

Asymmetric and Directed Bidirectional Transformation for Container Orchestrations

Antonio Garcia-Dominguez
a.garcia-dominguez@york.ac.uk
University of York
York, UK

ABSTRACT

In many DevOps scenarios, tools operate from declarative models of intended system configuration (e.g. Ansible/Puppet/Chef descriptions of infrastructure-as-code, or Kubernetes and Docker Compose descriptions of orchestrations of containers). DevOps-oriented domain-specific modeling notations will typically only cover a subset of all the capabilities in these configuration formats: this means users will need to manually edit the configuration files generated from the higher-level models. In many editing sessions, users will also touch upon parts that came from the high-level model, and will want that high-level model to be updated accordingly. Likewise, a user may want to introduce a change through the high-level model and not lose the YAML customisations that are unrelated to the high-level model. These requirements imply a need for a bidirectional transformation (“bx”) which is asymmetric (the configuration file contains all the information in the high-level model and more), and directed (changes are only applied to one side at a time). This case proposes revisiting the current state of bx tools for asymmetric and directed transformations, and complements the prior Families to Persons case from TTC 2017, which focused on a symmetrical and directed transformation. The case will reuse the Benchmarx framework from the TTC 2017 case.

CCS CONCEPTS

• Software and its engineering → Domain specific languages.

KEYWORDS

container orchestration, bidirectional transformations, model merging, graphical models, YAML

ACM Reference Format:

Antonio Garcia-Dominguez. 2023. Asymmetric and Directed Bidirectional Transformation for Container Orchestrations. In *Proceedings of 15th Transformation Tool Contest (TTC’23)*. ACM, New York, NY, USA, 5 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

DevOps was defined by Leite et al. [5] as a “collaborative and multidisciplinary effort within an organization to automate continuous

delivery of new software versions, while guaranteeing their correctness and reliability”. The rising interest in DevOps (with over 10% of the 61,302 responses to the Stack Overflow 2022 Developer Survey¹ considering themselves “DevOps specialists”) has motivated the creation of a number of domain-specific modelling notations for it, covering aspects such as microservice architectures [6], DevOps processes [3], or multi-cloud applications [4].

At a technical level, the automated continuous delivery efforts in DevOps typically require using tools to automate deployment. These include infrastructure-as-code tools (e.g. Puppet² or Ansible³), and container orchestration tools such as Kubernetes⁴ or Docker Compose⁵. Many of these tools operate by reading a declarative description of the desired system state or the intended combination of containers, usually written in a structured format (e.g. YAML⁶) according to a loosely defined schema (c.f. the Docker Compose file format reference, which evolves from version to version⁷).

It stands to reason that DevOps model-driven approaches would often aim to generate at least some of these configuration files from the high-level descriptions of the intended service compositions. Most likely, the high-level descriptions will only model the subset of the capabilities of the underlying tools that is relevant for their abstractions, as trying to capture all capabilities would overcomplicate the models and make them more brittle to minor changes in the underlying configuration file formats. From this limitation, it follows that users would typically manually customise the generated configuration files to cover the aspects not described by the high-level model. Users may later want to update the high-level model from the configuration file, to use it for visualisation (e.g. for onboarding new developers) or for reorganising the system in a more approachable notation with domain-specific validation rules.

This paper proposes a case based on the above scenario, focusing on container orchestration with Docker Compose. A high-level graphical domain-specific model (implemented with Sirius) is transformed into a Docker Compose YAML file, which can be customised by the user using a plain text editor. The high-level model can be updated from the YAML file at any time. It should also be possible to edit the high-level model and push the changes to the YAML file, while retaining any elements that were not part of the high-level graphical DSML.

At an essential level, this case implies the definition of a bidirectional transformation (“bx” from now on) between the high-level DSML and Docker Compose YAML files. In TTC 2017, the

¹<https://survey.stackoverflow.co/2022/>

²<https://www.puppet.com/>

³<https://www.ansible.com/>

⁴<https://kubernetes.io/>

⁵<https://docs.docker.com/compose/>

⁶<https://yaml.org/>

⁷<https://docs.docker.com/compose/compose-file/>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

