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

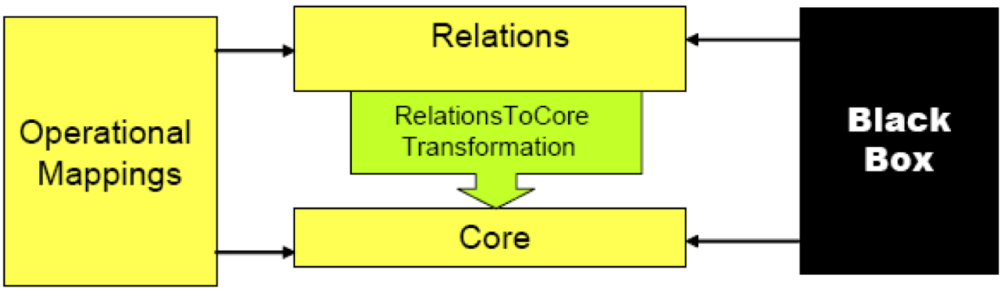

Fig. 1. QVT architecture from [1]

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 {
```

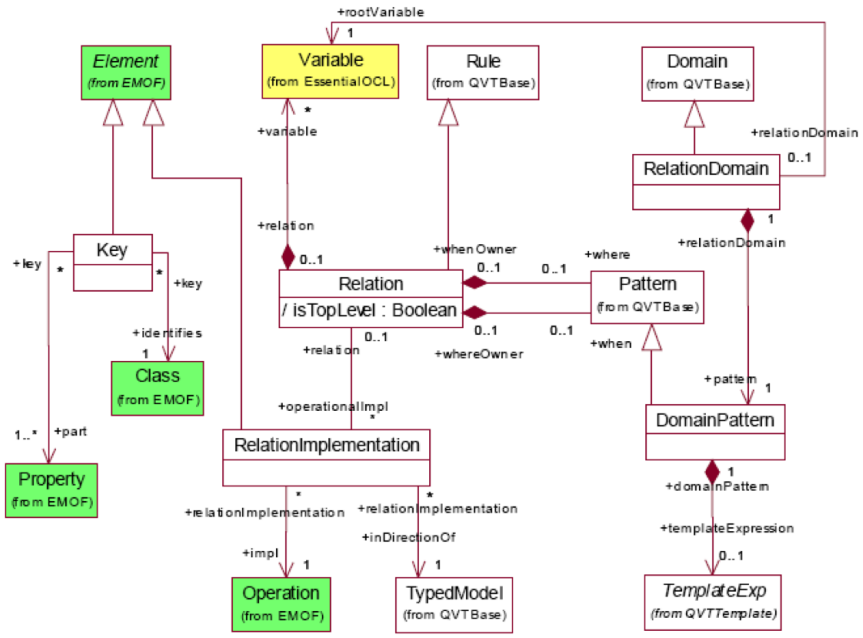

Fig. 2. QVT Relations abstract syntax

```
        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.

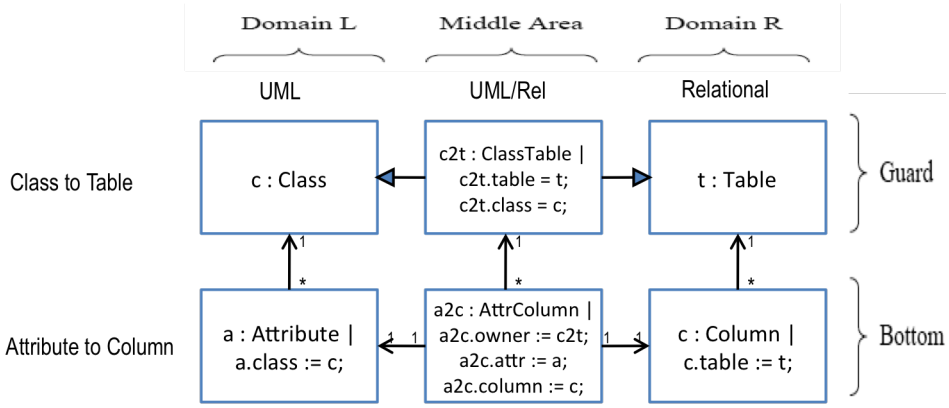

Fig. 3. QVT Core: mapping rule example

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 for BX

In this section we will consider techniques and tools for requirements engineering for BX. We will motivate the benefits of considering requirements for BX in general, before discussing some of the general questions to be addressed when building a BX. These questions will help us motivate a discussion on the general properties of BX (which may be the source of constraints on requirements for a BX), as well as examples of functional and non-functional requirements for BX. This is followed by a broad overview of requirements engineering processes for BX, which leads in to a discussion on MDE languages suitable for requirements engineering for BX.

3.1 Motivation

Requirements engineering is the process of identifying, documenting and maintaining requirements in systems engineering. The typical tasks involved in requirements engineering are:

- *identification:* where new requirements to address a problem are clarified
- *analysis:* where the requirements are assessed to ensure they accurately capture what is needed for the system under consideration, and conflicts between stakeholders are resolved
- *specification:* where the requirements are documented in a precise (but not necessarily formal) way
- *validation:* where the requirements are checked to ensure they are consistent and address stakeholder needs

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

– *maintenance:* where the requirements are considered for update as the system under consideration is constructed, deployed and changed.

BX are software systems and as such will benefit from a clear understanding of requirements; for large or complicated BX, there may be benefits to following a rigorous requirements engineering process as well. In particular, an understanding of requirements for BX can help in mapping BX problems to tools that are suitable for implementation (and vice versa). An understanding of requirements for BX can also help in contrasting different potential solutions in terms of their tradeoffs in how they satisfy requirements.

3.2 Questions and Properties for BX

A typical first phase of requirements engineering is *identification*, where engineers attempt to determine what requirements a software system should exhibit. This in turn may help determine properties or constraints that the ultimate system will satisfy. There are numerous ways in which requirements can be identified, e.g., via stakeholder interview, by reviewing existing similar systems, by following questionnaires or checklists, or by using testing techniques to derive requirements. Based on [6] we suggest some general questions that could be addressed when constructing a BX, the answers to which could help derive requirements.

1. What needs to be transformed into what?
2. What mechanisms can be used for building the BX? (i.e., theory, tools, techniques)
3. What are the application domains for the BX?
4. What are the specific characteristics of the BX (e.g, what patterns are appropriate to use)?
5. What are the quality requirements (e.g., performance) for the BX?
6. What are the success criteria for the BX?

Questions 3, 4 and 6 are possibly the most opaque. Question 3 is designed to help identify constraints on the scope of use for the BX, e.g., will the BX be used in developing hard real-time systems, or interactive systems? Question 4 is designed to help identify functional requirements, e.g., should the BX be parameterised, should it be interactive? This in turn may help identify suitable patterns that can be used in specifying or designing the BX. Question 6 is the "stopping condition": how will we know if we have successfully solved the BX problem?

BX exhibit various properties (such as least-change, or determinism). When considering requirements for a BX, there are general properties that may be of interest, particularly in determining constraints that the ultimate BX must satisfy. Some examples are:

– Size: is the BX small (e.g., a single reversible refactoring) or large (e.g., a reversible code generator)?
– Level of automation: is the BX meant to be fully automated, or involve a human-in-the-loop?

- Visualisation: how is the BX, its results, and its input presented to users?
- Level of industry application: to what extent is the BX to be deployed in an industrial context?
- Maturity level: should the BX be implemented in a tool? Should the BX be a theoretical construct?

Understanding the relative importance of these properties will be helpful in deciding on what theory or tool to choose for defining a BX.

3.3 Functional and Non-functional Requirements

In the classical requirements engineering literature, functional requirements specify what a system must, could or should provide. Non-functional (or behavioural) requirements specify criteria against which we can judge the quality of a system. In a requirements document, functional and non-functional requirements are typically presented separately, with suitable tests given that can be used to assess the coverage and completeness of fulfilment of requirements.

There has been little published research on examples of requirements for transformations in general, let alone BX, but based on [6, 7] we can propose some examples for BX. We start with functional requirements. For simplicity of presentation, we assume that a BX under development is defined between two models (a source and a target).

