Families to Persons Case with UML-RSDS

K. Lano,
Dept. of Informatics, King's College London
S. Kolahdouz-Rahimi,
Dept. of Software Engineering, University of Isfahan, Iran

Abstract. In this paper we describe a solution to the families to persons bx case using the UML-RSDS subset of UML and OCL.

1 Introduction

UML-RSDS uses a subset of UML 2 and OCL 2.4 to specify general applications, and model transformations. Application data and MT metamodels are defined by class diagrams, and application functionality and transformations are defined by the postconditions of use cases [3]. Executable code in Java, C#, C and C++ can be automatically generated from UML-RSDS specifications.

Bx specification is supported by the automated synthesis of inverse transformations from the forward direction of a transformation (provided this satisfies some restrictions) [2]. Transformations can also be applied incrementally.

In terms of the case description, ours is a correspondence-based approach. Change-propagation of all four types is supported in principle, although we have only implemented add and attribute changes for this case.

2 Forward transformation

Figure 1 shows the class diagram of the person and family models, enhanced by primary keys *personId* and *memberId* recording the source-target correspondence, according to the Auxiliary Correspondence Model pattern [3]. The forward transformation is defined by the use case *person2family*, its inverse is defined by *family2person*.

According to the case statement, the following invariants (I1), (I2), (I3), (I4) must be established and maintained by the transformations:

$$
\begin{aligned}
(I1):\ & \mathit{Family}{\rightarrow}\mathit{forAll}(\mathit{fam}\ | \\
& \quad \mathit{FamilyMember}{\rightarrow}\mathit{forAll}(m\ |\ m \in \mathit{fam.mother}{\rightarrow}\mathit{union}(\mathit{fam.daughters})\ \Rightarrow \\
& \qquad \mathit{Female}{\rightarrow}\mathit{exists}(f\ |\ f.\mathit{personId} = m.\mathit{memberId}\ \& \\
& \qquad\quad f.\mathit{familyId} = \mathit{fam.id}\ \& \\
& \qquad\quad f.\mathit{name} = \mathit{fam.name} + \text{", "} + m.\mathit{name})))
\end{aligned}
$$

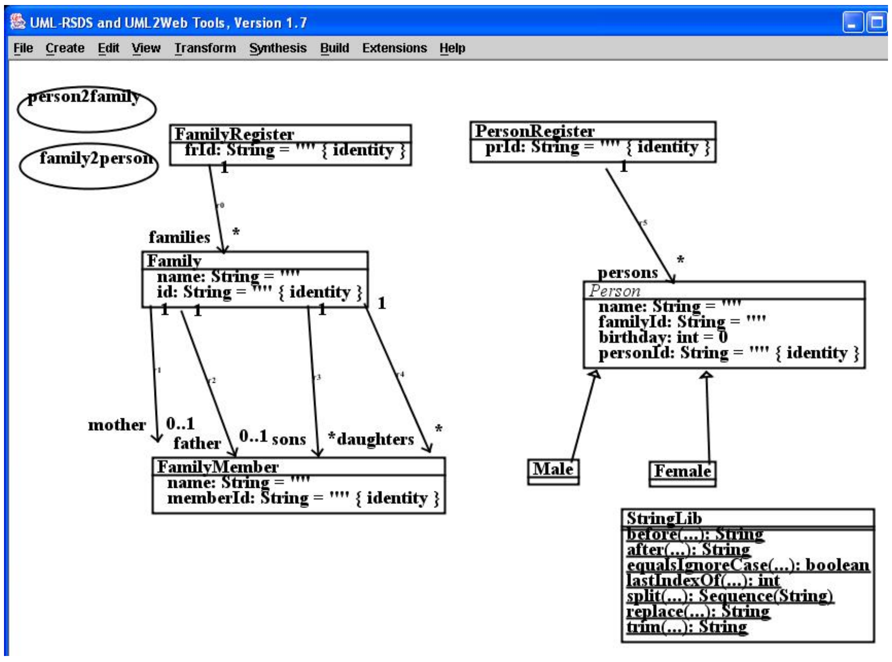

Fig. 1. Metamodels of Families to Persons case

($I2$):

