Handling Conflicting Changes in the Families to Persons Case with Synchronization Blocks

Georg Hinkel[1]

[1]RheinMain University of Applied Sciences and Arts, Unter den Eichen 5, 65195 Wiesbaden, Germany

Abstract

Modern systems are often developed in views and in a collaborative manner. If changes can occur in multiple locations, they can easily conflict with each other. The extended version of the Families to Persons case sets a benchmark for different synchronization approaches how such conflicts can be resolved. In this paper, we present a benchmark solution using NMF Synchronizations.

Keywords

TTC, Synchronization Blocks, NMF

1. Introduction

The lack of tools is a repeatedly reported shortcoming of model-driven development and in particular, collaborative tools are often asked for. If two people collaborate on a model, they may make conflicting changes, particularly if they are working on different models and the conflict only arises from the synchronization of these models. To gather insights on how different tools solve this challenge, the *families-to-persons* benchmark [1] was extended [2] for conflicting changes.

In this paper, we show how the Families-to-Persons case can be solved using Synchronization Blocks, using their implementation in NMF Synchronizations. The implementation is based on the 2017 version of the benchmark [1] where it achieved very good performance results [3], showing that propagating additions can be done in constant time even if model sizes get large. Inspired from these results, we extended the NMF solution to handle conflicting changes.

The implementation is available online[1] and a pull request exists to integrate it into the benchmark framework.

The remainder of this paper is structured as follows: Section 2 shortly introduces NMF Expressions and NMF Synchronizations. Section 3 describes the architecture of how to integrate them into a Java/EMF-based benchmark framework. Section 4 introduces the actual solution. Section 5 discusses how conflicting changes are handled. Section 6 discusses limitations of the solution before Section 8 concludes the paper.

2. NMF Expressions and NMF Synchronizations

NMF Expressions [4] is an incrementalization system integrated into the C# language. It takes expressions of functions and automatically and implicitly derives an incremental change propagation algorithm. This works by setting up a dynamic dependency graph that keeps track of the models' state and is adapted when necessary. The incrementalization system is extensible and supports large parts of the Standard Query Operators (SQO[2]).

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026

georg.hinkel@hs-rm.de (G. Hinkel)

https://www.hs-rm.de/design-informatik-medien/ueber-uns/personen-am-fachbereich/person/georg-hinkel (G. Hinkel)

0000-0002-6462-5208 (G. Hinkel)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]https://github.com/georghinkel/benchmarx/tree/nmf-ttc2026

[2]http://msdn.microsoft.com/en-us/library/bb394939.aspx; SQO is a set of language-independent standard APIs for queries, specifically defined for the .NET platform.

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/TTC-short2.pdf

NMF Synchronizations is a model synchronization approach based on the algebraic theory of synchronization blocks. Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way [5]. They combine a slightly modified notion of lenses [6] 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 Ω.

A (well-behaved) in-model lens $l : A \hookrightarrow B$ 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. Meanwhile, the second line is a bit weaker than the original GETPUT law because the global state may have changed. In particular, we allow the PUT function to change the global state.

A (single-valued) synchronization block $\boldsymbol{S}$ is an 8-tuple $(A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ that declares a synchronization action given a pair $(a, c) \in \Phi_{A-C} : A \cong C$ of corresponding elements in a base isomorphism Φ_{A-C}. For each such tuple in states (ω_L, ω_R), the synchronization block specifies that the elements $(f(a, \omega_L), g \nearrow (b, \omega_R)) \in B \times D$ gained by the lenses f and g are isomorphic with respect to Φ_{B-D}.

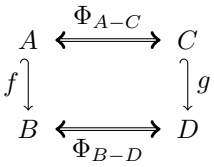


Figure 1: Schematic overview of unidirectional synchronization blocks

A schematic overview of a synchronization block is depicted in Figure 1. The usage of lenses allows these declarations to be enforced automatically and in both directions, if required. The engine computes the value that the right selector should have and enforces it using the PUT operation. Similarly, 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 $f : A \hookrightarrow B*$ and $g : C \hookrightarrow D*$ where stars denote free monoids.

