An NMF Solution to the TTC2021 Incremental Recompilation of Laboratory Workflows Case

Georg Hinkel
georg.hinkel@tecan.com
Tecan Software Competence Center GmbH
Wiesbaden, Germany

ABSTRACT

This paper presents a solution to the Incremental Recompilation Laboratory Workflows Case at the TTC 2021 using the .NET Modeling Framework (NMF). This solution is able to derive an incremental change propagation almost 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

The transformation of high-level process models to low-level jobs actually executed on machines is a common problem not only in laboratory automation but also in other domains such as smart production. In these domains, it is desirable to adapt an executed process in case of errors or at least avoid wasting resources if it is clear that the complete workflow cannot be performed completely. Because there are typically a lot of things that could go wrong, it is desirable to design a transformation system in such a way that an incremental change propagation can be inferred, i.e. does not have to be specified by the developer.

To assess to what degree current model transformation tools are able to infer an incremental change propagation in such scenarios, the Transformation Tool Contest[1] 2021 hosts a case for incremental recompilation of laboratory automation workflows. This paper presents a solution to this case using the .NET Modeling Framework (NMF, [3]).

NMF is a framework built for support of model-driven engineering, incremental model analyses and incremental model transformations. In particular, NMF Expressions [6] is an incrementalization system able to incrementalize arbitrary function expressions and NMF Synchronizations [2, 4] is an incremental model transformation approach. Using both tools in combination, it is possible to solve the incremental laboratory workflows case in a very declarative manner such that the required change propagations can be derived mostly implicitly.

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 evaluates the solution against the reference solution.

[1]https://www.transformation-tool-contest.eu

2 NMF EXPRESSIONS AND NMF SYNCHRONIZATIONS

NMF Expressions [6] 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[2]).

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 [4]. 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 $\omega \in \Omega$:

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

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 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

[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.

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# [2, 4]. 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].

3 SOLUTION

3.1 Assignments of plates, columns and wells

As a first step, we need to group the samples to process into plates and columns that can be pipetted at the same time. This is done using the `Chunk` operation recently built into NMF. This comes in two versions, `Chunk` and `ChunkIndexed` where the latter also keeps the original index in the original collection. The code for calculating the assignments of samples to plates, columns and tubes is depicted in Listing 1. This listing shows how to register a collection of samples with a synchronization context. That is, because the assignment of plates is needed in many places throughout the synchronization, we put it as context.

The input type `IEnumerable<ISample>` used in line 1 of Listing 1 denotes an incrementalizable collection of samples. NMF essentially implements the Standard Query Operators of C# and a few more operators on top of this interface in order to derive an incremental change propagation for a given query. That is, the system allows developers to obtain incremental updates of the results upon changes of the input models, such as adding a sample.

Lines 3–5 in Listing 1 calculate columns as chunks of samples. These columns are then chunked into microplates. Line 6 forces the incrementalization of this collection. The idea of this order as opposed to chunking the samples into plates and then further into columns is to allow NMF to rebalance samples between columns and then rebalance columns between plates. However, we did not specify a balancing strategy and thus, NMF will not try to rebalance the chunks. Similar, lines 8–9 calculate the collection of tube runners from different chunks of the input samples.

```
1  public void InitializeContext(IEnumerableExpression<ISample>
       samples, ISynchronizationContext context) {
2    context.Data.Add( _platesKey, samples
3      .ChunkIndexed( 8, ( samples, column ) => new ProcessColumn(
           column,
4                    samples.Select( tuple => new ProcessWell( tuple
                         .Item2 % 96, tuple.Item1 ) ) ) )
5      .Chunk( 12, ( columns, plateIndex ) => new ProcessPlate( $"
           Plate{plateIndex+1:00}", columns ) )
6      .AsNotifiable() );
7    context.Data.Add( _tubesKey, samples
8      .ChunkIndexed( 16, ( samples, tubeIndex ) => new Tubes( $"Tube
           {tubeIndex+1:00}",
9                    samples.Select( tuple => new ProcessWell( tuple
                         .Item2 % 16, tuple.Item1 ) ) ) )
10     .AsNotifiable());
11 }
```

