An NMF solution to the Smart Grid 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 Smart Grid case at the Transformation Tool Contest (TTC) 2017 using the .NET Modeling Framework (NMF). The goal of this case was to create incremental views of multiple models relevant in the area of smart grids. Our solution uses the incremental model transformation language NMF Synchronizations and the underlying incrementalization system NMF Expressions.

1 Introduction

Models should represent a system in a very abstract form. However, very often, the model is still too complex for humans to understand it or make use of it. Furthermore, necessary information is split among multiple models. Therefore, it is often beneficial for practical applications to reduce the complexity for human modelers through the use of views that combine the information from multiple models and reduce it to those parts of a model that are relevant for a particular task.

The Smart Grid Case of the Transformation Tool Contest (TTC) 2017 proposes a benchmark for such a scenario. Here, the modeled system is a smart grid where the necessary information to detect or predict is split among multiple models according to existing standards. The views originate from a model-based outage management system [Mit, BMK16] implemented using existing model view technology [BHK$^+$14].

If the source model changes, the view has to be adapted to the changed source. For large models, it becomes very slow to recompute the entire model from scratch, in particular, since changes usually only affect small parts of the model. Rather, it is much more efficient to only propagate the changes to the view in an incremental manner. However, implementing such a change propagation manually can be a very laborious task that further conceals the code intention, i.e. the view that is actually being computed.

This paper presents a solution to the proposed benchmark using the incremental model transformation language NMF Synchronizations [Hin15], integrated into the .NET Modeling Framework (NMF, [Hin16]). The solution is publicly available on Github[1]. We first give a very brief introduction into synchronization blocks, the formalism underneath NMF Synchronizations in Section 2 before Section 3 presents the solution. Section 4 evaluates the solution against the reference solution in ModelJoin and finally Section 5 concludes the paper.

2 Synchronization Blocks

Synchronization blocks are a formal tool to run model transformations in an incremental (and bidirectional) way. They combine a slightly modified notion of lenses [FGM$^+$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 Ω.

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

[1] `https://github.com/georghinkel/ttc2017smartGrids`

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 $a \in A$ and $\omega \in \Omega$:

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

The first condition is a direct translation of the original PUTGET law. Meanwhile, the second line is a bit weaker than the original GETPUT because the global state may have changed. In particular, we allow the PUT function to change the global state.

An unidirectional (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 function f and the lens g are in the dependent isomorphism Φ_{B-D}.

$$\begin{tikzcd} A \arrow[r, Leftrightarrow, "\Phi_{A-C}"] \arrow[d, "f"'] & C \arrow[d, hook, "g"] \\ B \arrow[r, Leftrightarrow, "\Phi_{B-D}"'] & D \end{tikzcd}$$

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 this declarations to be enforced automatically[2]. The engine simply computes the value that the right selector should have and enforces it using the PUT operation.

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# [Hin15].

3 Solution

We discuss the solutions to the outage detection and the outage prevention tasks separately in Sections 3.1 and 3.2.

3.1 Outage Detection

In NMF Synchronizations, the support for multiple input pattern elements is rather limited. As a reason, we experienced with NTL [Hin13] that multiple input elements is a rare case, but required a tremendous amount of code to support it. At the same time, the advantages of a true support for multiple input elements over transformation of tuples is limited.

Therefore, the easiest way to support multiple input pattern elements in NMF Synchronizations is to simply use tuples as inputs. Then, the model matching has to be adapted to match tuples instead of elements. Therefore, the main rule synchronizes a tuple of the CIM model and the COSEM model with the resulting view model.

$$\begin{tikzcd} CIMRoot \times COSEMRoot \arrow[r, Leftrightarrow, "\Phi_{MainRule}"] \arrow[d, "(join)"'] & Model \arrow[d, hook, ".RootElements.OfType < EnergyConsumer >"] \\ (MeterAsset \times PhysicalDevice)* \arrow[r, Leftrightarrow, "\Phi_{AssetToConsumer}"'] & EnergyConsumer* \end{tikzcd}$$

Figure 2: The join in the outage detection task formulated in a synchronization block

In a synchronization block, the main join of meter assets with physical devices is depicted in Figure 2, where we abbreviated the join expression. The implementation of this matching is depicted in Listing 1.