Synchronization blocks have been implemented in NMF Synchronizations, an internal DSL hosted by C# [7, 5]. For the incrementalization, it uses the extensible incrementalization system NMF Expressions. This DSL is able to lift the specification of a model transformation/synchronization in three orthogonal dimensions:

- **Direction:** A client may choose between transformation from left to right, right to left or in check-only mode
- **Change Propagation:** A client may choose whether changes to the input model should be propagated to the output model, also vice versa or not at all
- **Synchronization:** A client may execute the transformation in synchronization mode between a left and a right model. In that case, the engine finds differences between the models and handles them according to the given strategy (only add missing elements to either side, also delete superfluous elements on the other or full duplex synchronization)

This flexibility makes it possible to reuse the specification of a transformation for a broad range of different use cases. Furthermore, the fact that NMF Synchronizations is an internal language means that a wide range of advantages from mainstream languages, most notably modularity and tool support, can be inherited [8].

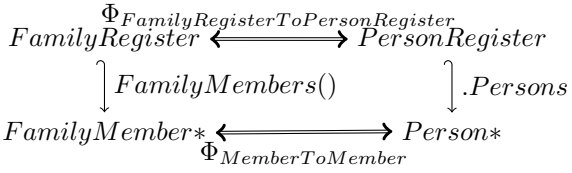


Figure 2: Synchronization block to synchronize family members with person elements

3. Integrating NMF into a Java-based benchmark environment

Like the 2017 edition of the benchmark, the NMF solution works with a companion process: Because Benchmarx runs as a JUnit test suite with models based on EMF, the solution records the changes, serializes them into the change format of NMF in a temporary file and then asks the companion process to propagate the changes and to serialize the resulting models into other temporary files. While this works, it adds considerable measurement overhead which is why we use a separate time measurement in the companion process, in order to get meaningful comparison results.

In the past [3], obtaining the changes worked by listening to the models via the EMF adapter infrastructure. This has the disadvantage that it turns movements into a series of deletion and addition. Because the 2026 edition of the benchmark featured conflicting changes of moves with deletions, this is no longer viable which is why we now changed the implementation to process the edit operations created by the benchmark. Unfortunately, the edits drop collection positions. To avoid confounding factors to the time measurements in the companion process, we restore collection positions when serializing the changes, but it keeps being a bias for Java/EMF-based solutions.

4. Synchronizing Families and Persons using Synchronization Blocks

Given that the 2026 version of the benchmark is an extension of the 2017 version, the description is heavily based on the original description [3]. In fact, the implementation of the actual synchronization is only changed in few places, because handling conflicting changes requires brownfield synchronization primitives.

In the *families to persons* synchronization, we see two correspondences that need to be synchronized [3]:

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.

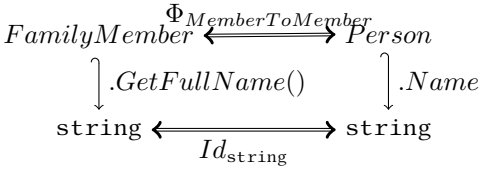


Figure 3: Synchronization block to synchronize names

Using synchronization blocks, these correspondences can be formulated in the diagrams of Figure 2 and Figure 3. The implementation in NMFSynchronizations is depicted in Listing 1.

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 – these lenses are highly specific to the given scenario. Therefore, we have to explicitly provide an implementation of Put for these two lenses.

```
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

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

Listing 2: Implementation of the `GetFullName` lens

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.

In Listing 2, we create an `ObservingFunc` for the function to obtain a family members full name. This syntax allows NMF to get a model of the actual function and therefore, NMF Expressions is able to incrementalize it. In Lines 6–11, we use this object to represent the actual extension method and its incrementalization, which essentially forwards to the same `ObservingFunc` object. The incrementalization is connected with the original function through the annotation in line 5. Furthermore, we annotate the PUT method also using an annotation in line 4 where we reference the `SetFullName` method in line 12–14. The return type `void` of this method makes it clear that this is a persistent lens. This method may contain arbitrary C# code and is called when NMF Synchronizations needs to write a value, for instance as a consequence of an update in the *Persons* model where the last name of a person changed.

In the case of `FamilyMemberCollection` which as the name implies is a collection, we only have to provide the query how the results of this collection are obtained and implement the methods `Add`, `Remove` and `Clear`. NMF Expressions is able to automatically incrementalize the query when this is necessary and uses the provided model manipulation methods in case a model element has to be added to the collection. A schematic implementation is depicted in Listing 3.

Again, the model manipulation methods `Add`, `Remove` and `Clear` may contain arbitrary C# code. 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