Listing 1: Setting up the mapping of samples to plates and wells

```
1  public class JobRequestToJobCollection : SynchronizationRule<
       IJobRequest, IJobCollection> {
2    public override void DeclareSynchronization() {
3      SynchronizeManyLeftToRightOnly( SyncRule<ReagentToTrough>(),
4        request => request.Assay.Reagents, jobCollection =>
             jobCollection.Labware.OfType<ILabware, Trough>() );
5      SynchronizeManyLeftToRightOnly( SyncRule<
           ProcessPlateToMicroplate>(),
6        (request, context) => GetPlates(context),
7        (jobCollection,_) => jobCollection.Labware.OfType<ILabware,
             Microplate>() );
8      SynchronizeManyLeftToRightOnly( SyncRule<SamplesToTubeRunner
           >(),
9        (request, context) => GetTubes(context),
10       (jobCollection,_) => jobCollection.Labware.OfType<ILabware,
             TubeRunner>() );
11     SynchronizeManyLeftToRightOnly( SyncRule<
           ProtocolStepToJobsRule>(),
12       (request, _) => request.Assay.Steps,
13       (jobCollection, context) => new CollectionOfJobCollections(
             jobCollection, context ) );
14   }
15 }
```

Listing 2: The entry point synchronization rule

3.2 The model synchronization

The actual model synchronization is split into several synchronization blocks that act as isomorphisms. Each synchronization rule defines a list of synchronization blocks that define what data should be synchronized. The entry point synchronization rule, the one synchronizing an overall high-level job request with a low-level job collection, is depicted in Listing 2. In this listing, lines 3–4 denote that the reagents are mapped to troughs and lines 5–10 denote that that tube runners should be created to host the samples as well as microplates for processing. For the tubes and the microplates, we consume a second parameter in the lens to access the plate collections stored in the context (cf. Listing 1).

Lastly in lines 11-13 of Listing 2, we define that the steps of the requested assay should be synchronized with the job collections in the low-level model. For this, we use a custom collection implementation that essentially groups the low-level jobs of the resulting job collection by name. This needs access to the transformation context as we will store information such as the affected samples of a job in there. The four calls to `SynchronizeManyLeftToRightOnly`

```
1  public class AddReagentToJobsRule : SynchronizationRule<AddReagent
       , JobsOfProtocolStep> {
2    public override void DeclareSynchronization() {
3      MarkInstantiatingFor( SyncRule<ProtocolStepToJobsRule>() );
4      SynchronizeManyLeftToRightOnly(
5         SyncRule<AddReagentLiquidTransferToLiquidTransfer>(),
6         ( step, context ) => GetPlates( context )
7           .SelectMany(p => p.Columns, (plate, column) => new
                AddReagentLiquidTransfer(column, plate, step))
8           .Where(transfer => transfer.Column.AnyValidSample.Value),
9         ( jobsOfStep, _ ) => jobsOfStep.Jobs.OfType<IJob,
              LiquidTransferJob>() );
10   }
11 }
```

Listing 3: Synchronizing the jobs for an AddReagent protocol step

```
private static ObservingFunc<ProcessColumn, bool>
    _anyNonErrorSample = new ObservingFunc<ProcessColumn, bool>(
    c => c.AllSamples.Any( s => s.State != SampleState.Error ) );
...
AnyValidSample = _anyNonErrorSample.Observe( this );
```

Listing 4: Calculating whether a column has any sample that is not in the error state.

basically define collection-valued unidirectional synchronization blocks that are only enforced from the left to the right.

3.3 Synchronization of AddReagent

The actual high-level process steps are translated using separate synchronization rules. That is, we synchronize a protocol step with the jobs implementing this protocol step. The approach to transform the other types of high-level jobs is conceptually similar, although the different complexity of the job types leads to a different complexity of the synchronization rules required. For AddReagent, the synchronization rule for this synchronization is depicted in Listing 3.

In this listing, line 3 marks the synchronization rule as instantiating for `ProtocolStepToJobsRule`, which means that the synchronization rule is used when the `ProtocolStepToJobsRule` is executed with an AddReagent protocol step. Lines 4–9 denote the synchronization block that computes the elements from which to create the jobs, using a dedicated class to represent the request for a liquid transfer. The query calculates all columns of all plates that have at least any valid (i.e., not failed) sample.

