Asymmetric and Directed Bidirectional Transformation for Container Orchestrations with YAMTL and EMF-Syncer

Artur Boronat[1]

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

Abstract

Container orchestration plays a vital role in DevOps practices, enabling efficient management of containers within complex application architectures. However, a challenge arises in bridging the gap between high-level graphical representations of container orchestration models and the concrete configuration files required by container orchestration tools. This paper proposes a bidirectional and asymmetric transformation approach from container orchestrations to MiniYAML using YAMTL, a unidirectional model-to-model transformation language, and the EMFSyncer, a bidirectional object syncer. We explore the integration of YAMTL and the EMFSyncer to leverage their complementary strengths. The paper outlines the solution, presents the transformation rules, and discusses the evaluation of the solution using benchmark criteria.

Keywords

Incremental model-to-model transformation, asymmetric transformation, EMF.

1. Introduction

In recent years, DevOps practices have gained significant traction in software development, emphasizing the collaboration between development and operations teams to achieve automated and continuous delivery of software. As part of the DevOps process, container orchestration has become a crucial aspect, enabling the efficient management of containers within complex application architectures. Docker Compose, a popular container orchestration tool, allows developers to define and manage multi-container applications.

However, a challenge arises when attempting to bridge the gap between the high-level graphical representation of container orchestration models and the concrete configuration files required by container orchestration tools. This transformation problem [1] involves translating a domain-specific model, representing container orchestrations, into YAML documents that adhere to the specific requirements of Docker Compose.

The proposed case is bidirectional and asymmetric. The transformation should not only support the conversion from the container orchestration model to the YAML document but also enable the reverse process. This bidirectional nature allows developers to update and synchronize changes made in either the model or the YAML document. However, the Docker Compose YAML file contains strictly more information than the high-level model, and any changes made in the high-level model should preserve those changes made in the configuration files.

In this paper, we present our solution for the transformation problem from container orchestrations to MiniYAML using YAMTL [2], a unidirectional model-to-model transformation language, and the EMF-Syncer [3, 4], a bidirectional object syncer. While YAMTL and the EMF-Syncer are designed for distinct use cases, in this paper, we explore their integration to harness their complementary strengths.

The structure of the paper is as follows: Section 2 provides a brief introduction to the YAMTL language and to the EMF-Syncer ; Section 3 describes an outline of the YAMTL solution; Section 4 presents the transformation definitions; Sections 5 and 6 show how to setup the synchronization process; Section 7 assesses the solution against the evaluation criteria of the benchmark; and Section 8 discusses the evaluation of the solution with the benchmark criteria.

2. YAMTL and EMF-syncer

2.1. YAMTL

YAMTL [5, 2] is a model transformation language for EMF models, with support for incremental execution [4, 3], which can be used as an internal language of any JVM language. For this paper, we have chosen its Groovy dialect.

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 input pattern for matching variables and an output pattern for creating objects. An input pattern consists of

TTC'23: 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, United Kingdom.

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

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 Groovy closures[1].

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.

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.

In YAMTL transformations, changes made to the input model after the initialization phase can be tracked using the EMF adapter framework, and these changes are linked to specific **in** or **out** elements in a rule. As a result, only the action blocks of those elements are re-executed. However, when an **out** element is re-executed, the features of its matched object are reset to execute the associated action block. While this approach prevents the introduction of duplicities and ensures correctness, it does not retain changes made in the output model through independent concurrent changes. Furthermore, resetting features that are subsequently updated in the action block means that large parts of the input model are removed via containment references and added again. Such changes are also propagated to the output model. Hence, this is the reason for using the EMF-Syncer to synchronize the output model Y_1 of the transformation with the target model Y_2.

2.2. EMF-Syncer

Given two EMF models, EMF-Syncer matches them and then synchronizes their contents.

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

Model synchronization can be performed from source to target via the operation `forwardSync` or from target to source via the operation `backSync`. EMF-Syncer automatically infers structural similarities between object-oriented data models by mapping object structural features by name, when found, translating both attribute and reference values during the synchronization process. When explicit matching is used, structural similarities are inferred using the similarity relation via the operation `match()`. The generic similarity relation defined over objects takes into account their attribute features and the absolute position of the object within the model.

The aforementioned generic mapping strategy that is built in EMF-Syncer can be customized in order to allow for more complex data transformations between the data models involved. A domain-specific mapping strategy is declared with a mapping specification that maps a source feature type to a target feature type, possibly including feature value transformations, either from source to target, or from target to source, or both. Two main custom mapping strategies can be declared: renaming of feature types and transformation of feature values.

