

# An NMF solution to the Families to Persons case at the TTC 2017

Georg Hinkel

FZI Research Center of Information Technologies  
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany  
hinkel@fzi.de

## Abstract

This paper presents a solution to the Families to Persons case at the Transformation Tool Contest (TTC) 2017 using the .NET Modeling Framework (NMF). The goal of this case was to bidirectionally synchronize a simple model of family relationships with a simple person register. We propose a solution based on the bidirectional and incremental model transformation language NMF Synchronizations.

## 1 Introduction

According to the general model theory of Stachowiak [Sta73], models always have a purpose. In many practical applications, one tries to reuse existing metamodels, usually because valuable infrastructure is built on top of it. However, because the different purposes require different abstractions, the information concerning an entity is often split among multiple of these models, which makes a pure transformation approach infeasible as neither source nor target model can be fully reconstructed from the respective other model. Instead, the models must be synchronized to make sure that they are consistent with regard to some correspondence rules.

The Families to Persons case at the Transformation Tool Contest (TTC) 2017[ABW17] demonstrates this problem in the scenario of a well known example model transformation, the Families to Persons transformation, and provides a benchmark framework to compare solutions in terms of abilities and performance. Here, a structured model of family relations should correspond to a flat model of persons where the information of a persons family is encoded only through that person's full name. Further, the information of the gender of a person is encoded differently: While the Families model encodes this information through the position of a model element, the same information is represented through subtypes in the Persons model. Finally, the Persons model contains the birthday of a person, an information that is missing in the Families model. Therefore, the information contained in one model cannot be fully reconstructed using the other model. Nevertheless, there is a clear correspondence as there should be a family member for each person and vice versa.

In this paper, we present a solution to this model synchronization challenge using the incremental and bidirectional model transformation language NMF Synchronizations [Hin15, HB17], part of the .NET Modeling Framework (NMF, [Hin16]). There, incremental model transformations run as a live transformation [HLR06] that continuously monitors both source and target models for incremental changes and propagates them to the other model. Because the benchmark framework runs in Java and NMF Synchronizations only runs on the .NET platform, our solution runs as a separate process that communicates with the benchmark framework using standard input and standard output. However, this adds a considerable serialization and deserialization overhead on the solution, which is why the benchmark times cannot be compared with other solutions. Instead, we included a custom time management and discuss how large the serialization overhead actually is.

---

*Copyright © by the paper's authors. Copying permitted for private and academic purposes.*

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>

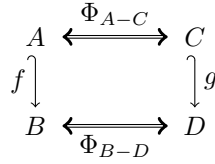


Figure 1: Schematic overview of synchronization blocks

Our solution is publicly available on GitHub<sup>1</sup>.

The remainder of this paper is structured as follows: Section 2 gives a very brief introduction to synchronization blocks, the formalism used in NMF Synchronizations. Section 3 presents our solution. Section 4 evaluates our approach before Section 5 concludes the paper.

## 2 Synchronization Blocks

Synchronization blocks are a formal tool to run model transformations in a bidirectional way [HB17]. They combine a slightly modified notion of lenses [FGM<sup>+</sup>07] with incrementalization systems. Model properties and methods are considered morphisms between objects of a category that are set-theoretic products of a type (a set of instances) and a global state space  $\Omega$ .

A (well-behaved) in-model lens  $l : A \leftrightarrow B$  between types  $A$  and  $B$  consists of a side-effect free GET morphism  $l \nearrow \in Mor(A, B)$  (that does not change the global state) and a morphism  $l \searrow \in Mor(A \times B, A)$  called the PUT function that satisfy the following conditions for all  $a \in A, b \in B$  and  $\omega \in \Omega$ :

$$\begin{aligned}
l \searrow (a, l \nearrow (a)) &= (a, \omega) \\
l \nearrow (l \searrow (a, b, \omega)) &= (b, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega.
\end{aligned}$$

The first condition is a direct translation of the original PUTGET law [FGM<sup>+</sup>07]. Meanwhile, the second line is a bit weaker than the original GETPUT because the global state may have changed. In particular, we allow the PUT function to change the global state.

