# The SDMLib Solution to the TTC 2017 Families 2 Persons Case

Albert Zündorf
Kassel University
zuendorf@uni-kassel.de

Alexander Weidt
Kassel University
alexander.weidt@uni-kassel.de

## 1 Introduction

The TTC 2017 Family to Persons Case asks for bidirectional transformations between a Family model and a Persons model. Each model provides informations that is not contained in the other model. Thus, the case asks to keep some kind of correspondences between the elements of the two models. In addition, the case asks for incremental handling of model changes.

Figure 1 shows the SDMLib class model for this case. SDMLib comes with its own code generation, i.e. we do not use the EMF class model nor the EMF code generation. SDMLib class models do not provide aggregation as this is an unusual concept within a graph like object model. Thus, the four associations between `Family` and `FamilyMember` are modeled as bidirectional associations with explicit roles at the `Family` side. To deal with these four associations easily, `FamilyMember` provides the additional method `getFamily()` that looks up all four associations in order to find the corresponding family.

To maintain the correspondences between the two model parts, we have added a famReg–persReg association between `FamilyRegister` and `PersonRegister`. In addition we use a cp–cfm association connecting `FamilyMember` objects with corresponding `Person` objects.

Adressing the incremental requirement, we use unidirectional `c`hanged links between the register objects and their `FamilyMember` and **Person** objects, respectively. The set methods mark changed objects with c links, as a side effect. Furthermore, the cp–cfm association has a life-dependency semantics: if one object is explicitly destroyed, the corresponding partner is deleted as well. Again, this is implemented via side effects in the corresponding `removeYou()` methods.

## 2 The rules

Figure 2 shows our forward transformation rule in graphical notation. Pattern matching starts with the pattern object `fr` that matches the `FamilyRegister` object passed as parameter. Pattern object `pr` matches the corresponding `PersonRegister`. Next, pattern object `fm` looks for `FamilyMember` objects marked as changed via a `c` link. This `c` link is shown in red color as it is deleted on rule execution (in order to unmark handled objects).

Now there are two cases: there might already exist a corresponding `Person` object or not. The first case is handled by the optional subpattern 1 shown in a dashed box in the bottom right corner of Figure 2. If pattern object `oldP` finds a match, the pseudo condition `ensureNameAndGender(oldP)` is checked. This methods, is shown below:

```
1    public boolean ensureNameAndGender(Person p) {
2       FamilyMember fm = p.getCfm();
3       Family f = fm.getFamily();
4       Class gender = Male.class;
5       if (fm.getMotherOf() != null || fm.getDaughterOf() != null) {
6          gender = Female.class;
```



In: A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.): Proceedings of the 10th Transformation Tool Contest, Marburg, Germany, 21-07-2017, published at http://ceur-ws.org