---

[2] If f was also a lens, then the synchronization block can be enforced in both directions.

```
public class MainRule : SynchronizationRule<Tuple<CIMRoot, COSEMRoot>, Model> {
  public override void DeclareSynchronization() {
    SynchronizeManyLeftToRightOnly(SyncRule<AssetToConsumer>(),
      sg => from pd in sg.Item2.PhysicalDevice
            join ma in sg.Item1.IDobject.OfType<IMeterAsset>()
            on pd.ID equals ma.MRID
            select new Tuple<IMeterAsset, IPhysicalDevice>(ma, pd),
      target => target.RootElements.OfType<IModelElement, OutageDetectionJointarget.IEnergyConsumer>());
  }
}
```

Listing 1: The implementation of the main rule for outage the outage detection task

Because .NET has a hard implementation of generics[3], a type filter can be easily specified by passing generic type arguments. NMF also contains an overload of the `OfType` type filter that accepts two type arguments and keeps the collection interface.

In particular, the incrementalization system NMF Expressions that is underlying NMF Synchronizations does support joins, available also through the query syntax of C#. A second synchronization rule then implements the kept attributes for every such a tuple, as depicted in Listing 2.

```
public class AssetToConsumer : SynchronizationRule<Tuple<IMeterAsset, IPhysicalDevice>, IEnergyConsumer> {
  public override void DeclareSynchronization() {
    SynchronizeLeftToRightOnly(
      asset => Convert.ToInt32(asset.Item2.AutoConnect.Connection), e => e.Reachability);
    SynchronizeLeftToRightOnly(asset => asset.Item2.ElectricityValues.ApparentPowermL1, e => e.PowerA);
    SynchronizeLeftToRightOnly(asset => asset.Item1.ServiceDeliveryPoint.EnergyConsumer.MRID, e => e.ID);
    SynchronizeLeftToRightOnly(
        asset => asset.Item1.ServiceDeliveryPoint.EnergyConsumer is ConformLoad ?
        ((ConformLoad)asset.Item1.ServiceDeliveryPoint.EnergyConsumer)
               .LoadGroup.SubLoadArea.LoadArea.ControlArea.MRID :
        ((NonConformLoad)asset.Item1.ServiceDeliveryPoint.EnergyConsumer)
               .LoadGroup.SubLoadArea.LoadArea.ControlArea.MRID,
      e => e.ControlAreaID);
    SynchronizeLeftToRightOnly(SyncRule<LocationToLocation>(),
      asset => asset.Item1.Location, e => e.Location);
  }
}
```

Listing 2: Implementation of kept attributes and references in the outage detection task

Two further synchronization rules synchronize location and position point.

3.2 Outage Prevention

In the implementation of the outage prevention task, the principle approach to use tuples to synchronize multiple inputs is the very same approach as in the outage detection task. The implementation of the main rule is depicted in Listing 3.

```
public class MainRule :
    SynchronizationRule<Tuple<CIMRoot, COSEMRoot, Substandard>, Model> {
  public override void DeclareSynchronization() {
    SynchronizeManyLeftToRightOnly(SyncRule<MMXUAssetToVoltageMeter>(),
      dr => dr.Item1.IDobject.OfType<IMeterAsset>()
              .Join(dr.Item3.LN.OfType<IMMXU>(),
                    asset => asset.MRID,
                    mmxu => mmxu.NamePlt.IdNs,
                    (asset, mmxu) => new Tuple<IMeterAsset, IMMXU>(asset, mmxu)),
      model => model.RootElements.OfType<IModelElement, IPMUVoltageMeter>());

    SynchronizeManyLeftToRightOnly(SyncRule<DeviceAssetToPrivateMeterVoltage>(),
      dr => dr.Item1.IDobject.OfType<IEndDeviceAsset>()
              .Join(dr.Item2.PhysicalDevice,
                    asset => asset.MRID,
                    pd => pd.ID,
                    (asset, pd) => new Tuple<IEndDeviceAsset, IPhysicalDevice>(asset, pd)),
      model => model.RootElements.OfType<IModelElement, IPrivateMeterVoltage>());
  }
}
```