```
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`

```
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

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.

In particular, lines 3 and 4 mark the synchronization rule `MemberToMale` as instantiating for the rule `MemberToMember` on the condition that the family member is either a father or a son of a family. Further, we override the creation of an output model element for the left side by overriding the method `CreateLeftOutput`. This method calls the base implementation which simply uses the default constructor to create a new `FamilyMember` element. Then, it adds a case-specific extension called `TemporaryStereotype` that carries the information on the gender (through the `IsMale` attribute) and the last name.

Unlike the original TTC case, the synchronization also needs to be able to see that concurrently added elements correspond. Therefore, it is necessary to also implement the method `ShouldCorrespond` to decide whether two already existing model elements should match. This can be implemented by just matching the name of the person with the full name of the family member. The method can be implemented either in the `MemberToMember`-rule or in both `MemberToMale` and `MemberToFemale`, as the former by default delegates to the latter. Listing 5 depicts an implementation of the latter.

5. Resolution of Conflicting Changes

In order to process conflicting changes, we use a feature of the incrementalization system NMF Expressions that has never before been applied for synchronization tasks: NMF Expressions can process changes in transactions. This means, instead of propagating changes immediately, changes to models

```
1 public override bool ShouldCorrespond(IFamilyMember left, IMale right, ISynchronizationContext
      context)
2 {
3    return left.GetFullName() == right.Name;
4 }
```

Listing 5: Specifying the correspondences initially

are only collected first and then propagated in a transaction-like batch. This feature was originally developed to reduce phantom changes. If the property *ItemCount* of a model element *m* is an integer-valued property, it is clear that the expression `m.ItemCount + m.ItemCount` can never be odd, but during change propagation, it can if the first property access is propagated before the second property access. To defeat this problem, NMF Expressions features a transaction mode.

In the presence of conflicting changes, this transaction mode is also helpful as it allows to correctly perform the changes and then coordinate potential conflicts between them. In particular for the case of *families-to-persons*, it allows the synchronization to see that moving a person between different families does not have an impact on the person register besides name change: The dependency graph node for the `SelectMany` operator (line 4 in Listing 3) sees both the addition of a person in one family and the removal from another and concludes that the person is only moved within the collection but because NMF Synchronizations currently is only implemented for collections without order, moves are ignored.