Because the latter needs to be calculated incrementally for each `ProcessColumn`, the calculation (and its incrementalization) is separated into a static function (see Listing 4).

The reason to separate the logic into an `ObservingFunc` instance here is that the incrementalization of a method in NMF involves some reflection and takes a bit of time while applying it to a particular element is rather cheap. Using a static instance essentially caches the incrementalization and applies it to multiple instances. This is also the reason that, although supported by NMF, nested queries are currently rather slow and hence we refrain from using the C# query syntax in the mappings such as Listing 3.

The child synchronization rule `AddReagentLiquidTransferToLiquidTransfer` then defines how the instances of this intermediate class are transformed into a low-level job as depicted in Listing 5. Line 3 defines that the source of the liquid transfer should be

```
1  public class AddReagentLiquidTransferToLiquidTransfer :
       SynchronizationRule<AddReagentLiquidTransfer,
       LiquidTransferJob> {
2    public override void DeclareSynchronization() {
3      SynchronizeLeftToRightOnly( SyncRule<ReagentToTrough>(), step
           => step.AddReagent.Reagent, liquidTransfer =>
           liquidTransfer.Source as Trough );
4      SynchronizeLeftToRightOnly( SyncRule<ProcessPlateToMicroplate
           >(), step => step.Plate, liquidTransfer => liquidTransfer
           .Target as Microplate );
5      SynchronizeManyLeftToRightOnly( SyncRule<
           AddReagentTipToTipTransfer>(),
6        step => step.Column.Samples
7          .Where( s => s.Sample.State != SampleState.Error )
8          .Select( s => new AddReagentTip( step, s ) ),
9        liquidTransfer => new TipCollection( liquidTransfer.Tips ) )
           ;
10     SynchronizeManyLeftToRightOnly(
11       ( step, _ ) => step.Column.AllSamples,
12       ( liquidTransfer, context ) => GetAffectedSamples( context,
             liquidTransfer ) );
13   }
14 }
```

Listing 5: The synchronization of add reagent elements to actual LiquidTransferJob elements.

```
1  public class AddReagentTipToTipTransfer : SynchronizationRule<
       AddReagentTip, ITipLiquidTransfer> {
2    public override void DeclareSynchronization() {
3      SynchronizeLeftToRightOnly( well => well.AddReagent.Volume,
           transfer => transfer.Volume );
4      SynchronizeLeftToRightOnly( well => well.TargetWell.Well,
           transfer => transfer.TargetCavityIndex );
5      SynchronizeRightToLeftOnly( well => IsSampleFailed( well.
           TargetWell.Sample ), transfer => transfer.Status ==
           JobStatus.Failed );
6    }
7  }
```

Listing 6: Synchronization rule `AddReagentTipToTipTransfer`

synchronized with the trough created for the reagent. Line 4 specifies that the reagent should be pipetted into the microplate created for the processing requst. In lines 5–9, the synchronization block denotes the which tips exactly need to be created. Here, we again use an intermediate class and a custom collection in line 9 in order to control that a tip liquid transfer is only removed when it is still planned. Lines 10–12 specify that the samples created for this liquid transfer are stored inside the transformation context.

The synchronization rule `AddReagentTipToTipTransfer` that specifies the transformation of the tip liquid transfers is depicted in Listing 6. Lines 3–4 synchronize the volumes and the target cavity (the source cavity is always 0 for a trough).

The last synchronization block in line 5 denotes that the information whether the status of the tip transfer is failed should be synchronized back to the high-level job request model.

3.4 Synchronization of DistributeSample

The synchronization of DistributeSample elements works exactly like the synchronization of AddReagent with one important exception: While the source labware of an AddReagent is accessible easily via the transformation trace from the reagent, this is unfortunately not as easy for DistributeSample.