$$\begin{array}{l} \mathit{Family}{\rightarrow}\mathit{forAll}(\mathit{fam} \mid \\ \quad\quad \mathit{FamilyMember}{\rightarrow}\mathit{forAll}(m \mid m \in \mathit{fam.father}{\rightarrow}\mathit{union}(\mathit{fam.sons}) \;\Rightarrow \\ \quad\quad\quad\quad \mathit{Male}{\rightarrow}\mathit{exists}(f \mid f.\mathit{personId} = m.\mathit{memberId}\ \& \\ \quad\quad\quad\quad\quad\quad f.\mathit{familyId} = \mathit{fam.id}\ \& \\ \quad\quad\quad\quad\quad\quad f.\mathit{name} = \mathit{fam.name} + \text{", "} + m.\mathit{name}))) \end{array}$$

These express that the families model is consistent wrt the persons model.

($I3$):

$$\begin{array}{l} \mathit{Female}{\rightarrow}\mathit{forAll}(f \mid \\ \quad\quad \mathit{FamilyMember}{\rightarrow}\mathit{exists}(m \mid m.\mathit{memberId} = f.\mathit{personId}\ \& \\ \quad\quad\quad\quad \mathit{Family}{\rightarrow}\mathit{exists}(\mathit{fam} \mid \mathit{fam.id} = f.\mathit{familyId}\ \& \\ \quad\quad\quad\quad\quad\quad m \in \mathit{fam.mother}{\rightarrow}\mathit{union}(\mathit{fam.daughters})\ \& \\ \quad\quad\quad\quad\quad\quad f.\mathit{name} = \mathit{fam.name} + \text{", "} + m.\mathit{name}))) \end{array}$$

($I4$):

$$\begin{array}{l} \mathit{Male}{\rightarrow}\mathit{forAll}(f \mid \\ \quad\quad \mathit{FamilyMember}{\rightarrow}\mathit{exists}(m \mid m.\mathit{memberId} = f.\mathit{personId}\ \& \\ \quad\quad\quad\quad \mathit{Family}{\rightarrow}\mathit{exists}(\mathit{fam} \mid \mathit{fam.id} = f.\mathit{familyId}\ \& \\ \quad\quad\quad\quad\quad\quad m \in \mathit{fam.father}{\rightarrow}\mathit{union}(\mathit{fam.sons})\ \& \\ \quad\quad\quad\quad\quad\quad f.\mathit{name} = \mathit{fam.name} + \text{", "} + m.\mathit{name}))) \end{array}$$

These constraints express that the persons model is consistent wrt the families model.

As usual for bx, the relations (I3) and (I4) characterising one direction of the transformation are logical duals of the relations (I1) and (I2) characterising the opposite direction. (I3) can be mechanically derived from (I1), and vice-versa, and similarly for (I4) and (I2). The conjunction of (I1), (I2), (I3), (I4) expresses the bijective correspondence of persons and family members, and the bx relation R of the transformation.

The mapping from families to persons is already defined explicitly by (I1) and (I2). These can be expressed in OCL as postconditions (transformation rules) of the use case for the *family2person* transformation:

```
Family::
 m : mother->union(daughters) =>
    Female->exists( f | f.personId = m.memberId &
      f.familyId = id &
      f.name = name + ", " + m.name )

Family::
 m : father->union(sons) =>
    Male->exists( f | f.personId = m.memberId &
      f.familyId = id &
      f.name = name + ", " + m.name )
```

The semantics of $E \rightarrow exists(e \mid e.eId = v \ \& \ P)$ in the case that *eId* is an identity attribute of E is that the E object with *eId* value equal to v is looked up, if it exists, and is then modified according to P. If the object does not exist, it is created and then modified.

(I3) and (I4) are invariants of this transformation (they are preserved by each application of either of the above rules). On the other hand, if (I1) and (I2) already hold between the family and person models, then re-applying *family*2*person* should have no effect. In particular, the birthday of existing persons is not modified by *family*2*person* if (I1) and (I2) hold.