The synchronization process can be performed using a push-based model (EAGER mode), where the entire source program state is migrated, or using a pull-based model (LAZY mode), where only those feature values accessed in the target program are migrated.

Once two models have been synchronized, changes that have been applied to an EMF model can be incrementally propagated to their model counterpart. Such changes are detected using the EMF adapter framework. Incrementality in the EMF-Syncer entails that only the changes performed in a model that was synchronized are propagated and merged within the counterpart model.

3. Solution outline

The solution consists of a pattern formed by a incremental YAMTL transformation t from source model S to a new target model T_1, from different metamodels, and an EMF forward syncing process $syncF_0$ with explicit matching between the transformation target model T_1 and a possibly existing target model T_2, conforming to the same metamodel. The model T_2 may receive changes ϕ_T directly, in addition to those received via t. The diagram in Figure 1 depicts the pattern of chained transformations based on commutative diagrams.

A source model change δ_S and a target model change ϕ_T can occur concurrently. The source model change δ_S is propagated along the YAMTL transformation t that has already been initialized via YAMTL's operation `propagate` (p for short in the diagram), inducing a target change δ_T. δ_T is represented as a dashed arrow in the diagram because it handled by the tool internally.

$$\begin{tikzcd}
S \arrow[r, "t"] \arrow[d, "\delta_{S,1}"] & T_1 \arrow[r, "syncF_0"] \arrow[d, dashed, "\delta_{T,1}"] & T_2 \arrow[d, "\phi_{T,1}"] \\
S' \arrow[r, "p_1"] \arrow[d, "\delta_{S,i}"] & T'_1 \arrow[r, "syncF_1"] \arrow[d, dashed, "\delta_{T,i}"] & T'_2 \arrow[d, "\phi_{T,i}"] \\
S^i \arrow[r, "p_i"] & T^i_1 \arrow[r, "syncF_i"] & T^i_2
\end{tikzcd}$$

Figure 1: Pattern of chained transformations.

$$\begin{tikzcd}
C \arrow[r, "c2y"] \arrow[d, "\delta_S"] & Y_1 \arrow[r, "syncF_0"] \arrow[d, dashed, "\delta_T"] & Y_2 \arrow[d, "\phi_T"] \\
C \arrow[r, "p_1"] & Y'_1 \arrow[r, "syncF_1"] & Y'_2
\end{tikzcd}$$

Figure 2: Instantiation of the pattern.

A second syncing process $syncF_p$ with explicit matching is used to reconcile the changes δ_T and ϕ_T. Subsequent change propagations are represented by additional diagrams, where transformations are labelled with the exponent i.

The parameters of the pattern are the source model S (and its metamodel), the target model T (and its metamodel), the transformation definition t, and the source and target model changes δ_S and ϕ_T. The pattern needs to be instantiated twice, once for each direction of the transformation.

Figure 2 illustrates how the pattern is instantiated to transform a container model into a YAML configuration. Figure 3 shows how the reconciliation process works in $syncF_1$, which is embodied by an explicit matching process, which computes the alignment of the models Y'_1 and Y'_2 and the parts that are dissimilar. The part of the model present in Y'_1 but not in Y'_2 is propagated to the model Y'_2 via $syncF_1$. Finally the delta δ_T applied to Y'_2 is propagated to Y''_2.

Section 4 shows the model transformation definition $c2y$. Section 5 shows how the top row of transformations are configured and executed in the method `initiateSynchronisationDialogue` of the benchmark. Section 6 shows how the subsequent rows of change propagations are executed in the method `performAndPropagateSourceEdit` of the benchmark.

$$\begin{tikzcd}
Y'_1 \arrow[r, leftrightarrow, "match"] \arrow[dr, "syncF_1"] & Y'_2 \arrow[d, dashed, "\delta_T"] \\
 & Y''_2
\end{tikzcd}$$

Figure 3: Syncing process $syncF_p$ with explicit matching.

4. Transformation Container2MiniYAML

The transformation from Containers to MiniYAML is defined in YAMTLGroovy. The module defining the forward transformation is defined as follows:

```
class YAMTLContainersToMiniYAML extends YAMTLModule {
  public YAMTLContainersToMiniYAML_helpers(EPackage CMM,
          EPackage YMM) {
    header().in('cmm',CMM).out('ymm',YMM)
    ruleStore([ /* rule declaration */ ])
    helperStore([ /* helper operations */ ])
  }
}
```

Listing 1: Transformation definition: module declaration.

