Incremental execution of relational transformation specifications in YAMTL: a case with laboratory workflows

Artur Boronat[1]

[1]*School of Computing and Mathematical Sciences, University of Leicester, University Rd, Leicester, LE1 7RH, UK*

Abstract

In this paper, we present the YAMTL solution to the Laboratory Workflows case of TTC 2021. This solution illustrates how to specify a consistency relation between two metamodels that may map one object of the input model to several objects of the output model using a declarative style. In addition, the solution makes use of generated boilerplate code and rule inheritance for the sake of conciseness. The initial experiments show that YAMTL introduces little overhead over the reference solution, implemented in plain code on the .NET Framework, and yet it addresses its main problems: change propagation is encoded using declarative rules and traceability is handled implicitly by YAMTL.

Keywords

Incremental model-to-model transformation, EMF.

1. Introduction

The TTC 2021 case on incremental recompilation of laboratory workflows [1] aims at investigating solutions for a key logistic issue at the start of the Covid-19 pandemic - availability of test capacity - from a model-driven engineering perspective. The problem consists in providing an automated approach to manage the evolution of laboratory workflows involving robotic liquid handlers (RLHs), e.g. to repurpose existing laboratory workflows in order to cope with a sudden demand of Covid-19 testing. To facilitate this task declarative high-level processess, specifying the configuration of a laboratory workflow, are to be mapped to low-level jobs that are executable by the laboratory RLHs. Whenever there is an error in a RLH low-level job (e.g. due a hardware problem), the corresponding high-level process change needs to be identified and this change needs to be propagated to the low-level job. The challenge is that such low-level jobs may not be able to start from scratch to fix the error, and the current state of the low-level job (e.g. current distribution of samples processed) needs to be maintained.

High-level models are defined in terms of `JobRequests` that use several `Samples` and that have `Assays`, each of which represents a protocol where there are four different `ProtocolSteps`: distributing sample, adding a reagent, washing or incubating. Protocol steps have dependencies to previous and next steps.

Low-level models capture the execution of the workflow on a RLH. A process is represented with a `JobCollection`, where each `Job` could refer to `LiquidTransfer`, `Wash` or `Incubate`. A `LiquidTransfer` manages a microplate, either by distributing either samples from tubes or reagents from troughs into microplate cavities. Each job `LiquidTransfer` has up to eight `LiquidTransferTips` in order to transfer samples or reagents at the same time.

Specific logic on the behaviour of a job `LiquidTransfer` and the mapping is documented in the case [1]. Solutions for specifying and managing such mapping are required: to be highly performant during change propagation, to be understandable in order to enhance maintenability, to keep some form of memory of the state of low-level jobs, and to have low or no specification overhead for making the mapping from high-level process to low-level jobs incremental with respect to changes in high-level processes.

The structure of the paper is as follows: section 2 provides a brief introduction to the YAMTL language; section 3 describes an outline of the YAMTL solution; section 4 presents the main challenges faced; section 5 presents the transformation rules used in the YAMTL solution; and section 6 discusses the evaluation of the solution with the benchmark criteria.

2. YAMTL outline

YAMTL [2] is a model transformation language for EMF models, with support for incremental execution [3], designed as an internal DSL of Xtend.

A YAMTL model transformation is defined as a module, a class specializing the class `YAMTLModule`, containing the declaration of transformation rules. Each rule has an in-

TTC'21: Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 25 June 2021, ~~Bergen, Norway~~ (online).

artur.boronat@leicester.ac.uk (A. Boronat)
https://arturboronat.info (A. Boronat)
0000-0003-2024-1736 (A. Boronat)

put pattern for matching variables and an output pattern for creating objects. An input pattern consists of **in** elements together with a global rule filter condition, which is true if not specified. Each of the **in** elements is declared with a variable name, a type and a local filter condition, which is true if not specified. An output pattern consists of **out** elements, each of which is declared with a variable name, a type and an action block. Filter conditions and action blocks are specified as non-pure lambda expressions in Xtend[1]. In such expressions, the YAMTL language uses the expression `'variable'.fetch as Type` to fetch the value of a `'variable'` from the execution environment and for obtaining referenes to objects created by other rules. This is normally used to access matched objects in a filter expressions and in output expressions.

When applying a YAMTL transformation to an input model, the pattern matcher finds all rule matches that satisfy local filter conditions. When a total match is found, the satisfaction of that match is finally asserted by the rule filter condition. Once all matches are found, the transformation engine computes an output match for each input match using the expressions in the **out** action blocks of the corresponding rule.

