An NMF solution to the TTC 2020 roundtrip engineering case

Georg Hinkel[1]

[1]Am Rathaus 4b, 65207 Wiesbaden, Germany

Abstract

This paper presents a solution to the Roundtrip Engineering case at the Transformation Tool Contest (TTC) 2020. I demonstrate how synchronization blocks can be easily used to specify the relationships in a bidirectional manner. Through a superimposition concept, the migrations can concentrate on those parts of the metamodel that have actually changed. The performance results on the provided model shows that the solution has a very good performance on the provided input models, although these are very small.

Keywords

Model Migration, Roundtrip, BX, Transformation

1. Introduction

Just like every software artifact, metamodels are also subject to evolution. However, in large entities, metamodel change do not take place immediately but rather, one has to accept a period where both the old and the new schema version are used throughout the organization. During this period, changes can occur either in the new or in the old form of a model, which leads to a problem that models have to be maintained both in the new and in the old schema.

In the TTC 2020 Roundtrip benchmark [1], the task was to synchronize instances of evolving metamodels in a range of minimal example evolution scenarios.

In this paper, I present a solution to this benchmark using synchronization blocks and their implementation in NMF Synchronizations [2]. NMF Synchronizations allows us to create very declarative and fully bidirectional specifications of commonalities between two versions of a metamodel. Furthermore, because NMF Synchronizations is implemented as an internal DSL, it allows to shorten the specification of commonalities such that developers of roundtrip migrations can focus on the actual differences between the two versions of a metamodel. The solution is available on GitHub: https://github.com/georghinkel/ttc2020-roundtrips.

TTC'20: Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 17 July 2020, ~~Bergen, Norway~~ (online).

georg.hinkel@gmail.com (G. Hinkel)

2. Synchronization blocks and NMF Synchronizations

Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way [2]. They combine a slightly modified notion of lenses [3] 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 $\omega \in \Omega$:

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

The first condition is a direct translation of the original PutGet law. 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 $\boldsymbol{S}$ 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 Φ_{A-C}. For each such a 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 regard to Φ_{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 and in both directions. The engine simply computes the value that

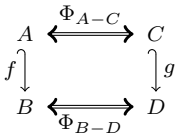

Figure 1: Schematic overview of unidirectional synchronization blocks

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 Kleene closures.

Synchronization blocks have been implemented in NMF Synchronizations, an internal DSL hosted by C# [4, 2]. For the incrementalization, it uses the extensible incrementalization system NMF Expressions [5]. 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 in 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 [6].

3. Solution

Our solution consists of two parts: At first, I describe the solutions to all four of the scenarios using vanilla NMF Synchronizations, that is, using explicit coding. Afterwards, I explain the necessary steps to turn this into a generic solution.

```
1  public class Person2Person : SynchronizationRule<V1Person,
       V2Person> {
2    public override void DeclareSynchronization() {
3      Synchronize(p => p.Name, p => p.Name);
4      Synchronize(p => p.Age, p => 2020 - p.Ybirth);
5    }
6    public override bool ShouldCorrespond(V1Person left,
         V2Person right, ISynchronizationContext context) {
7      return left.Name == right.Name;
8    }
9  }
```

Listing 1: The synchronization block to specify the semantic overlap between the Person classes in scenario 1

3.1. Specific solution

The idea of synchronization blocks is to specify the semantic overlap between two metamodels, not their difference. For scenario 1, this overlap consists of the name, which is still the same, and the overlap that the age can be computed from the year of birth or vice versa. This is depicted in Listing 1.

In particular, both aspects of the semantic overlap can be specified with just one line of code each. Here, the calculation of the age from the year of birth (and vice versa which NMF is able to automatically infer) is very simple because the inversion of a subtraction is already built into NMF. However, NMF also allows to specify a custom conversion operation and an appropriate lens to put back the value, in case a metamodel evolution requires more sophisticated adaptions.

As a very simple example, such a conversion is used in scenario 3, because the coalescing operator is not reversible in NMF by default. The implementation of the custom conversion is shown in Listing 2.

```
1  public class Person2Person : SynchronizationRule<V1Person,
       V2Person> {
2    public override void DeclareSynchronization() {
3      Synchronize(p => p.Name, p => Coalesce(p.Name));
4    }
5  }
6  [LensPut(typeof(Scenario3Solution), nameof(CoalesceBack))]
7  public static string Coalesce(string value) {
8    return value ?? "";
9  }
10 public static string CoalesceBack(string value, string
       coalesced)
11 {
12   return coalesced;
13 }
```

Listing 2: The synchronization blocks to specify the semantic overlap between the Person classes in scenario 3

In particular, one only needs to annotate a given conversion method with a lens put annotation in order to tell NMF how to invert this function call.

