations between a UML class model and a database model. The top part of the diagram (labelled Class to Table) defines the *c2t* relation, which relates a class to a table. The bottom pattern is evaluated using variable values of a valid binding (a valid pair of class and table) from the top pattern. In effect, the top part of the mapping rule defines a guard which restricts the scope of the bottom part of the rule.

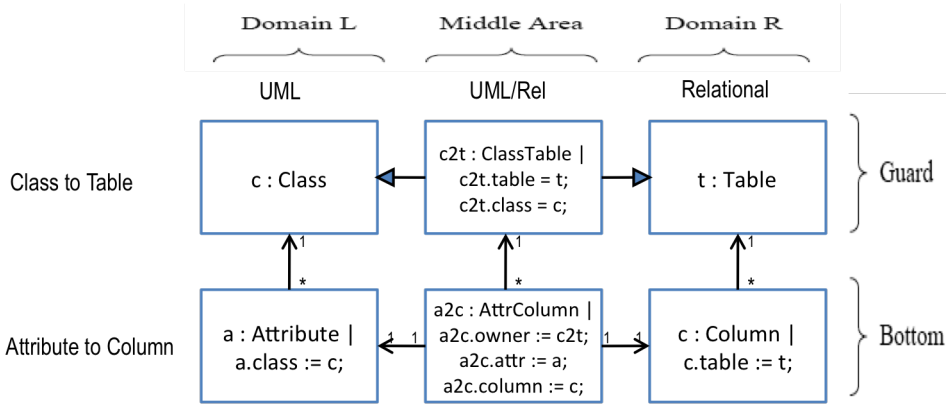


**Fig. 3.** QVT Core: mapping rule example

Once again, the mapping rule is directionless; it can be executed either way, checking a database table against a UML class, or vice versa.

The QVT standard is currently being further developed, both through the OMG standardisation efforts, but also through work on Eclipse QVT, an implementation of the different QVT languages. We briefly discuss the status of Eclipse QVT, and other MDE tools for BX, in the next subsection.

### 2.3 Tools

In this section we briefly outline some of the key tools, based on MDE technologies and principles, that either support or claim to support BX. As mentioned earlier, we exclude approaches based on triple graph grammars as these are covered elsewhere.

**Medini** Medini[1] claims to be a reasonably complete implementation of QVT Relations, but is currently unsupported. It is an EMF based transformation engine but also has a non-commercial licensed editor and debugger. While it uses the the QVT Relations syntax, it intentionally departs from the semantics of the OMG standard (e.g., how it supports deletion of elements, that it does not provide a checkonly mode). As such, we prefer not to label Medini as an implementation of QVT.

**ModelMorf** ModelMorf is a proprietary tool from Tata Consulting Services[2]. It also claims to faithfully implement the QVT Relations standard, but research by Stevens [2] shows that it does not fully implement the semantics specified in the standard. By some measures, it is more faithful than Medini, but it is still not a full implementation of QVT.

**jQVT** jQVT[3] is a QVT-like engine that is defined on top of the Java type system instead of using EMF. In turn, it uses Xbase (a partial programming language written in Xtext which compiles to Java and includes powerful features such as closures) instead of OCL for expressions. In essence, jQVT is a Java embedding of QVT; the jQVT engine generates native Java code from jQVT scripts. Of note is that it does provide support for bidirectional transformations. As of early 2016 jQVT was still being maintained.

**Echo** Echo[4] is an open-source EMF-based tool for model repair and transformation that exploits the Alloy model finder to determine models that satisfy relations. It in turn provides an implementation of the QVT Relations syntax, but the semantics intentionally departs from the OMG specification. Echo is also bidirectional.

**JTL** The Janus Transformation Language (JTL)[5] is a by-design bidirectional language with a QVT-like syntax, which propagates changes made in one model to the other. If a change made to one model makes the second model inconsistent, an approximation ("closest match") is calculated using answer set programming. As such, there can be several solutions to a transformation problem and the results provided by JTL may need to be constrained further.

**Eclipse QVT** Substantial engineering effort is being put into the development of Eclipse QVT, a project that aims to support the full OMG QVT specification

[1] `http://projects.ikv.de/qvt/wiki`
[2] Archived copy available at `https://web.archive.org/web/20120323171429/http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`
[3] `https://sourceforge.net/projects/jqvt/`
[4] `http://haslab.github.io/echo/`
[5] `http://jtl.di.univaq.it/`

(though with Ecore instead of MOF models). Currently, QVT Operations is well supported and active as part of the Eclipse M2M project. QVT Relations (in Eclipse terms, QVT Declarative) and QVT Core are work-in-progress. As work on these projects is ongoing and their status is changing regularly, we refer the reader to the Eclipse MMT project website[6] for the latest information. As of this writing, the intention with Eclipse QVT is that the Oxygen release in June 2017 will provide full support for QVT Relations.