TTC’23, July 20, 2023, Leicester, UK

© 2023 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06...\$0.00

<https://doi.org/XXXXXXX.XXXXXXX>

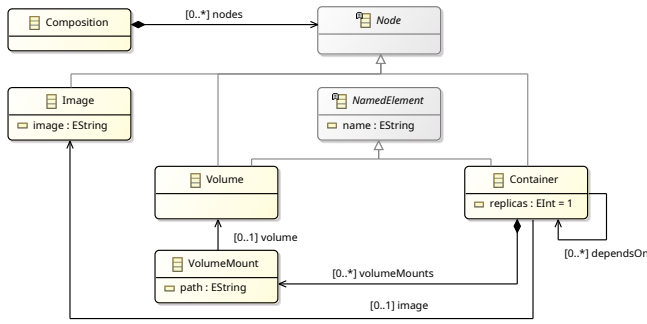


Figure 1: Class diagram for the Containers metamodel

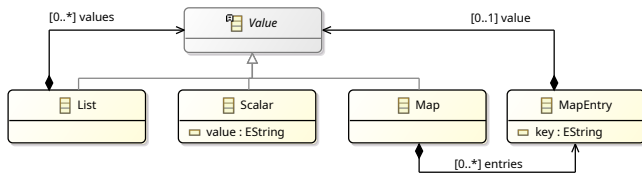


Figure 2: Class diagram for the MiniYAML metamodel

Families to Persons case by Anjorin et al. [1] evaluated the available approaches for symmetric and directed bx using the proposed Benchmarx framework. This work was later updated and expanded upon in a journal paper [2], which also collected a number of useful terms to describe bx, as well as a feature model to cover the variability of bx tools. Families to Persons was symmetric (neither side was a view of the other, with information loss happening in both directions), and directed (consistency-relevant changes were only applied to one side at a time). The proposed case is still directed, but it is asymmetrical (the Docker Compose YAML file contains strictly more information than the high-level model, so information loss only happens in one direction). While this should make it conceptually “easier” than the symmetric Families to Persons bx, the mapping is also more complicated, with some objects in the high-level model being turned into simple string concatenations in the target model. At the same time, it can be argued that the generation of Docker Compose YAML files is a more industrially relevant scenario: if the current state of the art in bx tools (which may have significantly evolved since the TTC 2017 case) can handle it well, this could prove to be an interesting application niche.

2 MODELING LANGUAGES

The proposed bx is between two languages: a “Containers” domain-specific modelling language (shown in Figure 1), and a simplification of the YAML data model called “MiniYAML” (shown in Figure 2).

2.1 Abstract syntax

Models conforming to the Containers metamodel (Figure 1) have a COMPOSITION as their root object, containing a number of NODES of various types. An IMAGE represents a specific Docker image by its full name including the registry (if it is not the Docker Hub) and tag, as stored in its image attribute. A CONTAINER is a component that runs one or more replicas of a certain IMAGE. A CONTAINER

may have VOLUMEMOUNTS of certain VOLUMES (units of persistent storage) at specific paths. CONTAINERS and VOLUMES are NAMEDELEMENTS, which have a name that also acts as their unique identifier. A CONTAINER may dependOn other CONTAINERS, meaning that it should only be started after its dependencies have been started.

On the other hand, a model conforming to the MiniYAML metamodel (Figure 2) has a MAP as its root object, which contains MAPENTRY objects. Each MAPENTRY has a key (a string, which should be unique within its containing MAP), and a value. Besides MAP, other types of VALUES include LISTS (of VALUES), and SCALAR values with a string (this is a simplification from YAML, which can support integer and floating point types through its JSON schema).

2.2 Concrete syntax

The concrete syntax of the Containers modelling language is implemented through Eclipse Sirius⁸ and exemplified in Figure 3, which models the container orchestration used by the AutoFeedback system developed by the author⁹. CONTAINERS are grey rectangles decorated with a puzzle piece icon, labelled after their name. A CONTAINER may contain yellow ovals representing their VOLUMEMOUNTS, labelled after their paths and decorated with a folder icon. An IMAGE is reflected as a green oval with a cardboard box icon, labelled after their image. VOLUMES are purple rectangles with a hard disk icon, labelled after their name. Note that the replicas of a CONTAINER is not part of its graphical syntax, but can be edited through the Properties view of the Sirius editor.

The MiniYAML language does not have an explicitly defined concrete syntax: while the case artifact includes a tree-based editor autogenerated from the metamodel, the ultimate concrete syntax is YAML itself. The case artifact includes an uk.ac.york.ttc.mini-yaml.model2yaml project with a MINICYAMLCONVERTER Java class which uses SnakeYAML¹⁰ to automatically convert between MiniYAML models in XMI format, and YAML files.