An additional postcondition sets *PersonRegister* :: *persons* to be all existing *Person* instances:

```
PersonRegister::
  persons = Person.allInstances
```

To define the mapping from persons to families, we logically strengthen (I3) and (I4) and turn them into explicit constraints by enforcing that persons are mapped to parents unless there is already a parent of that gender in a family, in which case they are mapped to children. We assume that *mother*, *father*, *sons*, *daughters* are pairwise disjoint. We also use String library operations *before* and *after* to split the person name:

```
Female::
 FamilyMember->exists( m | m.memberId = personId &
   Family->exists( fam | fam.id = familyId &
     (fam.mother@pre.size = 0 => m : fam.mother & m /: fam.daughters) &
     (fam.mother@pre.size > 0 & fam.mother@pre->excludes(m) => m : fam.daughters) &
     fam.name = StringLib.before(name, ", ") &
     m.name = StringLib.after(name, ", ") ) )

Male::
 FamilyMember->exists( m | m.memberId = personId &
   Family->exists( fam | fam.id = familyId &
     (fam.father@pre.size = 0 => m : fam.father & m /: fam.sons) &
     (fam.father@pre.size > 0 & fam.father@pre->excludes(m) => m : fam.sons) &
     fam.name = StringLib.before(name, ", ") &
     m.name = StringLib.after(name, ", ") ) )
```

(I1) and (I2) are clearly invariants of this transformation. Again, if (I3) and (I4) already hold, applying *person*2*family* should not modify the target model.

An additional postcondition sets the *FamilyRegister* families to be all existing *Family* instances:

```
FamilyRegister::
  families = Family.allInstances
```

3 Change propagation

Changes to the family model should propagate to the person model, and vice-versa, as follows (Table 1).

family model change	*person model change*
New *FamilyMember*	new *Male* or *Female*
New *Family*	no change
Changed *Family* :: *name*	changed *name* for each person from the family
Changed *FamilyMember* :: *name*	changed *name* for corresponding person
Move a member from *father* to *sons* in family	no change
person model change	*family model change*
New *Person*	new *FamilyMember*, possibly new *Family*
Changed *Person* :: *familyId*	moves corresponding member to new or modified *Family*
Changed *Person* :: *name*	changes *name* of corresponding member and possibly of its family
Changed *Person* :: *birthday*	no change

Table 1. Change-propagation between models

The choice between creating a new family or updating an existing family is made by using *Person* :: *familyId*. If a person p has $p.familyId$ equal to the *id* of some existing family f, then p is added to f by *person2family*, otherwise a new family is created and p is added to that family. In the first case, the family name of p should equal $f.name$.

4 Evaluation

In this section we evaluate the correctness and efficiency of the bx using different change-propagation scenarios. A Java 4 implementation of the transformation code and invariant checks was generated. All tests were carried out on a standard Windows 7 laptop with Intel i3 2.53GHz processor using 25% of processing capacity.

We wrote an implementation of the BXTool interface adapted to our test environment as follows:

– *initiateSynchronisationDialogue*() creates corresponding *FamilyRegister* and *PersonRegister* objects.
– *performAndPropagateTargetEdit*(*String f*) loads a delta model file f containing new person model elements or model changes, and runs *person2family* on the updated person model.
– *performAndPropagateSourceEdit*(*String f*) loads a file f containing new family model elements/model changes, and runs *family2person* on the updated family model.
– *performIdleTargetEdit*(*String f*) loads a file f containing new person model elements/model changes, and updates the person model with these changes.

- *performIdleSourceEdit*(*String f*) loads a file *f* containing new family model elements/model changes, and updates the family model with these changes.
- *setConfigurator*(*Object c*) has no effect because we use a fixed update strategy.
- *assertPostcondition*(*Object x*, *Object y*) checks if (I1) and (I2) hold.
- *assertPrecondition*(*Object x*, *Object y*) checks if (I3) and (I4) hold.
- *saveModels*(*String f*) saves the models in file *f*.