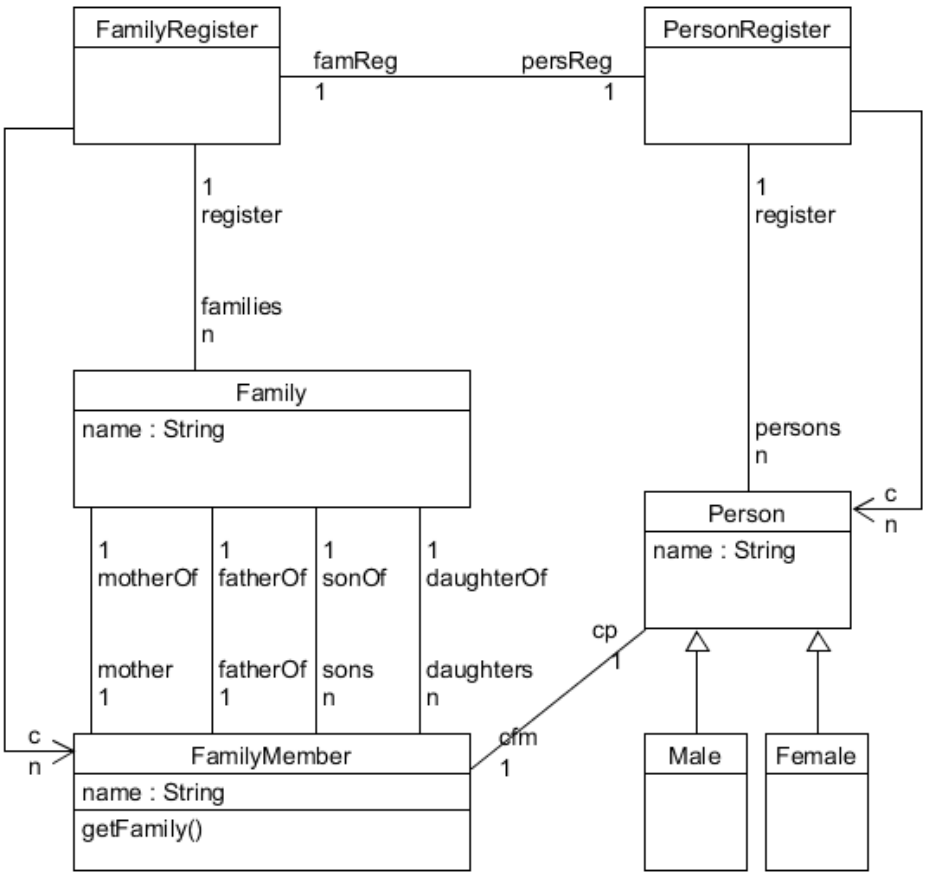


Figure 1: Adapted Class Model

```
7         }
8         if (p.getClass() != gender) {
9            try {
10              Person newP = ((Person) gender.newInstance())
11                    .withRegister(p.getRegister()).withBirthday(p.getBirthday())
12                    .withCfm(p.getCfm()).withName(p.getName());
13              p.removeYou();
14              p = newP;
15           } catch (InstantiationException | IllegalAccessException e) {
16              e.printStackTrace();
17           }
18        }
19        String fullName = String.format("s, s", f.getName(), fm.getName());
20        p.withName(fullName);
21        return true;
22     }
```

Listing 1: Example Graph Query in Java

The Families to Persons case requires that the mother and the daughters of a `Family` shall map to an object of type `Female` within the `PersonRegister`. The father and the sons shall map to an object of type `Male`. As a `FamilyMember` might have changed e.g. from the `sons` role of one family to the `mother` role of another family, the corresponding `Person` object needs to change from type `Male` to type `Female`. As Java objects are not able

