Asymmetric and Directed Bidirectional Transformation for Container Orchestrations

Antonio Garcia-Dominguez[1]

[1]*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.

Keywords

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

1. Introduction

DevOps was defined by Leite et al. [1] 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 Survey[1] considering themselves "DevOps specialists") has motivated the creation of a number of domain-specific modelling notations for it, covering aspects such as microservice architectures [2], 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. Puppet[2] or Ansible[3]), and container orchestration tools such as Kubernetes[4] or Docker Compose[5]. 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. YAML[6]) according to a loosely defined schema (c.f. the Docker Compose file format reference, which evolves from version to version[7]).

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. Piedade et al. observed a significant reduction in development effort with a visual notation for developing Docker Compose container orchestrations [5], while also noticing that several of the existing visual tools for Docker Compose lacked support for certain Docker Compose concepts (e.g. DockStation did not support specifying networks). Their 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.

It is worth noting that there are some agreed-upon specifications in cloud computing that have been adapted into model-driven approaches. Zalila et al. [6] proposed

TTC'23: 15th 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, UK.

[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/

OCCIware Studio, a model-driven toolchain that formalises the concepts in the Open Cloud Computing Interface (OCCI, a unified RESTful protocol for cloud computing management) into an OCCIware Ecore metamodel, and provides a runtime component for design, deployment, execution, and supervision of cloud applications. Challita et al. later proposed TOSCA Studio [7], also based on OCCIware, which provides a model-driven approach to design OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) descriptions: these are usually written in YAML and only describe the structure of cloud applications in a declarative manner, leaving the exact implementation up to the TOSCA-supporting cloud provider. OpenTOSCA[8] is an open-source end-to-end toolchain for deploying and managing cloud applications, which includes Eclipse Winery, a web-based environment for visual modeling of TOSCA cloud application topologies (which generates TOSCA YAML descriptions).

This paper proposes a case based on a scenario inspired by the findings of Piedade et al. [5], 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 Families to Persons case by Anjorin et al. [8] 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 [9], 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.

[8]https://www.opentosca.org/

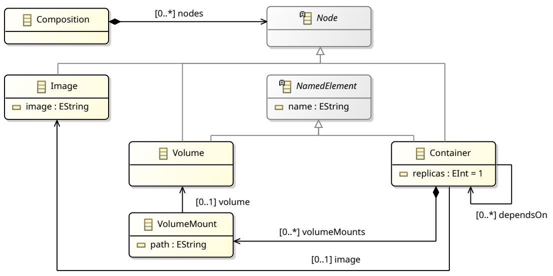

Figure 1: Class diagram for the Containers metamodel

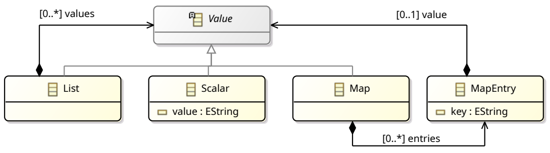

Figure 2: Class diagram for the MiniYAML metamodel

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

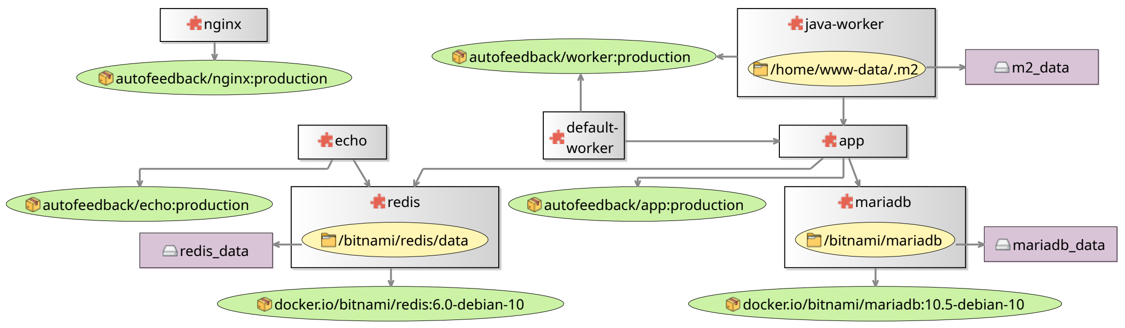
nginx
autofeedback/nginx:production
autofeedback/worker:production
java-worker
/home/www-data/.m2
m2_data
default-worker
app
echo
autofeedback/echo:production
redis
/bitnami/redis/data
redis_data
autofeedback/app:production
mariadb
/bitnami/mariadb
mariadb_data
docker.io/bitnami/redis:6.0-debian-10
docker.io/bitnami/mariadb:10.5-debian-10

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

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 Sirius[9] and exemplified in Figure 3, which models the container orchestration used by the AutoFeedback system developed by the author[10]. 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.miniyaml.model-2yaml` project with a MINIYAMLCONVERTER Java class which uses SnakeYAML[11] to automatically convert between MiniYAML models in XMI format, and YAML files.

