White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Towards Processing YAML Documents with Model Management Languages

Ionut Predoaia
University of York
United Kingdom
ionut.predoaia@york.ac.uk

Dimitris Kolovos
University of York
United Kingdom
dimitris.kolovos@york.ac.uk

Antonio García-Domínguez
University of York
United Kingdom
a.garcia-dominguez@york.ac.uk

Matthias Lenk
Upstream Security
Germany
matthias.lenk@posteo.net

Wolfram Ebel
NetApp
Germany
wolfram.ebel@netapp.com

Jan Burkl
NetApp
Germany
jan.burkl@netapp.com

ABSTRACT

YAML is a widely used textual format for capturing structured data. Despite its widespread use by software engineering practitioners, there is little support for YAML in model management (e.g. model-to-text, model-to-model) languages. This paper proposes an approach for bridging the conceptual gap between contemporary model management languages and YAML. A technical solution is presented for enabling the use of model management tasks over models captured in YAML. Our solution is evaluated in an industrial case study on cloud infrastructure automation, involving the use of model transformations that transform EMF models into YAML models, with the goal of producing Infrastructure as Code through Ansible Playbooks.

CCS CONCEPTS

• **Software and its engineering** → **Model-driven software engineering**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

Model Management, YAML, MDE, EMF, Infrastructure as Code, Ansible, Eclipse Epsilon, EMC Driver, Cloud Automation

1 INTRODUCTION

YAML is one of the most widely used languages for configuration files and data storage, largely due to its simplicity and readability. Many technologies and tools rely on YAML for DevOps (e.g., Kubernetes, Docker Compose, GitHub Actions, AWS CodeBuild, Microsoft Azure Pipelines), for infrastructure management (e.g., Ansible, Terraform, AWS CloudFormation, Google Cloud Deployment Manager), and for API specification (e.g., OpenAPI, Swagger UI), among other uses.

Many research efforts have involved domain-specific modelling languages expressed through a YAML-based syntax, e.g., for cloud provisioning [5], web services [9], DevOps configurations [3, 30], data science workflows [8], functional decomposition [17], graph-learning processes [33] and performance assessment [13]. Furthermore, YAML documents have been the object of research in many works involving model-driven engineering (MDE) workflows. Examples of such works involve model transformations producing DevOps artefacts [3, 6], the automatic code generation of software systems [23], web services [9, 36] and schemas [7], and bidirectional model transformations in the context of the 2023 Transformation Tools Contest [2, 4, 15, 16].

This paper proposes an approach for treating YAML documents as *first-class models* in MDE languages, by providing support to seamlessly integrate them in MDE processes alongside traditional models such as EMF-based models. Despite the fact that from an MDE technical perspective, YAML is inferior compared to MOF and Ecore for elaborating object-oriented metamodelling architectures (see Section 2), due to its simplicity and popularity, it has the potential to lower the entry barrier and can widen the adoption of automated model management and MDE. This work aims to make model management languages and MDE techniques more accessible to YAML-literate engineers, by contributing a driver that enables the management of schema-less YAML documents within Eclipse Epsilon [11], a mature and well-established family of model management languages and tools. We evaluate our approach on an industrial case study related to cloud infrastructure automation, by using model transformations for transforming EMF-based models into (YAML-based) Ansible Playbooks.

Section 2 introduces the necessary background. Section 3 proposes an approach to provide first-class support for YAML documents in model management languages. Section 4 illustrates an industrial case study in which the proposed approach is evaluated.

Section 5 presents related work and, finally, Section 6 concludes the paper, and provides directions for future work on this subject.

2 BACKGROUND

YAML[1] is a case-sensitive data serialisation language designed to be human-friendly and to improve readability by minimising the amount of structural characters, and allowing data to be structured in a natural and meaningful way through indentation. In the absence of a schema, it is technically inferior compared to contemporary metamodelling architectures such as EMF and MOF, as it lacks support for types. However, a YAML schema can be used to remedy this limitation, by defining the general structure to which the document must adhere.

Three basic primitives are supported by YAML: *scalars*, *mappings* (hashes or dictionaries) and *sequences* (arrays or lists). Scalars are used to represent a single atomic value (e.g., string, integer). Mappings represent unordered collections of unique key-to-value associations, similar to a dictionary or a map. The key-value pairs of a mapping can either be written on separate lines, or they can be written using an abbreviated form, as a comma-separated single line enclosed within curly braces. Sequences are used to represent an ordered list of nodes. Each entry from a list is defined on a separate line with a dash at the same indentation level, or alternatively, a list can be defined using an abbreviated form, as a comma-separated single line enclosed within square brackets.

YAML also supports more advanced features such as tags, anchors and aliases. Tags are used to associate metadata to nodes [39]. Anchors and aliases provide the ability to reference and use the same data multiple times within a single YAML document.

3 MANAGING YAML DOCUMENTS

This section presents an approach to provide support for model management of models captured in YAML within Eclipse Epsilon. Listing 1 will be used as a running example, representing a YAML document that defines a project plan, describing the involved people, tasks and effort allocation.