As a reason, the current design of the solution has no direct connection between a column of a processing microplate and the tube runner that holds the samples. We first created an approach

```
1  public abstract class MicroplateProtocolStepRule<TProtocol,
       TJobRule, TJob> : SynchronizationRule<TProtocol,
       JobsOfProtocolStep>
2    where TProtocol : IProtocolStep
3    where TJob : class, IJob
4    where TJobRule : MicroplateJobRule<TProtocol, TJob>
5  {
6    public override void DeclareSynchronization() {
7      MarkInstantiatingFor( SyncRule<ProtocolStepToJobsRule>() );
8      SynchronizeManyLeftToRightOnly(
9        SyncRule<TJobRule>(),
10       ( step, context ) => GetPlates( context )
11         .Where( plate => plate.AnyValidSample.Value )
12         .Select( plate => Tuple.Create( step, plate ) ),
13       ( jobsOfStep, _ ) => jobsOfStep.Jobs.OfType<IJob, TJob>() );
14   }
15 }
16 public abstract class MicroplateJobRule<TProtocol, TJob> :
       SynchronizationRule<Tuple<TProtocol, ProcessPlate>, TJob>
17   where TProtocol : IProtocolStep
18   where TJob : IJob
19 {
20   public override void DeclareSynchronization() {
21     SynchronizeManyLeftToRightOnly(
22       ( step, _ ) => step.Item2.AllSamples,
23       ( job, context ) => GetAffectedSamples( context, job ) );
24     SynchronizeRightToLeftOnly(
25       step => AreAllFailed( step.Item2.AllSamples ),
26       job => job.State == JobStatus.Failed );
27     SynchronizeLeftToRightOnly( SyncRule<ProcessPlateToMicroplate
           >(),
28       tuple => tuple.Item2, MicroplateProperty );
29   }
30   protected abstract Expression<Func<TJob, IMicroplate>>
         MicroplateProperty { get; }
31 }
```

Listing 8: Template for synchronization of microplate processing protocol steps

that would calculate the mapping incrementally, but this turned out to be very resource-intensive both in terms of time and memory.

```
private static Tubes GetSourceTube( ITransformationContext context
    , ProcessColumn column ) {
  return GetTubes( context )
    .AsEnumerable()
    .FirstOrDefault( t => t.Samples
      .AsEnumerable()
      .Any( s => column.Samples
        .AsEnumerable()
        .Any( s2 => s.Sample == s2.Sample ) ) );
}
```

Listing 7: Calculating the rube runner for a given column of a processing plate

The solution now is to break out of the incrementalization monad explicitly and calculate the source tube runner one-time as depicted in Listing 7: We explicitly call the `AsEnumerable` method here in order to instruct the compiler to actually compile the lambda expressions used to calculate the tube runner. This, however, breaks the support of rebalancing the chunks making up the columns and plates.

3.5 Synchronization of Wash and Incubate

The synchronization of Wash steps and Incubate steps is very similar, because both steps (as many in lab automation) operate on entire microplates. That is, the protocol step needs to be instantiated for each microplate that is used for sample processing.

The synchronization rule templates for protocol steps operating on a single microplate is depicted in Listing 8. There are two rule templates, one for synchronizing a protocol step with a collection of low-level jobs, the other for actually synchronizing the protocol step in for a given microplate into a given job. The `MicroplateProtocolStepRule` class already organizes the registration of the rule as instantiating and the calls to the child rule. The template for the latter, `MicroplateJobRule`, registers affected samples, sets the samples to failed (using another lens called `AreAllFailed` in line 25) and synchronizes the target microplate. Because the target metamodel does not use a shared base class for jobs operating on microplates, the rule template uses an abstract property such that instance rules have to specify the property used to store the microplate.

```
1  public class WashToJobsRule : MicroplateProtocolStepRule<Wash,
       WashToWashJob, WashJob> {}
2
3  public class WashToWashJob : MicroplateJobRule<Wash, WashJob> {
4    protected override Expression<Func<WashJob, IMicroplate>>
         MicroplateProperty => wash => wash.Microplate;
5    public override void DeclareSynchronization() {
6      base.DeclareSynchronization();
7      SynchronizeManyLeftToRightOnly(
8        tuple => tuple.Item2.Columns.SelectMany( c => c.Samples.
             Where( s => s.Sample.State != SampleState.Error ).
             Select( s => s.Well ) ),
9        wash => wash.Cavities );
10   }
11 }
```

Listing 9: Synchronization of Wash steps

The instantiation of the rule templates for Wash elements is depicted in Listing 9. Since the rule to synchronize Wash protocol steps is sufficiently described using the synchronization template, we do not need to provide any further specification other than the type parameters to be used, including a reference to the child rule. Unfortunately, the C# compiler is not (yet?) able to infer the type parameters `TProtocol` and `TJob`, so they must be specified explicitly.

