ETH zürich

ETH Library

Towards rigorously faking bidirectional model transformations

Conference Paper

Author(s):
Poskitt, Christopher M.; Dodds, Mike; Paige, Richard F.; Rensink, Arend

Publication date:
2014-10

Permanent link:
https://doi.org/10.3929/ethz-a-010217731

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
CEUR Workshop Proceedings 1277

Funding acknowledgement:
291389 - Concurrency Made Easy (EC)

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Towards Rigorously Faking Bidirectional Model Transformations

Christopher M. Poskitt[1*], Mike Dodds[2], Richard F. Paige[2], and Arend Rensink[3]

[1] Department of Computer Science, ETH Zürich, Switzerland
[2] Department of Computer Science, The University of York, UK
[3] Department of Computer Science, University of Twente, The Netherlands

Abstract. Bidirectional model transformations (*bx*) are mechanisms for automatically restoring consistency between multiple concurrently modified models. They are, however, challenging to implement; many model transformation languages not supporting them at all. In this paper, we propose an approach for automatically obtaining the consistency guarantees of *bx* without the complexities of a *bx* language. First, we show how to "fake" true bidirectionality using pairs of unidirectional transformations and inter-model consistency constraints in Epsilon. Then, we propose to automatically verify that these transformations are consistency preserving—thus indistinguishable from true *bx*—by defining translations to graph rewrite rules and nested conditions, and leveraging recent proof calculi for graph transformation verification.

1 Introduction and Motivation

Model transformations are operations for automatically translating models conforming to one language (i.e. a metamodel) into models conforming to another, in a way that maintains some sense of consistency between them. At their most basic, model transformations are unidirectional: given a source model (e.g. some high-level yet user-modifiable view of a system), they generate a target model (perhaps a lower-level view, such as code) whose data is "consistent" with the source, in a sense that is either left implicit, or captured by textual constraints or an inter-model consistency relation.

Many situations arise where the source and target models may *both* be modified by users in concurrent engineering activities, e.g. when integrating parts of systems that are modelled separately but must remain consistent. Bidirectional model transformations (*bx*) [5,17] are a mechanism for automatically restoring inter-model consistency in such a scenario; in particular, *bx* simultaneously describe transformations in both directions—from source to target and target to source—with their compatibility guaranteed by construction [5].

While this advantage is certainly an attractive one, *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 [3]), essentially restricting their use to models presenting identical data in

* Received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389.

different ways, whereas others require users to work with specific formalisms such as triple graph grammars (e.g. MOFLON [1]). The QVT-R language—part of the OMG's Queries, Views, and Transformations standard—allows *bx* to be expressed, but suffers an ambiguous semantics [18] and limited tool support (the most successful ones often departing from the original semantics [11]). Moreover, many modern transformation languages do not provide any support for *bx* (e.g. ATL [9]), meaning that users must express them as two separate unidirectional transformations. While this seems a practical workaround, it comes with the major risk that the compatibility of the transformations might not be maintained over time.

A trade-off between the benefits (but complexity) of *bx* and the practicality (but possible incoherence) of unidirectional transformations can be achieved in Epsilon, a platform of interoperable model management languages. Epsilon has languages supporting the specification of unidirectional transformations in either a rule-based (ETL), update-in-place (EWL), or operational (EOL) [12] style. Furthermore, it provides an inter-model consistency language (EVL [10]) that can be used to express and evaluate constraints between models conforming to different metamodels. With these languages together, *bx* can be "faked" in a practical way, by: (1) defining pairs of unidirectional transformations for separately updating the source and target models; and (2) defining "consistency" via inter-model constraints in EVL, the violation of which will trigger appropriate transformations to restore consistency.

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

We aim to address this shortcoming and obtain the guarantees of *bx* without the need for *bx* languages. Instead, we will use *rigorous* proof techniques to verify that faked *bx* are consistency preserving, and thus indistinguishable to users from true *bx*. To this end, we propose to apply techniques from graph transformation verification. Given a faked *bx* in Epsilon, we will model the unidirectional transformations as graph transformation rules, and EVL constraints as nested graph conditions [7]. Then, by leveraging graph transformation proof calculi [8,14,15] in a weakest precondition style, we aim to automatically prove compatibility of the unidirectional transformations with respect to the EVL constraints. Furthermore, we aim to exploit the model checker GROOVE [6] to automatically search for counterexamples when consistency preservation does not hold.