3.1 Eclipse Epsilon

Eclipse Epsilon [11, 18] is a framework of interoperable Java-based languages and tools that aim to simplify model-based software engineering tasks. Epsilon supports multiple types of model management operations, such as model-to-model transformation via the Epsilon Transformation Language (ETL) [20], model-to-text transformation via the Epsilon Generation Language (EGL) [34] and model validation via the Epsilon Validation Language (EVL) [21], among others.

The Epsilon Object Language (EOL) [19] is the core expression language used by Epsilon languages, and can additionally be used as a general-purpose standalone model management language for automating other tasks than the ones handled by the previously specified Epsilon languages. EOL provides features such as model modification, multiple model access, conventional programming constructs (e.g., variables, loops, branches), user interaction, profiling, and support for transactions [22].
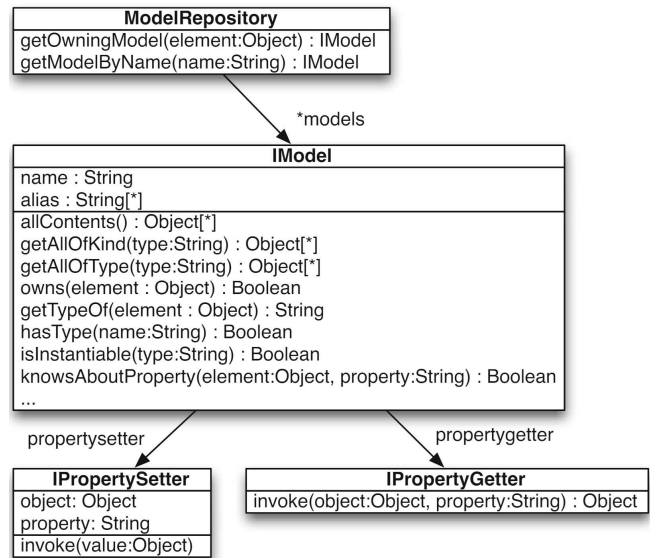
---

[1]https://yaml.org/

Figure 1: The Epsilon Model Connectivity Layer

The Epsilon Model Connectivity (EMC) [10] layer provides an interface for uniformly interacting with models conforming to a wide variety of technologies, e.g., the Eclipse Modelling Framework (EMF), XML, CSV, MATLAB Simulink. The concrete implementations that enable Epsilon to support various technologies are called EMC drivers, which are essentially classes that implement the `IModel` interface presented in the class diagram from Figure 1. Epsilon-based programs can perform model management on models of any technology for which an EMC driver is implemented.

3.2 EMC YAML Driver

An EMC driver has been implemented for YAML [12], allowing Epsilon-based programs to perform model management over YAML-based models. The implementation of the EMC driver relies on the SnakeYAML library [37]. Through the driver, a YAML document from the file system is loaded into memory as a model, which is then accessed, queried and modified. Finally, the in-memory representation of the model is serialised back to the file system as a YAML document, overwriting the original YAML document.

The EMC YAML driver supports the basic primitives of YAML: scalars, mappings and sequences. Note that the driver does not support more advanced features such as tags, anchors and aliases, in the sense that it ignores them and does not permit their manipulation. The driver follows the convention that each node consists of a key-value pair separated by a colon, where the type of the value dictates the type of the node. Therefore, a scalar node is a key-value pair having a value of type scalar, a mapping node is a key-value pair having a value of type mapping, and a list node is a key-value pair having a value of type list. For instance, line 2 from Listing 1 represents a scalar node, lines 3–5 represent a list node, and line 14 represents a mapping node. The root is considered a node with an empty key and its value is the entire YAML document. The value of the root node can be of type *scalar*, *mapping*, or *list*. The value of a list node is a list comprising either scalar values or other nodes.

In the following, we illustrate how Epsilon-based programs (e.g., EOL scripts) can use the EMC YAML driver to query and manipulate the YAML document in Listing 1. We consider the name of a node to be its key, e.g., the scalar node from line 2 has the name of *title*.

```
1   project:
2     title: IoT Home Automation
3     people:
4       - name: Alice
5       - name: Bob
6     tasks:
7       - title: Implementation
8         duration: 3
9         effort:
10          - person: Bob
11            percentage: 60
12          - person: Alice
13            percentage: 40
14    metadata: {year: 2024, reviewed: true}
```

Listing 1: Running Example — YAML Document

```
1   Node.all.println();
2   ScalarNode.all.println();
3   MappingNode.all.println();
4   ListNode.all.println();
```

Listing 2: Accessing all nodes

```
1   // last scalar node named "title"
2   s_title.all.last().println();
3
4   // first mapping node named "metadata"
5   m_metadata.all.first().println();
6
7   // first list node named "effort"
8   l_effort.all.first().println();
```