The transformation rules in the forward transformation module are defined as follows:

```
rule('Composition2MainMap')
  .in('c', CMM.composition)
  .out('m', YMM.map, {
    m.entries.add( mapEntry('version', scalar('2.4')) )
    m.entries.add(
      mapEntry('services', map(
        fetch(c.getNodes().findAll{it instanceof
                Container}) )) )
    m.entries.add(
      mapEntry('volumes', map(
        fetch(c.getNodes().findAll{it instanceof Volume})
                )) )
  }),

rule('Container2MapEntry')
  .in('cn', CMM.container)
  .out('meContainer', YMM.mapEntry, {
    def cn = fetch('cn')
        def meContainer = fetch('meContainer')
    meContainer.key = cn.name
    meContainer.value = map
  })
  .out('map', YMM.map, {
    def cn = fetch('cn')
        def map = fetch('map')
    if (cn.image)
      map.getEntries().add(
        mapEntry('image', scalar(cn.image.image)) )
    if (cn.replicas != 1)
      map.getEntries().add(
        mapEntry('replicas',
                scalar(cn.replicas.toString())) )
    if (cn.volumeMounts)
      map.getEntries().add(
        mapEntry('volumes', list(fetch(cn.volumeMounts))) )
    if (cn.dependsOn)
      map.getEntries().add(
        mapEntry('depends_on', list(
                      cn.dependsOn.collect{scalar(it.name)}
                      )) )
  }),

rule('VolumeMount2Scalar')
 .in('vm',CMM.volumeMount)
 .out('s', YMM.scalar, {
   s.value = "${vm.volume.name}:${vm.path}"
 }),

rule('Volume2MapEntry')
 .in('v',CMM.volume)
 .out('me', YMM.mapEntry, {
   me.key = v.name
 })
```

Listing 2: Transformation definition: rule declaration.

The helper operations in the forward transformation module are defined as follows:

```
def YFactory = MiniyamlFactory.eINSTANCE;

def scalar(String text) {
  def sc = YFactory.createScalar()
  sc.value = text
  sc
}
def map(entries) {
  def map = YFactory.createMap()
  map.entries += entries
  map
}
def mapEntry(key,value) {
  def me = YFactory.createMapEntry()
  me.key = key
  me.value = value
  me
}
def list(values) {
  def map = YFactory.createList()
  map.values += values
  map
}
```

Listing 3: Transformation definition: helper declaration.

The backward transformation, from the metamodel MiniYAML to the metamodel Containers, is declared in a separate module. The transformation rules in the backward transformation module are defined as follows:

```
rule('MainMap2Composition')
  .in('m', YMM.map).filter{ m.eContainer() == null }
  .out('c', CMM.composition),

rule('MapEntry2Container')
 .in('m', YMM.mapEntry)
  .filter { m.eContainer()?.eContainer()?.key == 'services'
               }
 .out('c', CMM.container, {
   c.name = m.key
   addToRoot(m, c)

   def image = getFieldValue(m, 'image')
   if (image)
     c.image = fetch(image, 'im', 'Scalar2Image',
       ['root': m.eContainer().eContainer().eContainer() ])

   def replicas = getFieldValue(m, 'replicas')?.value
   if (replicas) c.replicas = (replicas as int)

   def volumes = getFieldValue(m, 'volumes')?.values
   if (volumes)
     c.volumeMounts += fetch(volumes, 'vm',
                'Scalar2VolumeMount')

   def dependsOnItem = getFieldValue(m,
             'depends_on')?.values?.value
   if (dependsOnItem) {
     dependsOnItem.each{ depName ->
       def list = fetch(allInstances(YMM.mapEntry))
         .findAll{it instanceof Container}
       def dep = list.find{ it.name==depName}
       if (dep) c.dependsOn.add(dep)
     }
   }
 }),

rule('Scalar2Image').isUniqueLazy()
 .in('s', YMM.scalar)
 .out('im', CMM.image, {
   im.image = s.value
   def c = fetch(root)
   c.nodes.add( im )

 }),

rule('Scalar2VolumeMount').isUniqueLazy()
 .in('s', YMM.scalar).filter { s.value.contains(':') }
 .out('vm', CMM.volumeMount, {
   def parts = s.value.split(':')
   vm.volume = fetch(allInstances(YMM.mapEntry))
     .findAll{ it instanceof Volume }
     .find{ v -> v.name==parts[0]}
   vm.path = parts[1];
 }),

rule('MapEntry2Volume')
 .in('m', YMM.mapEntry)
   .filter { m.eContainer()?.eContainer()?.key == 'volumes' }
 .out('v', CMM.volume, {
   v.name = m.key
   addToRoot(m, v)
 })
```

Listing 4: Backward transformation definition.