A (single-valued) synchronization block  $\mathcal{S}$  is an octuple  $(A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$  that declares a synchronization action given a pair  $(a, c) \in \Phi_{A-C} : A \cong C$  of corresponding elements in a base isomorphism  $\Phi_{A-C}$ . For each such a tuple in states  $(\omega_L, \omega_R)$ , the synchronization block specifies that the elements  $(f \nearrow (a, \omega_L), g \nearrow (b, \omega_R)) \in B \times D$  gained by the lenses are in the dependent isomorphism  $\Phi_{B-D}$ .

A schematic overview of a synchronization block is depicted in Figure 1. The usage of lenses allows these declarations to be enforced automatically in both directions. The engine simply computes the value that for example the right selector should have and enforces it using the PUT operation. An unidirectional specification is also possible where  $f$  is no longer required to be a lens. More details can be found in previous publications [HB17].

A multi-valued synchronization block is a synchronization block where the lenses  $f$  and  $g$  are typed with collections of  $B$  and  $D$ , for example  $f : A \leftrightarrow B^*$  and  $g : C \leftrightarrow D^*$  where stars denote Kleene closures.

Synchronization Blocks have been implemented in NMF Synchronizations, an internal DSL hosted by C# [HB17, Hin15].

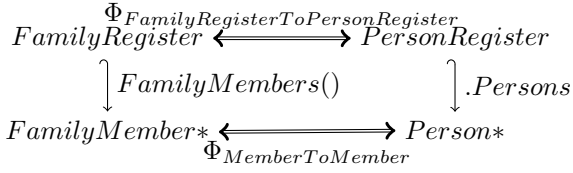
## 3 Solution

To solve the FamiliesToPersons case, we see two correspondencies that need to be synchronized:

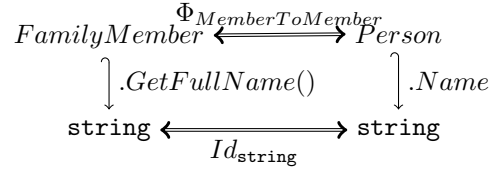
1. All family members contained in a family need to be synchronized with the people in the Persons model and
2. The full name of family members that consists of the name of the family and the name of the family member needs to be synchronized with the full name of the corresponding person.

Using synchronization blocks, these correspondencies can be formulated in the diagrams of Figure 2. In the internal DSL of NMF Synchronizations, the implementation is depicted in Listing 1.

<sup>1</sup><http://github.com/georghinkel/benchmarkx>



(a) Synchronization block to synchronize family members with person elements



(b) Synchronization block to synchronize names

Figure 2: Synchronization Blocks in the solution

```

1 public class FamilyRegisterToPersonRegister : SynchronizationRule<FamilyRegister, PersonRegister> {
2     public override void DeclareSynchronization() {
3         SynchronizeMany(SyncRule<MemberToMember>(),
4             fam => new FamilyMemberCollection(fam),
5             persons => persons.Persons);
6     }
7 }
8 public class MemberToMember : SynchronizationRule<IFamilyMember, IPerson> {
9     public override void DeclareSynchronization() {
10        Synchronize(m => m.GetFullName(), p => p.Name);
11    }
12 }

```

Listing 1: Implementation of main synchronization blocks

In particular, the definition of the synchronization blocks in Listing 1 are implemented in a call to the methods `Synchronize` and `SynchronizeMany`. The types and the base isomorphisms `FamilyRegisterToPersonRegister` and `MemberToMember` used in the synchronization block can be inferred from the context and the explicitly specified dependent synchronization rule `MemberToMember`. In the second synchronization block, the identity on strings is used as isomorphism.

While NMF is able to convert the simple member accesses for the persons into lenses, this does not hold for the helpers `FamilyMemberCollection` and `GetFullName` that we used in this implementation. Therefore, we have to explicitly provide an implementation of PUT for these two items.

