An NMF Solution to the TTC2023 Containers to MiniYAML Case

Georg Hinkel[1]

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

Abstract
This paper presents a solution to the Containers to MiniYAML Case at the TTC 2023 using the .NET Modeling Framework (NMF), especially NMF Synchronizations. This solution is able to derive an incremental change propagation entirely in an implicit manner.

Keywords
incremental, model-driven, transformation

1. Introduction

To denote the infrastructure of distributed systems, models are often used to capture deployment information at a high level. Often, very generic languages are used as they offer a great flexibility. However, to process the information contained in these models, often type-safe representations need to be extracted and may be individually maintained. If this is the case, a synchronization between both representations is necessary in order to keep both artifacts up to date. The TTC 2023 Containers to MiniYAML case poses an example where deployment information is stored in very generic YAML files that need to be synchronized as the deployment information may contain details not present in the conceptual model while the latter may contain information such as layouts for graphical editors that are not present in the original YAML file.

This case is particular interesting for NMF as it applies model synchronization to models of different levels of abstraction. While previous cases denoted a synchronization of models that contained essentially the same information in different ways, this case denotes a synchronization between a very specific model like the containers model and a very generic metamodel for YAML.

In this paper, I demonstrate a solution to this case using NMF Synchronizations. NMF Synchronizations makes it possible to use a simple and concise specification of consistencies to gain an efficient, bidirectional transformation with support for incremental updates on both ends.

The remainder of this paper is structured as follows: Section 2 gives a brief overview how NMF Expressions and NMF Synchronizations work. Section 3 explains the actual solution. Section 4 explains how the solution was integrated into the benchmark framework. Section 5 discusses results from the benchmark framework before Section 6 concludes the paper.

2. NMF Expressions and NMF Synchronizations

NMF Expressions [1] is an incrementalization system integrated into the C# language. That is, 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 adapt when necessary. The incrementalization system is extensible and supports large parts of the Standard Query Operators (SQO[1]).

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 [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 $l \searrow \in Mor(A \times B, A)$

TTC'23: 15th Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 20 July 2023, Leicester, UK.
georg.hinkel@hs-rm.de (G. Hinkel)
https://www.hs-rm.de/de/hochschule/personen/hinkel-georg (G. Hinkel)
0000-0002-6462-5208 (G. Hinkel)

[1] 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.

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 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}.

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

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 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 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. 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 [5].

Based on this formal notion of synchronization blocks and in-model lenses, one can prove that model synchronizations built with well-behaved in-model lenses are correct and hippocratic [2]. That is, updates of either model can be propagated to the other model such that the consistency relationships are restored and an update to an already consistent model does not perform any changes.

3. Solution

The solution consists of three synchronization rules adapted from the Epsilon solution and a couple of synchronization blocks, synchronizing details of the models. These synchronization rules are the `MainMap` rule as the start rule, the `Container2MapEntry` that synchronizes containers and the `Volume2MapEntry` rule to synchronize volumes.

The `MainMap` rule consists of three rather simple synchronization blocks depicted in Listing 1.

```
SynchronizeLeftToRightOnly(_ => "2.4", m => m.Scalar<string>(
       "version"));

SynchronizeMany(SyncRule<Volume2MapEntry>(),
    c => c.Nodes.OfType<INode, IVolume>(),
    m => m.ForceEntries("volumes"));
SynchronizeMany(SyncRule<Container2MapEntry>(),
    c => new ServicesCollection(c),
    m => m.ForceEntries("services"));
```

Listing 1: The MainMap rule

The first one simply sets the version scalar attribute to 2.4. The second and third synchronization blocks denote synchronization blocks to synchronize the containers and the volumes. The `Composition` metaclass only has a very generic `nodes` reference, therefore we need to work with a type filter. Due to type inference restrictions in .NET, unfortunately these type filters also need to specify the actual collection type. On the YAML side, we are working with a helper method to find the map entry with name *volumes* (or *services*, respectively), make sure it exists, make sure its value is a map and return the entries of that map. Because this is done outside of NMF Synchronizations, it has the downside that this is not being change-tracked. That is, if a client was to change the name of the map entry, NMF Synchronizations would not see that the elements would no longer be services/volumes.

When adding a container to the services because a corresponding entry was added to the YAML model, there is an additional task: We also need to make sure that an image element exists that corresponds to the image entry of the YAML container. Because this is a very imperative logic, this is implemented in a dedicated collection class `ServicesCollection` in order to take control over the behavior when NMF Synchronizations adds the container created for the Map entry to the container model.