- *Correctness:* when the BX is run in the forward direction, the target model must be well formed (defined in terms of conformance to the target metamodel and any corresponding constraints). Similarly, when the BX is run in the reverse direction, the source model must be well formed.
- *Inconsistency tolerance:* the BX should be able to support incomplete or inconsistent models, e.g., temporarily inconsistent models. This reflects the practical situation wherein a BX *gradually* re-establishes consistency over a sequence of steps.
- *Modularity:* it should be possible to compose BX into new transformations.
- *Traceability:* a BX should support the generation of trace-links (sometimes called a correspondence model) between source and target models, as well as between the steps of a transformation chain.
- *Change propagation:* a BX should provide support for propagating changes from one model to the other model.
- *Incrementality:* a BX should make it possible to update a model based only on the changes made to the other model (that is, the parts of the model that do not change are not used to make changes to the other model).
- *Uniqueness:* a BX could support the ability to generate a unique solution to the problem of ensuring consistency between two models.
- *Termination:* it should be possible to support the definition of terminating BX transformation executions.
- *Style:* a BX should be expressible in a particular style, i.e., declarative, operational or hybrid.

Note the wording of these requirements; we have used the words *must*, *should* and *could* to indicate the degree of importance or criticality of each type of requirement. As this suggests – and as is reinforced by [5] there is substantial variability in what BX provide (and also how they are implemented).

Non-functional requirements, recall, specify criteria against which we can judge the quality of a BX. As is the case for functional requirements for BX, there is limited research on non-functional requirements. Some examples have been proposed in [7], and we list a selection here.

- Extensibility: the extent to which the BX can be extended to support new functional requirements or a change in scope.
- Usability: is the BX judged to be usable by stakeholders?
- Robustness: can the BX manage invalid models (i.e., that do not conform to the metamodels involved in the BX), or deal with errors in models?
- Interoperability: can the BX be combined and used together with non-BX tools (e.g., other MDE tools and operations, such as model comparisons or mergings)?

Clearly, more research on requirements for BX is needed. As our experience with building BX grows, and our understanding of what constitutes a useful BX scenario increases, our ability to elaborate sensible functional and non-functional requirements for BX will improve.

3.4 Requirements Engineering Processes for BX

In this section we briefly outline typical stages of a requirements engineering process for BX and highlight the key artefacts and stakeholders that will be involved. We discuss elicitation in some detail, and evaluation briefly. This leads in to the next section where we give an overview of some of the key specification techniques that can be used within a requirements engineering process for BX.

The requirements engineering literature, e.g., [8], identifies the following generic phases in requirements engineering:

- *Domain analysis and elicitation:* Identify who are your stakeholders. From these stakeholders, gather information on the system domain and system requirements.
- *Evaluation and negotiation:* Identify imprecision, conflicts, omissions and redundancies in the informal requirements identified in the previous phase. Resolve these (if possible and appropriate) via negotiation and consultation.
- *Specification:* Document the formal requirements in a specification (we will consider this for BX in more detail later). The specification is often the basis for a contract between developers and customers.
- *Validation and Verification:* Check the specification for consistency, completeness and acceptability to stakeholders.

This is generic, applicable to any kind of software or systems engineering. What might a requirements engineering process for BX look like? Tehrani et al [6] propose a process for transformations, which is depicted in Figure 4.

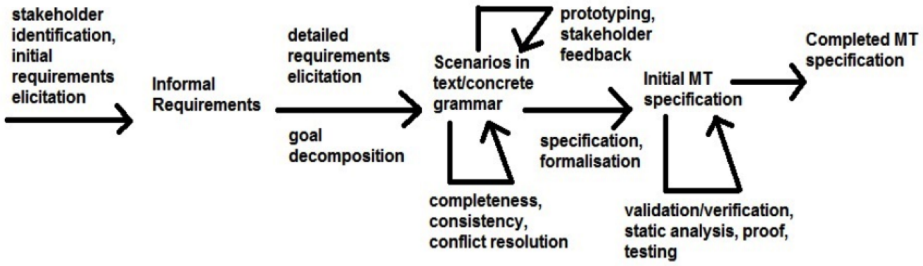

Fig. 4. A requirements engineering process from [6]

(It is worth emphasising that the process shown in Figure 4 is for transformations in general, not specifically for BX.) There are some points to note about the above process.

- The process is generic for the most part, and resembles the steps that are typically carried out for software systems.
- An interesting aspect is the use of scenarios as a concrete mechanism for driving the development of a requirements specification. In the context of BX this suggests that identifying and capturing more (and more detailed) BX scenarios will be very helpful in improving our understanding of BX requirements engineering.
- The process distinguishes between local and global requirements, as is often done in systems engineering. A local requirement may pertain to a particular transformation component (e.g., that correspondences are defined between elements of particular types), whereas a global requirement may apply to an entire transformation (e.g., a performance requirement, that a measure of complexity is reduced by running a BX, or a safety requirement).

3.5 Elicitation

Elicitation is an important first step in any requirements engineering process. What techniques might be applicable for BX? Many of the traditional elicitation techniques appear to be directly applicable to BX problems with little change, as argued by Tehrani et al [6]. For example, a classic elicitation technique is observation (an ethnographic method): observing an existing – possibly manual – BX technique or process could provide sensible requirements for an automated process. Consider a scenario wherein a BX is to be defined between an Excel spreadsheet and a SysML requirements diagram[7] A manual BX process between the two might involve (a) making changes to cells in an Excel column; (b) switching to a SysML editor; and (c) modifying attributes in a SysML class model.

[7] This is a sanitised version of a real problem encountered by the author.

This might indicate to a requirements engineer a sequence of steps that is needed to implement in an automated BX.

Another technique that can be used for elicitation is the *unstructured interview*, where open-ended questions are asked about the problem domain or the current (BX) process. This can be useful for identifying transformation goals, e.g., "ensure that the source and target models are inconsistent for no more than 10ms". In carrying out an unstructured interview regarding a transformation, Tehrani [6] suggests some generic open-ended questions that may be useful to consider; we have extended their questions with some of our own, based on our experience in the MONDO project[8].

- Is there a size range for the source and target models? This may suggest to the engineer the type of infrastructure that may be useful for the project (e.g., EMF to represent models).
- Does the encoding for the BX matter? For example, for very large scale models it may be necessary to consider binary formats.
- Are there any assumptions that are made about the source or target models? For example, are they always available? Are they read-only? Write-only? Are there confidentiality restrictions?

Along with unstructured interviews there are *structured* interviews, which involve asking pre-loaded questions about the domain and the BX, perhaps based around a checklist linked to a requirements pattern catalogue. For example, a checklist of questions may be divided into parts, one focusing on questions related to global functional requirements (e.g., is hippocraticness important, is semantics preservation important?) and another related to local non-functional requirements (e.g., should this rule satisfy a specific time bound?)

A final elicitation technique that we mention is *scenario-based analysis*, where scenarios are used to capture different requirements transformation processing cases. The benefit of using scenarios is that they are concrete: scenarios are usually presented in a concrete scenario language, often supplemented with sketches of sample models. For example, for BX we might specify a scenario for introducing or removing a pattern to change an object-oriented design. The forward transformation scenario could include a concrete example of introducing the pattern into an existing design.

3.6 Evaluation

Once we have elicited requirements for BX through any of the techniques described previously, we have a set of informal statements of what the BX must or should provide. These statements may be inconsistent, and ideally we should be identify this before we formalise the BX requirements in a specification. There is little to no published research on evaluation techniques for BX requirements. We may find some inspiration in the general requirements engineering literature. For example, one approach used for requirements evaluation is prototyping, i.e.,

[8] http://www.mondo-project.org/

engineers build a prototype (paper, mock-up, simulation) of a solution in order to help identify or reconcile inconsistencies. It is unclear whether the expense of building a BX prototype is greater than building a BX in the first place (because, for example, a BX prototype would need to be constructed using standard BX tools). Another approach that is sometimes used is goal-oriented analysis, but it is as of yet unclear how goal-oriented techniques apply to the definition of BX. There are significant open questions relating to how we evaluate requirements for BX.