Listing 3: Accessing nodes by key and type

```
1   var node = s_duration.all.first();
2   node.name.println(); // duration
3   node.type.println(); // ScalarNode
4   node.value.println(); // 3
5   (node.s_value + 1).println(); // 31
6   (node.i_value + 1).println(); // 4
7   (node.d_value + 1).println(); // 4.0
8   (node.f_value + 1).println(); // 4.0
9   node.b_value.println(); // false
```

Listing 4: Accessing node properties

```
1   // name=Alice
2   var nameNode1 = new ScalarNode("name", "Alice");
3   var nameNode2 = new s_name("Alice");
4
5   // project={}
6   var projectNode1 = new MappingNode("project");
7   var projectNode2 = new m_project;
8
9   // tasks=[{}, {}]
10  var tasksNode1 = new ListNode("tasks", 2);
11  var tasksNode2 = new l_tasks(2);
```

Listing 5: Creating nodes

Accessing All Nodes. Listing 2 shows how to access all nodes of a specific type from a YAML document. Line 1 retrieves and prints a sequence containing all nodes from the YAML document, regardless of their type. Line 2 retrieves and prints a sequence containing all scalar nodes, i.e., the ones with the key of *title*, *name*, *duration*, *person*, *percentage*, *year* and *reviewed*. Note that line 2 does not retrieve the atomic scalar values inside a list node, as they are not considered nodes by the driver. To access the scalar values from a list node, one has to retrieve the list node and then iterate its *value* property. Line 3 prints all mapping nodes, i.e., those with the key of *project* and *metadata*. Line 4 prints all list nodes, i.e., those having the key of *people*, *tasks* and *effort*.

Accessing Nodes by Key and Type. Listing 3 shows how to query nodes by their key and type. The *s_* prefix followed by the key of the node is used to query all scalar nodes with the specified key. Similarly, the *m_* prefix is used for mapping nodes, and the *l_* prefix for list nodes, to query nodes with the specified key. Line 2 from Listing 3 prints the last scalar node named *title* (line 7 from Listing 1), as follows: "`title=Implementation`". Line 5 prints the first mapping node named *metadata* (line 14 from Listing 1), as follows: "`metadata={year=2024, reviewed=true}`". Line 8 prints the first list node named *effort* (lines 9–13 from Listing 1): "`effort=[{person=Bob, percentage=60}, {person=Alice, percentage=40}]`".

Accessing Node Properties. Listing 4 presents a set of properties that can be accessed on a node: *name*, *type*, *value*, *s_value*, *i_value*, *d_value*, *f_value* and *b_value*. The comments from the listing represent the output of each line. Line 1 from Listing 4 fetches the first scalar node named *duration* (line 8 from Listing 1). The *name* of the node is printed at line 2 from Listing 4. Line 3 prints the *type* of the node, i.e., *ScalarNode*. Line 4 prints the *value* of the node, whereas lines 5–9 fetch the value and then convert it to a different data type. Respectively, line 5 converts the value to a string, line 6 converts it to an integer, line 7 converts it to a double, line 8 converts it to a float and line 9 converts it to a boolean. If the value of the node named *duration* would have been a non-numeric string (e.g., "weeks"), casting it to the incorrect type would not have broken the program, e.g., casting the string to an integer would return a value of *1*, or casting it to a boolean would return a value of *false*. The conversion prefixes from lines 5–9 are necessary for managing schema-less YAML documents, as they have no typing information.

Creating Nodes. Listing 5 creates nodes of each type and the comments represent the output of printing each node. To instantiate a new node, the *new* operator must be used, followed by the type of the node, either *ScalarNode*, *MappingNode* or *ListNode*. Alternatively, the convention for accessing and querying nodes can be used for creating nodes. For example, using the *new* operator followed by *s_name* creates a scalar node with the key of *name*. Lines 2–3 create two identical scalar nodes with the key of *name* and value of *Alice*. Lines 6–7 create two identical mapping nodes with the key of *project* and an empty value. Finally, lines 10–11 create two identical list nodes named *tasks* containing two empty entries. The created mapping nodes and list nodes can be populated programmatically at a later point.

```
1   var s = new ScalarNode("status", "complete");
2
3   // set scalar node value
4   s_duration.all.first().value = 5;
5
6   // add scalar node to mapping node
7   m_metadata.all.first().value.appendNode(s);
8
9   // add scalar node to list node
10  l_tasks.all.first().value.addRow();
11  l_tasks.all.first().value.at(1).appendNode(s);
```

Listing 6: Modifying nodes

```
1   // delete all scalar nodes named "name"
2   delete s_name.all;
3
4   // delete the first mapping node named "metadata"
5   delete m_metadata.all.first();
6
7   // delete the first list node named "effort"
8   delete l_effort.all.first();
```

Listing 7: Deleting nodes

Modifying Nodes. Listing 6 presents a way for modifying a scalar node, a mapping node and a list node. Line 4 from Listing 6 sets the value of the first scalar node named *duration* (line 8 from Listing 1) to *5*. Line 7 from Listing 6 appends the scalar node *s*, which was defined at line 1, to the mapping node named *metadata* (line 14 from Listing 1). Line 10 from Listing 6 adds a new entry to the first list node named *tasks* (lines 6–13 from Listing 1), and line 11 from Listing 6 appends the scalar node *s* to the newly added entry.