Listing 3: The implementation of the main rule in the outage prevention task

---

[3] This means that the generic type arguments are still available at runtime.

In this listing, we used the alternative method chaining syntax for the join. Both syntaxes are equivalent, as the compiler converts the query syntax into the method chaining syntax.

To handle the different transformation of the various subtypes of a power system resource, we utilize the rule instantiation feature of NMF Synchronizations. With a rule instantiation, the isomorphism represented by a synchronization rule can be refined for a subset of model elements.

```
public class PowerSystemResource2PowerSystemResource
      : SynchronizationRule<IPowerSystemResource, IPowerSystemResource> {
  public override void DeclareSynchronization() {}
}
public class ConductingEquipment2ConductingEquipment
      : SynchronizationRule<IConductingEquipment, IConductingEquipment> {
  public override void DeclareSynchronization() {
    SynchronizeManyLeftToRightOnly(SyncRule<Terminal2Terminal>(),
        conductingEquipment => conductingEquipment.Terminals, equipment => equipment.Terminals);
    MarkInstantiatingFor(SyncRule<PowerSystemResource2PowerSystemResource>());
  }
}
```

Listing 4: Transforming power system resources

An example of synchronization rule instantiation for conducting equipment is depicted in Listing 4. This means that whenever a power system resource is a conducting equipment, also its terminals are synchronized.

4 Evaluation

Our solution is quite concise as it only consists of 58 lines of code for the outage detection scenario and 195 lines of code for the outage prevention scenario. Both numbers include empty lines as well as lines that only contain braces. Another 140 lines of code actually run the benchmark.

The performance results are depicted in Figure 3. They show that after an initial overhead, the time to propagate updates is nearly constant, indicating no larger bottleneck.

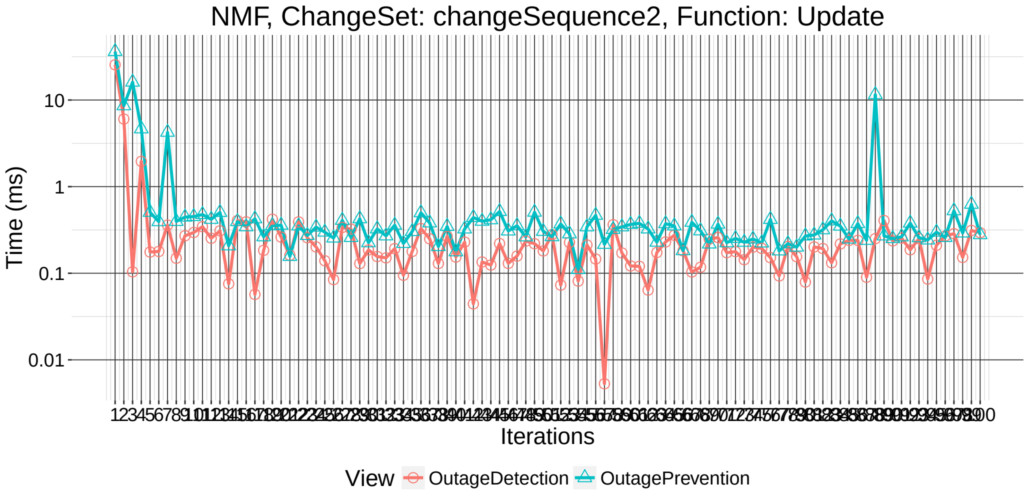

Figure 3: Update times for change sequence 2

The comparison with the reference solution in ModelJoin reveals that the incremental update processing of our solution is more than an order of magnitude faster than recomputing the view after every change sequence. The results are depicted in Figure 4.

