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

3 Requirements Engineering

4 Architecture and Design

5 Verification

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[1] 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).

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