In order to run these synchronization blocks, I instruct NMF to enforce the consistency relations specified using

```
1  var repository = new ModelRepository();
2  var input = LoadModel<Scenario1.V1.Model.Person>(repository)
       ;
3
4  var transformation = new Scenario1Solution();
5  transformation.Initialize();
6
7  Scenario1.V2.Model.Person result = null;
8  // this call is the migrate step
9  transformation.Synchronize(ref input, ref result,
       SynchronizationDirection.LeftToRightForced,
       ChangePropagationMode.None);
10 // this call is the migrate back step
11 transformation.Synchronize(ref input, ref result,
       SynchronizationDirection.RightToLeftForced,
       ChangePropagationMode.None);
12
13 repository.Save(input, Output);
```

Listing 3: Running the transformation: Migrate and migrate back

synchronization blocks either from left to right or from right to left. In the context of this solution, left means V1 (because I always noted the V1 type on the left) and right means V2. Migrate and Migrate back therefore translate to simply calling the transformation with direction *left to right forced* and *right to left forced*. Here, forced means that also null values are propagated.

The execution of the synchronization is depicted in Listing 3. I first create a model repository in which I load the input model (lines 1 and 2), then create and initialize the transformation (lines 4/5). Then, I create a new and empty variable that I use to hold the migrated V2 model in line 7. In line 9, I force NMF to override this variable and put the migrated `Person` model element. Then, I immediately migrate the model back, using the same pattern. As the last parameter suggests, NMF is also able to obtain an incremental change propagation in case the models are to be used in-memory, but offline synchronization is also supported (by just disabling the change propagation).

A new information as in scenario 2 simply can be implemented by not synchronizing this attribute. Multiple edit operations as in scenario 4 simply means to combine the necessary synchronization blocks.

3.2. Generic solution

The biggest problem that I see with the specific solution is that the identical parts of the metamodel have to be specified over and over again. While of course not a problem for very small metamodels such as the ones in the benchmark, this can become a problem once the idea is applied to big metamodels with hundreds of classes as one has to create a separate synchronization rule for each metaclass and a synchronization block for every feature.

The code for generating such synchronization blocks for single-valued attributes and references is depicted in

```
1  foreach (var att in oldModelClass.Attributes) {
2    var newAtt = newModelClass.Attributes.FirstOrDefault(a =>
         a.Name == att.Name);
3    if (newAtt != null && att.Type == newAtt.Type && newAtt.
         LowerBound == att.LowerBound) {
4      // Create Synchronize call
5      var lambda = CreateLambdaFor<TOld>(att);
6      singleAttribute
7        .MakeGenericMethod(lambda.ReturnType)
8        .Invoke(this, new object[] { lambda,
             CreateLambdaFor<TNew>(newAtt) });
9    }
10 }
11 foreach (var oldReference in oldModelClass.References) {
12   var newReference = newModelClass.References.
         FirstOrDefault(r => r.Name == oldReference.Name);
13   if (newReference != null && newReference.LowerBound ==
         oldReference.LowerBound) {
14     // Create Synchronize call
15     var oldLambda = CreateLambdaFor<TOld>(oldReference);
16     var newLambda = CreateLambdaFor<TNew>(newReference);
17     var rule = Synchronization.
           GetSynchronizationRuleForSignature(oldLambda.
           ReturnType, newLambda.ReturnType);
18     singleReference
19       .MakeGenericMethod(oldLambda.ReturnType, newLambda.
             ReturnType)
20       .Invoke(this, new object[] { rule, oldLambda,
             newLambda, null });
21   }
22 }
```

Listing 4: Generating a synchronization block for each unchanged attribute and reference