The resulting behavior is currently that in case of what the benchmark calls conflicting changes, simply both changes are performed. In the case of a rename/delete conflict, the element is simply renamed and then deleted, achieving a behavior consistent with the benchmark expectations.

6. Limitations

In its current state, the implementation is only a prototype to show that the resolution of conflicting changes can work in this way. In particular, NMF Synchronizations only uses the event API of NMF Expressions rather than implementing nodes in the dependency tree. The latter would allow NMF Synchronizations to see the conflicts and handle them in a more controlled way, allowing a user to specify a custom 3-way merge algorithm, a functionality that could then also be exposed via the DSL.

However, even with these extensions, it would still be difficult for a developer to control which of the expected models should be created. Using synchronization blocks, renames and deletions or additions are handled by different synchronization blocks and we even supporting custom 3-way merges would not change that. We currently lack a good formalism to describe interactions between multiple synchronization blocks.

The fact that the 2026 version of the benchmark solution now properly captures moves instead of turning them into a combination of delete and add has a side-effect to existing test cases. NMF Synchronizations currently does not support breaking correspondences of existing elements. Hence, moving a family member such that it changes the gender of the person is not supported. In the tests, moving Maggie Simpson to the *sons* of a different family does not replace the element in the Person model, hence it is still an instance of `Female`.

7. Evaluation

In order for a fair comparison, we modified the time measurements of Benchmarx to return microseconds instead of seconds and in the case of the NMF solution, to delegate the time measurements to the companion process. The result obtained on a system equipped with an AMD Ryzen 7 Pro 6850H clocked at 3201 Mhz in a system with 32GB RAM running Windows 11 (10.0.26100) are depicted in Figure 4.

The results show that the runtime is generally very low due to the low number of model elements. The actual runtime of the benchmark is much higher, due to the high synchronization overhead involved.

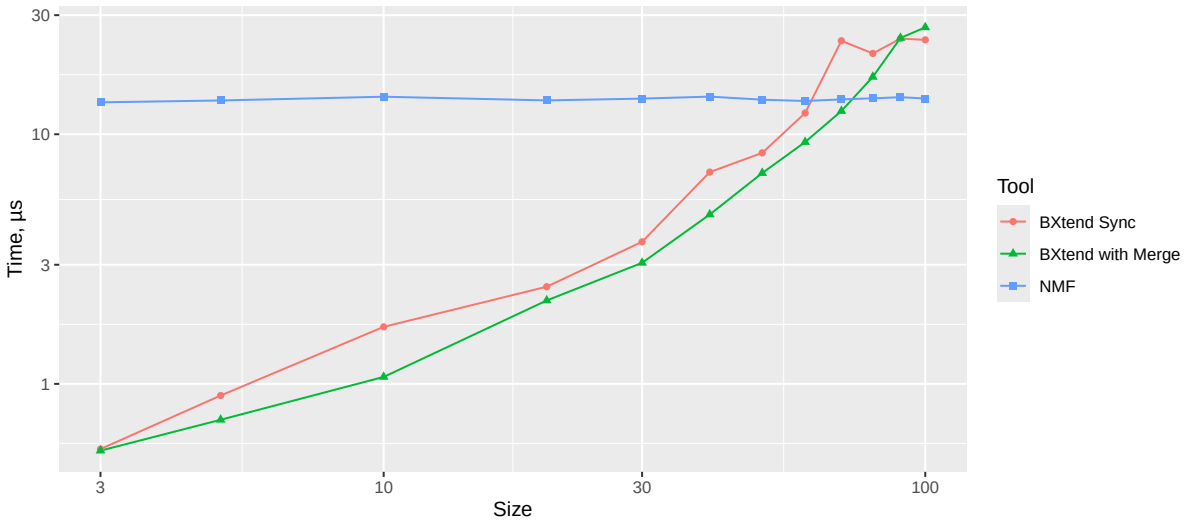


Figure 4: Results from the scalability tests for concurrent synchronisation cases with increasing model size and constant number of conflict-free changes

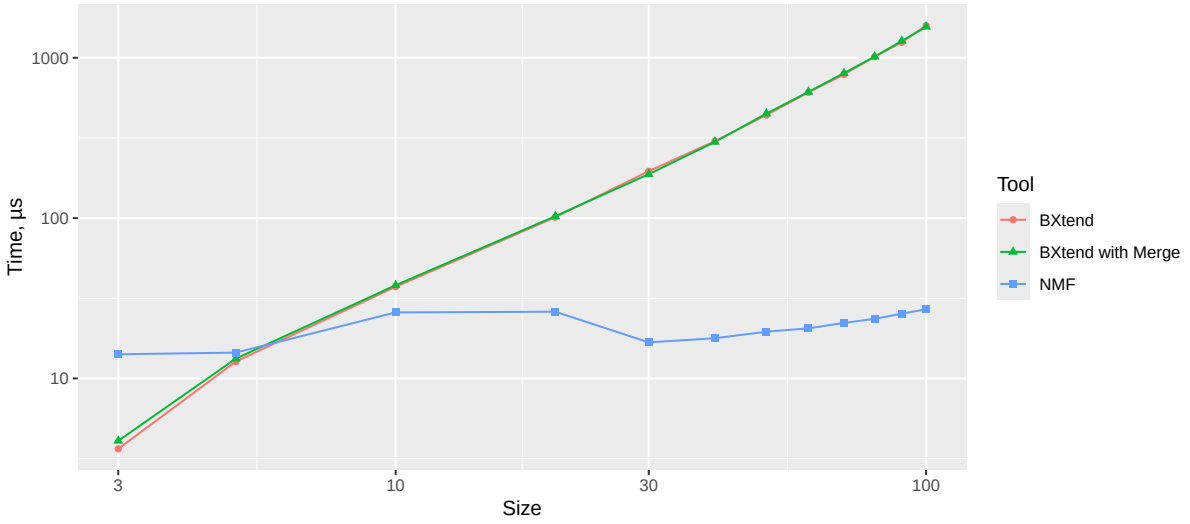


Figure 5: Results from the scalability tests for constant model size and a growing number of conflict-free changes