Deleting Nodes. Listing 7 deletes scalar nodes, a mapping node and a list node by using the *delete* operator. Line 2 from Listing 7 deletes all scalar nodes with the key of *name* (lines 4 and 5 from Listing 1). Line 5 from Listing 7 deletes the first mapping node named *metadata* (line 14 from Listing 1). Finally, line 8 from Listing 7 deletes the first list node named *effort* (lines 9–13 from Listing 1). Note that deletion occurs by value and not by reference, e.g., if the same scalar node is appended to multiple list nodes, and then one of those list nodes is deleted, then all other list nodes will still contain the appended scalar node.

Creating complete YAML documents. Listing 8 programmatically creates from scratch the complete YAML document from Listing 1. Note that at(index:Integer) is an operation[2] that returns the element at a specific index from a collection. Line 2 sets the YAML document to be a mapping node. Lines 5–20 from Listing 8 create all nodes corresponding to the data of the YAML document from Listing 1. Lines 23 and 24 populate the list node named *people*, lines 27–30 populate the list node named *effort* and lines 33–35 populate the list node named *tasks*. Lines 38 and 39 populate the mapping node named *metadata* and lines 42–45 populate the mapping node named *project*. Finally, line 48 adds the mapping node named *project* to the YAML document.

---
[2]Epsilon supports a range of built-in operations on collections and sequences: https://eclipse.dev/epsilon/doc/eol/#collections-and-maps

```
1   // set the YAML document as a mapping node
2   YAMLDoc.setRootAsMap();
3
4   // create all nodes from the YAML document
5   var project = new m_project;
6   var title0 = new s_title("IoT Home Automation");
7   var people = new l_people(2);
8   var name1 = new s_name("Alice");
9   var name2 = new s_name("Bob");
10  var tasks = new l_tasks(1);
11  var title = new s_title("Implementation");
12  var duration = new s_duration(3);
13  var effort = new l_effort(2);
14  var person1 = new s_person("Bob");
15  var person2 = new s_person("Alice");
16  var percentage1 = new s_percentage(60);
17  var percentage2 = new s_percentage(40);
18  var metadata = new m_metadata;
19  var year = new s_year(2024);
20  var reviewed = new s_reviewed(true);
21
22  // populate the "people" list node
23  people.value.at(0).appendNode(name1);
24  people.value.at(1).appendNode(name2);
25
26  // populate the "effort" list node
27  effort.value.at(0).appendNode(person1);
28  effort.value.at(0).appendNode(percentage1);
29  effort.value.at(1).appendNode(person2);
30  effort.value.at(1).appendNode(percentage2);
31
32  // populate the "tasks" list node
33  tasks.value.at(0).appendNode(title);
34  tasks.value.at(0).appendNode(duration);
35  tasks.value.at(0).appendNode(effort);
36
37  // populate the "metadata" mapping node
38  metadata.value.appendNode(year);
39  metadata.value.appendNode(reviewed);
40
41  // populate the "project" mapping node
42  project.value.appendNode(title0);
43  project.value.appendNode(people);
44  project.value.appendNode(tasks);
45  project.value.appendNode(metadata);
46
47  // add the "project" node to the YAML document
48  YAMLDoc.getRoot().value.appendNode(project);
```

Listing 8: Creating a complete YAML document from scratch

4 CASE STUDY

The use case is an industrial case study provided by NetApp [29], a global software company that delivers hybrid cloud data services and data management services. Note that a part of this case study has been briefly presented in [32].

4.1 Description

Infrastructure automation can be enabled using a DevOps practice called Infrastructure as Code (IaC). The management and provisioning of infrastructure is performed through definition files that contain declarative code. Once the infrastructure is defined using code, it is rolled out to systems through automated processes [24]. The main benefit of IaC is automation, as it automates the deployment, configuration, and management of infrastructure, and this prevents unforeseen issues that can be caused by human errors.

One of the most widely used infrastructure automation technologies today is Ansible [1]. Ansible is an automation and orchestration
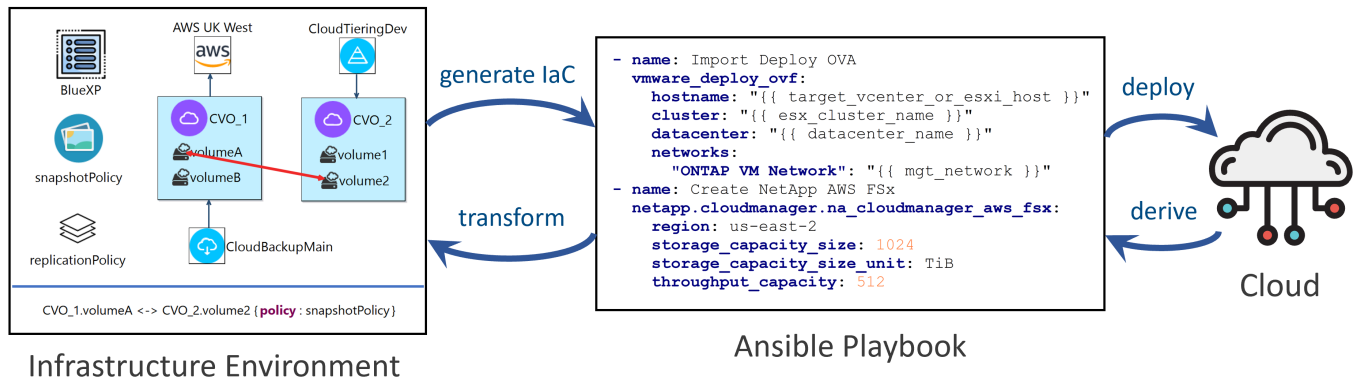
Figure 2: Case Study — Model Transformations