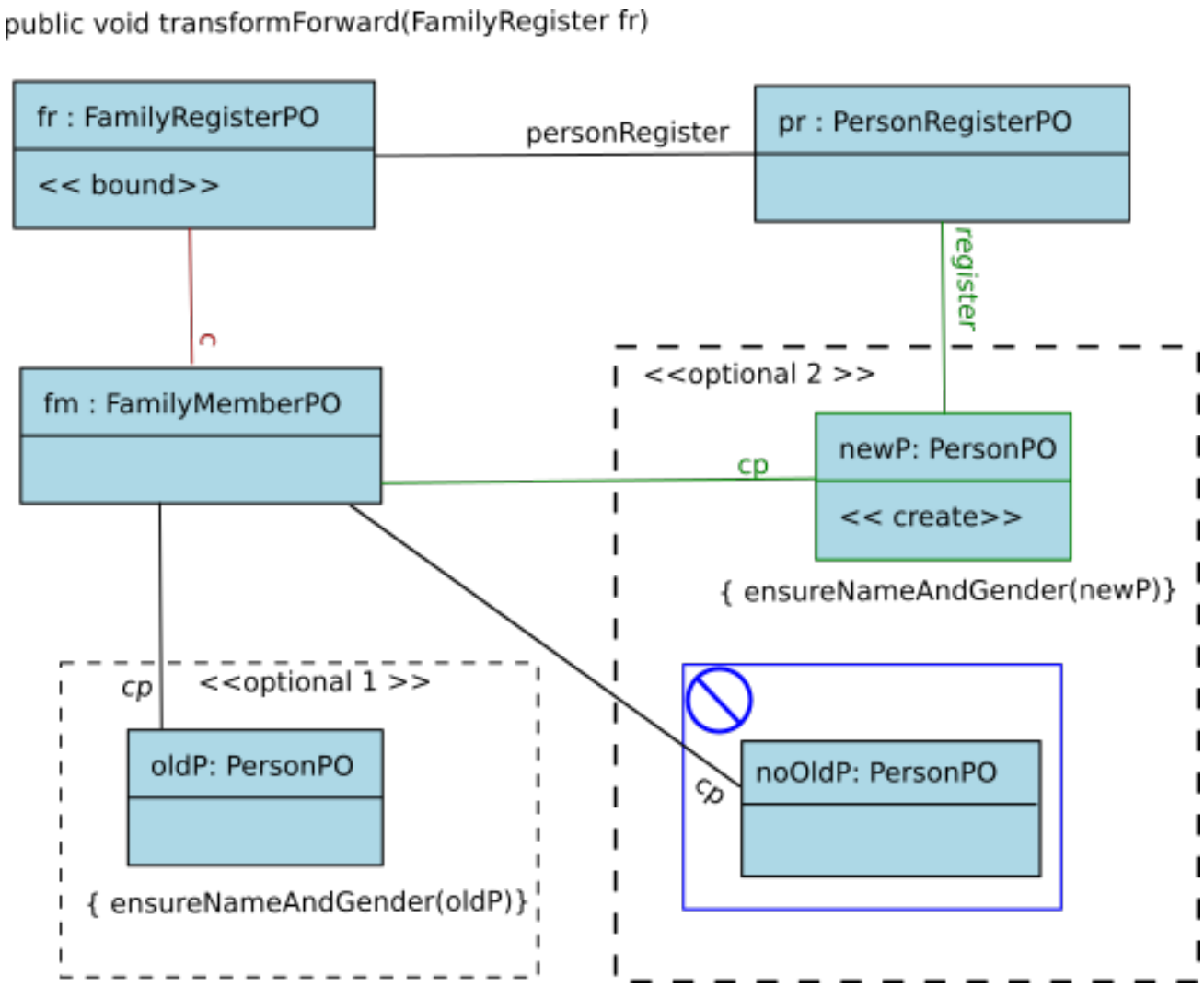


Figure 2: Forward Transformation Rule

to change their type, lines 10 to 12 of Listing 1 simply create a new Object of the appropriate gender and copy all attributes and references from the old object. Finally, method `ensureNameAndGender()` assigns the current full name to the `Person` object.

The second case is handled by the second optional subpattern of Figure 2. The second subpattern contains a negative application condition shown as a solid blue box with a forbidden sign in its upper left corner. This negative application condition again looks for an old corresponding `Person` if there is no match the negative application is not violated and the surrounding pattern continues its matching. In our case, the second optional subpattern just creates a new object of type `Person`. As discussed, we actually need an object either of type `Female` or of type `Male`. This is again handled by calling method `ensureNameAndGender()` within the pseudo condition shown below pattern object `newP`.

Figure 3 shows our backward transformation rule. Pattern matchin starts with the `PersonRegisterPO` pattern object `pr` shown in the up right corner. The match for `pr` is provided as parameter. The rule first looks for a `Person p` that is marked as changed via a `c` link. This `c` link is destroyed (as indicated by the red color). The backward transformation rule now handles two cases: there is an old corresponding `FamilyMember` or not. The first case is handled by the optional subpattern shown at the bottom of Figure 3. The pattern object `oldFM` matches a `FamilyMember` that has a `cfm` (corresponding family member) link to our `Person p`. Next pattern object `oldF` looks up the corresponding `Family`. This uses method `getFamily()` which tries `motherOf`, `fatherOf`, `daughterOf`, and `sonOf` links to navigate from a `FamilyMember` to its `Family`. Now there are two

conditions. Condition 1 is a pseudo condition that actually assigns the given name of the current person to the current `FamilyMember`. After that, the second condition checks, whether the current `Family` has the right name. The second condition holds, if the current Family's does not match current person's family name. In that case, we need to look for another Family with a matching name. This is done by the second subpattern of our rule. Thus, in case of a conflict with the family name, the first subpattern matches (all conditions) and it is executed, i.e. the `oldFM` object is destroyed. (Causing the second subpattern to create a new `FamilyMember` object, see below). If the family name matches, the second condition fails and the pattern is not executed but the `oldFM` object survives. In that case, the first pseudo condition has already adjusted the given name and the case of an old existing `FamilyMember` is handled, completely.