5. Initiate Synchronization Dialogue

The synchronization dialogue starts by configuring the YAMTL engine, shown in Listing 5, and the EMF-Syncer engine, shown in Listing 6.

The transformation is initialized by instantiating the transformation module. `YAMTLGroovyExtensions.init(xform)` initializes the transformation module adding syntactic sugar for calling helper operations. The transformation is instantiated as `INCREMENTAL` with granularity level `ELEMENT` and feature calls are tracked within the package `containers`. These configuration options enable YAMTL to track feature calls within the package `containers` for the Container metamodel, and incremental evaluation will be performed at the level of `in` and `out` elements, without having to match/re-execute the entire rule.

The input model is loaded using the operation `loadInputResource()`, the transformation is executed via the operation `execute()`, and the input model is adapted to listen for notifications using `adaptInputModel("cmm")`, where "cmm" is the name of the domain to be adapted.

```
xform = new YAMTLContainersToMiniYAML(
  ContainersPackage.eINSTANCE,
  MiniyamlPackage.eINSTANCE);
YAMTLGroovyExtensions.init(xform);
xform.setExecutionMode(ExecutionMode.INCREMENTAL);
xform.setIncrementalGranularity(
  IncrementalGranularity.ELEMENT);
xform.adviseWithinThisNamespaceExpressions(
  List.of("containers..*"));
xform.loadInputResources(Map.of("cmm", source));
xform.execute();
xform.adaptInputModel("cmm");
```

Listing 5: YAMTL configuration.

The EMF-Syncer is configured with pushed-based synchronization mode via `enableEagerMode`. Lines 3-7 in Listing 6 configure the two domains of the syncer, which correspond to the EMF metamodel MiniYAML. The output model Y_1 of the YAMTL transformation is set as the source model of the syncer in lines 8-10, whereas the target model is set to the target model Y_2 in lines 11-13. The synchronization is then performed by matching the overlapping elements in Y_1 and Y_2 and the complement $Y_1 \backslash Y_2$ is then merged into Y_2 via the operation `forwardSync`.

```
1  syncer = new EMFSyncer();
2  syncer.enableEagerMode();
3  var miniyamlDomain = new EMFSyncerParameter_EMF(
4    "miniyaml",
5    Map.of("pk", MiniyamlPackage.eINSTANCE));
6  syncer.setSourceModelHandler(miniyamlDomain);
7  syncer.setTargetModelHandler(miniyamlDomain);
8  syncer.setSourceModel(
9    xform.getOutputModel("ymm").getContents()
10    .stream().map(o ->
                (Object)o).collect(Collectors.toList()));
11 syncer.setTargetModel(
12   target.getContents()
13    .stream().map(o ->
                (Object)o).collect(Collectors.toList()));
14 syncer.match();
15 syncer.forwardSync();
```

Listing 6: EMF-Syncer configuration.

6. Performing and propagating source change

The propagation of the source edit is then performed by propagating all changes tracked for the model C' in domain "cmm" via `xform.propagateDelta("cmm")`. The synchronization is then performed by matching the overlapping elements in Y_1' and Y_2' and the complement $Y_1' \backslash Y_2'$ is then merged into Y_2' via the operation `forwardSync`.

```
edit.accept(getSourceModel());
xform.propagateDelta("cmm");
syncer.match();
syncer.forwardSync();
```

Listing 7: Propagating source changes.

7. Evaluation

The current solution, available at https://github.com/arturboronat/benchmarx/ (as a fork of the case repository), implements the pattern that synchronizes a container model C with an existing miniYAML model Y_2. This includes accommodating subsequent edits on both the source C and the target Y_2. Reverse synchronization has also been implemented, with the pattern partially used as the case only needs batch backward transformations.

Correctness. The solution successfully passes batch forward, forward incremental, and batch backward tests, without considering the order of the references.

Principle of Least Change. In this solution, the EMF-Syncer calculates the components of the model Y_1 that differ from those in Y_2. Differencing is based on model matching using a generic similarity relation, which is computed considering the shape of an object and its position relative to the containment structure, excluding the full graphical structure. Once two objects are matched as similar, the emerging graphical structure from each object is compared, and differing references are marked as conflicts that need to be resolved. After performing a `match()`, when the EMF conducts a `forwardSync()`, it merges the differing parts of the source model into the target model. The operation `match()` is performed incrementally in subsequent runs.