The overarching goal of our work is to achieve the ideal that Stevens [17] contemplated in her survey of *bx*: that *"if a framework existed in which it were possible to write the directions of a transformation separately and then check, easily, that they were coherent, we might be able to have the best of both worlds."*

2 Example *bx* in Epsilon: Class Diagrams to Databases

To illustrate the ideas of our proposal, we recall a common model transformation problem that concerns the consistency of class diagram and relational database models (CD2RDBM). Class diagram models conform to a simple language describing famil-

iar object-oriented concepts (e.g. classes, attributes, relationships), whereas relational database models conform to a language describing how databases are constructed (e.g. tables, columns, primary keys). Here, 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 1 contains two simple models that are consistent in this sense (we omit the metamodels for lack of space).

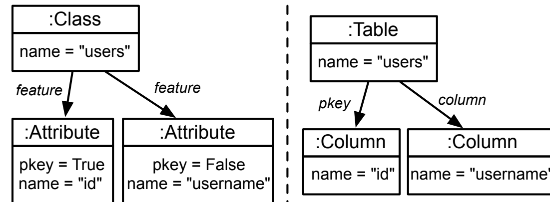

Fig. 1. Two consistent CD and RDB models

Users of the models should be able to create new classes (or tables) whilst maintaining inter-model consistency. A *bx* would be well suited for this: 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 fake this 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, 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 the relevant notion of 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 deliberately simple one chosen to illustrate the concepts. Note even that the CD2RDBM problem can lead to more interesting (i.e. less symmetric) *bx*, e.g. manipulating inheritance in the class model.

3 Checking Compatibility of the Transformations

The 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 in this sense. 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 (or not) of the transformations can be checked, and—crucially—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 [13], which augment the abstraction with expressions over labels and familiar control constructs (e.g. sequential composition, branching) for restricting the application of rules.

Habel et al. [7,8] developed weakest precondition calculi for proving specifications of the form $\{pre\}\ P\ \{post\}$, which express that if a graph satisfies the precondition pre, then any graph resulting from the execution of graph program P will satisfy the postcondition $post$; these pre- and postconditions expressed using nested conditions, a graphical formalism for first-order (FO) structural properties over graphs. They defined constructions that, given a nested condition $post$ and program P, would return a weakest liberal precondition $\mathrm{Wlp}(P, post)$, representing the weakest property that must hold for successful executions of P to establish $post$. The specification would then be (dis)proven by checking the validity of $pre \Rightarrow \mathrm{Wlp}(P, post)$ in an automatic FO theorem prover. Poskitt and Plump developed proof calculi in a similar spirit, separately addressing two extensions: programs and properties involving attribute manipulation [14,15], and reasoning about non-local structural properties [16].

We aim to exploit this work to check the compatibility of transformations in the Epsilon approach to *bx*. In particular, we are developing automatic translations of EWL transformations to graph programs (denoted P_S, P_T for the source and target updates respectively), and translations from EVL constraints to nested conditions (denoted evl). Then, the task of checking compatibility of the transformations, as shown in Figure 2, reduces to proving the specifications $\{evl\}\ P_S;\ P_T\ \{evl\}$ and $\{evl\}\ P_T;\ P_S\ \{evl\}$ (here we assume the input graphs to be disjoint unions of the two models). Intuitively: if the models are consistent to start with, and executing the transformations in either order maintains consistency, then the transformations are compatible.