However, the results also indicate that more change sequences available for different sizes are necessary to evaluate the scalability of our solution.

Unfortunately, the reference solution in ModelJoin produced compilation errors for the generated QVTo-transformations, which is why we cannot compare with this implementation in the OutagePrevention task, but we look forward to compare our solution with other solutions of the case.

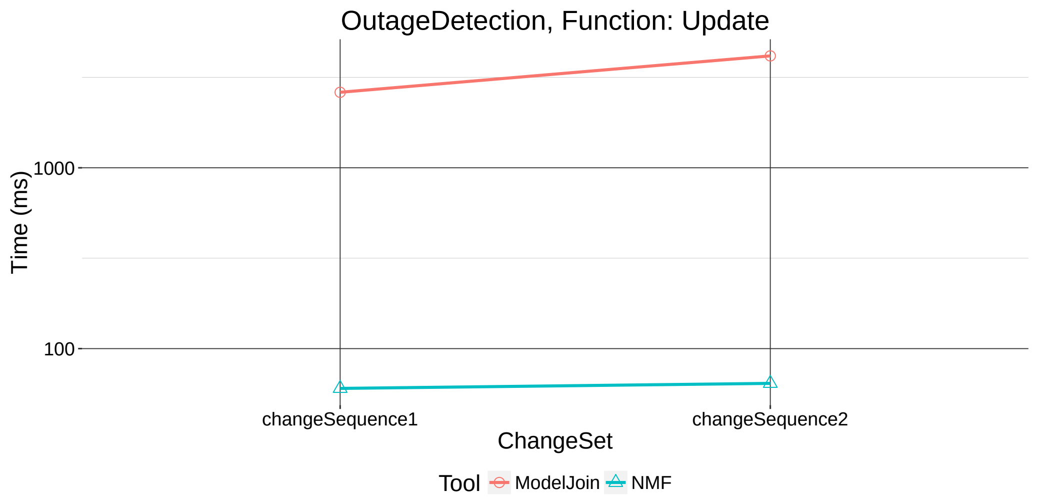

Figure 4: Results for the outage detection task

5 Conclusion

In this paper, we presented the NMF solution to the Smart Grid case at the TTC 2017. The solution shows how synchronization blocks, in particular their implementation in NMF Synchronizations can be used to perform incremental view computations. The resulting solution is faster than the reference implementation by multiple orders of magnitude.

References

[BHK+14] Erik Burger, Jörg Henß, Martin Küster, Steffen Kruse, and Lucia Happe. View-Based Model-Driven Software Development with ModelJoin. *Software & Systems Modeling*, 15(2):472–496, 2014.

[BMK16] Erik Burger, Victoria Mittelbach, and Anne Koziolek. Model-driven consistency preservation in cyber-physical systems. In *Proceedings of the 11th Workshop on Models@run.time co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. CEUR Workshop Proceedings, October 2016.

[FGM+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.

[Hin13] Georg Hinkel. An approach to maintainable model transformations using internal DSLs. Master thesis, 2013.

[Hin15] Georg Hinkel. *Change Propagation in an Internal Model Transformation Language*, pages 3–17. Springer International Publishing, Cham, 2015.

[Hin16] Georg Hinkel. NMF: A Modeling Framework for the .NET Platform. Technical report, Karlsruhe Institute of Technology, Karlsruhe, 2016.

[Mit] Victoria Mittelbach. Model-driven Consistency Preservation in Cyber-Physical Systems. Master's thesis, Karlsruhe Institute of Technology (KIT), Germany.