For the synchronization of a Wash in conjunction with a specific processing plate, we need to specify the property holding the microplate (in line 4) and take care of the speciality of the Wash that it holds a reference to the cavities that should be washed. For this, we need to override the declaration of the synchronization rule. Because we do want to inherit the declaration of the template, we need to call the base declaration in line 6. Then, we add the synchronization of the cavities in lines 7–9.

The synchronization of Incubate protocol steps works in the same way, except that the child rule extends the template with synchronization blocks for temperature and duration.

3.6 Synchronization of the `Next` reference

In order for the scheduler to be able to actually schedule the low-level jobs, the base class for jobs keeps a reference to the next and previous jobs. That is, the scheduler may only schedule a job if all previous jobs are completed and in the opposite direction, the job is a prerequisite for all next jobs.

To aid this situation, we use a utility class called `CollectionBinding` that essentially enforces the synchronization of elements between an incrementalizable source collection (typically a query) and a target collection that should be adapted. The implementation is depicted in Listing 10. The query calculates the jobs for which the set of affected samples intersects the affected samples of the current

```
CollectionBinding.Create(
  _nextJobs.Jobs.Where( j => ProtocolSynchronization.
       GetAffectedSamples( _context, j ).Intersect( samples ).Any
       () ),
  item.Next )
```

Listing 10: Binding the next low-level jobs to the jobs of the next job collection that affect the same samples

job. The return value is an instance of the `IDisposable` interface, the typical interface in .NET to dispose objects. In this case, the binding is stopped when disposed. Because NMF supports bidirectional references, only one direction of the the association has to be set manually, the other is set automatically by NMF.

Unfortunately, the management of the collection binding currently has to be done by handling change events of the jobs created for a job collection manually.

3.7 Starting the synchronization

To run the solution, we create a new context for the model synchronization, initialize the samples and start the model synchronization in the direction *LeftToRight* with change propagation in both directions.

```
_context = new SynchronizationContext( _synchronization,
    SynchronizationDirection.LeftToRight,
    ChangePropagationMode.TwoWay );
_synchronization.InitializeContext( _jobRequest.Samples, _context
    );
_synchronization.Synchronize( ref _jobRequest, ref _jobCollection,
      _context );
```

Listing 11: Starting the model synchronization

4 EVALUATION

We see the strongest point of the presented solution in fact that the change propagation can be inherited mostly from a declarative specification. As a consequence, essentially all types of changes are supported, not just the change types executed by the benchmark framework. This means, that new types of error handling do not necessarily have an effect on the transformation but are supported by default. The declarative specification, however, keeps the understandability of the solution at a good level. The attendees of the TTC will judge on the understandability in relation to the reference solution.

In addition to the inherited change propagation, the solution also means that no changes to the metamodel code are necessary and the model representation can be reused independently of the transformation.

Before we describe the results in terms of performance, remind that the reference solution is a solution tailored manually and explicitly for the given types of changes, without any incrementalization system or alike. Therefore, it is hard to beat it in terms of performance and the strengths of the solution in this paper are rather in the declarativeness and in fact that any type of change is supported.

To evaluate the solution in terms of performance, we have run the benchmark on a system equipped with a Intel Core i7-8850H CPU clocked at 2.6Ghz and 32GB RAM, running Windows 10. The results are discussed in the remainder of this section.