Conciseness vs Runtime. For this solution, we used an experimental version of YAMTL and EMF-Syncer that employ a Groovy dialect of the YAMTL transformation language and replace native AspectJ for detecting feature calls with a proxy-based mechanism using SpringAOP. Model transformations are thus defined using Groovy as the host language, with YAMTL serving as an internal DSL for developing model transformations.

Both changes simplify the configuration and improve readability of transformations, at the cost of sacrificing runtime performance. Groovy provides syntactic flexibility for customizing syntax. For instance, the matched object of an `in` or `out` pattern element in a rule can be accessed within the corresponding filter or action block without the need to explicitly declare the variable. This is possible due to dynamic typing in Groovy, which, however, means that typing information is not available to assist in traditional code completion. On the other hand, by using SpringAOP, there is no need to define aspects explicitly when defining a new transformation.

Groovy classes are compiled to Java bytecode using the Gradle plugin 'groovy', resulting in poorer runtime performance compared to when Java classes are used. However, YAMTL enables the definition of model transformations in Xtend or Java, using native aspects, for optimizing transformations defined in YAMTL.

Scalability. The scalability of the transformation is shown in Figure 4 using benchmark scalability tests. The runtime of the batch forward transformation seems to grow approximately quadratically with the number of containers. Conversely, the incremental forward transformation demonstrates a more linear relationship with the number of containers. Analyzing the plotted data, we observe that the *Incr. FWD* runtime increases more slowly with the number of containers compared to the *Batch FWD* runtime. This suggests that the *Incr. FWD* transformation has a near-constant runtime when compared to the *Batch FWD* transformation. This is a positive result considering that the incremental step involves re-executing a YAMTL transformation for C', matching the incrementally updated miniYAML model Y_1' with the edited one Y_2', and propagating additions from Y_1 to Y_1'.

8. Conclusions

The solution presented in this paper successfully addresses the transformation problem from container orchestrations to MiniYAML. The implementation supports the synchronization of a container model with an existing MiniYAML model, facilitating subsequent edits on either the source or the target. The solution also incorporates the principle of least change, where only differing parts of the models are updated, ensuring efficient model synchronization.

The solution demonstrates correctness, passing batch forward, forward incremental, and batch backward tests. The pattern used for synchronization is partially used for reverse synchronization, as the case only requires batch backward transformations. Despite its correctness, the solution trades off runtime performance for the sake of conciseness and readability. Using the Groovy dialect for YAMTL transformation language and replacing native AspectJ with a proxy-based mechanism for detecting feature calls resulted in easier configuration and readability, but at the expense of runtime performance. Despite the performance trade-off, the solution exhibits good scalability.

Overall, the proposed solution effectively solves the transformation problem, providing a robust, scalable, and bidirectional transformation mechanism between container orchestration models and MiniYAML documents. Future work could explore further optimizing the runtime performance while maintaining the benefits of readability and ease of configuration.

References

[1] A. Garcia-Dominguez, Asymmetric and directed bidirectional transformation for container orchestrations, 2023. URL: https://www.transformation-tool-contest.eu/TTC_2023_paper_3-v2.pdf.

[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, Code-first model-driven engineering: On the agile adoption of MDE tooling, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019), San Diego, CA, November 11-15, ACM, 2019.

[4] A. Boronat, EMF-Syncer: scalable maintenance of view models over heterogeneous data-centric software systems at run time 1619-1374 (2023). URL: https://doi.org/10.1007/s10270-023-01111-7.

[5] 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.

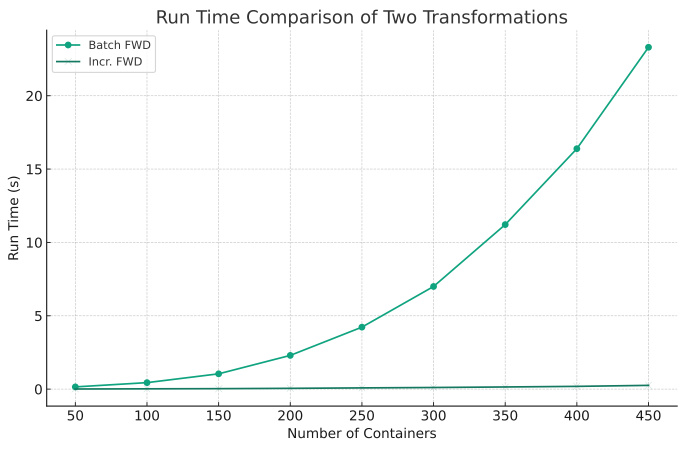

Figure 4: Runtime (seconds) for the scalability tests (forward and incremental forward)