Note that the diagram in Figure 4 is logarithmic in both axes. The runtime of the NMF solution appears constant, which is clear since the time for the synchronization depends on the size of the change rather than the model size. Unlike the Java solutions that continuously run in the same process, also overhead such as JIT compilation might explain the relatively high runtimes for the smaller sizes.

Figure 5 shows the results for propagating a growing number of conflicts in a model of a constant size. Here, the times for the NMF solution also begin to rise for larger model sizes, but are still dominated by other effects such as JIT compilation and apparently confounding factors during measurement (running the benchmark requires at least an open Eclipse). For the larger models, the NMF solution is faster than the reference solutions by multiple orders of magnitude, confirming results obtained for the 2017 edition of the benchmark [3].

8. Conclusion

The extension was an interesting challenge to NMF. While the solution shows that the benchmark can be solved using essentially the exact same synchronization as in the 2017 edition of the benchmark, it gives good directions on how to extend NMF Synchronizations to support conflicting changes more generically. The performance measurements indicate that the solution is fast for the model sizes considered.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, A. Zündorf, Benchmarking bidirectional transformations: theory, implementation, application, and assessment, Softw. Syst. Model. 19 (2020) 647–691. URL: https://doi.org/10.1007/s10270-019-00752-x. doi:10.1007/s10270-019-00752-x.

[2] A. Anjorin, T. Buchmann, The Families to Persons Case – Revisited, CEUR Workshop Proceedings, CEUR-WS.org, 2026. To appear.

[3] G. Hinkel, An NMF solution to the Families to Persons case at the TTC 2017, in: A. Garcia-Dominguez, G. Hinkel, F. Krikava (Eds.), 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, 2017.

[4] G. Hinkel, R. Heinrich, R. Reussner, An extensible approach to implicit incremental model analyses, Software & Systems Modeling (2019). URL: https://doi.org/10.1007/s10270-019-00719-y. doi:10.1007/s10270-019-00719-y.

[5] G. Hinkel, E. Burger, Change propagation and bidirectionality in internal transformation DSLs, Softw. Syst. Model. 18 (2019) 249–278. URL: https://doi.org/10.1007/s10270-017-0617-6. doi:10.1007/s10270-017-0617-6.

[6] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, ACM Transactions on Programming Languages and Systems (TOPLAS) 29 (2007). URL: http://doi.acm.org/10.1145/1232420.1232424. doi:10.1145/1232420.1232424.

[7] G. Hinkel, Change Propagation in an Internal Model Transformation Language, in: D. Kolovos, M. Wimmer (Eds.), 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, Springer International Publishing, Cham, 2015, pp. 3–17. URL: http://dx.doi.org/10.1007/978-3-319-21155-8_1. doi:10.1007/978-3-319-21155-8_1.

[8] G. Hinkel, T. Goldschmidt, E. Burger, R. Reussner, Using Internal Domain-Specific Languages to Inherit Tool Support and Modularity for Model Transformations, Software & Systems Modeling (2017) 1–27. URL: http://rdcu.be/oTED. doi:10.1007/s10270-017-0578-9.