YAMTL also supports multiple rule inheritance. When a rule specializes another one, its execution semantics is altered as follows: in matched input elements, filter expressions are inherited using a leftmost top-down evaluation strategy w.r.t. the inheritance hierarchy; in output elements, action expressions are also inherited following a leftmost topdown evaluation strategy w.r.t. the inheritance hierarchy by default, and they can be can be overriden in descendant rule.

The YAMTL engine has been extended with an incremental execution mode, which consists of two phases: the *initial phase*, the transformation is executed in batch mode but, additionally, tracks feature calls in objects of the source model involved in transformation steps as *dependencies*; and the *propagation phase*, the transformation is executed incrementally for a given source update and only those transformation steps affected by the update are (re-)executed. This means that components of YAMTL's execution model have been extended but the syntax used to define model transformations is preserved. Hence, a YAMTL batch model transformation can be executed in incremental mode, without any additional user specification overhead.

3. Solution outline

The mapping in the case study is specified by providing EMF metamodels for the high-level and low-level process languages (available in the case description [1]), and by defining mappings as rules. When applying a rule, performing a transformation step, YAMTL implicitly mantains a traceability model that remembers which rules were applied to which parts of source models. When a change is applied to the high-level model, YAMTL automatically re-evaluates the affected transformation steps, if needed.

From an outline perspective, the model transformation is performed as follows: initially labware is initialized by setting tube runners with samples, troughs with reagens and microplates with cavities, and samples are allocated from tube cavities to microplate cavities. In a second phase, for each `ProtocolStep` and `Sample`, a `LiquidTransferTip` is initialized in order to perform the transfer of a sample or of a reagent. In addition, jobs are initialized according to the type of `ProtocolStep`s in the high-level model.

A strength of our solution is that it is fully declarative, without involving explicit invocation of rules or imposing control flow. It is also non-intrusive, as it does not require the modification of the original metamodels or adding external glueing code to execute parts of the transformation.

4. Challenges and YAMTL extensions

In the YAMTL solution, the main challenge was a consequence of the additional information that low-level jobs need to deal with: specific source and target labware for each job. Such additional information means that for each `ProtocolStep` a transformation rule has to create several `Job`s, depending on the nature of the job. That is, the mapping between high-level and low-level models is one-to-many for some elements. Originally, as many other model transformation languages with support for functional model transformations, YAMTL did not support such type of mappings. This case has prompted the development of additional YAMTL extensions in order to enable their specification.

The new features that have been added to YAMTL are as follows:

- Matched rules `toMany` that enable repeated rule application for the same input object subject to a valid termination condition `toManyCap` based on the match count. The count of the present input match, to which the rule is being applied, can be retrieved with the expression `'matchCount'.fetch()`. Whenever a rule `toMany` is involved in a rule inheritance hierarchy, all rules in that hierarchy must be `toMany` too. All variants of the operator `fetch()` have been aug-

[1]For a more detailed description of the YAMTL language, the reader is referred to [2], including programmable execution strategies and multiple inheritance.

mented with an additional parameter, the occurrence of the transformation step from which the target object must be fetched. By default, the value of this parameter is 0, corresponding to the first transformation step that is found for the input object or input match. Hence the `fetch` operator is equipped for working with several matches of a rule `toMany`.

- When the global correctness check, which ensures that a model transformation is a mapping, is disabled, there may be several rules transforming the same input object. This allows YAMTL to represent relational model transformations that are more expressive than mapping model transformations. The scope of the correctness check is at rule level though. A rule cannot be applied more than once to the same input object, unless the rule is declared as `toMany` and has a valid termination condition for the repetition.

- Boilerplate code generation to reduce the amount of code needed to access matched objects and created output objects within filter expressions and output initialization actions. For example, expressions of the type `val in_sample = 'in_sample'.fetch()as Sample` can be skipped. Syntactic helpers use the names in input and output elements, so that matched objects can be accessed in filters by using the name of the corresponding input element, and both matched objects and created output objects can be accessed in output initialization actions using the name of the corresponding input/output elements. Code generation takes into account rule inheritance and appends the name of the type of the corresponding element when there are ambiguities (an element is declared in different rules with different output types). Whenever, the same name has been used for an input/output element in unrelated rules and these have different types, then the name of the rule is appended to resolve ambiguities. Matched objects and built-in helpers (like `matchCount`) are also available as syntactic helpers.