A sketch of the implementation for `ServicesCollection` is depicted in Listing 2. The same type filter operation is passed into the constructor of this class (with just one generic type argument because this time, the collection is readonly), but with custom implementations for collection modifications: adding, removing or entirely clearing the collection. In NMF, collection classes like `ServicesCollection` are called virtual collections.

```
private class ServicesCollection : CustomCollection<
      IContainer> {
    public ServicesCollection(Composition comp)
      : base(comp.Nodes.OfType<IContainer>())
    { _comp = comp; }

    public override void Add(IContainer item) { ... }
    public override void Clear() { ... }
    public override bool Remove(IContainer item) { ... }
}
```

Listing 2: A sketch of the `ServicesCollection` implementation

The second rule and maybe the most interesting one is `Containers2MapEntry`. This rule controls the synchronization of a container with a map entry in the YAML model. It consists of five synchronization blocks as depicted in Listing 3.

```
Synchronize(c => c.Name, me => me.Key);

Synchronize(c => GetImage(c), me => me.Scalar<string>("image"
      ));
Synchronize(c => c.Replicas.WithDefault(1), me => me.Scalar<
      int?>("replicas"));

SynchronizeMany(
  c => new VolumeMountCollection(c),
  me => new ScalarCollection(me, "volumes"));
SynchronizeMany(
  c => new DependsOnNameCollection(c),
  me => new ScalarCollection(me, "depends_on"));
```

Listing 3: The synchronization blocks of `Containers2MapEntry`

The first synchronization block just denotes that the names of the container and the map entry generated for it should be synchronized. The next two synchronization blocks utilize a helper function to read and write entries of the map entries map as a given type and denote that this value should be synchronized with values from the container. This applies to the image of the container and the replicas. For the replicas, we want to treat no definition of replicas as 1, for which we created another helper function `WithDefaults`. This helper function essentially changes the default value for a given type and is sufficiently generic that we will take it over into the source code of NMF.

In order to run the synchronization block bidirectionally, these helper functions need to be specified as in-model lenses. For this, NMF uses dedicated annotations as depicted in Listing 4.

```
[LensPut(typeof(YamlHelpers), nameof(SetScalar))]
public static T? Scalar<T>(this IMapEntry? entry, string key)
{ ... }

public static void SetScalar<T>(this IMapEntry? entry, string
      key, T? value)
{ ... }
```

Listing 4: Signature of the `Scalar` helper method and Lens put

The definition of Listing 4 is what NMF calls a persistent lens [2]. This denotes that the put function entirely propagates the value. An alternative is a non-persistent lens, which in this case would have to return a value of type `T?` that NMF would then propagate to the next lens.

Lenses are used as black boxes in the synchronization. That is, even though the implementation of the `Scalar` method depicted in Listing 4 of course casts the value of the map entry to a map and then looks for the map entry with the given name, casting its value to a scalar, these accesses are not recorded and the transformation will therefore not react on changes in this chain. For instance, if one accidentally or not renames the `image` element, NMF Synchronization would not reset the image of the container because it does not notice that the scalar element is no longer the correct one.

```
[LensPut(typeof(YamlHelpers), nameof(SetScalar))]
[ObservableProxy(typeof(YamlHelpers), nameof(
      ScalarIncremental))]
public static T? Scalar<T>(this IMapEntry? entry, string key)
{ ... }

public static INotifyValue<T?> ScalarIncremental<T>(IMapEntry
      entry, string key)
{ ... }

public static void SetScalar<T>(this IMapEntry? entry, string
      key, T? value)
{ ... }
```

Listing 5: Signature of the `Scalar` helper method and Lens put, revised

To achieve this, a second annotation is used to tell NMF Expressions when the scalar value changes, depicted in Listing 5. To implement the incremental version of the helper, we use a standard implementation

`NotifyExpression` and instruct it to reevaluate the scalar value whenever the contents of the `Entries` collection of the underlying map changes. This implementation is slightly incomplete as also value changes of the scalar would have to taken into account, but this is not implemented because it was not needed for the benchmark. Implementing this kind of helper methods with lens put operations and incremental proxies is admittedly complex, but could be reused for any model synchronization targeting YAML, thus potentially in a wide range of projects, justifying a slightly higher implementation effort.

However, YAML is not the only metamodel where this kind of key-value storage appears, so we are also considering to add more generic primitives to NMF synchronizations that generically target metamodels with key-value-like storage.