The main class, *Controller*, of the Java executable implements this interface.

We define test cases for change scenarios in Table 1, the starting models for these tests are in *test*1*.txt*, ..., *test*7*.txt* and the delta model fragments are in *test*1*delta.txt*, ..., *test*7*delta.txt*. The transformation implementation is in *Controller.java*, the *main* method of this program runs test number *n* when invoked with argument *n*:

```
java Controller n
```

Models and model deltas are written in OCL text format, eg.:

```
m : Male
m.name = "Windsor, Charles"
m.personId = "1"
m.birthday = 19471025
m : pregister.persons
```

defines a person with a given set of attribute values belonging to the collection *persons* of the (previously created) *pregister*.

The results of the tests are shown in Table 2. The result files are in *out*1*.txt*, ..., *out*7*.txt*. Inspection of the result files confirmed that the expected updates had been performed.

Test	*Description*	*(I1) & (I2)*	*(I3) & (I4)*	*Execution time*
1	Changed Person name, familyId	true	true	10ms
2	Changed Person birthday	true	true	10ms
3	New Persons (10000)	true	true	35s
4	Changed Family name	true	true	10ms
5	Changed FamilyMember name	true	true	10ms
6	New Family	true	true	10ms
7	New FamilyMember (50000)	true	true	27s

Table 2. Test case correctness and efficiency

Figure 2 shows the time (in ms) for test 7 delta files of increasing size (in terms of number of person instances). Similar time-complexity graphs are observed for other test cases.

The code of the case solution and the test files may be found at nms.kcl.ac.uk/kevin.lano/uml2web/persons2fam.zip. These have also been uploaded to SHARE.

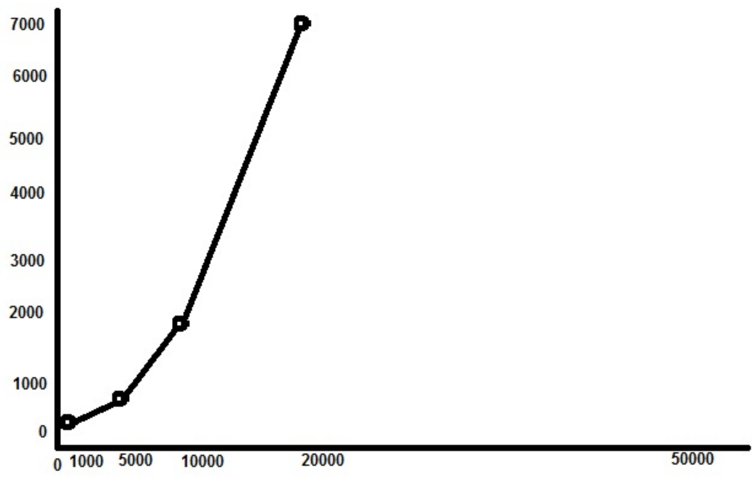

Fig. 2. Execution time of test7

The file mm.txt contains the complete solution specification, including metamodels, transformation rules and invariants.

The size metrics for the transformation rules of the solution are:

- Lines of code: 30
- Number of words: 182
- Number of characters: 936

5 Conclusions

We have shown that the case can be solved by the bx facilities supported by UML-RSDS, with the limitation that manual refinement of the forward transformation was necessary in this case to express the preference for mapping persons to parents. Deletion changes are not yet supported. The solution has the advantage of being declarative and closely related to the logical expression of the problem. The efficiency is practical for model sizes up to 50,000 elements.

References

1. K. Lano, S. Kolahdouz-Rahimi, *Model-transformation Design Patterns*, IEEE Transactions in Software Engineering, vol 40, 2014.
2. K. Lano, S. Yassipour-Tehrani, *Verified bidirectional transformations by construction*, VOLT '16, MODELS 2016.
3. K. Lano, *Agile Model-based Development using UML-RSDS*, Taylor and Francis, 2016.