3.7 MDE Languages for Requirements Engineering for BX

In this section we move from a mostly abstract discussion on requirements engineering for BX and focus on the more concrete topic of languages that can be used to support requirements engineering for BX. There has been some work in this area – i.e., on different MDE languages and tools for specifying transformation requirements – though there is still very limited experience of specifying requirements for BX in the specific. Here, we will focus on presenting details of one approach – *trans*ML – which is a family of languages that can be used for engineering model transformations. *trans*ML can, as we will show, be used to specify different aspects of the requirements for a BX. We will also use *trans*ML in the next section to specify different facets of the architecture and design of a BX. For an alternative approach to specifying requirements for transformations, based on mind-maps, the interested is referred to the DSL-Maps approach [9].

*trans*ML [10] by way of introduction, is a family of MDE languages to support the lifecycle of transformation development, from requirements through to implementation. It is technology agnostic, and can be used with any transformation implementation language (there is published experience of using *trans*ML with QVT, EOL, ETL and ATL). The overall architecture of *trans*ML – that is, the set of languages and their inter-relationships – is depicted in Figure 5. The parts of *trans*ML relevant to this section are the Requirements language (at the top) and the languages to support Analysis (Simple Scenarios and Formal Specification).

We focus on the requirements language and those languages of *trans*ML that support analysis in this section. The former is used primarily to support the description of the results of elicitation.The latter are used to support detailed specification.

To support description of the results of elicitation, *trans*ML provides a diagrammatic representation of (BX) requirements that is derived from SysML requirements diagrams. Such representations can be produced using any of the aforementioned techniques for elicitation. Because *trans*ML is an MDE language, it is defined using metamodels. The *trans*ML requirements metamodel is shown in Figure 6.

The requirements metamodel is very simple, but defines an expressive requirements language for BX. The language explicitly supports hierarchical decomposition of requirements, as well as classification, refinement, and traceability. Of particular note is the *ReqSource* element, which identifies where a require-

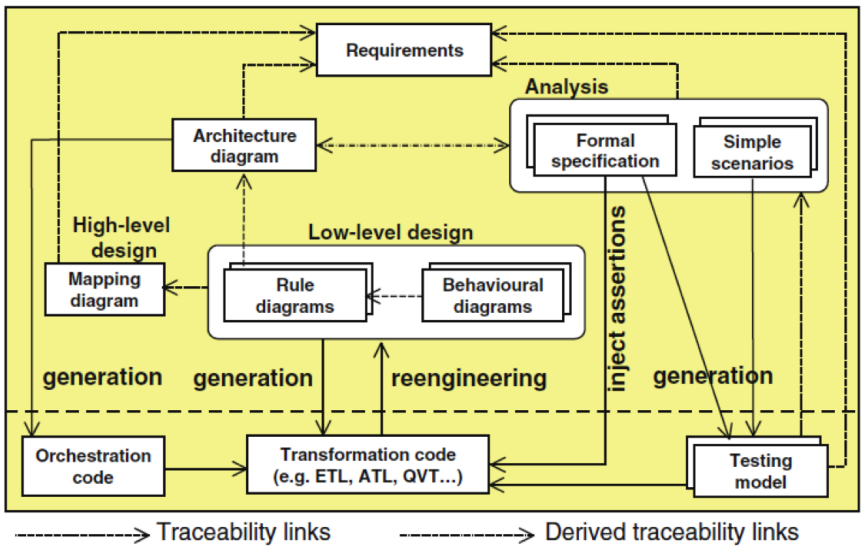

Fig. 5. *trans*ML architecture; boxes represent languages (or sets of languages) and arrows represent dependencies, typically traceability links

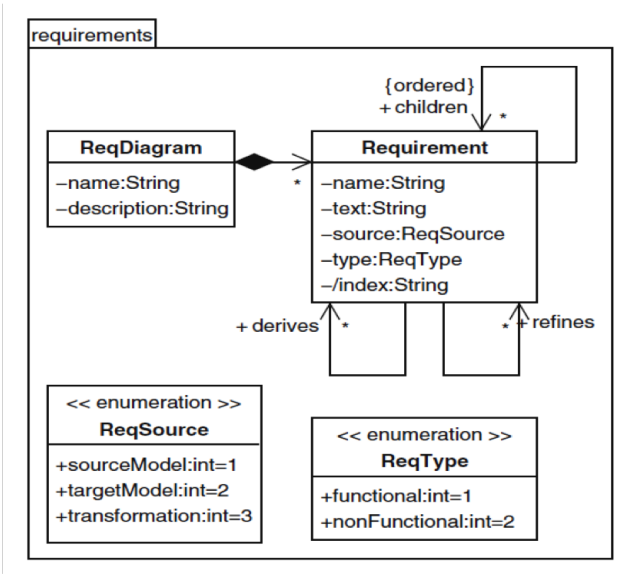

Fig. 6. *trans*ML requirements metamodel

ment arises, i.e., in the source of a transformation, the target of a transformation, or from the transformation itself (it is generated by the transformation).

We illustrate the requirements metamodel with two examples, the first from Guerra et al [10] which shows an example requirements model for a unidirectional transformation (Figure 7), and the second which shows an example for a BX (Figure 8). In both cases, the examples involve transformations from and between object-oriented and database models. We observe that different concrete syntaxes are used in each example. The first concrete syntax is based on SysML, whereas the second is a box-and-arrow domain-specific requirements language which makes use of elements of UML (particularly dependencies and stereotypes).

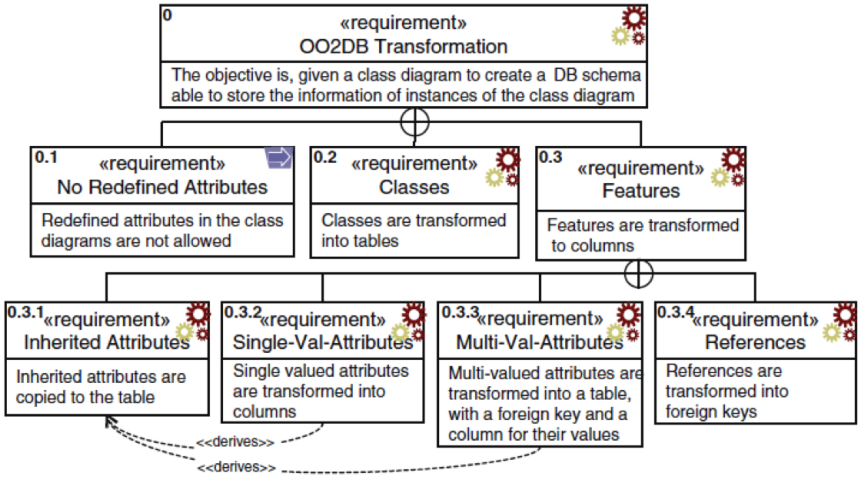

Fig. 7. *trans*ML requirements model example (SysML-like concrete syntax)

The top-level requirement (OO2DB Transformation) in Figure 7 is decomposed into the set of requirements below (i.e., No Redefined Attributes, Classes, Features). The Features requirement is further decomposed in the last level of the diagram. Note that derived requirements are also noted, i.e., that the Inherited Attributes requirement is derived from the Single-Val-Attributes and Multi-Val-Attributes requirements.

The example in Figure 8 illustrates a requirements specification for a BX. It has a similar structure to the previous example for a unidirectional transformation. The main difference is in the expression of the individual requirements, which are expressed in terms of consistency relationships rather than transformation features.