tool used for software provisioning, configuration management, and application deployment that delivers IaC. One describes the desired end state of a system, and then Ansible's processes reliably configure the system to the desired end state [38]. Ansible Playbooks are the core component of Ansible, and their syntax is based on YAML.

For infrastructure automation, NetApp uses Ansible Playbooks, among other technologies. NetApp's Ansible Playbooks comprise data of infrastructure components and are consumed by Python-based scripts which deploy the described infrastructures to the cloud. Ansible Playbooks often contain hundreds of lines of code, therefore it can be time-consuming to get a clear high-level mental image of an infrastructure environment. Note that documentation and examples of concrete Ansible Playbooks of NetApp are publicly available in [25]. Although Ansible Playbooks use the human-readable syntax of YAML, professionals without a technical background often have difficulties in fully understanding them. As such, there is some collective hesitation and lack of trust in the adoption of automation, therefore, manual operations are generally preferred when planning for complex infrastructure environments.

The design and configuration of hybrid multi-cloud infrastructure environments is complex and requires specialist cloud computing expertise. When planning enterprise data storage environments, a multitude of parties are involved, such as security experts, cloud and storage architects, and operational and legal teams. Due to the involvement of some non-technical professionals, there is often a lack of shared understanding of the described infrastructure between these parties. Therefore, instead of using Ansible Playbooks for the automation of infrastructure environments, it would be beneficial to use a set of higher-level abstractions that could potentially be understood by all parties involved and can lower the entry barrier to the adoption of cloud services. To this end, a dedicated hybrid graphical-textual DSL has been developed, that covers a set of higher-level abstractions over NetApp's Public Cloud Services (PCS). Note that a hybrid graphical-textual DSL operates over a single semantic model through a part-graphical and part-textual syntax [31]. Such DSLs are effectively used through hybrid graphical-textual model editors, as they allow for editing some parts

of the model through graphical representations and other parts of the model through textual representations.

The DSL is used for modelling infrastructure specifications, as it provides a graphical syntax for the abstraction of high-level infrastructure components and textual syntaxes for defining behaviour and additional lower-level details. NetApp's staff and customers that are involved in the planning phase of enterprise infrastructure environments can quickly get a high-level understanding of an environment by looking at the modelled infrastructure specification.

The workflow of the desired solution for the case study is illustrated in Figure 2. The solution is divided into a forward engineering process, a reverse engineering process, and a round-trip engineering process. The rationale of the desired solution is to leverage model transformations for automatically generating IaC (i.e., an Ansible Playbook) that is equivalent to the infrastructure specification defined in the hybrid graphical-textual model editor, and vice versa.

In the forward engineering process, a storage designer will model the infrastructure environment using the hybrid graphical-textual DSL. The modelled infrastructure specification will be used to generate a corresponding Ansible Playbook. Optionally, a DevOps engineer may manually edit the Ansible Playbook for fine-grained adjustments or for including sensitive credentials. Next, the Ansible Playbook is executed, and the corresponding infrastructure components will be automatically deployed and configured in the cloud.

In the reverse engineering process, an Ansible Playbook is derived according to the infrastructure components managed in the cloud. As in the forward engineering process, a DevOps engineer can optionally edit the Ansible Playbook. Finally, by using the Ansible Playbook as input, an infrastructure specification will be derived and displayed in the hybrid graphical-textual model editor.

