An NMF Solutions to the TTC2023 Containers to MiniYAML Case

Georg Hinkel
georg.hinkel@hs-rm.de
RheinMain University of Applied Sciences
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.

CCS CONCEPTS

• **Software and its engineering → Object oriented frameworks**; **Specialized application languages**; *API languages*.

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 discusses results from the benchmark framework before Section 5 concludes the paper.

2 NMF EXPRESSIONS AND NMF SYNCHRONIZATIONS

NMF Expressions [5] 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 [3]. They combine a slightly modified notion of lenses [1] 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 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}.

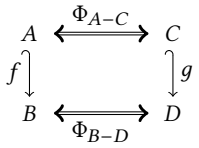

Figure 1: Schematic overview of unidirectional synchronization blocks

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

[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.

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# [2, 3]. 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 [4].

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 [3]. 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<Container2MapEntry>(),
    c => c.Nodes.OfType<INode, IContainer>(),
    m => m.ForceEntries("services"));
SynchronizeMany(SyncRule<Volume2MapEntry>(),
    c => c.Nodes.OfType<INode, IVolume>(),
    m => m.ForceEntries("volumes"));
```

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 `Container` metaclass only has a very generic `nodes` reference, therefore we need to work with a type filter. On the YAML side, we are working with a helper method to find the map entry with name *services* (or *volumes*, 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.

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 2.

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

Synchronize(c => c.Image.Image_, 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 2: 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 3.

```
[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 3: Signature of the `Scalar` helper method and Lens put

The definition of Listing 3 is what NMF calls a persistent lens [3]. 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 3 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 does not reset the image of the container because it does not notice that the scalar element is no longer the correct one. This can be changed through dedicated annotations, but this is not done,

hence changes such as renaming key elements in the YAML is not supported.

The third pair of synchronization blocks denote the synchronization of collections. Here, we use three helper classes that denote virtual collections. That is, we wrap a collection of elements with custom methods for changing collection contents. As an example for these collections, the custom collection for volume mounts is depicted in Listing 4.

```
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 4: 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 EVALUATION

The integration of the presented solution into the benchmark framework is still pending. Unfortunately, this integration is rather difficult due to the instability of Eclipse, which sometimes just fails to resolve the basic types of the benchmark framework. 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, 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 did not perform measurements so far.

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.

There is one shortcoming in this transformation, which is also the reason that several tests fail. It is the handling of null values. If the property of a model element is not set, NMF assigns a null reference and the standard behaviour in .NET is to throw a `NullReferenceException` when trying to access a property of a null reference (whereas this returns OclUndefined in OCL). In the incremental execution of NMF, this changed behaviour would be quite easy to implement, but in case no change propagation is required, NMF compiles the code expressions to normal .NET bytecode. However, even though in particular the C# programming languages does have a null-coalescing operator `?.`, this is unfortunately not exposed in the expression sub-language that NMF Synchronizations is using heavily. Also, the put operation in these cases is not clear, because a single in-model lens usually cannot generically know when a model element created on the fly can be removed as there might be other lenses requiring it.

As a workaround, the transformation needs to customize the creation of containers in order to ensure that a container always has the `image` reference set to an element, in order to avoid a possible `NullReferenceException` when accessing the image of the newly created container, depicted in Listing 5.

```
protected override IContainer CreateLeftOutput(IMapEntry input,
      IEnumerable<IContainer> candidates, ISynchronizationContext
      context, out bool existing)
{
  existing = false;
  return new Container { Image = new Image() };
}
```

Listing 5: Overriding the creation of containers

In this listing, we make use of the fact that the transformation is always running as an online transformation, meaning that changes to the either model are always performed immediately (no idle updates). Idle updates in this solution mean that the models are just not propagated from NMF to the Eclipse tests.

Another problem with respect to the benchmark framework is that NMF 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, yet. 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. Therefore, all tests that require an exact order are going to fail.

It is generally quite hard to work with the benchmark framework as the target platform for Eclipse seems rather unstable and Eclipse often fails to run the unit tests until one restarts the system.

5 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] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 3, Article 17 (May 2007). https://doi.org/10.1145/1232420.1232424

[2] Georg Hinkel. 2015. Change Propagation in an Internal Model Transformation Language. In *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*, Dimitris Kolovos and Manuel Wimmer (Eds.). Springer International Publishing, Cham, 3–17. https://doi.org/10.1007/978-3-319-21155-8_1

[3] Georg Hinkel and Erik Burger. 2019. Change propagation and bidirectionality in internal transformation DSLs. *Softw. Syst. Model.* 18, 1 (2019), 249–278. https://doi.org/10.1007/s10270-017-0617-6

[4] Georg Hinkel, Thomas Goldschmidt, Erik Burger, and Ralf Reussner. 2017. Using Internal Domain-Specific Languages to Inherit Tool Support and Modularity for Model Transformations. *Software & Systems Modeling* (2017), 1–27. https://doi.org/10.1007/s10270-017-0578-9

[5] Georg Hinkel, Robert Heinrich, and Ralf Reussner. 2019. An extensible approach to implicit incremental model analyses. *Software & Systems Modeling* (29 Jan 2019). https://doi.org/10.1007/s10270-019-00719-y