5. Detailed solution

The transformation is available at https://github.com/arturboronat/ttc21incrementalLabWorkflows.

In the following we briefly describe the transformation rules as they appear.

Rule `jobRequest_->_jobCollection` transforms the `JobRequest` into a `JobCollection`.

```
rule('jobRequest_->_jobCollection')
  .in('in_jobRequest', LAB.jobRequest)
  .out('out_jobCollection', JOB.jobCollection)
```

Listing 1: Rule `jobRequest_->_jobCollection`.

Rule `reagent_trough` generates a `Trough` for each input `Reagent`.

```
rule('reagent_->_trough')
  .in('in_reagent', LAB.reagent)
  .out('out_trough', JOB.trough) [
    val in_jobRequest = in_reagent.eContainer.eContainer as
               JobRequest
    val out_jobCollection = in_jobRequest
      .fetch('out_jobCollection',
               'jobRequest_->_jobCollection') as
               JobCollection
    out_trough.name = in_reagent.name
    out_jobCollection.labware.add(out_trough)
  ],
```

Listing 2: Rule `reagent_trough`.

Rule `jobRequest_->_tubeRunner` generates as many `TubeRunners` as required for the input `JobRequest`, according to the expression `jobRequest.samples.size / TUBE_RUNNER_CAPACITY`. The helper `max_count` implements that expression, determining the maximum number of times a rule should be applied by taking into account the capacity of `tubeRunner` and the number of samples in the jobRequest, which need to be distributed in `tubeRunners`.

```
rule('jobRequest_->_tubeRunner').toMany
  .in('jobRequest', LAB.jobRequest)
  .toManyCap[ max_count(jobRequest.samples.size,
               TUBE_RUNNER_CAPACITY) ]
  .out('tubeRunner', JOB.tubeRunner)[
    val out_jobCollection = jobRequest
      .fetch('out_jobCollection',
               'jobRequest_->_jobCollection') as
               JobCollection

    var tubeRunner_list = out_jobCollection.labware.filter[
               it instanceof TubeRunner ]
    tubeRunner.name = String.format('''TubeRunner%02d''',
               tubeRunner_list.size + 1)
    out_jobCollection.labware.add(tubeRunner)
  ],
```

Listing 3: Rule `jobRequest_->_tubeRunner`.

Rule `jobRequest_->_microplate` generates as many `Microplatess` as required for the input `JobRequest`, according to the expression `jobRequest.samples.size / MICROPLATE_CAPACITY`, in a similar way as above.

```
rule('jobRequest_->_microplate').toMany
  .in('jobRequest', LAB.jobRequest)
  .toManyCap[max_count(jobRequest.samples.size,
               MICROPLATE_CAPACITY)]
  .out('microplate', JOB.microplate)[
    val out_jobCollection = jobRequest
```

```
      .fetch('out_jobCollection',
                 'jobRequest_->_jobCollection') as
                 JobCollection
    var microplate_list = out_jobCollection.labware.filter[
                 it instanceof Microplate]
    microplate.name = String.format('''Plate%02d''',
                 microplate_list.size+1)
    out_jobCollection.labware.add(microplate)
  ]
```

Listing 4: Rule `jobRequest_->_microplate`.

Rule `sample_->_allocation` computes the allocation of samples to tube runner and microplate cavities. This rule is used to complete the initialization of tube runners and microplates. The rule is transient and the `JobCollection` that is created is immaterial. Therefore, this rule is only used to perform some additional initialization in other objects. In addition, the rule remembers what sample is stored in each cavity in order to implement the application of backward changes. This rule has an undo action, which updates these backward traces when the sample is no longer allocated.

```
rule('sample_->_allocation').transient
.in('in_sample', LAB.sample).filter [
  in_sample.state == SampleState.WAITING
]
.out('out_aux', JOB.jobCollection)[
  val in_jobRequest =
    in_sample.eContainer as JobRequest
  val tubeRunnerNumber =
    getTubeRunner_number(in_jobRequest, in_sample)
  val tubeRunner = in_jobRequest
    .fetch('tubeRunner', 'jobRequest_->_tubeRunner',
                 tubeRunnerNumber) as TubeRunner
  tubeRunner.barcodes += in_sample.sampleID
  val microplateNumber =
    getMicroplate_number(in_jobRequest, in_sample)
  val microplateCavity =
    getMicroplate_cavity(in_jobRequest, in_sample)
  val microplate =
    in_jobRequest.fetch('microplate',
                 'jobRequest_->_microplate',
                 microplateNumber) as Microplate

  // to facilitate backward propagation, which is external
              to YAMTL
  // track how to retrieve sample from its cavity on a
              microplate
  backward_insert(microplate.name, microplateCavity,
              in_sample)
].undo[
  val in_jobRequest = in_sample.eContainer as JobRequest
  val microplateNumber =
    getMicroplate_number(in_jobRequest, in_sample)
  val microplateCavity =
    getMicroplate_cavity(in_jobRequest, in_sample)
  val microplate = in_jobRequest
    .fetch('microplate', 'jobRequest_->_microplate',
                 microplateNumber) as Microplate
  microplate_cavity_to_sample.get(microplate.name)
    .remove(microplateCavity)
]
```