The round-trip engineering process is an ongoing and bidirectional process, as each change in the modelled infrastructure environment will be directly reflected in the Ansible Playbook, and each change in the Ansible Playbook will be reflected in the modelled infrastructure environment displayed in the hybrid graphical-textual model editor.
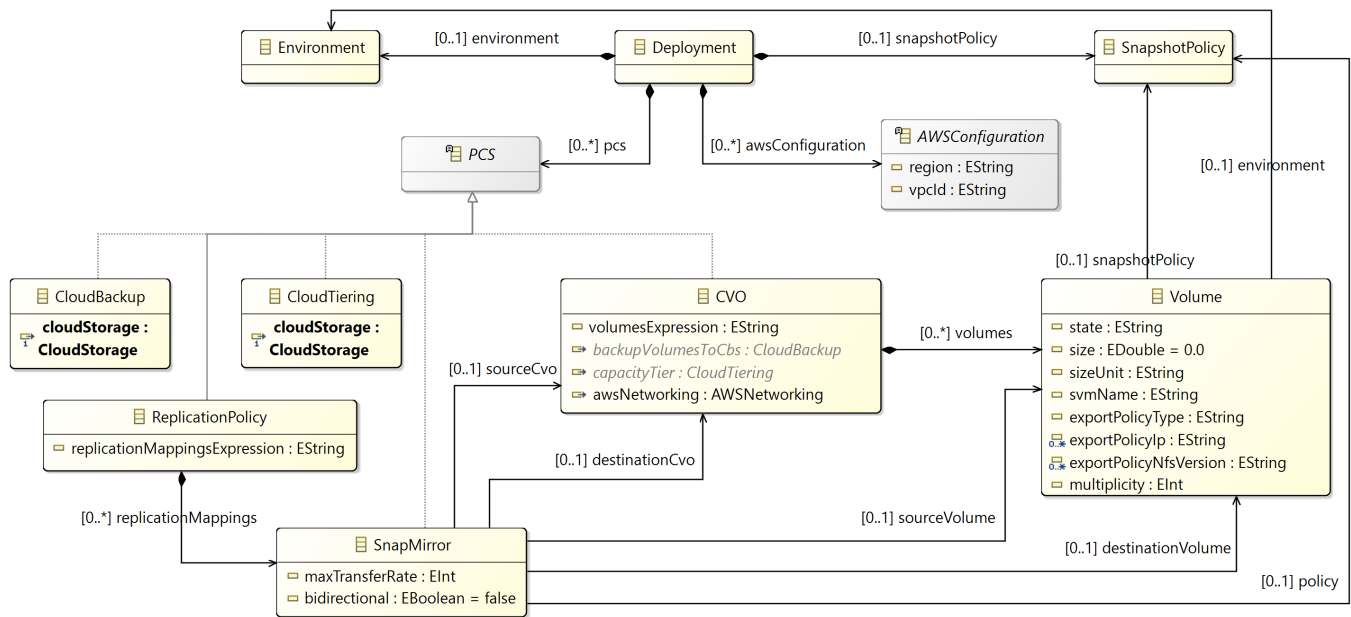
Figure 3: Metamodel Excerpt of the *Infrastructure Services DSL*

4.2 Infrastructure Services DSL

The DSL is used for modelling the infrastructure services and components that will be deployed to the cloud using IaC. Figure 3 illustrates a metamodel excerpt of the developed DSL, which contains the domain's core abstractions. The DSL is based on EMF: accordingly, for the solution of the case study we interact with EMF-based models conforming to the Ecore metamodel that is described below.

Open Network Technology for Appliance Products (ONTAP) [28] is NetApp's proprietary operating system that provides optimised storage functions, and it can be deployed on physical or virtual appliances. Cloud Volumes ONTAP (CVO) [27] is a cloud instance of ONTAP. A *Deployment* configuration contains a *SnapshotPolicy* and a list of infrastructure services of type *PCS* that are deployed to an *Environment* (e.g., NetApp BlueXP [26]), in a specific public cloud (e.g., AWS). The types of infrastructure services are: *CVO*, *CloudBackup*, *CloudTiering*, *ReplicationPolicy* and *SnapMirror*. A *CVO* instance contains a list of *volumes* that define logical storage areas. A *CloudBackup* service provides a single control plane that facilitates the implementation of custom and efficient backup and recovery strategies. A *CloudTiering* service automatically tiers inactive data from on-premises ONTAP clusters to cloud object storage. *SnapMirror* is a proprietary protocol that replicates data from a source *volume* to a target *volume* of a *CVO* instance, based on a *SnapshotPolicy*. A *ReplicationPolicy* defines a replication strategy comprising *replication mappings* of type *SnapMirror*.

4.3 Model Transformations

In the context of the case study, we implemented the forward engineering process by leveraging the EMC YAML driver. The reverse and round-trip engineering processes have not been implemented

yet in our work. Accordingly, we demonstrate how the driver enables model management, by using a model-to-model transformation to transform an EMF-based model conforming to the metamodel of the DSL into a YAML model, representing an Ansible Playbook. By using a model-to-model transformation such as the one from Listing 9, we are able to transform an EMF model as the one from Listing 10, into the YAML model from Listing 11, which represents an Ansible Playbook used by NetApp engineers to create and deploy *CVO* instances to the cloud. Note that Listing 9 does not represent the complete model-to-model transformation, but rather a relevant excerpt. Likewise, Listing 10 does not represent the complete EMF model, as it includes only the relevant model elements and attributes for showcasing the model transformation from Listing 9.