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}
```

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.

[9] https://www.eclipse.org/sirius/
[10] https://gitlab.com/autofeedback/autofeedback-webapp/-/blob/master/docker-compose.yml
[11] https://bitbucket.org/snakeyaml/snakeyaml

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 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 VolumeMounts of the Container.
   - `depends_on`, set to a List of Scalars with the names of the Containers that this Container depends upon.
3. A VolumeMount 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 VolumeMounts 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[12] 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 "prioritary" 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 metamodel) 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.

[12]https://github.com/agarciadom/benchmarx/tree/main/examples/containerstominiyaml

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 will 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 large[13]. Performance would likely not be an issue for this bidirectional transformation. Instead, maintainability and keeping to the principle of "least change" would be the most important aspects to tackle. Still, the case materials include an experiment for evaluating the scalability of the solutions to larger models.

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

Listing 2: Sample code for measuring AST/ASG side of transformation rules modelled in EMF

```
public int countNodes(Resource resource) {
  final TreeIterator<EObject> it = resource.getAllContents();
  int size = 0;
  while (it.hasNext()) {
    it.next();
    ++size;
  }
  return size;
}
```

5. Evaluation criteria

Solutions will be evaluated across the following criteria:

1. *Correctness*: following the approach from the authors of the Benchmarx benchmark [9], 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 languages, the metric will be "number of nodes in the transformation's abstract syntax, ignoring comments". This differs from the approach that was followed in the Benchmarx "Families to Persons" study that is the base of this case [8], which counted words while ignoring comments. This is to accommodate both textual and graphical transformation notations (e.g. triple graph grammars). For instance, if the transformation was implemented as an Eclipse Modeling Framework (EMF) model, the metric would be equivalent to the code in Listing 2.
   The case includes an `ast-counter` Maven project which can count the number of AST nodes in Java code and in the Epsilon languages used for the reference solution. Participants are encouraged to extend this project to measure their source languages (e.g. by counting the number of elements in an XMI-serialised model), by adding the relevant implementations of the IFILEMEASURER interface and associating it to the appropriate extension inside the static block of the FOL-

derMeasurer class. It is also acceptable to produce these AST measurements separately as part of their solution (e.g. if no JVM-friendly parsers exist for a transformation language).

The reference implementation includes an Eclipse launch configuration that measures the number of AST nodes in its Java and Epsilon source code. Participants are encouraged to duplicate this launch configuration for their own solutions, providing it with the root folder of their transformation source code.

Note that the reference implementation also includes a `count-words.sh` script which uses the C preprocessor to remove comments for Epsilon / Java programs. This is only to emulate what the old approach (based on words) would have produced, for the sake of comparison: it will not be used for the contest, as results may not be directly comparable. As an example, these are the results of the two measurement methods at the time of writing for the reference implementation:

- *AST node counting*: 86 nodes in ECL, 241 nodes in EML, 92 nodes in EOL, 805 nodes in ETL, and 1772 nodes in Java, for a total of 2996 nodes.
- *Word counting*: 84 words in ECL, 162 words in EML, 81 words in EOL, 485 words in ETL, and 809 words in Java, for a total of 1621 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.
   To measure this, the test suite has been designed to be run in two modes: 1) requiring that if the YAML model already exists, the relative order of map entries and list items is preserved, and 2) waiving this requirement. Mode 1 is intended for evaluating "least change" (in terms of % of tests passed in this mode), whereas Mode 2 is for evaluating general correctness of the transformation.
4. *Scalability*: the transformations should be able to scale to models with increasing numbers of containers, volumes, and images. The case materials include a ScalabilityMeasurements class to measure this in the forward and backward directions, both in batch and in incremental situations.

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 least nodes, while still passing the correctness tests for adding and deleting elements (the tests for renaming elements will not be considered). The "Most Scalable" prize will go to the solution with the lowest execution times, which is still correct in the batch scenarios and in the incremental addition and removal of containers, volumes, volume mounts, and images.

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 the "batch" context in both directions, and in the "incremental" context in regard to addition and removal of containers, volumes, volume mounts, and images. Conciseness, "least change", and scalability 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] L. Leite, C. Rocha, F. Kon, D. Milojicic, P. Meirelles, A Survey of DevOps Concepts and Challenges, ACM Computing Surveys 52 (2020). doi:10.1145/3359981.

[2] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, A. Zündorf, Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations, SN Computer Science 2 (2021). doi:10.1007/s42979-021-00825-z.

[3] A. Colantoni, L. Berardinelli, M. Wimmer, 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, 2020. doi:10.1145/3417990.3420203.

[4] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, A. Solberg, CloudMF: Model-Driven Management of Multi-Cloud Applications, ACM Transactions on Internet Technology 18 (2018). doi:10.1145/3125621.

[5] B. Piedade, J. P. Dias, F. F. Correia, Visual notations in container orchestrations: an empirical study with Docker Compose, Software and Systems Modeling 21 (2022) 1983–2005. doi:10.1007/s10270-022-01027-8.

[6] F. Zalila, S. Challita, P. Merle, Model-driven cloud resource management with OCCIware, Future Generation Computer Systems 99 (2019) 260–277. doi:10.1016/j.future.2019.04.015.

[7] S. Challita, F. Korte, J. Erbel, F. Zalila, J. Grabowski, P. Merle, Model-based cloud resource management with TOSCA and OCCI, Software and Systems Modeling 20 (2021) 1609–1631. doi:10.1007/s10270-021-00869-y.

[8] A. Anjorin, T. Buchmann, B. Westfechtel, The Families to Persons Case, in: Proceedings of the 10th Transformation Tool Contest, volume 2026, CEUR-WS.org, Marburg, Germany, 2017, pp. 27–34. URL: http://ceur-ws.org/Vol-2026/paper2.pdf.

[9] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H.-S. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, A. Zündorf, Benchmarking bidirectional transformations: theory, implementation, application, and assessment, Software and Systems Modeling 19 (2020) 647–691. doi:10.1007/s10270-019-00752-x.