Both of these examples are informal, in the sense that they rely substantial on natural language, and are the result of applying elicitation techniques; they may contain imprecision or inconsistencies, which may be resolved by analy-

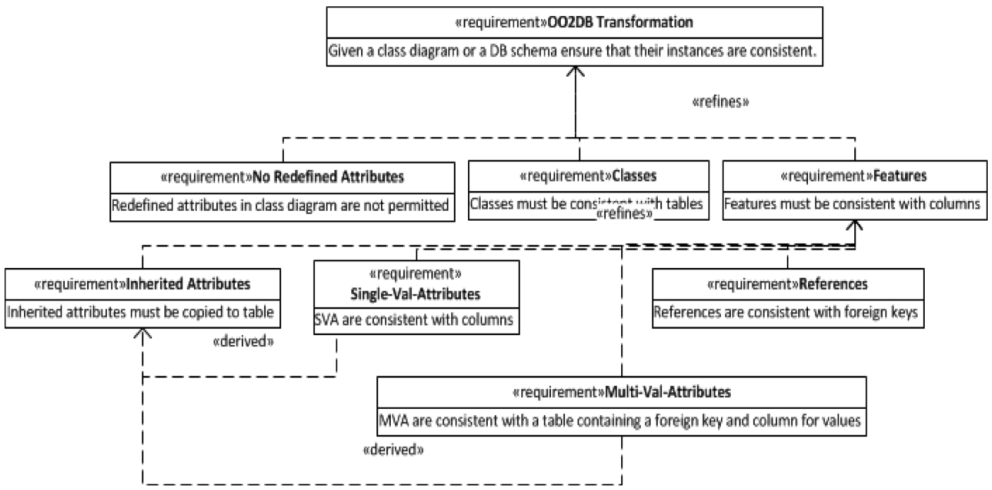

Fig. 8. *trans*ML requirements model example (box-and-arrow concrete syntax)

sis. *trans*ML supports two sets of languages for requirements analysis: a simple scenario language, and a formal specification language for requirements.

The simple scenario language of *trans*ML supports description of *concrete cases* for transformation, i.e., how examples are meant to be related by the BX. *trans*ML is applicable to both models or fragments of models, the latter of which is essential for incremental development and for working with large monolithic models. An example of a transformation case (i.e., a scenario) for part of an object-oriented to database BX is shown in Figure 9.

On the left side of the example is the object-oriented model fragment, consisting of a class with a multi-valued attribute; on the right side is a database model fragment, consisting of two tables containing columns and foreign keys. This is an example of a BX scenario involving a class with a multi-valued attribute and a consistent database model that resolves the multi-valued aspect using a foreign key (there are other solutions).

The second *trans*ML language for requirements analysis supports formal specification of requirements; it is used to specify what a transformation has to do. It captures correctness properties and specifies restrictions on the models involved in the BX (for example, the consistency relations specified in the BX may only be applicable when the source or target models obey various constraints). *trans*ML s formal specification language supports all of this via use of declarative patterns, a concept taken from triple graph grammars. Patterns express allowed and permitted relations between elements from the involved models. The pattern language itself is expressive and can include conditions on attribute values as well as constraint.

The metamodel for the *trans*ML formal specification language for requirements is depicted in Figure 10.

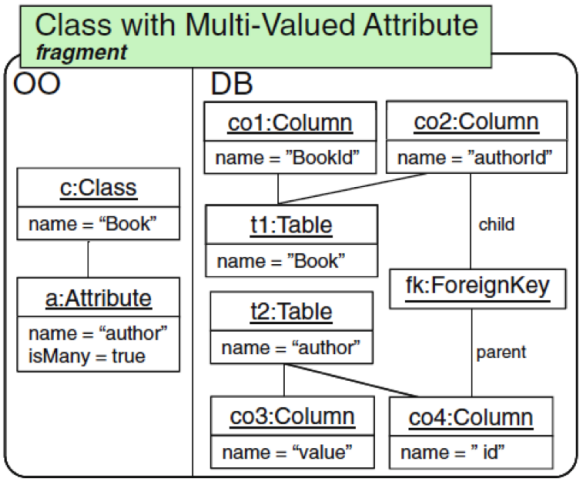

Fig. 9. *trans*ML scenario (example case)

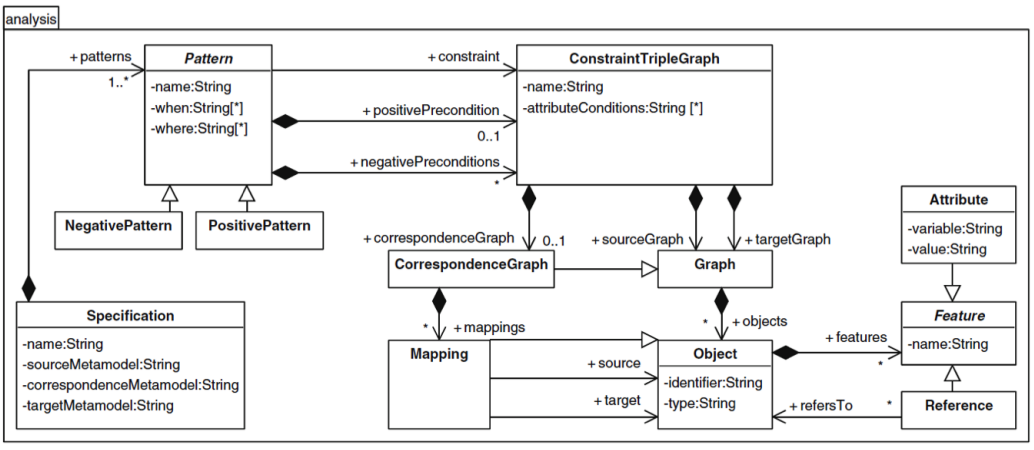

Fig. 10. *trans*ML formal specification language metamodel

A requirements specification (the Specification element in Figure 10 is made up of a number of patterns. A pattern may be a positive or a negative precondition, which are similar to both the *when*-clauses of QVT Relations, as well as triple graph grammar's negative application conditions. The Constraint Triple Graph element encodes these clauses, and also include correspondence graphs (which is effectively traceability information) as well as links to source and target graphs.

An example of a pattern for a BX is shown in Figure 11.

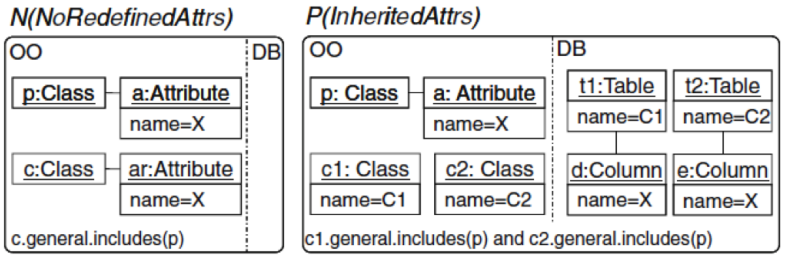

Fig. 11. *trans*ML example pattern

The example pattern is, once again, taken from the object-oriented to database BX example that we have used several times before. In this example, the left side of the diagram is a negative pattern: it checks for the existence of two classes c and p such that p is an ancestor or c, while both have an attribute with the same name (X). On the right is a positive pattern: it expresses the inherited attribute property (in this case, the inherited attribute named X is mapped into two columns in the database model). More detailed examples of patterns and specifications can be found in [10].

In the next section we consider the next phases of the BX engineering lifecycle, focusing on architecture and design; we will explore further aspects of *trans*ML for supporting these phases.

4 Architecture and Design

In this section we motivate and present an approach for developing the architecture and design of a BX, including MDE languages to capture detailed designs of BX, as well as techniques for expressing and applying design patterns for BX. What we present here builds on the techniques introduced in the last section, where we used *trans*ML to capture requirements for BX.

As discussed earlier, large and complicated BX are similar to large and complicated software systems: they involve many parts (e.g., transformation components, rules) with complicated inter-relationships and dependencies. Many BX have sophisticated behaviour which can be difficult to interpret from their concrete syntax. They are also difficult to engineer correctly. Large software systems are usually not monolithic: they are built as a set of interrelated components. Arguably, BX should be constructed in the same way.

Nevertheless, architecture for BX – and transformations in general – can be complicated. Some of the issues are as follows.