Listing 4. For brevity, we do not handle inheritance, multi-valued attributes or references and only check whether an attribute or reference with the same name exists and whether the lower bound is the same (in order to account for the difference between null values and empty strings in scenario 3).

```
1  foreach (var oldClass in oldModel.Descendants().OfType<
       IClass>()) {
2    var newClass = newModel.Descendants().OfType<IClass>().
         FirstOrDefault(c => c.Name == oldClass.Name);
3    if (newClass != null) {
4      var oldMapping = oldClass.GetExtension<MappedType>();
5      var newMapping = newClass.GetExtension<MappedType>();
6
7      if (oldMapping?.SystemType != null && newMapping?.
           SystemType != null) {
8        var rule = (SynchronizationRuleBase)Activator.
             CreateInstance(typeof(MigrationRule<,>).
             MakeGenericType(oldMapping.SystemType,
             newMapping.SystemType));
9        yield return rule;
10     }
11   }
12 }
```

Listing 5: Generating synchronization rules

What we need to do is to generate synchronization rules to house the generated synchronization blocks. For this, we simply iterate over the classes of the metamodel and check whether there is a corresponding class in the new metamodel.

```
1  public class Scenario4Solution : Migration<V1.Container, V2.
       Container> {
2    [OverrideRule]
3    public class Person2Person : MigrationRule<V1.IPerson, V2
       .IPerson> {
4      public override void DeclareSynchronization() {
5        base.DeclareSynchronization();
6        Synchronize(p => p.Age, p => 2020 - p.Ybirth);
7      }
8    }
9  }
```

Listing 6: Superimposition of the migration for person elements

With these two artifacts, we get a model synchronization that automatically synchronizes all classes and features that have not changed (meaning that a feature with the same name exists), but we still need to specify the semantic overlap that is contained in different attributes such as the corespondence between age and year of birth.

To do that, we use the superimposition concept that is available in NMF Synchronizations, depicted in Listing 6. That is, we inherit from our new migration class that spawns the synchronization rules to synchronize the unchanged bits and then superimpose this rule by a more detailed rule that inherits the synchronization of unchanged attributes and references (line 5) and add the synchronization of the age with the year of birth by subtracting from 2020.

4. Evaluation

I ran performance measurements of the solution on a Intel Core i7-8550U CPU on a system with 8GB RAM running Windows 10. The results are depicted in Figure 2. The figure shows the required time to synchronize the model changes forward and backward. The suffix indicates the direction that is executed first, i.e. *Scenario1Backward* means that the V2 version of scenario 1 is migrated back to V1 and then migrated to V2 again, *Scenario1Forward* means that the V1 version of scenario 1 is migrated to V2 and back to V1 again.

As the figure shows, the solution is generally very fast. If the synchronization is only called once, the effects of just-in-time compilation and assembly loading cause an average of little more than 10ms but if the synchronization is repeated often, the runtime anneals to roughly 10ns per iteration. However, the input model size is also trivially small and therefore, the results are hardly meaningful. A thourough performance evaluation would require larger models that the benchmark did not provide.

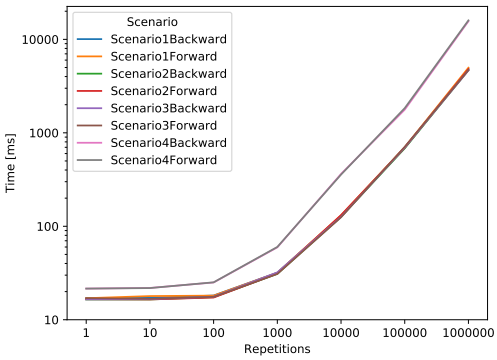

Figure 2: Performance results for running the Transformation step of the solution multiple times.

5. Conclusion

I think that the NMF solution highlights the advantages model transformations based on synchronization blocks can offer in terms of flexibility. A single specification of consistency relationships between the evolution steps of a metamodel suffices to transform instances forwards and backwards. Boilerplate rules can be calculated automatically while the essential differences between two evolution steps (the actual migration) is specified manually with the full flexibility.

One may think that the generic solution and the reflection it performs must lead to a slow solution. However, this is not true because NMF uses the .NET expression compiler under the hood to compile the expressions that are built through reflection. Therefore, the reflection only affects the initialization of such a transformation, the runtime is completely identical.

References

[1] L. Beurer-Kellner, J. von Pilgrim, T. Kehrer, Round-Trip Migration of Object-Oriented Data Model Instances, http://www.transformation-tool-contest.eu/2020_roundtrip.pdf, 2020.

[2] G. Hinkel, E. Burger, Change Propagation and Bidirectionality in Internal Transformation DSLs, Software & Systems Modeling (2017). URL: http://rdcu.be/u9PT. doi:10.1007/s10270-017-0617-6.

[3] 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.

[4] 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.

[5] 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.

[6] 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.