Listing 5: Rule `sample_->_allocation`.

Rule `tip_creation` creates a `TipLiquidTransfer` for each input sample and adds itself to the corresponding `LiquidTransferJob`. When the sample has failed, the `TipLiquidTransfer` is removed in the undo action.

```
rule('tip_creation')
.in('in_sample', LAB.sample).filter[
  in_sample.state != SampleState.ERROR
]
.in('in_step', LAB.protocolStep).filter[
  (in_step instanceof DistributeSample || in_step
              instanceof AddReagent)
]
.out('out_tip', JOB.tipLiquidTransfer) [
  val step = in_step
  val tip = out_tip
  val in_jobRequest =
    step.eContainer.eContainer as JobRequest
  val occurrence =
    getTipContainerIndex(in_jobRequest, in_sample)
  val out_job =
    step.fetch(occurrence) as LiquidTransferJob

  switch(step) {
    DistributeSample: {
      tip.volume = step.volume
      // SOURCE
      out_job.source =
        in_jobRequest.getTubeRunner(in_sample)
      tip.sourceCavityIndex =
        in_jobRequest.getTubeRunner_cavity(in_sample)
    }
    AddReagent: {
      tip.volume = step.volume
      // SOURCE
      tip.sourceCavityIndex = 0
      val reagent = step.reagent
      val trough = reagent.fetch() as Trough
      out_job.source = trough
    }
  }
  // TARGET
  out_job.target =
    in_jobRequest.getMicroplate(in_sample)
  tip.targetCavityIndex =
    in_jobRequest.getMicroplate_cavity(in_sample)
  // SET CONTAINER
  out_job.tips.add(tip)
].undo[
  // SET CONTAINER
  val in_jobRequest = in_step.eContainer.eContainer as
              JobRequest
  val occurrence = getTipContainerIndex(in_jobRequest,
              in_sample)
  val out_job = in_step.fetch(occurrence) as
              LiquidTransferJob
  out_job.tips.remove(out_tip)
]
```

Listing 6: Rule 'tip_creation'.

Rule `job` is an abstract rule that declares how to create a job, adding it to the output job collection. This rule is `toMany` and all its child rules are so too.

```
rule('job').isAbstract.toMany
  .in('in_step', LAB.protocolStep)
  .out('out_job', JOB.job) [
    out_job.protocolStepName = in_step.id
    // set container
    val in_jobRequest = in_step.eContainer.eContainer as
            JobRequest
    val out_jobCollection = in_jobRequest
      .fetch('out_jobCollection',
            'jobRequest_->_jobCollection') as
            JobCollection
    out_jobCollection.jobs.add(out_job)

    if (in_step.previous !== null)
      out_job.previous.add(in_step.previous.fetch() as Job)
  ]
```

Listing 7: Rule 'job'.

Rule `tipContainer` is an abstract rule that computes the number of times the rule needs to be applied for creating liquid transfer jobs. Rule `distributeSample` and `addReagent` cast down the input and output pattern elements to concrete types.

```
rule('tipContainer').isAbstract.toMany
  .inheritsFrom(#['job'])
  .in('in_step', LAB.protocolStep).filter[
    (in_step instanceof DistributeSample ||
    in_step instanceof AddReagent)
  ]
  .out('out_job', JOB.job),

rule('distributeSample').toMany
  .inheritsFrom(#['tipContainer'])
  .in('in_step', LAB.distributeSample)
  .toManyCap[max_count(sampleCount,TIP_CAVITIES)]
  .out('out_job', JOB.liquidTransferJob),

rule('addReagent').toMany
  .inheritsFrom(#['tipContainer'])
  .in('in_step', LAB.addReagent)
  .toManyCap[max_count(sampleCount,TIP_CAVITIES)]
  .out('out_job', JOB.liquidTransferJob)
```

Listing 8: Rules 'tipContainer'.