Similarly, the function `GetImage` in the second synchronization block is also a lens. However, in this case the incrementalization of the method can be done by NMF directly, which is why we can use this functionality.

```
private static readonly ObservingFunc<IContainer, string?>
_getImage = ObservingFunc<IContainer, string?>.FromExpression
      (
    c => c.Image != null ? c.Image.Image_ : null);

[ObservableProxy(typeof(Container2MapEntry), nameof(
      GetImageIncremental))]
[LensPut(typeof(Container2MapEntry), nameof(SetImage))]
public static string? GetImage
(IContainer container) => _getImage.Evaluate(container);

public static INotifyValue<string?> GetImageIncremental
(IContainer container) => _getImage.Observe(container);

public static void SetImage(IContainer container, string
      image)
{ ... }
```

Listing 6: The `GetImage` helper method

This implementation is depicted in Listing 6. NMF encapsulates both incremental and non-incremental implementation of the actual lambda expression into an object that can be used to either evaluate the lambda expression for a given input or observe the parameters and return an object that can be used to fetch updates when the value changes. Here, NMF is able to infer that the image changes either if a image element is assigned or the actual image reference of the image element changes. Also note that NMF automatically suffixed the `image` property with a _ because properties cannot have the same name as their declaring class in C# and the naming convention of Pascal case would cause a name conflict between the `Image` class and property name.

The third pair of synchronization blocks denote the synchronization of collections. Here, we again use three helper classes that denote virtual collections. As an example for these collections, the custom collection for volume mounts is depicted in Listing 7.

```
private class VolumeMountCollection : CustomCollection<string
      > {
  private readonly IContainer _container;

  public VolumeMountCollection(IContainer container)
    : base(container.VolumeMounts.Select(vm => \$"{vm.Volume.
          Name}:{vm.Path}"))
  {  _container = container; }

  public override void Add(string item)
  { ... }

  public override void Clear()
  { ... }

  public override bool Remove(string item)
  { ... }
}
```

Listing 7: Sketch of the custom collection for the volume mounts of a container

Custom collections are initialized with an expression that NMF is able to incrementalize but unable to infer generic operations to add elements. In the case of the collection of volume mounts, this is a select call from the volume mounts to format them into strings. However, the reverse of such operations is usually not clear, in this case it is not obvious how to convert the string representation of a volume mount back to the model. Rather, this logic is very application specific, in this case that we know that the colon is always the separator between the volume name (which must not contain colons) and the path. Therefore, NMF requires the developer to explicitly specify what should happen in these cases, but at least the developer does not have to care where these changes come from.

The third synchronization rule to synchronize volumes to map entries only contains a synchronization rule to synchronize the names of the volume and the corresponding map entry.

4. Integration into Benchmarx

While the benchmark framework Benchmarx is implemented in Java, the presented NMF solution runs in .NET. NMF also uses its own model representation implementing the standard change notification interfaces present in the .NET platform. Therefore, meanwhile technologies exist to expose Java objects to the .NET runtime, these approaches would not help because the code generated by EMF does not implement these (.NET) interfaces. Therefore, similarly to the FamiliesToPersons case in 2017, we have chosen an approach to register to the EMF change notifications, generate an NMF change model out of it, serialize the change model into an XMI file and then load these change descriptions within a companion process that runs the NMF solution.

The communication to this companion process is done through a mixture of stdin/stdout and file-based communication. When Benchmarx starts the synchronization dialogue, the solution starts the companion process, which in turn creates empty source and target models. Whenever changes are made to source or target model, the (Java) NMF implementation of the benchmark uses an adapter attached to the in-memory models to obtain the changes in the format of NMF change models and serializes these change models to a temporary file. Once this is completed, the solution notifies the companion process to load the changes by writing the path to the change model to stdin. The companion app then deserializes the changes. Because NMF Synchronizations is an online incremental solution, applying the changes causes the respective other model to get synchronized. The companion app therefore measures the time to apply the updates as these include propagating them to the other side. The measured time is then written to stdout from where the Java implementation of the benchmark can pick it up. The Java benchmark implementation then asks the companion process to serialize the current source or target model into a given file from which it deserializes this model in order to allow for the correctness checks from the benchmark.

The implementation of the change recorder is very generic. In particular, the implementation from the 2017 FamiliesToPersons case [6] could be reused but needed to be extended because the Containers to MiniYAML case includes more kinds of changes. However, with the changes for the TTC 2023, the change recorder implementation is quite feature-complete and could be reused for any application that needs to serialize changes of EMF models to the NMF change model format.