For the `GetFullName`-method, the PUT operation needs to be specified through an annotation. In addition, because NMF does not parse the contents of a method (only of lambda expressions), we need to specify an explicitly incrementalized version of the given helper method. To do this, we can reuse the implicit incrementalized lambda expression and also use that for the batch implementation to avoid code duplication. A sketched implementation is depicted in Listing 2.

```

1 private static ObservingFunc<IFamilyMember, string> fullName =
2     new ObservingFunc<IFamilyMember, string>(m =>
3         m.Name == null ? null : ((IFamily)m.Parent).Name + ", " + m.Name);
4
5 [LensPut(typeof(Helpers), "SetFullName")]
6 [ObservableProxy(typeof(Helpers), "GetFullNameInc")]
7 public static string GetFullName(this IFamilyMember member) {
8     return fullName.Evaluate(member);
9 }
10 public static INotifyValue<string> GetFullNameInc(this IFamilyMember member) {
11     return fullName.Observe(member);
12 }
13 public static void SetFullName(this IFamilyMember member, string newName) {
14     ...
15 }

```

Listing 2: Implementation of the `GetFullName` lens

In the case of `FamilyMemberCollection` which as the name implies is a collection, we only have to provide the formula how the results of this collection are obtained and implement the methods `Add`, `Remove` and `Clear`. A schematic implementation is depicted in Listing 3.

```

1 private class FamilyMemberCollection : CustomCollection<IFamilyMember> {
2     public FamilyRegister Register { get; private set; }
3     public FamilyMemberCollection(FamilyRegister register)
4         : base(register.Families.SelectMany(fam => fam.Children.OfType<IFamilyMember>()))
5     { Register = register; }
6 }

```

```

7 | public override void Add(IFamilyMember item) { ... }
8 | public override bool Remove(IFamilyMember item) { ... }
9 | public override void Clear() { ... }
10| }

```

Listing 3: Implementation of the FamilyMemberCollection

However, to add a family member to a family, the `Add` method has to know the family name of a person as well as its gender – information that is encoded using the containment hierarchy in the Families model and therefore unavailable before the element is added to a family. Therefore, we carry this information over from the corresponding element of the Person metamodel using a temporary stereotype: In NMF, all model elements are allowed to carry extensions. We use this to add an extension that specifies the last name and whether the given element is male. The stereotype is deleted as soon as a family member is added to a family.

Furthermore, the fact that different genders are modeled through different classes in the Persons model, the synchronization rule `MemberToMember` needs to be refined to allow NMF Synchronizations to decide whether to create a `Male` or `Female` output element. This can be done in NMF Synchronizations through an instantiating rule.

The implementation of both of these concepts is depicted in Listing 4.

```

1 | public class MemberToMale : SynchronizationRule<IFamilyMember, IMale> {
2 |     public override void DeclareSynchronization() {
3 |         MarkInstantiatingFor(SyncRule<MemberToMember>(),
4 |             leftPredicate: m => m.FatherInverse != null || m.SonsInverse != null);
5 |     }
6 |     protected override IFamilyMember CreateLeftOutput(IMale input, ...) {
7 |         var member = base.CreateLeftOutput(input, candidates, context, out existing);
8 |         member.Extensions.Add(new TemporaryStereotype(member) {
9 |             IsMale = true,
10 |             LastName = input.Name.Substring(0, input.Name.IndexOf(','))
11 |         });
12 |         return member;
13 |     }
14 | }

```

Listing 4: The MemberToMale-rule

## 4 Evaluation

The actual transformation consists of 196 lines of which 73 are either empty or consist only of a single brace and whitespaces. We therefore think that our solution is very concise.

The solution passes all batch test cases defined in the test suite. For the incremental test cases, there are still some failures, but these are not due to the inabilities of the transformation but rather due to inaccurate change models: A move for example is transcribed as a deletion and an addition. However, for a new family member, NMF Synchronization will not reuse the `Person` element for the deleted member and thus, the information on the birthday is lost. Unfortunately, detecting a move and distinguishing it from deleting an element and inserting a new one is more complex and so far, we did not implement such a case distinction for the TTC.<sup>2</sup>

Due to the fact that the NMF solution does not use EMF and does not use Java, there is a large serialization overhead attached to it. Depending on the test case, this overhead can outweigh the change propagation by multiple orders of magnitude. Thus, we adjusted the time measurement for our solution to eliminate this overhead.