– *Components:* what are appropriate component models for BX? For software systems we have a reasonable understanding of what a component in a software architecture is, how it may be implemented, and how it can be precisely combined with other components. Our understanding of components for BX

and transformations in general is underdeveloped. Most transformation languages offer a notion of a *rule*, and some languages have a notion of *module*, but richer and deeper understanding (e.g., of ports, protocols, and architectural styles) is missing.
- *Relationships:* what are appropriate relationships that can be defined between BX components? For software systems we have a comprehensive library of component connectors (e.g., protocols, buffers, compositions, containments) that can be deployed; a similar understanding for BX is not yet available.
- *Interoperability:* a key aspect of software architecture is what it provides in terms of interoperation with external systems. For BX, the question is: how can a BX be integrated with other components or architectures, e.g., code generators, verification tools, etc.

We will now present an approach to transformation architecture, embodied in *trans*ML and present several small examples of both BX architecture and unidirectional transformation architecture. We then describe an approach for detailed design for transformations.

4.1 BX Architecture in *trans*ML

In Section 3 we introduced the *trans*ML approach and explained its support for requirements specification (including scenarios and formal requirement specification). As illustrated in Figure 5, *trans*ML provides support for expressing transformation architectures and designs.

Architecture in *trans*ML is embodied in a traditional architectural modelling approach: an architecture is a set of components and connectors that interact via directional interfaces. Component types are given in terms of metamodels, or event types (for supporting event-driven architectures or for events generated by sensors) or other components (to support higher-order transformations). The component model is general in the sense that it can be used to represent transformations, black-box components (e.g., non-transformation or non-MDE components), or actors (e.g., human users).

The *trans*ML metamodel for architectures is illustrated in Figure 12. It is worth noting the *direction* attribute on the Interface element; components of BX may both generate and receive information via interfaces.

Constraints on interfaces can be used to impose a concept of contract, e.g., to restrict expected inputs and outputs, but also to support conformance checking.

Figure 13 shows an example of a unidirectional transformation architecture, using a simple component-based concrete syntax from UML. This example illustrates a transformation-centric view, i.e., the components in the architecture are themselves transformations. This can be contrasted with a type-centric architecture, shown in Figure 14, where the components are types (or metamodels). In both cases, the example architecture is for a chain of transformations between an object-oriented model and SQL code.

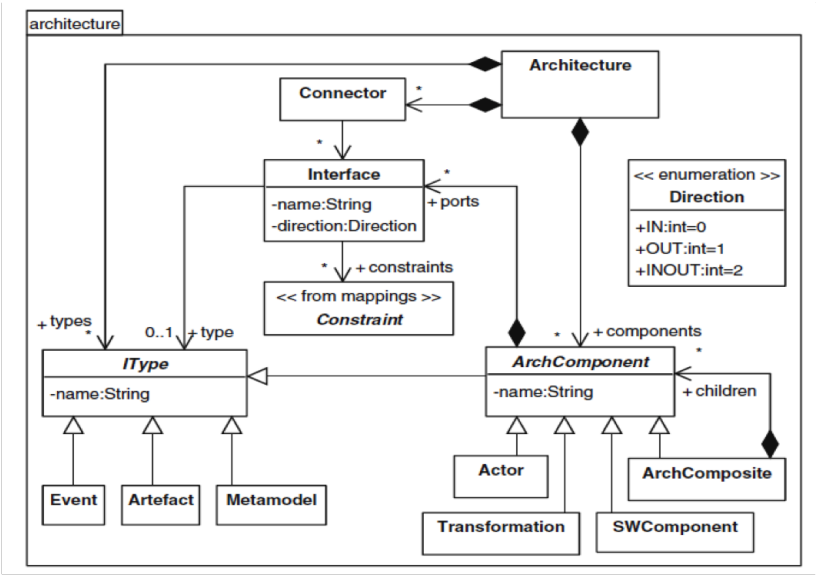

Fig. 12. *trans*ML architecture metamodel

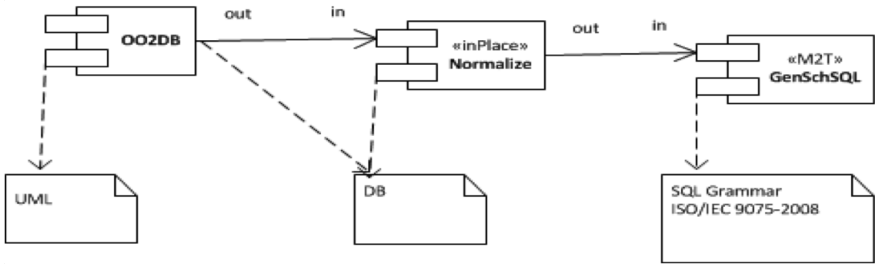

Fig. 13. *trans*ML architecture example (transformation-centric)

In the above example, firstly a unidirectional OO2DB transformation is executed (taking a UML model as input and producing a DB model as output). Then, a normalising update-in-place transformation is executed on the DB model. Finally, a model-to-text transformation is executed on the DB model, producing SQL compliant to a specific grammar.

The type-centric view represents the individual transformations as relationships between components. We have extended this example to represent bidirectional transformations throughout: i.e., the OO2DB, Normalise and GenSchSQL (the model-to-text transformation) could be executed in either direction. We could, of course, present the same BX in a transformation-centric style. In this case, the architecture in Figure 13 would have bidirectional dependencies on the relevant input and output models, as depicted in Figure 18.

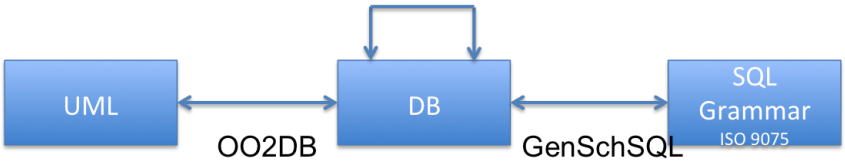

Fig. 14. *trans*ML architecture example (type-centric, bidirectional)

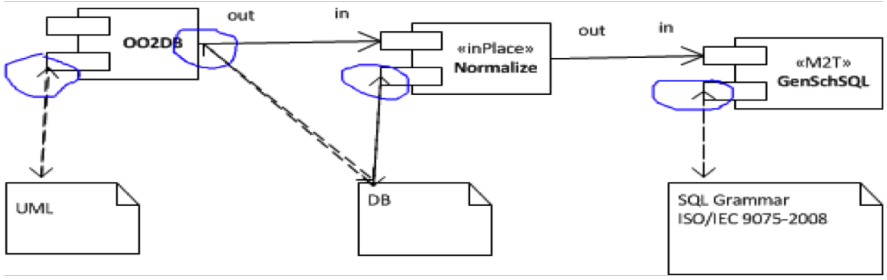

Fig. 15. *trans*ML architecture example (transformation-centric, bidirectional)

4.2 Design of BX

The architecture of a software system captures the key components and their interrelationships. In the case of a BX this includes the connections between transformation components, the ports through which components communicate, and restrictions and constraints on that communication. The engineering process for BX continues with design, which can be broken into two parts: *high-level design*, which focuses on capturing *what is transformed into what*; and *low-level design*, which focuses on capturing *how* the transformation is to be carried out. We briefly consider *trans*ML support for each aspect.

High-level design of a BX, once again, aims to capture what is transformed into what. To represent this, *trans*ML introduces a *mapping diagram*, inspired by triple graph grammars. These capture the mappings between arbitrary model elements involved in the transformation. However, mappings are not meant to be used as a implementation model – specifically, they are not meant to be used as a tracing mechanism to guide the execution of code (this, as we will soon see, is the purpose of the low-level design features of *trans*ML .

The *trans*ML metamodel for mapping diagrams is illustrated in Figure 16. Mappings have ends which are associated with modelling elements. Navigability is a property of mappings; BX will involve navigation to both source and target. Constraints can be attached to mappings in order to define conditions on when (part of) a mapping can hold.