Rule `plateJobs` is an abstract rule that computes the number of times the rule needs to be applied for creating jobs that work with a whole microplate (wash and incubate). Rule `wash` and `incubate` cast down the input and output pattern elements to concrete types and complete the initialization of output objects.

```
rule('plateJobs').isAbstract.toMany
  .inheritsFrom(#['job'])
  .in('in_step', LAB.protocolStep).filter[
    (in_step instanceof Wash ||
    in_step instanceof Incubate)
  ]
  .out('out_job', JOB.job),

rule('wash').toMany
  .inheritsFrom(#['plateJobs'])
  .in('in_step', LAB.wash)
  .toManyCap[max_count(sampleCount, MICROPLATE_CAPACITY)]
  .out('out_job', JOB.washJob) [
    val out_job = out_job_WashJob // set to vble to avoid
              fetching several times
    val microplate = getMicroplateFromMatchCount(in_step,
              matchCount)
    out_job.microplate = microplate
    val start = MICROPLATE_CAPACITY * matchCount
    val end = sampleCount - 1
    for (i: start..end)
      out_job.cavities += i % MICROPLATE_CAPACITY
  ],
rule('incubate').toMany
  .inheritsFrom(#['plateJobs'])
  .in('in_step', LAB.incubate)
  .toManyCap[max_count(sampleCount, MICROPLATE_CAPACITY)]
  .out('out_job', JOB.incubateJob) [
    val in_step = in_step_Incubate
    val out_job = out_job_IncubateJob
    // incubate
    out_job.temperature = in_step.temperature
    out_job.duration = in_step.duration
    val microplate = getMicroplateFromMatchCount(in_step,
              matchCount)
    out_job.microplate = microplate
  ]
```

Listing 9: Rules 'plateJobs'.

6. Evaluation

All in all, YAMTL extensions allowed a declarative specification of the mapping from high-level protocols to low-level jobs, involving one-to-many mappings. The implementation of the solution was not intrusive, as the metamodels provided did not need to be modified. In the following we will consider the evaluation criteria of the article:

Understandability. The new features of YAMTL helped in specifying the transformation in a declarative way, managing traceability implicitly. Such type of specification can be challenging for model transformation languages with strong correctness criteria, where matches need to be unique: e.g. for ATL. This approach for defining the transformation should help in defining more complex behaviour in the liquid transfer robot, where additional cases are defined with additional rules. Furthermore, such additional cases can be treated using specialized rules that reuse behaviour from existing ones.

Conciseness. The transformation rules for generating jobs reuse logic, both for matching and for initializing the resulting objects, via rule inheritance. In this case, rule inheritance helped reduce the amount of code needed for each type of job. On the other hand, boilerplate code, generated by YAMTL, has helped reduce the amount of code required for fetching values from the execution environment from within filter conditions and output initialization actions.

Overhead specification. The incremental execution of a YAMTL model transformation does not require additional code. However, incrementality is only achieved when executing the mapping from the high-level model to the low-level model. Backward propagation requires explicit coding, as in the rule `sample_->_allocation`, which remembers the allocation in an auxiliary map, external to YAMTL, and the explicit `undo` action is required to update this external data structure.

Number of elements in the low-level model. The solution remembers the state of the samples in specific cavities thanks to the internal traceability model in YAMTL. When applying changes, only changes corresponding to samples that failed (that is, whose state are error) and new samples were applied. However, the consistency relation between high-level and low-level models is a bit brittle as it relates specific positions within labware, implemented as list indexes. The EMF null constraint does not allow to unset a specific list position, hampering the reuse of microplate cavities.

[2]The evaluation has focussed on the stages involving model transformations, engine initialization and model loading have not been considered as there were no relevant challenges in those stages.

Execution time. After two runs on a MacBookPro11,5 "Core i7" 2.5 GHz with 16 GB RAM, the median run times (in milliseconds) both of the initial model transformation and of the updates are shown in Figure 1[2]. The solutions available in the main repository were used to compare our solution. In the preliminary results discussed at the workshop, YAMTL adds a little overhead over the reference solution and it shows a very reasonably performance for both for the initial transformation and for propagating updates forward. However, a more thorough inspection on how changes are considered for the different tools is required in order to be able to infer reliable conclusions.

