Towards Rigorously Faking Bidirectional Model Transformations
AMT at MODELS 2014, Valencia

Chris Poskitt
Mike Dodds
Richard Paige
Arend Rensink

ETH Zürich  University of York
UNIVERSITY OF TWENTE.
Unidirectional model transformations

• translate models in some source language to models in some target language

• maintain some sense of consistency between the models
Unidirectional model transformations

- translate models in some source language to models in some target language
- maintain some sense of consistency between the models
What if users modify both models?

• in some scenarios, users may modify both models in concurrent engineering activities

• e.g. database view problem, system integration

• maintaining consistency still important - but harder
Bidirectional transformations (bx)

- bidirectional model transformations (bx) simultaneously describe transformations in both directions

- compatibility of the directions guaranteed
  => *i.e. both directions maintain consistency of models*

- **BUT**: inherently complex and difficult to implement
  => *many model transformation languages do not support bx*
  => *others do, with conditions (e.g. bijective, TGGs)*
  => *QVT-R supports bx, but suffers an ambiguous semantics*
Is there another way?

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

“Faking” bx in Epsilon

• **Epsilon** is a platform of interoperable model management languages

• no direct support for bx, but:
  => languages for unidirectional transformations (ETL, EWL, EOL)
  => an inter-model consistency language (EVL)

• bx can be faked in Epsilon by:
  (1) defining pairs of unidirectional transformations
  (2) defining consistency via inter-model constraints

![Diagram showing update transformation, constraint violation, and repair transformation]
Class Diagrams to Relational Databases
(*the forbidden example*)

- two metamodels: class diagram and relational DB

- consistency defined in terms of a correspondence between the data (attributes) in the models

```
Fig. 1. Two consistent CD and RDB models in the previous section and a true.
```

```
| :Class          | :Table
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name = &quot;users&quot;</td>
<td>name = &quot;users&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| :Attribute      | :Column
| pkey = True     | pkey = False    |
| name = "id"    | name = "username" |
|                 |                 |
```
Example bx “faked” in Epsilon

- users of the models should be able to **create new classes (or tables)** whilst maintaining consistency

- first, we specify a **pair of unidirectional transformations** in Epsilon’s update-in-place language

```plaintext
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...
then, we specify and monitor inter-model constraints that express what it means to be consistent

```
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
    }
}
```
then, we specify and monitor inter-model constraints that express what it means to be consistent

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
  }
}
We didn’t quite fake everything yet...

- fake bx lack the consistency guarantees that true bx have by construction

- what does this mean?
  => compatibility of the directions might not be maintained
  => repair transformations might not actually restore consistency

- our example is obviously compatible, but we should be able to check this easily and automatically
Our proposal: exploit graph transformation verification techniques to check compatibility

• graph transformation (GT) is a computation abstraction
  => state is represented as a graph
  => computational steps represented as GT rule applications
Our proposal: exploit graph transformation verification techniques to check compatibility

- **graph transformation (GT)** is a computation abstraction
  - => state is represented as a graph
  - => computational steps represented as GT rule applications

```plaintext
init:
  ∅  ⇒  •

grow:
  •  ⇒  •
```

Example 2.
Our proposal: exploit graph transformation verification techniques to check compatibility

- **graph transformation (GT)** is a computation abstraction
  => state is represented as a graph
  => computational steps represented as GT rule applications

```
init:
\emptyset \Rightarrow \bullet

grow:
1 \Rightarrow 1 \rightarrow \bullet
```

```
\emptyset \Rightarrow \bullet \Rightarrow \bullet \rightarrow \bullet
```

```grow
\Rightarrow \bullet \rightarrow \bullet
```

```grow
\Rightarrow \bullet \rightarrow \bullet
```

```grow
\Rightarrow \bullet \rightarrow \bullet
```
GT verification techniques

- functional correctness of GT rules can be verified in a weakest precondition style

- pre- and postconditions are expressed in the graph-based logic of nested conditions, equiv. to FO logic

- roughly, to verify \{pre\} P \{post\}:
  
  \[ WP(P, post) \Rightarrow \text{pre} \]
How we will **rigorously fake bx**

- translate the unidirectional transformations to **GT rules**
  => denoted $P_S$ and $P_T$

- translate the inter-model constraints to **nested conditions**
  => denoted $evl$

- automatically discharge the following specifications using the **weakest precondition calculi**

\[
\{evl\} P_S; P_T \{evl\} \quad \{evl\} P_T; P_S \{evl\}
\]
Proving consistency of our CD/DB bx

\[
P_S \quad \Rightarrow \quad \text{\begin{tabular}{|c|}
\hline
:Class \\
\hline
name = newName \\
\hline
\end{tabular}}
\]

\[
P_T \quad \Rightarrow \quad \text{\begin{tabular}{|c|}
\hline
:Table \\
\hline
name = newName \\
\hline
\end{tabular}}
\]

\[
evl
\forall (\forall (\exists (\exists (\forall (\exists ))))
\]

\[
\forall (\forall (\exists (\exists (\forall (\exists ))))
\]

\[
\forall (\forall (\exists (\exists (\forall (\exists ))))
\]
Proving consistency of our CD/DB bx

\[ \forall (\text{:Class} \_1, \exists (\text{:Class} \_1, \text{:Table} \_2, \exists (\text{:Table} \_2, \text{:Class} \_2))) \]
\[ \land \forall (\text{:Table} \_2, \exists (\text{:Table} \_2, \text{:Class} \_2)) \]

\( \text{compatible: } WP(P_S; P_T, evl) \equiv WP(P_T; P_S, evl) \equiv evl \)
Putting it all together

we need to do this bit

exploit existing theorem provers here
Our next steps

• identify a selection of bx case studies

• fake them in Epsilon, manually translate them into GT rules and nested conditions, and verify compatibility

• implement the translations for an expressive subset of the Epsilon languages; implement the WP calculation

• challenges and open questions:
  => finding counterexamples (e.g. using GROOVE)
  => theoretical / practical limitations (e.g. is FO expressive enough?)
In summary

• bx simultaneously describe transformations in both directions - compatible by construction

• but they are inherently complex and challenging to implement

• can be faked in Epsilon as pairs of unidirectional transformations and inter-model consistency constraints

• we will leverage GT proof technology to obtain compatibility guarantees for faked bx