Figure 17 illustrates a mapping, for the Class2Relational BX. On the left of the diagram is a package containing key modelling elements of an OO model; on the right, a database model. In the centre are the mappings along with some informal English text explaining the purpose of each distinct mapping. Note the

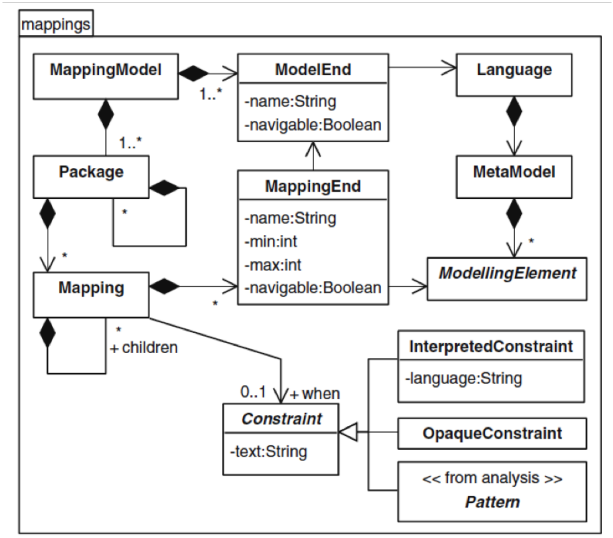

Fig. 16. *trans*ML mapping diagram metamodel

navigability of each rule; these can be executed from a DB model to an OO model, or vice versa.

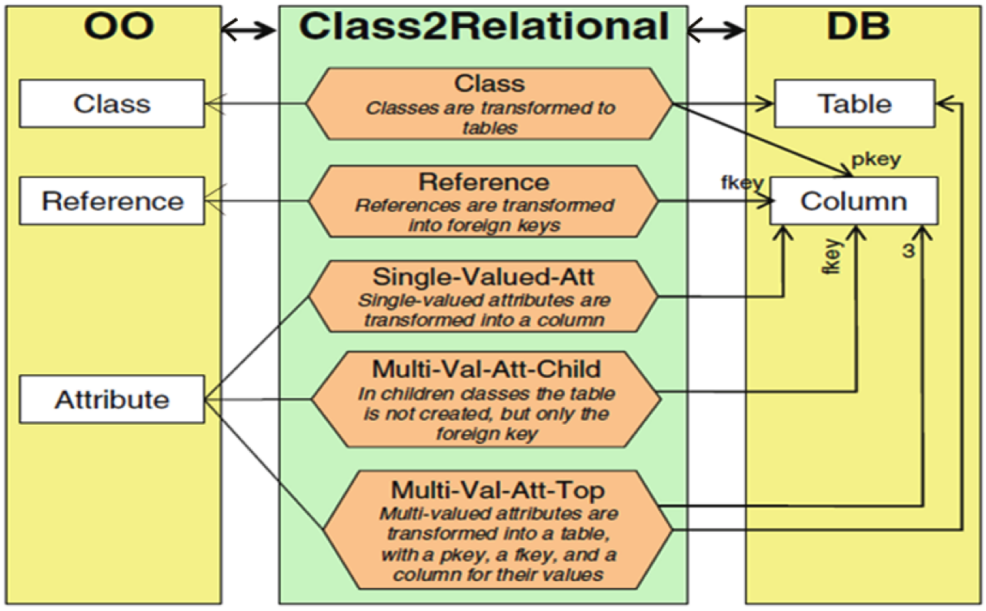

Fig. 17. *trans*ML mapping example

The next example elaborates what is presented in Figure 17 and imposes a constraint on the very last mapping, Multi-Val-Att-Top. The constraint, expressed in OCL, states that the owner of an attribute cannot have any parent classes; this is so that multi-valued attributes can be appropriately flattened into a table.

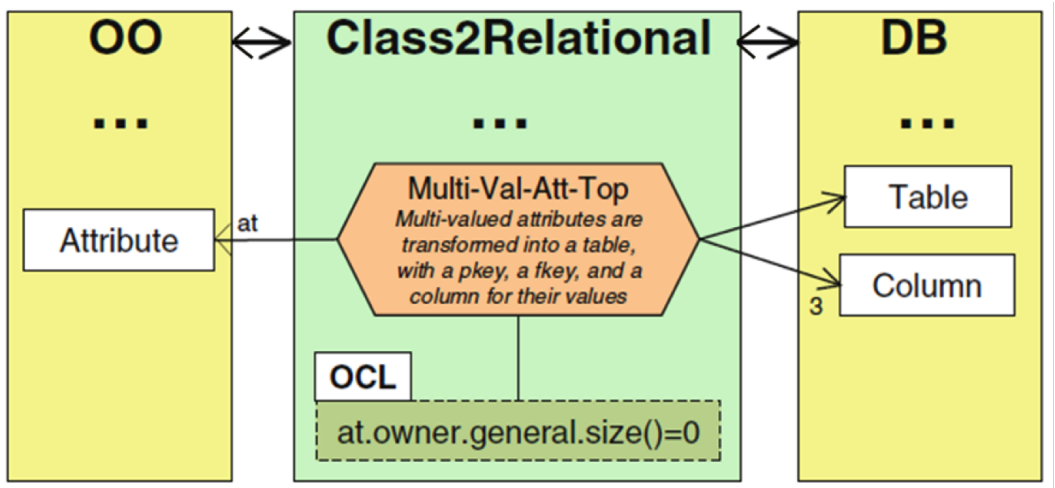

Fig. 18. *trans*ML mapping example (constraint)

While high-level design is supported in *trans*ML via mapping diagrams, low-level design – which is where the transition to implementation begins – is supported by more detailed diagrams. Technically, low-level design *could* be supported by using a favourite BX programming language. But it may be preferable - for reasons of process - to maintain a degree of platform independence while still focusing on the essential aspects of BX development. As such, *trans*ML provides low-level design languages for capturing the *structure* of BX rules, control flow, and blocks. These are encapsulated in two diagrams: the *rule structure diagram* and the *rule behaviour diagram*.

The rule structure diagram (metamodel in Figure 19 is used to refine a mapping diagram. A rule in such a diagram can contribute to the implementation of one or more mappings. Rules themselves may be unidirectional or bidirectional. Structure diagrams also allow for explicit or implicit (e.g., nondeterministic) capture of execution flow, via subclasses of the *Flow* metaclass. In particular, a set of rules can be placed inside a nondeterministic block, for example, as in graph transformation programs.

Effectively, rule structure diagrams capture the structure rules, execution flow and data dependencies. This is illustrated in Figure 20, which shows a directional transformation from an object-oriented model to a database model. The structure in particular is tailored to a representation of rules in the Epsilon

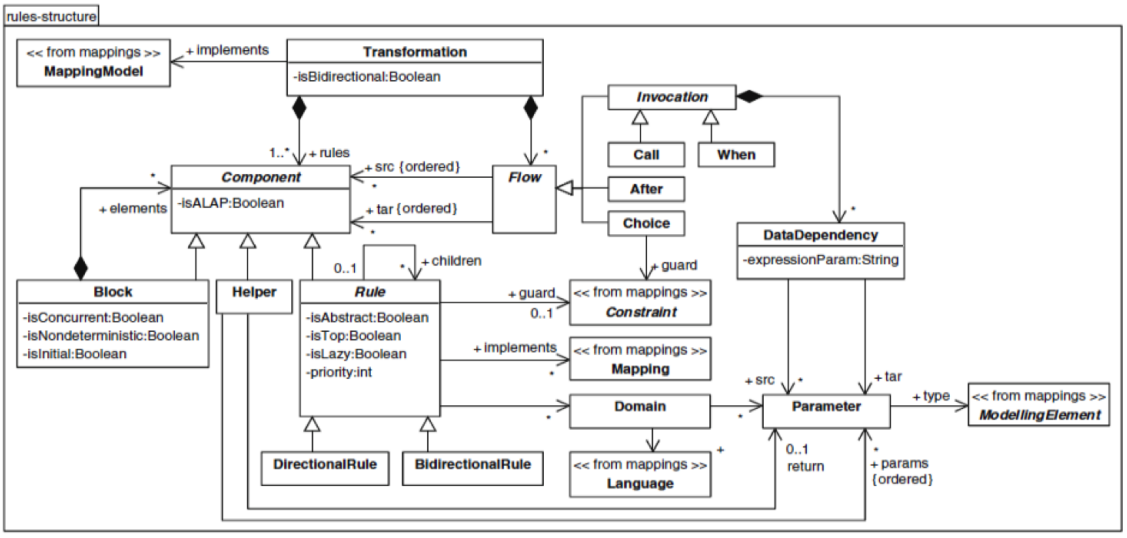

Fig. 19. *trans*ML rule structure diagram metamodel

Transformation Language (ETL). There is a top-level rule (Class2Table) that is executed initially; its execution is followed by a block of rules that execute nondeterministically; these populate the structure of a database table (i.e., Reference2Column, SingleValuedAtt2Column, MultiValuedAtt2Table). Note that blocks can be a useful mechanism for design, even if the ultimate implementation language does not support them (for example, ETL does not support blocks directly).