Listing 9 is written in ETL, containing a set of transformation rules that specify how to transform EMF model elements into their YAML counterparts. The transformation rule at lines 1–13 specifies that a *Deployment* model element, i.e., the root of the EMF model, must be transformed into a list node representing the root of the YAML model, containing one entry for each *CVOConfiguration*. The EMF model contains two *CVOConfiguration* model elements (*CVO* and *CVO_HA* are subtypes of *CVOConfiguration*), therefore, the list node representing the root of the YAML model is populated with two entries: lines 1–16 from Listing 11 represent the first entry and lines 17–24, the second.

The transformation rules at lines 15–69 specify how EMF model elements of type *CVO_HA*, *AWSNetworkingHA*, *CloudBackup* and *CloudTiering* are transformed into corresponding mapping nodes, whose entries are appended to the root list node of the YAML model. For brevity, Listing 9 does not include the transformation rules for the EMF model elements of type *CVO* and *AWSNetworking*, as they are similar to the ones for *CVO_HA* and *AWSNetworkingHA*. Note

```
1   rule Deployment_Rule
2     transform x : Emf!Deployment
3     to y : Yaml!ListNode {
4       if (x.pcs.isDefined()) {
5         for (pcs_item in x.pcs) {
6           if(pcs_item.isKindOf(Emf!CVOConfiguration)) {
7             y.value.addRow();
8             y.value.last().appendEntries(pcs_item.equivalent().value);
9           }
10        }
11      }
12      Yaml.root.value = y;
13  }
14
15  rule CVO_HA_Rule
16    transform x : Emf!CVO_HA
17    to y : Yaml!MappingNode {
18      y.value.appendNode(new Yaml!s_name("Create NetApp CVO for AWS HA"));
19      var configuration_node = new Yaml!`m_netapp.cloudmanager.na_cloudmanager_cvo_aws`;
20      configuration_node.value.appendNode(new Yaml!s_is_ha(true));
21      if (x.name.isDefined())
22        configuration_node.value.appendNode(new Yaml!s_name(x.name));
23      if (x.awsnetworkingha.isDefined())
24        configuration_node.value.appendEntries(x.awsnetworkingha.equivalent().value);
25      if (x.backup_volumes_to_cbs.isDefined())
26        configuration_node.value.appendEntries(x.backup_volumes_to_cbs.equivalent().value);
27      if (x.capacity_tier.isDefined())
28        configuration_node.value.appendEntries(x.capacity_tier.equivalent().value);
29      y.value.appendNode(configuration_node);
30  }
31
32  rule AWSNetworkingHA_Rule
33    transform x : Emf!AWSNetworkingHA
34    to y : Yaml!MappingNode {
35      if (x.region.isDefined())
36        y.value.appendNode(new Yaml!s_region(x.region));
37      if (x.vpc_id.isDefined())
38        y.value.appendNode(new Yaml!s_vpc_id(x.vpc_id));
39      if (x.node1_subnet_id.isDefined())
40        y.value.appendNode(new Yaml!s_node1_subnet_id(x.node1_subnet_id));
41      if (x.node2_subnet_id.isDefined())
42        y.value.appendNode(new Yaml!s_node2_subnet_id(x.node2_subnet_id));
43      if (x.failover_mode.isDefined())
44        y.value.appendNode(new Yaml!s_failover_mode(x.failover_mode));
45      if (x.mediator_subnet_id.isDefined())
46        y.value.appendNode(new Yaml!s_mediator_subnet_id(x.mediator_subnet_id));
47      if (x.mediator_key_pair_name.isDefined())
48        y.value.appendNode(new Yaml!s_mediator_key_pair_name(x.mediator_key_pair_name));
49      if (x.cluster_floating_ip.isDefined())
50        y.value.appendNode(new Yaml!s_cluster_floating_ip(x.cluster_floating_ip));
51      if (x.data_floating_ip.isDefined())
52        y.value.appendNode(new Yaml!s_data_floating_ip(x.data_floating_ip));
53      if (x.data_floating_ip2.isDefined())
54        y.value.appendNode(new Yaml!s_data_floating_ip2(x.data_floating_ip2));
55      if (x.svm_floating_ip.isDefined())
56        y.value.appendNode(new Yaml!s_svm_floating_ip(x.svm_floating_ip));
57  }
58
59  rule CloudBackup_Rule
60    transform x : Emf!CloudBackup
61    to y : Yaml!MappingNode {
62      y.value.appendNode(new Yaml!s_backup_volumes_to_cbs("yes"));
63  }
64
65  rule CloudTiering_Rule
66    transform x : Emf!CloudTiering
67    to y : Yaml!MappingNode {
68      y.value.appendNode(new Yaml!s_capacity_tier("S3"));
69  }
70
71  operation Native("java.util.LinkedHashMap") appendEntries(source : Native("java.util.LinkedHashMap")) {
72    for (entry in source.entrySet())
73      self.appendNode(entry);
74  }
```

Listing 9: Model-to-Model Transformation in ETL from an EMF Model to a YAML Model