4.1 Scaling Samples

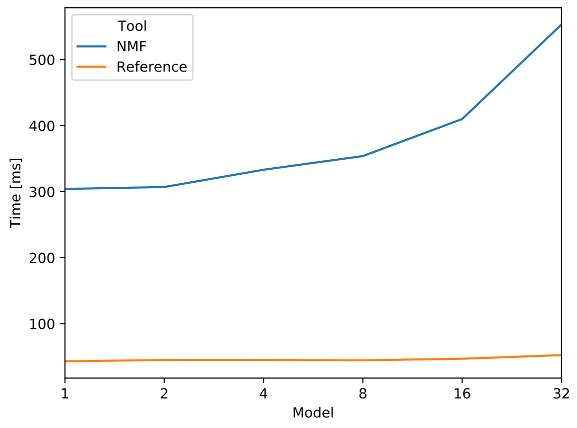

Figure 2: Time for the initial transformation in the scale samples scenario

The results in terms of time to execute the initial transformation for the scaling samples scenario are depicted in Figure 2. In this scenario, the different models represent loads of 8 samples (size 1) to 256 samples (size 32), applied to a simple ELISA assay model. The results show that whereas the time for the reference solution is essentially constant at around 50ms, the initial time for the NMF solution grows worse than linear, it is more like quadratic.

We did profile the NMF solution. The results show that much of the time is lost because NMF Synchronizations executes the incrementalization of the queries used to specify the synchronizations over and over again instead of reusing it. Further, the collection binding depicted in Listing 10 also requires the system to be incrementalized over and over again. We expect that the performance gap could be reduced, if the frameworks can be adapted to cache the incrementalization properly.

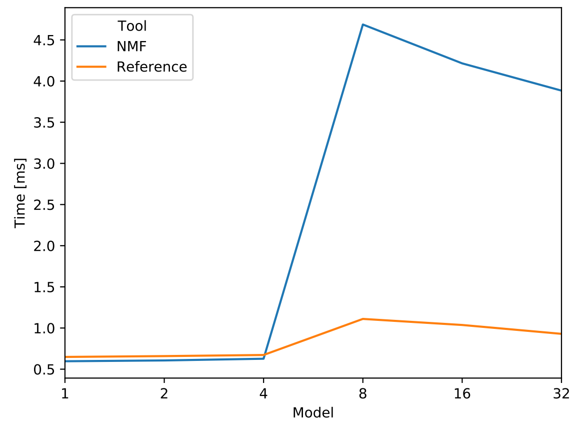

Figure 3: Results for the average time for an update in the scale samples scenario

The results for propagating the state of changes of low-level elements including both what this means in terms of changes to sample states and also what low-levels become obsolete because of this are depicted in Figure 3. Meanwhile again, the NMF solution is slower, it is still within few milliseconds even for the largest models considered.

4.2 Scaling the Assay

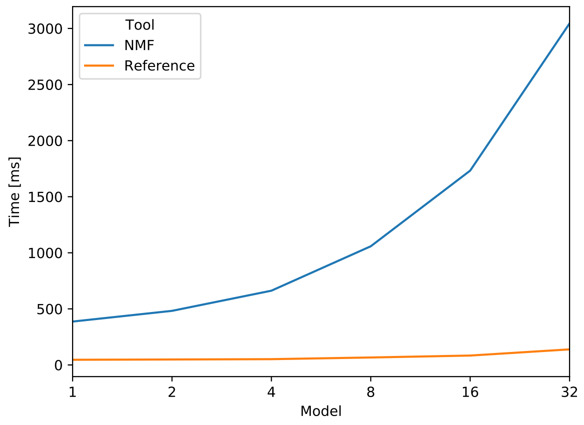

Figure 4: Time for the initial transformation in the scale assay scenario

In the scale assay scenario, all model sizes use 96 samples, but the number of protocol steps varies: Whereas the smallest model (size 1) uses a simplified ELISA assay model with 8 steps, the largest model uses 32 repetitions (256 protocol steps in total). The execution time for the initial transformation is depicted in Figure 4. Again, the runtime of the NMF solution grows quadratic with the size of the model meanwhile the reference solution stays fast.