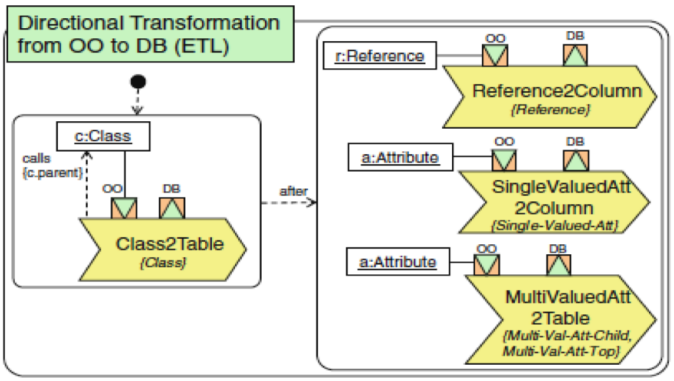

Fig. 20. *trans*ML rule structure diagram example

A second example is shown in Listing 1.2. In this case, a small domain-specific BX language is used to specify parts of a transformation between trees and graphs. The transformation is divided into two nondeterministic blocks; these blocks encapsulate bidirectional rules between elements of one model (e.g., Tree) and elements of a second model (e.g., Node).

Listing 1.2. An example of a BX using blocks

```
transformation Tree2Graph {
  nondeterministic RuleBlockForward {
     bidirectional Tree2Node { ... };
     bidirectional TreeEdge2GraphEdge {...};
   }

   nondeterministic RuleBlockBackward {
      bidirectional TreeLabelsfromNodeLabels {...};
      bidirectional TreeEdgesfromGraphEdges { ... };
    }
}
```

Rule structure diagrams in particular need to take into account the choice of ultimate implementation language. This is because these diagrams capture execution flow, which is platform specific. For example, consider ETL: the execution flow model is such that each rule is executed once at each instance of input; by comparison, in a graph transformation language, execution is for "as long as possible", i.e., until a fix-point is reached. As such, a specific rule structure diagram may be transformed easily to one implementation language, but not another. The metamodel for rule structures is, in our experience, sufficiently generic to capture a number of transformation implementation languages, but there may be specific features of specific implementation languages that we have not considered that are not easily supported.

The rule structure diagram treats rules as black boxes, ignoring their behaviour. As such, concepts such as attribute contribution, object creation, or link configuration will be ignored. These can all be specified using implementation languages such as ETL, but *trans*ML also provides a diagram for their specification: the rule behaviour diagram. This allows the behaviour of rules to be captured using an action language, or declarative graphical pre- and post-conditions, or object diagrams annotated with operations (similar in a sense to Catalysis snapshots). An example unidirectional rule behaviour diagram is shown in Figure 21. On the left of the figure is a snapshot with annotations indicating creation of objects. On the right is the ETL program that would correspond to such a diagram.

It should be noted that while we have broad and quite deep experience of using *trans*ML for engineering unidirectional transformations, we have much less experience of using it for engineering BX. Using some of the features of *trans*ML for capturing different aspects of BX may be a useful contribution to the BX community, as they provide platform-independent ways of specifying different features.

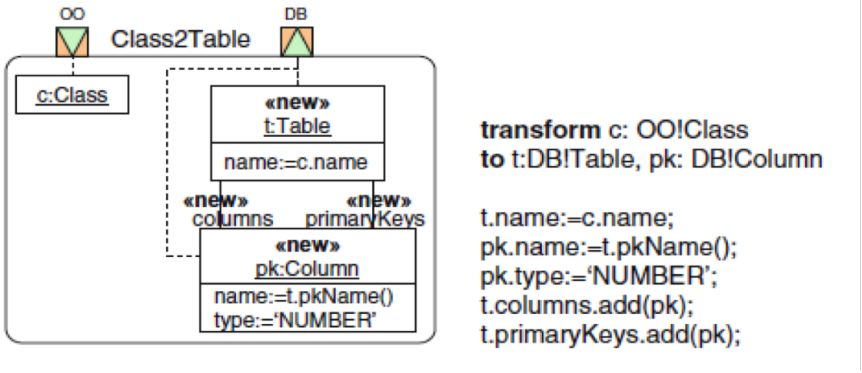

Fig. 21. *trans*ML rule behaviour diagram example

4.3 Design Patterns for BX

In this section we very briefly discuss several design patterns [**?**] for BX. Design patterns in general capture recurring design problems (e.g., in object-oriented design) and their solutions. Solutions generally need to be instantiated for particular problem concepts. Many different patterns have been developed and captured in the literature, including some for model transformations. In this section we present three examples, taken from [11] with some customisation for our context.

Auxiliary Correspondence Model Pattern A special kind of model transformation is a *merging* or *weaving*, where two or more models are combined into a single model. This weaving process can be carried out in batch mode or via a change propagation approach, where changes from the models being combined can be propagated to others. In doing so, most such transformations make use of a so-called *auxiliary correspondence model*. This is a design pattern: the auxiliary correspondence model defines auxiliary model elements and associations that link source and target elements. It can be used to record mappings performed by a BX and to propagate modifications when one model changes. The benefits of using such a pattern is that it separates concerns: the source and target models are kept separate from the connections that link their elements. In turn, these explicit links between source and target model can make it easier to check correctness and coverage in the transformation. The disadvantage of applying this pattern is that it requires maintenance of an additional model.

Unique Instantiation Pattern This pattern focuses on improving the efficiency of transformations. In particular, it is applied to avoid duplicating model elements in either the source or the target of a BX. In particular, the pattern imposes a check that an element satisfying specified properties does not exist, before the element is actually created in the source or target. For example, in a

QVT-Relations transformation that has applied this pattern, new elements will not be created if there are already elements that satisfy the relations specified; this is really at the heart of check-before-enforce mode in QVT-Relations. The benefit of using this pattern is that it can help ensure hippocraticness; the disadvantage is the test for existence, which can degrade performance. However, we note that other patterns, e.g., related to indexing [11] – and model indexing frameworks like Hawk – can help offset this.

Map Objects Before Links Pattern This pattern is used to separate the relation between elements in source and target models from the relations between *links* in the models. A particular application of this pattern would be to structure a transformation wherein model elements are transformed before the relations between model elements (i.e., nodes before edges). Such an execution flow may be useful in cases where models may have self-associations or circular dependencies. The benefit of using this pattern is similar to that of the Visitor pattern in object-oriented design: the specification of the transformation is modular and processing for a new type of association in a modelling language can be more easily handled. The disadvantage of using this pattern is that while edges (relations) are treated modularly, nodes (model elements) may not be, and if a new feature is added to a language, it may require significant restructuring to the transformation that has used this pattern.

4.4 Summary

In this section we have discussed different aspects of the architecture and design of BX, covering abstract architecture for transformations, through high-level and low-level design, including behaviour of individual transformation rules, as well as a selection of design patterns that can be used to help increase cohesion and decrease coupling in our BX. We will next briefly discuss an approach to verification of BX, focusing on use of mathematical techniques.

5 Verification

In this section we explore a specific 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 [12]), whereas others require users to work with specific formalisms such as triple graph grammars (e.g. MOFLON [13]). Many modern transformation languages do not provide any support for BX (e.g. ATL [14]), meaning that users must express them as two unidirectional transformations. While this seems a practical workaround,

the two transformations may diverge over time – that is, there are no constructive guarantees that the two unidirectional transformations maintain the consistency relationship between the models being manipulated.