References

[1] G. Hinkel, Incremental recompilation of laboratory workflows, in: Proceedings of the 14th Transformation Tool Contest (TTC@STAF), CEUR Workshop Proceedings, 2021. (To Appear).

[2] A. Boronat, Expressive and efficient model transformation with an internal dsl of xtend, in: Proceedings of the 21th ACM/IEEE International Conference on MoDELS, ACM, 2018, pp. 78–88.

[3] A. Boronat, Incremental execution of rule-based model transformation, International Journal on Software Tools for Technology Transfer 1433-2787 (2020). URL: https://doi.org/10.1007/s10009-020-00583-y. doi:10.1007/s10009-020-00583-y.

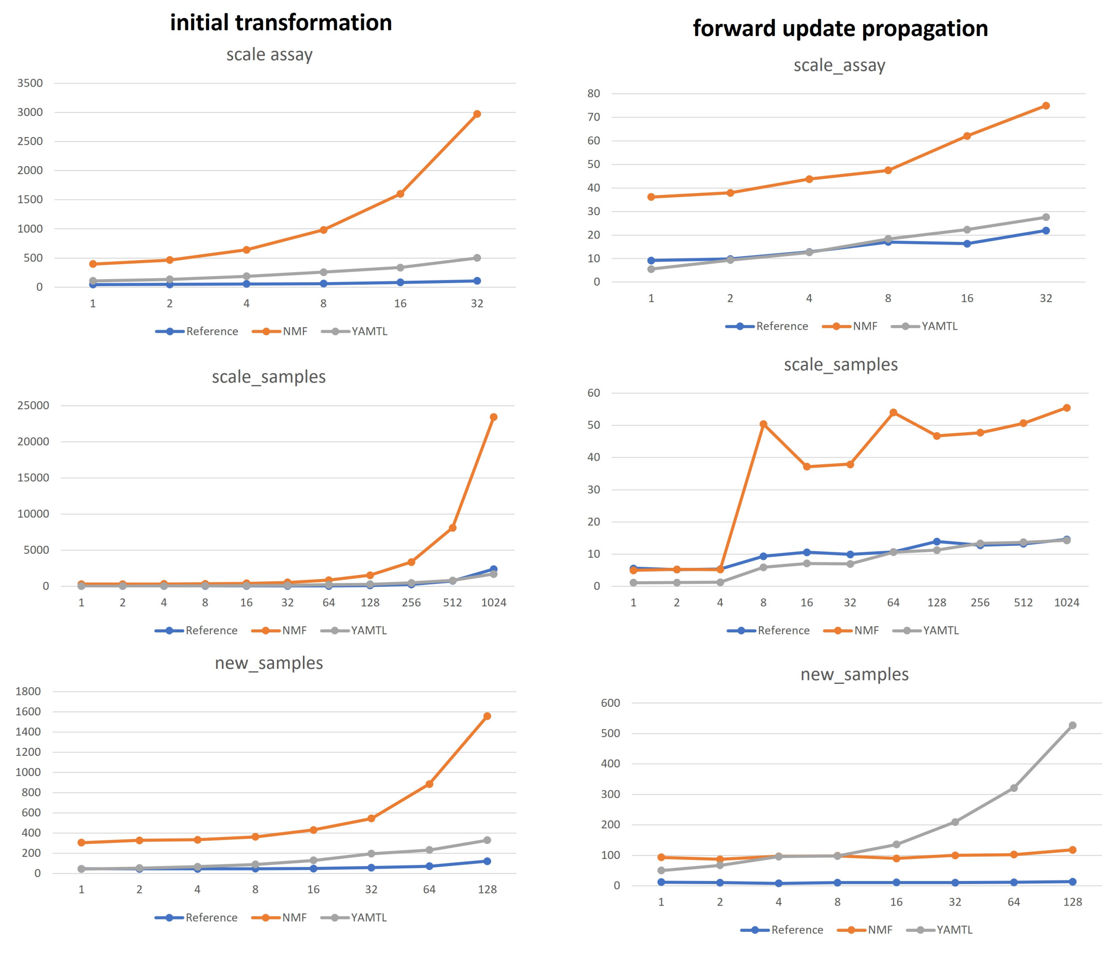

Figure 1: Preliminary results in milliseconds: initial transformation (left) and all updates per model (right).