The results for the incremental scalability test cases for NMF and the fastest reference solutions EMOFLON and BXTEND recorded on an Intel i7-4710MQ clocked at 2.50Ghz in a system with 16GB RAM is depicted in Figure 3. The plot shows the time to add a family member or a person in a model of the given number of families (each consisting of five members). Both axes are logarithmic. We only depicted the times for the fastest reference solutions as other provided reference solutions were much slower.

In the results for the *Incremental Forward* scenario, we can see that the resulting curve for the NMF solution is flat meanwhile the curve for EMOFLON and BXTEND is not. This indicates that unlike the other two, the NMF solution is the only fully incremental solution as the time to add a new family member does not depend on the size of the model. Furthermore, even for the smallest model with only 10 families, the incremental update

<sup>2</sup>Even without this case distinction, the class to convert EMF notifications to model changes in the NMF format is already more complex than the entire model synchronization.

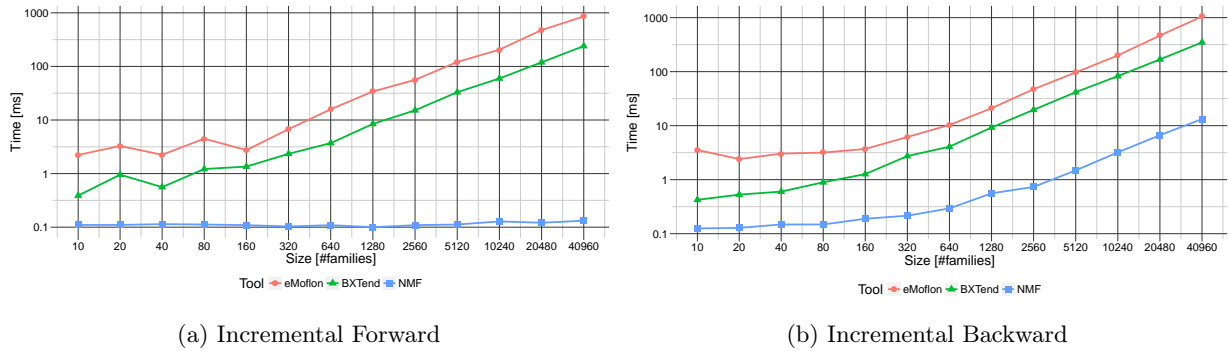


Figure 3: Performance results for the incremental scalability tests

is faster than recomputing the *Persons* model from scratch. For the largest size, this leads to a speedup of more than three orders of magnitude.

The results for the *Incremental Backward* benchmark in the opposite direction are depicted in Figure 3b. Unlike the incremental forward where the scalability tests inserts a new family member and the propagation simply adds the corresponding person which can be done in amortized  $O(1)$ , the incremental backward test has to look for a suitable family, which implies a  $\Omega(n)$  complexity. Therefore, the speedups in this case do not grow equally with the size. Instead, a saturation happens at a speedup of slightly more than one order of magnitude.

## 5 Conclusion

In this paper, we applied the bidirectional and incremental model transformation language NMF Synchronizations to the Families to Persons synchronization example. The solution demonstrates the easy adaptation of NMF Synchronizations by providing custom lens implementations.

Our solution is a live transformation that runs on the .NET platform, not on a JVM, and therefore requires a serialization and deserialization overhead to communicate with the benchmark driver. After eliminating this overhead, the results show that our solution is fully incremental and is faster than BXTEND and EMOFLOM in the incremental scalability tests by multiple orders of magnitude.

## References

- [ABW17] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. The Families to Persons Case. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org, July 2017.
- [FGM<sup>+</sup>07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3), May 2007.
- [HB17] Georg Hinkel and Erik Burger. Change Propagation and Bidirectionality in Internal Transformation DSLs. *Software & Systems Modeling*, 2017.
- [Hin15] Georg Hinkel. Change Propagation in an Internal Model Transformation Language. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, pages 3–17, Cham, 2015. Springer International Publishing.
- [Hin16] Georg Hinkel. NMF: A Modeling Framework for the .NET Platform. Technical report, Karlsruhe Institute of Technology, Karlsruhe, 2016.
- [HLR06] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 321–335. Springer, 2006.
- [Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag, Wien, 1973.