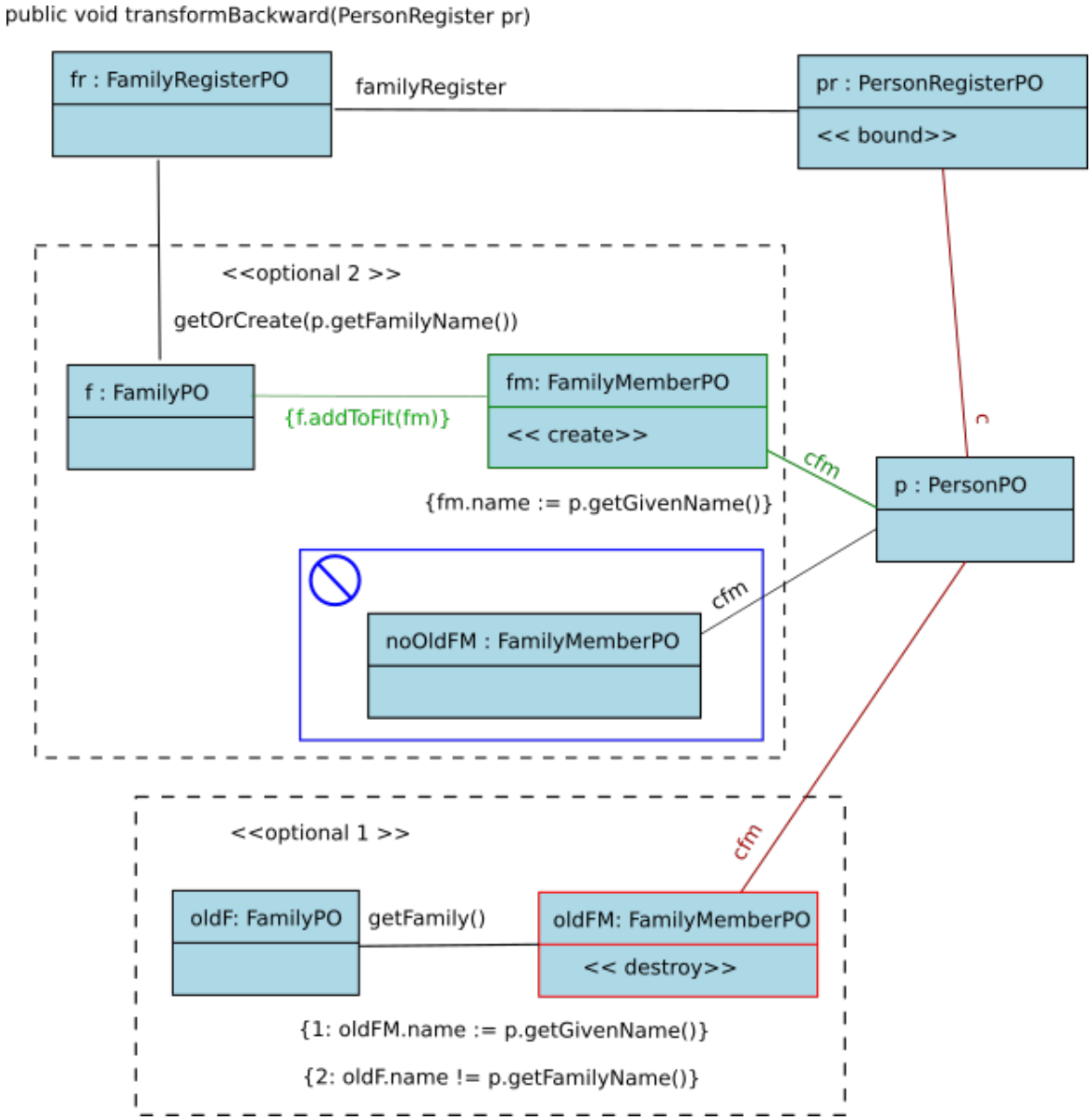


Figure 3: Backward Transformation Rule

The second optional subpattern of our backward transformation rule creates a new `FamilyMember` object that corresponds to our `Person p`. This is done by the pattern object `fm`. The pseudo condition below `fm` assigns the given name. We also need a `Family f` for the new `FamilyMember`. The `Family` is derived from the `FamilyRegister fr` via the path expression `getOrCreate(p.getFamilyName())`. This operation searches our `FamilyRegister` for a matching `Family`. If there is no `Family` with the right name, a new `Family` object is

created. Actually, the `getOrCreate()` methods also respects the `Decisions` parameter required by the case description and creates always a new `Family`, if required. Now, we have to add our new `FamilyMember` to its `Family`. This is done using the pseudo condition `f.addToFit(fm)`. Method addToFit() just looks for the gender of the current person and whether the corresponding parent role is still vacant and whether it should be used or whether the new `FamilyMember` should become a daughter or a son, respectively. Note, method `addToFit()` might easily have been modeled as a model transformation rule, but the cases are pretty straight forward and thus we just programmed it in plain Java.