**Bidirectionalisation** There have been several approaches to so-called *bidirectionalisation* of transformations. In these approaches, a forward transformation (from source to target) is written and the backward transformation is calculated or computed automatically. Examples of this approach include that of Hoisl [3]. The GRoundTram approach of Sasano is another example [4].

For further details, and a more in-depth classification of MDE approaches to BX, the interested reader is referred to Hidaka et al's excellent survey of BX [5].

## 3 Requirements Engineering

## 4 Architecture and Design

## 5 Verification

In this section we explore an approach to verifying BX. The approach we present is intended to be pragmatic, meant to be used with existing MDE tools and technologies. As such we do not consider issues such as soundness or completeness, though the mechanisms are present to prove conjectures related to these properties if so desired.

BX are challenging to implement on account of the inherent complexity that they must encode. Model transformation languages supporting them often do so with conditions: some require that BX are bijective (e.g. BOTL [6]), whereas others require users to work with specific formalisms such as triple graph grammars (e.g. MOFLON [7]). Many modern transformation languages do not provide any support for BX (e.g. ATL [8]), meaning that users must express them as two separate unidirectional transformations. While this seems a practical workaround, the compatibility of the transformations might not be preserved over time.

A trade-off between the benefits (but complexity) of pure BX languages and the practicality (but possible incoherence) of unidirectional transformations can be achieved in Epsilon, a platform of interoperable model management languages. Epsilon has languages supporting the specification of unidirectional transformations in either a rule-based (ETL), update-in-place (EWL), or operational (EOL) [9] style. Furthermore, it provides an inter-model consistency

[6] `https://projects.eclipse.org/projects/modeling.mmt`

language (EVL [10]) that can be used to express and evaluate constraints between models of different languages. With these languages together, BX can be "faked" by: (1) defining pairs of unidirectional transformations for separately updating the source and target models; and (2) defining inter-model constraints in EVL, the violation of which will trigger appropriate transformations to restore consistency.

Although this process gives us a means of checking consistency and automatically triggering a transformation to restore it, we lack the important guarantee that BX give us: the compatibility of the transformations. It might be the case that after the execution of one transformation, the other does not actually restore consistency, leading to further EVL violations. How do we check for, and maintain, compatibility?

We aim to address this shortcoming and obtain the guarantees of BX without the need for BX languages. Instead, we will use *rigorous* proof techniques to verify that faked BX are consistency preserving, and thus indistinguishable to users from true BX. To this end, we propose to apply techniques from graph transformation verification. Given a faked BX in Epsilon, we will model the unidirectional transformations as graph transformation rules, and EVL constraints as nested graph conditions [11]. Then, by leveraging graph transformation proof calculi [12–14] in a weakest precondition style, we aim to automatically prove compatibility of the unidirectional transformations with respect to the EVL constraints. Furthermore, we aim to exploit the model checker GROOVE [15] to automatically search for counterexamples when consistency preservation does not hold.

## 6 Conclusions and Perspectives

Bidirectional transformations must be engineered, as must unidirectional transformations and other programs that manipulate models in MDE. The state-of-the-art in engineering BX is piecemeal at the moment: there are some specific techniques for supporting different engineering phases – such as requirements engineering or design – but very coarse understanding of efficient and effective engineering lifecycles, and alternative process models. This paper attempts to capture some of the current thinking on engineering BX. It summarised some of the state-of-the-art in BX design and implementation, presented some approaches for requirements specification and analysis, and suggested some ideas for capturing the architecture of complicated BX, and the detailed design of BX in general. It also presented some ideas on an approach for verification of BX; this approach is pragmatic, in the sense that it is meant to be used within an engineering process and it acknowledges tradeoffs between completeness and soundness.

MDE for BX possesses some sound theory – such as delta lenses – and some pragmatic, if incomplete, tools (such as Eclipse QVT-Relations) but these are still siloed: the theory needs to inform the enhancement of tools, and the tools need to be used to test the corners of the theory. A good example of research

that attempts to link BX theory and practice is that combining triple graph grammars and delta lenses, but more needs to be done. What is really needed is tools that *evidently* implement the theory in a systematic and audited way.