The technical challenges of the process fall into two main parts: computing the abstractions, and checking validity. Defining translations for the former requires care: we need to determine how much of the EWL language can be handled, we need to ensure that the graph-based semantics we abstract them to is "correct", and we need to adapt the proof technology to our specific needs. The work in [14,15,16], for example, does not presently support type graphs (causing more effort to encode conformance to

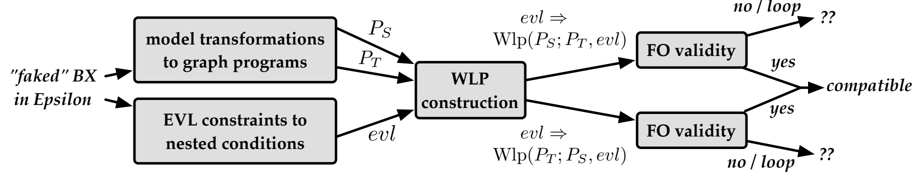

Fig. 2. Overview of the process for checking compatibility of the transformations

metamodels). Similar concerns must be addressed for the translations of EVL to nested conditions (we can take inspiration from recent work on such translations for core OCL [2]). For the challenge of checking validity, we aim to leverage existing FO theorem provers (e.g. Vampire) as much as possible, adapting existing translations of nested conditions to FO logic [7,14]. Given the undecidability of FO validity, we also aim to explore the use of the GROOVE model checker [6] in finding counterexamples when the theorem provers respond with "no", or do not appear to terminate.

Our example *bx* for the CD2RDBM problem is easily translated into graph programs and nested conditions, as given in Figure 3. 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:

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

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.

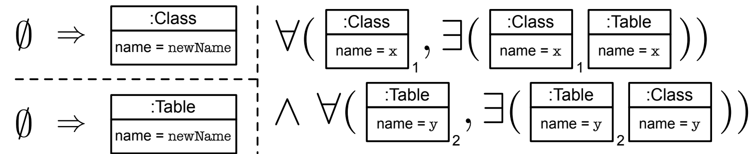

Fig. 3. Our CD2RDBM *bx* expressed as graph transformation rules and a nested condition

4 Next Steps

After further exploring the CD2RDBM example, we will identify a selection of *bx* case studies—from the community repository [4] and beyond—that exhibit a broader range of characteristics and challenges to address. We will implement these *bx* using EWL transformations and EVL constraints, then manually translate them into graph transformations and nested conditions. These will serve as a proof of concept, but also as

guidance, helping us to determine how far we should adapt the proof calculi to support our goals (e.g. introducing type graphs for capturing the metamodels). After implementing the weakest precondition calculations and translations to FO logic, we will design and implement automatic translations from Epsilon *bx* to their corresponding graph-based abstractions, initially focusing on a core (but expressive) subset of the languages. Finally, we will explore the use of GROOVE in finding counterexamples when verification fails, by exploring executions of the graph transformation rules.

References

1. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A standard-compliant metamodeling framework with graph transformations. In: ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer (2006)
2. Arendt, T., Habel, A., Radke, H., Taentzer, G.: From core OCL invariants to nested graph constraints. In: ICGT 2014. LNCS, vol. 8571, pp. 97–112. Springer (2014)
3. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. In: GT-VMT 2002. ENTCS, vol. 72, pp. 103–117. Elsevier (2003)
4. Cheney, J., McKinna, J., Stevens, P., Gibbons, J.: Towards a repository of Bx examples. In: EDBT/ICDT Workshops. vol. 1133, pp. 87–91. CEUR-WS.org (2014)
5. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer (2009)
6. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Software Tools for Technology Transfer 14(1), 15–40 (2012)
7. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19(2), 245–296 (2009)
8. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer (2006)
9. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
10. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 204–218. Springer (2009)
11. Macedo, N., Cunha, A.: Implementing QVT-R bidirectional model transformations using Alloy. In: FASE 2013. LNCS, vol. 7793, pp. 297–311. Springer (2013)
12. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: ICECCS 2009. pp. 162–171. IEEE Computer Society (2009)
13. Plump, D.: The design of GP 2. In: WRS 2011. EPTCS, vol. 82, pp. 1–16 (2012)
14. Poskitt, C.M.: Verification of Graph Programs. Ph.D. thesis, The University of York (2013)
15. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae 118(1-2), 135–175 (2012)
16. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: ICGT 2014. LNCS, vol. 8571, pp. 33–48. Springer (2014)
17. Stevens, P.: A landscape of bidirectional model transformations. In: GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer (2007)
18. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and System Modeling 9(1), 7–20 (2010)