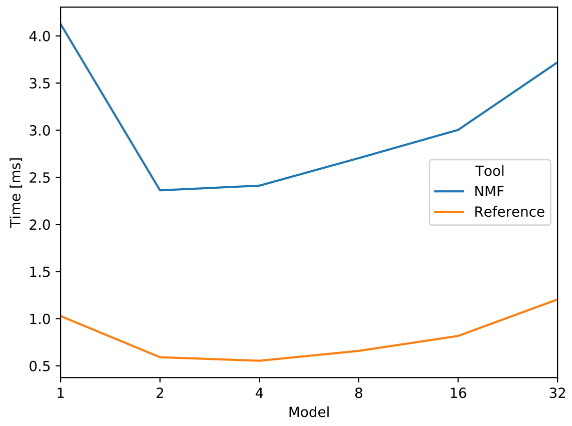

Figure 5: Results for the average time for an update in the scale assay scenario

The results for updates are depicted in Figure 5. The results look awkward because as it appears, the change propagation takes longer for smaller models up to some point. This is because the benchmark framework uses Pythons `subprocess.POpen` to spawn the processes for the model sizes and they can make use of JIT optimizations of earlier runs. Again, the propagation of the changes happens in a few milliseconds, both for NMF and the reference solution, the reference solution being slightly faster.

4.3 New samples

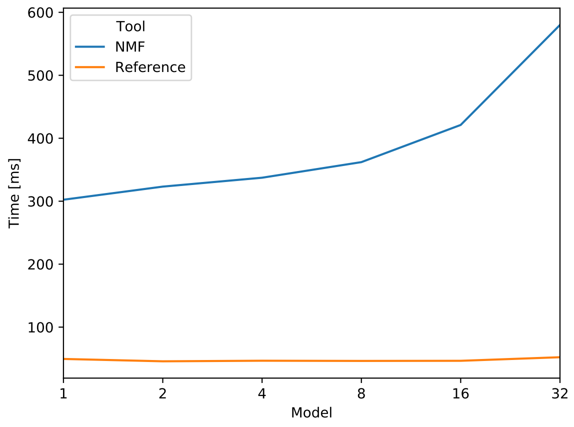

Figure 6: Time for the initial transformation in the new samples scenario

The results for the initial transformation in the new samples scenario are depicted in Figure 6. Not very surprising, they are very similar to the scaling samples case because the parameters for the initial model are exactly the same.

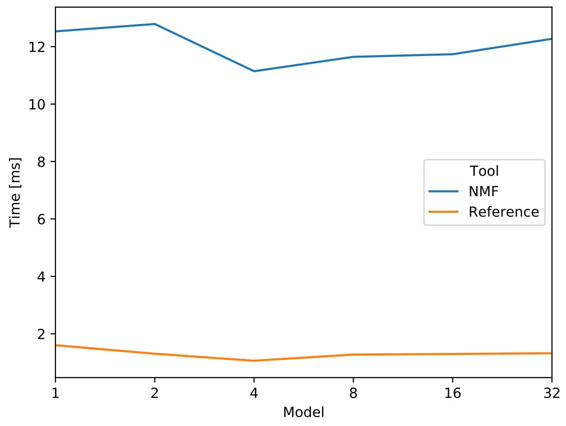

Figure 7: Results for the average time for an update in the new samples scenario

The difference to the scenario that plainly scales the number of samples is that new samples are introduced during the runtime of the benchmark. The results for propagating these changes are depicted in Figure 7. Still, the changes are propagated within a few milliseconds, but this time this is much slower than in the other two scenarios, in line with the performance issues in the initial transformation.

5 CONCLUSION

The solution has shown that in fact, it is possible to derive an incremental change propagation for most of the transformation. Only for smaller parts such as the synchronization of the next reference, manual code is necessary, although rather limited. The evaluation shows that meanwhile the performance for actually making changes almost keeps up with the reference solution, meanwhile supporting much more types of changes. However, the solution also shows a performance problem caused by the current inability of NMF to cache the query incrementalization properly.

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. 2018. NMF: A Multi-platform Modeling Framework. In *Theory and Practice of Model Transformation*, Arend Rensink and Jesús Sánchez Cuadrado (Eds.). Springer International Publishing, Cham, 184–194.

[4] 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

[5] 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

[6] 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