3 INTENDED TRANSFORMATIONS

The general intent of the transformation is to start from a model as the one in Figure 3, and produce a YAML document such as the one in Listing 1. This YAML document can be edited manually in various ways: a user could do a find-and-replace to rename a given container, or they could add extra options for a given container or volume which are not part of the Containers metamodel. It should be possible for the user to update the Containers model from the YAML file at any point. It should also be possible to edit a Containers model and update the YAML file from it, while keeping any customisations that are unrelated to the Containers metamodel.

3.1 High-level description

In its forward direction (from Containers to MiniYAML), operating in batch mode (where the MiniYAML model does not exist yet), the transformation should follow these rules:

- (1) A COMPOSITION should be transformed into a MAP with three keys: version set to a “2.4” SCALAR, services set to

⁸<https://www.eclipse.org/sirius/>

⁹<https://gitlab.com/autofeedback/autofeedback-webapp/-/blob/master/docker-compose.yml>

¹⁰<https://bitbucket.org/snakeyaml/snakeyaml>

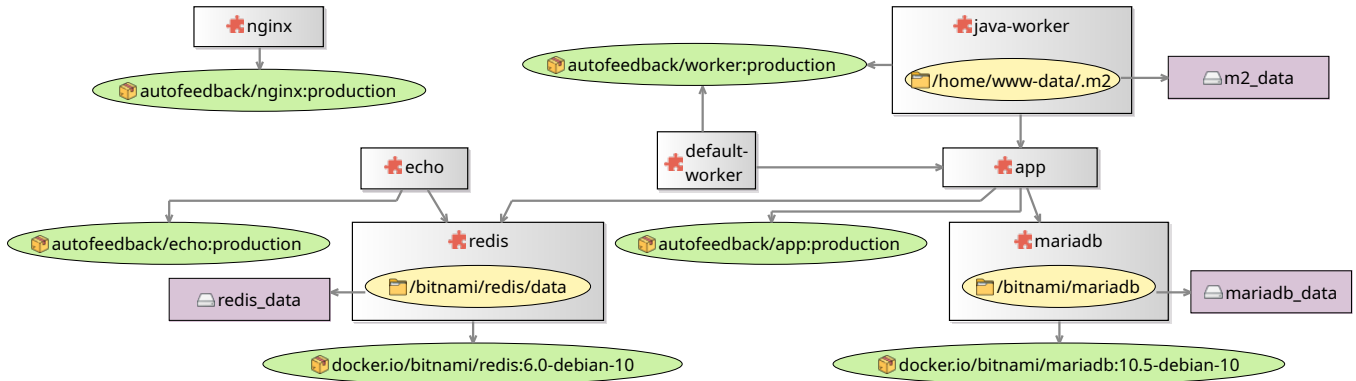


Figure 3: Example containers model, based on the AutoFeedback open-source system

Listing 1: Example YAML from Figure 3

```

version: '2.4'
services:
  mariadb:
    image: docker.io/bitnami/mariadb:10.5-debian-10
    volumes: ['mariadb_data:/bitnami/mariadb']
  redis:
    image: docker.io/bitnami/redis:6.0-debian-10
    volumes: ['redis_data:/bitnami/redis/data']
  nginx: {image: 'autofeedback/nginx:production'}
  app:
    image: autofeedback/app:production
    depends_on: [mariadb, redis]
  java-worker:
    image: autofeedback/worker:production
    replicas: '2'
    volumes: ['m2_data:/home/www-data/.m2']
    depends_on: [app]
  default-worker:
    image: autofeedback/worker:production
    replicas: '2'
    depends_on: [app]
  echo:
    image: autofeedback/echo:production
    depends_on: [redis]
volumes: {mariadb_data: null, redis_data: null, m2_data: null}

```

a MAP whose MAPENTRY objects are produced from the CONTAINERS, and volumes set to a MAP produced from the VOLUMES.

- (2) A CONTAINER should be transformed into a MAPENTRY where the key is equal to its name. The value of the MAPENTRY should be a MAP of its own, with at least the image key set to the image of the IMAGE of the CONTAINER. The MAP may also have keys for:
 - replicas, if the value is different from 1.

- volumes, set to a LIST produced from the VOLUME MOUNTS of the CONTAINER.
 - depends_on, set to a LIST of SCALARS with the names of the CONTAINERS that this CONTAINER depends upon.
- (3) A VOLUME MOUNT should be transformed into a SCALAR whose value should be of the form “volumeName:path”.
 - (4) A VOLUME should be turned into a MAPENTRY whose key should be its name. The MAPENTRY should not have a value.

