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, to some extent.

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 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 [1], a unidirectional model-to-model transformation language, and the EMF-Syncer [2, 3], 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 3 presents an outline of the solution; section **??** presents the transformation rules used in the YAMTL solution; and section **??** discusses the evaluation of the solution with the benchmark criteria.

2. YAMTL and EMF-syncer

2.1. YAMTL

YAMTL [4, 1] is a model transformation language for EMF models, with support for incremental execution [3, 2], 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 **`in`** elements together with a global rule filter condition, which is true if not specified. Each of the **`in`** elements is

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)
© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

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 [1], including programmable execution strategies and multiple inheritance.

The matching process is based on a generic similarity relation defined over objects that takes into account their attribute features and the absolute position of the object within the model, and is performed with the `match()` operation.

Model synchronization can be performed from source to target via the operation `forwardSync` or from target to source via the operation `syncBack`. 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 under the operation `match()` instead.

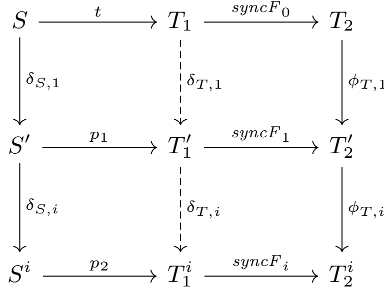
The synchronization process can be performed using a push-based model (EAGER mode), where the entire source program state is migrated, or using a pull-based model (LAZY mode), where only those feature values accessed in the target program are migrated.

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.

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 back and merged within the counterpart model.

3. Solution outline

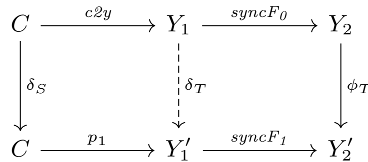
The solution consists of a pattern formed by a incremental YAMTL transformation t from source model S to a new target model T_1, from different metamodels, and an EMF forward syncing process $syncF_0$ with explicit matching between the transformation target model T_1 and a possibly existing target model 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 below 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. 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.

The diagram below shows how the pattern is instantiated to transform a container model into a YAML configuration. Section 4 shows the metamodel and 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.



4. Transformation Container2MiniYAML

The transformation from Containers to MiniYAML is defined in YAMTLGroovy. Listing 1 shows the module declaration.

```
1 class YAMTLContainersToMiniYAML extends YAMTLModule {
2   public YAMTLContainersToMiniYAML_helpers(EPackage CMM,
              EPackage YMM) {
3     header().in('cmm',CMM).out('ymm',YMM)
4     ruleStore([ /* rule declaration */ ])
5     helperStore([ /* helper operations */ ])
6   }
7 }
```

Listing 1: Transformation definition: module declaration.

Listing 1 shows the rule declaration.

```
1  rule('Composition2MainMap')
2    .in('c', CMM.composition)
3    .out('m', YMM.map, {
4      m.entries.add( mapEntry([
5        'key': 'version',
6        'value': scalar(['text': '2.4']) ]) )
7      m.entries.add( mapEntry([
8        'key': 'services',
9        'value': map([
10     'n': 0,
11     'entries': fetch(c.getNodes().findAll{n ->
               CMM.container.isInstance(n)})])]))
12     m.entries.add( mapEntry([
13       'key': 'volumes',
14       'value': map([
15     'n': 1,
16     'entries': fetch(c.nodes.findAll{n ->
               CMM.volume.isInstance(n)})])]))
17   }),
18
19 rule('Container2MapEntry')
20   .in('cn', CMM.container)
21   .out('meContainer', YMM.mapEntry, {
22     meContainer.key = cn.name
23     meContainer.value = map
24   })
25   .out('map', YMM.map, {
26     if (cn.image != null)
27       map.entries.add( mapEntry([
28     'key': 'image',
29     'value': scalar(['text': cn.image.image])]) )
30     if (cn.replicas != 1)
31       map.entries.add( mapEntry([
32     'key': 'replicas',
33     'value': scalar(['text': cn.replicas.toString()])]) )
34     if (!cn.volumeMounts.isEmpty())
35       map.entries.add( mapEntry([
36     'key': 'volumes',
37     'value': list(['values': fetch(cn.volumeMounts)])]) )
38     if (!cn.dependsOn.isEmpty())
39       map.entries.add( mapEntry([
40     'key': 'depends_on',
41     'value':  list(['values':
               cn.dependsOn.collect{scalar(['text':
               it.name])}])]) )
42   }),
43
44 rule('VolumeMount2Scalar')
45  .in('vm',CMM.volumeMount)
46  .out('s', YMM.scalar, {
47    s.value = "${vm.volume.name}:${vm.path}"
48 }),
49
50 rule('Volume2MapEntry')
51  .in('v',CMM.volume)
52  .out('me', YMM.mapEntry, {
```

```
53    me.key = v.name
54 })
```

Listing 2: Transformation definition: rule declaration.

Listing 1 shows the helper declaration.

```
1  def YFactory = MiniyamlFactory.eINSTANCE;
2
3  staticOperation('scalar', { argMap ->
4    def sc = YFactory.createScalar()
5    sc.value = argMap.text
6    sc
7  }),
8
9  staticOperation('map', { argMap ->
10   def map = YFactory.createMap()
11   map.entries += argMap.entries
12   map
13 }),
14
15 staticOperation('mapEntry', { argMap ->
16   def me = YFactory.createMapEntry()
17   me.key = argMap.key
18   me.value = argMap.value
19   me
20 }),
21
22 staticOperation('list', { argMap ->
23   def map = YFactory.createList()
24   map.values += argMap.values
25   map
26 })
```

Listing 3: Transformation definition: helper declaration.

5. Initiate synchronization Dialogue

The synchronization dialogue starts by configuring the YAMTL engine, shown in Listing 4, and the EMF-Syncer engine, shown in Listing 5.

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.

```
1  xform = new YAMTLContainersToMiniYAML(
2    ContainersPackage.eINSTANCE,
3    MiniyamlPackage.eINSTANCE);
4  YAMTLGroovyExtensions.init(xform);
5  xform.setExecutionMode(ExecutionMode.INCREMENTAL);
6  xform.setIncrementalGranularity(
7    IncrementalGranularity.ELEMENT);
8  xform.adviseWithinThisNamespaceExpressions(
9    List.of("containers..*"));
10 xform.loadInputResources(Map.of("cmm", source));
11 xform.execute();
12 xform.adaptInputModel("cmm");
```

Listing 4: YAMTL configuration.

The EMF-Syncer is configured with pushed-based synchronization mode via `enableEagerMode`. Lines 3-7 in Listing5 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 5: 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`.

```
1 edit.accept(getSourceModel());
2 xform.propagateDelta("cmm");
3 syncer.match();
4 syncer.forwardSync();
```

Listing 6: Propagating source changes.

7. Conclusions

The current solution provided at https://github.com/arturboronat/ttc2023-bx implements the instantiation of the pattern that synchronizes a container model C with an existing miniYAML model Y_2, including subsequent edits on the source C and on the target Y_2.

The reverse synchronization is yet to be implemented by developing an opposite YAMTL transformation from miniYAML configuration to the corresponding container model. The rest of the arguments for the pattern parameters are provided as in the presented solution above.

The solution passes the tests batch forward and forward incremental, without considering the order of the references. Regarding the principle of least change, the EMF-Syncer computes the parts of the model Y_1 that differ from those in Y_2 merging only the parts that were changed.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.