## 3 Results

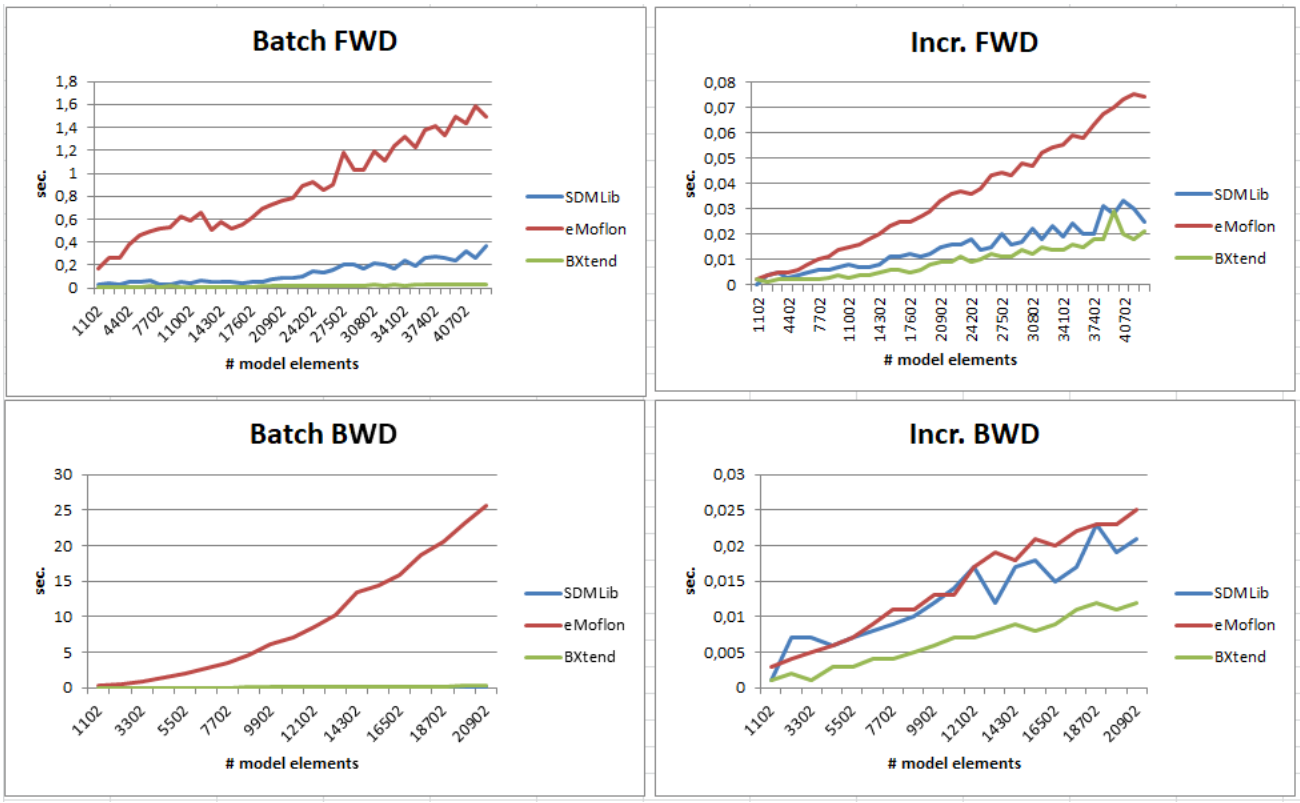


Figure 4: Performance

Figure 4 shows the performance measured by the predefined scalability test of the Families to Persons case. We have measured this performance on a laptop with 2 GHz Intel I7 CPU giving the Java process 6 GB heap space. We compared SDMLib only to eMoflon and BXtend as the Medini and the FunnyQT approach used a magnitude more time. Final performance results will be delivered by the case organizers that are able to compare all solutions on one computer. Thus, Figure 4 gives some relative results, only. Still, SDMLib performance almost similar to BXtend while eMoflon is 2 or 3 times slower. All tools deal well with the incremental transformations.

## References

[SDMLib] SDMLib - Story Driven Modeling Library *www.sdmlib.org* May, 2017.

[F2PCase] Anthony Anjorin, Thomas Buchmann and Bernhard Westfechtel The Family to Persons Case *www.transformation-tool-contest.eu/TTC_2017_paper_2.pdf* May, 2017.