If the MiniYAML model already exists before running the transformation forward, then the containers, volumes, volume mounts, replicas, and inter-container dependencies of the Containers model should replace those of the MiniYAML model, while preserving any other elements outside the Containers metamodel (e.g. a custom restart entry in a container’s MAP). At the very least, adding or removing one of these elements from the Containers model should add or remove the relevant element in the MiniYAML model. Ideally, the transformation should be able to handle the renaming of a CONTAINER or VOLUME while preserving the additional content that is unrelated to the Containers metamodel. Furthermore, the transformation should minimise unnecessary changes in the YAML file (e.g. changes in the order of the map entries).

In its backward direction (from MiniYAML to Containers) in batch mode, the transformation should recover the COMPOSITIONS, IMAGES, VOLUMES and VOLUME MOUNTS from the same MiniYAML elements that would have been produced in the forward direction. These will replace the contents of the Containers model entirely. Ideally, the transformation should minimise unnecessary changes (e.g. changing the path of an IMAGE in the model, which would cause unnecessary changes in the Sirius diagrams).

3.2 Reference implementation

Besides the above high-level description, the case materials include EMF-based implementations of the Containers and MiniYAML metamodels, and a reference implementation of the transformation using a combination of languages from the Eclipse Epsilon open-source project:

- An ETL (Epsilon Transformation Language) script transforming Containers models to MiniYAML models (containers2miniyaml.etl).

- An ETL script transforming MiniYAML models to Containers models (`miniyaml2containers.etl`).
 - A combination of an Epsilon Merging Language (EML) script, an Epsilon Comparison Language (ECL) script, and an ETL script which can merge two MiniYAML models together (`mergeMiniyaml.eml`, `compareMiniyaml.ecl`, and `mergeMiniyaml.etl`) respectively.
- In this transformation, the “left” MiniYAML model is the “priority” one: its containers, volumes, volume mounts, replicas, and inter-container dependencies will take precedence over those of the “right” MiniYAML model. Any other content (e.g. customisations outside the Containers meta-model) will be merged.

At a high-level, the ECL script computes a match between the “left” and “right” models based on name-based paths (e.g. `services.redis.image`), where SCALARS also consider their value. The EML script merges matching elements together, and the ETL script copies non-matching elements from either side.

These transformations are then encapsulated as Java classes:

- `CONTAINERSToMINIYAML` implements the batch forward transformation, `MERGINGCONTAINERSToMINIYAML` implements the forward transformation with merging if the MiniYAML model already exists, and `MERGINGCONTAINERSToYAML` class implements the forward transformation with merging if the YAML file already exists.
- `MINIYAMLToCONTAINERS` implements the batch backward transformation from a MiniYAML model to a Containers model, and `YAMLToCONTAINERS` also transforms the YAML file into a MiniYAML model before transforming it into a Containers model. The reference implementation does not have a “merging” version of the backward transformation: it replaces the Containers model if it exists.

4 RESEARCH QUESTIONS

The aim of this case is to explore the capabilities of the current state of the art of transformation tools in an asymmetric and directed bx. Specifically, the case is intended to answer these questions:

- (1) How concisely can we specify such a bx with current tools? Having to maintain separate one-way transformations as in the reference implementation would incur significant cost when scaling up to the full complexity of real-world metamodels. Ideally, it should be possible to implement the bx through a single set of relationships, without repetition. This could be done through explicit consistency relationships, through triple graph grammars, or through static analysis of a one-way transformation (with perhaps some use of heuristics).
- (2) How well can such a bx preserve customisations in the YAML which are outside of the bx, across various types of changes in the models? The reference implementation can handle well the case where elements are added and removed, but it cannot handle renames well: renaming a container in the Containers model would most likely result in losing the additional content in the YAML file. A bx tool that can operate with

operational deltas (“o-deltas”) would most likely be able to handle this case in a more robust manner.

- (3) How would such a bx scale to larger models, with more containers, more volumes, and more custom YAML elements outside of the transformation’s control?

In the reference implementation, the merging process of the MiniYAML model newly created from the Containers model with the previously existing (and potentially customised) MiniYAML model requires pairwise object matching, with $O(n^2)$ path comparisons per type. Is such a cost unavoidable, or are there more efficient ways to establish and maintain the relationships between the Containers and MiniYAML models?