While this kind of integration means that the solution has a high overhead in terms of serializing and deserializing model changes, it also allows to implement a requirement from the benchmark framework: As an online synchronization approach, NMF Synchronizations always propagates changes. However, the Containers to MiniYAML benchmark mandates also so called idle edits, which in the Containers to MiniYAML case are interpreted as changes that should not be propagated. In the NMF solution, this is implemented by just not propagating the updated models back to the benchmark framework. However, if the behavior to defer the propagation of changes is really necessary, NMF has multiple options to implement this: One could either create all the changes in a transaction, which means that the change propagation is only done once the transaction is completed. Alternatively, NMF Synchronizations can also run in an incremental check-only mode in which it records inconsistencies and keeps a incrementally maintained list of inconsistencies with methods allowing a user to resolve these inconsistencies (by updating either source or target model).

5. Evaluation

As expected, the solution passes all tests that ignore the order in the YAML while all tests that check that the solution preserves the order in the YAML files fail. The reason here is that NMF Synchronizations differentiates strongly between synchronization blocks that use the identity as the isomorphism (such as typically used to synchronize attributes) and those that use other synchronization rules. This is because NMF Synchronizations uses NMF Transformations under the hoods, which defers the execution of the rule bodies in order to turn possible trace resolves to "late resolve" operations as in QVT-R. Further, NMF Expressions generally does not have support for ordering elements, yet. That is, while synchronization blocks are processed in the order in which they occur, the collection-valued lens implementations generally ignore the order of elements. This is not a limitation of the theory but rather only a limitation of the implementation that currently does not support order. The change interface that NMF is using in fact does report indices where elements have been inserted or removed but there is no functionality in place to ensure that orderings are kept across lenses as they are sometimes hard to implement. As an example, it is quite hard to get the index of an added element in a filtered collection, given the index of the element in the source collection, compared at least to propagating the change in constant time when order is not required. Currently, the infrastructure of NMF Expressions is not able to calculate whether the order of elements is required, particularly because this information is not present in the .NET collection interfaces. Therefore, all tests that require an exact order are going to fail.

Traditionally, the time measurements of the benchmark framework is entirely odd as the actual propagation of changes only takes a fraction of the actual runtime, which is mainly used for recording the changes, serializing them, deserializing them in NMF, serializing the result model in NMF and deserializing it in the benchmark framework. In the FamiliesToPersons case from 2017 [6, 7], this serialization effort took more than 90% of the runtime while the actual propagation was very fast. I do not see a reason why the actual change propagation time should be higher in this case, but performance was not a concern of the case and therefore, no measurements have been performed.

A problem of the NMF solution here is certainly that although synchronization blocks are by themselves very concise, the synchronization needs a lot of customization which contribute to the entire solution being more verbose than others in the end. Here, especially the fact that the YAML model is extremely generic plays a crucial role and makes it necessary for the solution to search for the required information instead of directly accessing it as in a more closed metamodel. However, in the wild, there are a lot of standards that – like MiniYAML – operate rather on an open-world assumption and essentially allow users to denote whatever they want rather than having a fixed schema (like the container model does). Therefore, supporting such open-world metamodels will be an important subject of future work.

I see the major advantage of this solution that it does combine both directions into a single transformation, even though it may make the synchronization a bit more difficult to write sometimes. Essentially, NMF Synchronization breaks up the bidirectionality of the transformation into smaller pieces. Instead of multiple largely independent transformations of entire models that need to fit together, NMF forces developers to work implement bidirectionality in smaller chunks, mostly in-model lenses or their collection-valued equivalents, custom collections. Because the transformation crosses abstraction boundaries, we often needed to implement our own in-model lenses, particularly on the rather generic metamodel, which here is the YAML metamodel. These in-model lenses are a lot easier to review and test and the formal foundation of synchronization blocks gives a clear notion of what properties these pairs of functions need to fulfill. Ideally, these notions could be proved by theorem provers, but this has not been done, yet and may be subject of future work.

6. Conclusion

The NMF solution shows how the Containers to YAML transformation from the case study can be implemented in a bidirectional fashion by decomposing it into multiple isomorphisms with synchronization blocks. The advantage of this decomposition is that it allows developers to break down the bidirectionality into smaller pieces that are easier to implement and review while ensuring correctness of the resulting transformation through theoretical proofs. However, the solution also shows that the support of NMF when synchronizing models at different abstraction levels is complicated and requires a lot of helper functions. To provide a better support in such cases and to support order of elements will be subject of future research.

References

[1] 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.

[2] 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.

[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, 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.

[6] 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.

[7] 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.