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 and executable 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) [15] style. Furthermore, it provides an inter-model consistency language (EVL [16]) that can be used to express and evaluate constraints between models. With these languages, 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. Thus, how do we check for, and maintain, compatibility?

We aim to obtain the guarantees of BX without the need for BX languages. Instead, we can 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 [17]. Then, by leveraging graph transformation proof calculi [18–20] in a weakest precondition style, we aim to automatically prove compatibility of the unidirectional transformations with respect to the EVL constraints.

5.1 Illustration

To illustrate the idea, consider yet again the OO to RDBMS problem. Class diagram models conform to a simple language describing familiar object-oriented concepts, whereas relational database models conform to a language describing how databases are constructed. Consistency is defined in terms of a correspondence between the data in the models, e.g. every table n corresponds to a class n, and every column m corresponds to an attribute m. Figure 22 contains two simple models that are consistent in this sense (we omit the metamodels, but they are obvious).

Users of the models should be able to create new classes (or tables) whilst maintaining inter-model consistency. Upon the creation of a new class (resp. table), a table (resp. class) should be created with the same name to restore consistency. We can implement such a simple BX in Epsilon with a pair of unidirectional transformations (one for updating the class diagram model, one for updating the relational database) and a set of EVL constraints. For the former,

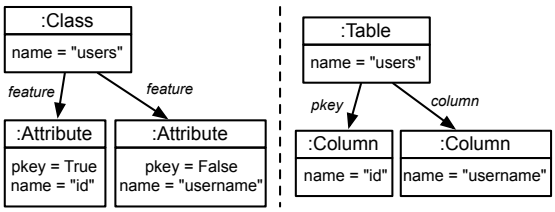

Fig. 22. Two consistent OO and RDBMS models

we can use the Epsilon Wizard Language (EWL) to define a pair of update-in-place transformations, `AddClass` and `AddTable` (for simplicity, here we assume the new class/table name `newName` to be pre-determined and unique, but Epsilon does support the capturing and sharing of such data between wizards).

```
wizard AddClass {
 do {
    var c: new Class;
    c.name = newName;
    self.Class.all.first().contents.add(c);
 }}


wizard AddTable {
 do {
    var table: new Table;
    table.name = newName;
    self.Table.all.first().contents.add(table);
 }}
```

Using the Epsilon Validation Language (EVL), we express inter-model consistency: that for every class n, there exists a table named n (and vice versa). If one of the constraints is violated, Epsilon can automatically trigger the relevant transformation to attempt to restore consistency. For example, after executing the transformation `AddClass`, the constraint `TableExists` will be violated, indicating that the transformation `AddTable` should be executed to restore consistency.

```
context OO!Class {
 constraint TableExists {
    check : DB!Table.all.select(t|t.name = self.name).size() > 0
}}


context DB!Table {
 constraint ClassExists {
    check : OO!Class.all.select(c|c.name = self.name).size() > 0
}}
```

This example of a BX, "faked" in Epsilon, is a simple one chosen to illustrate the concepts. Even what appears to be a simple BXcan lead to more interesting (i.e. less symmetric) BX, e.g. manipulating inheritance in the class model.

5.2 Checking Compatibility

A critical difference between the "faked" BX in the previous section and a true BX is the absence of guarantees about the compatibility of the transformations: upon the violation of `TableExists`, for example, does the execution of `AddTable` actually restore consistency? For this simple example, a manual inspection will quickly confirm that the transformations are indeed compatible. But what about more intricate BX? And what about BX that evolve and change over time? For the Epsilon-based approach to be a convincing alternative to a BX language, it is imperative that the compatibility of the transformations can be checked, and that this can be done in a simple and automatic way. To this end, we propose to leverage and adapt some recent developments in the verification of graph transformations.

Graph transformation is a computation abstraction: the state of a computation is represented as a graph, and the computational steps as applications of rules (i.e. akin to string rewriting in Chomsky grammars, but lifted to graphs). Modelling a problem using graph transformation brings an immediate benefit in visualisation, but also an important one in terms of semantics: the abstraction has a well-developed algebraic theory that can be used for formal reasoning. This has been exploited to facilitate the verification of graph transformation systems, i.e. calculi for systematically proving specifications about graph properties before and after any execution of some given rules. Furthermore, such calculi have been generalised to graph programs [21], which augment the abstraction with expressions over labels and familiar control constructs (e.g. sequential composition, branching) for restricting the application of rules. In particular, we look to exploit work by Poskitt and Plump who developed proof calculi for graph programs, separately addressing reasoning about programs and properties involving attribute manipulation [19, 20], as well as reasoning about non-local structural properties [22].

Our example BX for the OO2RDBMS problem is easily translated into graph programs and nested conditions, as given in Figure 23. The programs P_S, P_T are the individual rules creating respectively a class or table node labelled `newName` (here, $\emptyset$ denotes the empty graph, indicating that the rules can be applied without first matching any structure, i.e. unconditionally). The nested condition evl, given on the right, expresses that for every class (resp. table) node, there is a table (resp. class) node with the same name (we do not define here a formal interpretation, but note that `x`, `y` are variables, and that the numbers indicate when nodes are the same down the nesting of the formula). Were the weakest liberal preconditions to be constructed, we would find:

$$\mathrm{Wlp}(P_S; P_T, evl) \equiv \mathrm{Wlp}(P_T; P_S, evl) \equiv evl.$$

Since $evl \Rightarrow evl$ is clearly valid, both $\{evl\}\ P_S;\ P_T\ \{evl\}$ and $\{evl\}\ P_T;\ P_S\ \{evl\}$ must hold, and—assuming correctness of the abstractions—the original EWL transformations are therefore compatible with respect to the EVL constraints.

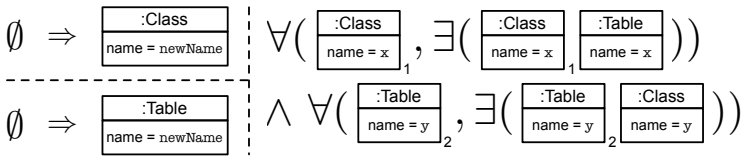

Fig. 23. Our CD2RDBM BX expressed as graph transformation rules and a nested condition

A key challenge with an approach such as this is what to do when the verification step fails, i.e., the implication above does not hold. We are exploring the use of the GROOVE tool to generate counterexamples when verification fails, via exploring executions of the graph transformation rules.

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[9] 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 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).

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

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. Sobhan Yassipour Tehrani, Steffen Zschaler, and Kevin Lano. Requirements engineering in model-transformation development: An interview-based study. In *Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings*, pages 123–137, 2016.
7. Soroosh Nalchigar, Rick Salay, and Marsha Chechik. Towards a catalog of non-functional requirements in model transformation languages. In *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, 2013.
8. IEEE 29148-2011. Systems and software engineering lifecycle processes requirements engineering, 2011.
9. Ana Pescador and Juan de Lara. Dsl-maps: from requirements to design of domain-specific languages. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 438–443, 2016.
10. Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. Engineering model transformations with transml. *Software and System Modeling*, 12(3):555–577, 2013.
11. Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Model transformation design patterns. *IEEE Transactions on Software Engineering*, 40(12), 2014.
12. Peter Braun and Frank Marschall. Transforming object oriented models with BOTL. In *GT-VMT 2002*, volume 72 of *ENTCS*, pages 103–117. Elsevier, 2003.
13. 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.
14. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
15. 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.

16. 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.
17. 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.
18. 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.
19. Christopher M. Poskitt. *Verification of Graph Programs.* PhD thesis, The University of York, 2013.
20. Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
21. Detlef Plump. The design of GP 2. In *WRS 2011*, volume 82 of *EPTCS*, pages 1–16, 2012.
22. Christopher M. Poskitt and Detlef Plump. Verifying monadic second-order properties of graph programs. In *ICGT 2014*, volume 8571 of *LNCS*, pages 33–48. Springer, 2014.