In practice, it is unlikely that the YAML documents will grow particularly large¹¹. Performance would likely not be an issue for this bidirectional transformation. Instead, maintainability and keeping to the principles of “least change” and “least surprise” would be the most important aspects to tackle.

5 EVALUATION CRITERIA

Solutions will be evaluated across the following criteria:

- (1) *Correctness*: following the approach from the authors of the Benchmarx benchmark [2], test cases will check that the dependent model is consistent with the master model. This means that they should have the same containers, volumes, volume mounts, and images. This criteria will be measured according to the % of test cases that are passed. The test cases will cover various scenarios, e.g. an initial “batch” execution in either direction, or the update of the dependent side after a certain change in the master side.
- (2) *Conciseness*: a more concise description of the transformation should in principle be more maintainable. Since the statement structure can be significantly different across language, this criteria will be measured as the number of words in the transformation rules, ignoring comments. This is the same approach that was followed in the Families to Persons study cited above. For reference, the reference implementation includes a `count-words.sh` script which uses the C preprocessor to remove comments for Epsilon / Java programs. In its current version, the reference implementation uses 749 words in its Java code, 480 words in ETL, 162 words in EML, 81 words in EOL and 84 words in ECL, for a total of 1556 words.
- (3) *Least Change*: beyond just correctness, the transformations should avoid making any unnecessary changes that do not impact the consistency of the master and dependent model. For instance, in the forward direction, they should preserve the additional information in the existing YAML file, and the relative order of the keys in the YAML document. In the backward direction, they should also preserve the locations of the various nodes, avoiding disturbing existing Sirius diagrams whenever possible.

¹¹The average size of the `composer.yaml` files in the `docker/awesome-compose` Github project is 609B: <https://github.com/docker/awesome-compose>.

This will be measured by checking in each test case the number of differences between the actual output and the expected output that do not impact consistency (e.g. YAML keys in a different relative position, or missing YAML keys that are unrelated to the transformation). Ideally, no such differences should be produced.

6 TARGET PRIZES

The prizes will be based on a combination of the three criteria above. The “Most Complete” prize will go to the solution that passes the most tests (resolving ties using the “Least Change” criterion). The “Most Concise” prize will go to the solution that requires the last words, while still passing the correctness tests for adding and deleting elements (the tests for renaming elements will not be considered).

If there are enough solutions, an overall ranking can be devised by adding their rankings in each category, and sorting in ascending order. Ties will be resolved by sorting in ascending order of standard deviation (therefore, a tool that is 2nd/2nd would be ranked above a tool that is 1st/3rd). Further ties will be resolved by the case author and TTC organizers.

7 JOURNAL-QUALITY SOLUTION CRITERIA

To be eligible for a follow-up journal publication, a solution must be correct in regard to addition and removal of containers, volumes,

volume mounts, and images. Conciseness and “least change” are desirable properties, but not required for such a publication. Ideally, declarative solutions that support maintainability by not requiring the specification of both transformation directions would be preferred.

REFERENCES

- [1] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. 2017. The Families to Persons Case. In *Proceedings of the 10th Transformation Tool Contest*, Vol. 2026. CEUR-WS.org, Marburg, Germany, 27–34. <http://ceur-ws.org/Vol-2026/paper2.pdf>
- [2] Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. 2020. Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software and Systems Modeling* 19, 3 (May 2020), 647–691. <https://doi.org/10.1007/s10270-019-00752-x>
- [3] Alessandro Colantoni, Luca Berardinelli, and Manuel Wimmer. 2020. DevOpsML: towards modeling DevOps processes and platforms. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, Virtual Event Canada. <https://doi.org/10.1145/3417990.3420203>
- [4] Nicolas Ferry, Franck Chauvel, Hui Song, Alessandro Rossini, Maksym Lushpenko, and Arnor Solberg. 2018. CloudMF: Model-Driven Management of Multi-Cloud Applications. *ACM Transactions on Internet Technology* 18, 2 (May 2018). <https://doi.org/10.1145/3125621>
- [5] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2020. A Survey of DevOps Concepts and Challenges. *Comput. Surveys* 52, 6 (Nov. 2020). <https://doi.org/10.1145/3359981>
- [6] Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. 2021. Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations. *SN Computer Science* 2, 6 (Sept. 2021). <https://doi.org/10.1007/s42979-021-00825-z>