A key challenge with connecting theory with practical tools is the limitations in our theories of metamodelling. It is questionable whether we have a sound and complete understanding of a type theory for MDE and metamodelling, but this would underpin any attempts to link a theory of BX with the pragmatic tools supporting BX.

We mentioned tools for BX throughout this paper. The standard tool in the MDE community is QVT-Relations; the Eclipse implementation is still under development. QVT-Relations has been criticised for being very complex, with substantial semantic ambiguity. The development of its Eclipse implementation is revealing some of these ambiguities, but this will only be convincing if supported by a sound theory, e.g., delta lenses. Perhaps applying delta lens theory to aspects of Eclipse QVT could even help simplify the tool.

It remains to be seen if we can develop a rich, compelling set of industrial scenarios for BX. In our substantial industrial experience of transformations and MDE, we have had only one precise requirement for a BX (over around 15 industrial projects and 13 years of experience), and that was for the results of various forms of analysis (e.g., failure analysis, performance analysis) to be reflected on source models after calculation. It is unclear if such scenarios benefit from the heavyweight machinery of BX. But it should also be noted that requirements for BX sometimes emerge as development proceeds and having ways in which transformations can be *extended* to become bidirectional may be useful.

In Section 5 we described an approach to BX that involved specification of inter-model consistency constraints between two models, and the definition of two separate but synchronised update-in-place transformations on the two models. When the constraints were violated and the models became inconsistent, the transformations would be triggered to re-establish consistency. This approach – two simple yet unidirectional transformations instead of a single bidirectional transformation – needs to be clearly related to the BX solution space: when is it more effective to use versus building a full BX?

Finally, we observe that many transformations developed in practice are operational (e.g., those written in EOL or subsets of ATL). As well, there are many model-to-text (or model-to-grammar) transformations that support code generation scenarios. How do these fit in to the BX space? Are they simply too hard to consider? Are there scenarios or types of transformations that simple *should not* (rather than *cannot*) be bidirectionalised? As a challenge, consider the EuGENia tool[7] which is a unidirectional model transformation written in EOL, which automatically generates three models needed by GMF to construct a graphical editor. These are generated by a transformation that takes as input a single annotated Ecore model. The transformation is defined entirely operationally, as we found that it would be too complex to implement using declarative rules (it is not a mapping transformation). Could EuGENia be turned into a bidirectional

[7] `https://eclipse.org/epsilon/doc/eugenia/`

transformation? Our intuition is no (and, more pragmatically, we cannot see any reason why you would want to do so), but it would be interesting to explore what, fundamentally, makes an operational or hybrid transformation difficult to bidirectionalise.

**Acknowledgements** Parts of this work were supported by the European Commission's 7th Framework Programme, through grant #611125 (MONDO).

## References

1. OMG. *MOF 2.0 QVT V1.3*. Object Management Group, 2016.
2. Perdita Stevens. A simple game-theoretic approach to checkonly QVT relations. *Software and System Modeling*, 12(1):175–199, 2013.
3. Bernhard Hoisl, Zhenjiang Hu, and Soichiro Hidaka. Towards bidirectional higher-order transformation for model-driven co-evolution. In *Model-Driven Engineering and Software Development - Second International Conference, MODELSWARD 2014, Lisbon, Portugal, January 7-9, 2014, Revised Selected Papers*, pages 153–167, 2014.
4. Isao Sasano, Zhenjiang Hu, Soichiro Hidaka, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. Toward bidirectionalization of ATL with groundtram. In *Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, pages 138–151, 2011.
5. Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-based classification of bidirectional transformation approaches. *Software and System Modeling*, 15(3):907–928, 2016.
6. Peter Braun and Frank Marschall. Transforming object oriented models with BOTL. In *GT-VMT 2002*, volume 72 of *ENTCS*, pages 103–117. Elsevier, 2003.
7. Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In *ECMDA-FA 2006*, volume 4066 of *LNCS*, pages 361–375. Springer, 2006.
8. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
9. Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nikolaos Drivalos, and Fiona A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS 2009*, pages 162–171. IEEE Computer Society, 2009.
10. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *LNCS*, pages 204–218. Springer, 2009.
11. Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.
12. Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *ICGT 2006*, volume 4178 of *LNCS*, pages 445–460. Springer, 2006.
13. Christopher M. Poskitt. *Verification of Graph Programs*. PhD thesis, The University of York, 2013.

14. Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
15. Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *Software Tools for Technology Transfer*, 14(1):15–40, 2012.