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <pcs:Deployment xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:pcs="pcs.netapp.org">
3       <environment name="BlueXP"/>
4       <snapshotPolicy name="snapshotPolicyDefault"/>
5       <pcs xsi:type="pcs:CloudBackup" name="CloudBackupMain" cloudstorage="CVO_1"/>
6       <pcs xsi:type="pcs:CloudTiering" name="CloudTieringDev" cloudstorage="CVO_HA_2"/>
7       <pcs xsi:type="pcs:CVO" name="CVO_1" backup_volumes_to_cbs="CloudBackupMain" awsnetworking="04674988-a733".../>
8       <pcs xsi:type="pcs:CVO_HA" name="CVO_HA_2" capacity_tier="CloudTieringDev" awsnetworkingha="604aae99-e8e8".../>
9       <awsconfiguration xsi:type="pcs:AWSNetworking" region="UK West" vpc_id="2fd70595-653d" subnet_id="04674988-a733"
10          cvo="CVO_1"/>
11      <awsconfiguration xsi:type="pcs:AWSNetworkingHA" region="US North" vpc_id="4d0ad751-ae69" cvo_ha="CVO_HA_2"
            node1_subnet_id="604aae99-e8e8" node2_subnet_id="c0136b8f-ee50" failover_mode="PrivateIP" mediator_subnet_id=
12          "0301dedc-d542" mediator_key_pair_name="mediator_key" cluster_floating_ip="237.95.233.7" data_floating_ip=
13          "6.118.90.145" data_floating_ip2="6.118.90.146" svm_floating_ip="234.119.77.179"/>
14      ...
15  </pcs:Deployment>
```

Listing 10: EMF Model in XMI Format Conforming to the Metamodel of the DSL

```
1   - name: Create NetApp CVO for AWS HA
2     netapp.cloudmanager.na_cloudmanager_cvo_aws:
3       is_ha: true
4       name: CVO_HA_2
5       region: US North
6       vpc_id: 4d0ad751-ae69
7       node1_subnet_id: 604aae99-e8e8
8       node2_subnet_id: c0136b8f-ee50
9       failover_mode: PrivateIP
10      mediator_subnet_id: 0301dedc-d542
11      mediator_key_pair_name: mediator_key
12      cluster_floating_ip: 237.95.233.7
13      data_floating_ip: 6.118.90.145
14      data_floating_ip2: 6.118.90.146
15      svm_floating_ip: 234.119.77.179
16      capacity_tier: S3
17  - name: Create NetApp CVO for AWS single
18    netapp.cloudmanager.na_cloudmanager_cvo_aws:
19      is_ha: false
20      name: CVO_1
21      region: UK West
22      vpc_id: 2fd70595-653d
23      subnet_id: 04674988-a733
24      backup_volumes_to_cbs: 'yes'
```

Listing 11: Ansible Playbook for Creating *CVO* Instances

that model element types with "*HA*" as a suffix are related to high availability. Furthermore, lines 71–74 define a utility method that appends the entries of a mapping node to the value of another mapping node, which is called by the transformation rules from lines 1–30.

Some data is lost during the forward engineering process, e.g., the transformation rule at lines 65–69 from Listing 9 produces from a model element of type *CloudTiering*, a scalar node with a hard-coded value (i.e., capacity_tier:S3), therefore the properties of the *CloudTiering* model element are lost. For the reverse engineering process, an opposite transformation of the one from Listing 9 would be required, to transform a YAML model into an EMF model. However, the reverse transformation must recover any data that was lost during the forward engineering process, e.g., a scalar node with the key of *capacity_tier* would have to be transformed into a *CloudTiering* model element having empty or predefined values for its properties. A challenge that we may encounter in the implementation of the round-trip engineering process is to maintain constant identifiers in the EMF model, to avoid losing visual information (e.g.,

diagram coordinates) of a model element displayed in a graphical diagram. Moreover, a challenge related to synchronisation could also arise, as merging the EMF model with the previously generated YAML model can be non-trivial in the case they both have been modified simultaneously.

5 RELATED WORK

YAML Documents as Models. At the 2023 edition of the Transformation Tools Contest, the case described in [15] presented the challenge of implementing asymmetric and directed bidirectional transformations between a YAML model and a model of Docker Compose specifications. An EMF-based reference solution is proposed in [15], which consists of model-to-model transformations that transform a model conforming to a *Containers DSL* metamodel to a model conforming to a metamodel of a simplified YAML specification, and vice versa. The implementation relies on a utility function based on SnakeYAML [37] for converting YAML documents into EMF models. Three papers proposed solutions to the challenge presented in [15], implementing asymmetric and directed bidirectional transformations involving YAML documents, by using YAMTL and EMF-SYNCER [2], BXtendDSL [4], and the .NET Modelling Framework (NMF) [16]. However, the approaches from [2, 4, 15, 16] do not describe a generic and uniform technique for interacting with YAML documents, such as the EMC YAML driver.

The work from [3] adopts the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard for modelling cloud resources in a technology-independent way. To this end, Xtext is used to define YAML-like textual syntaxes that are used to represent underlying models conforming to the TOSCA metamodel, a Docker Compose-inspired metamodel and a Dockerfile-inspired metamodel. This work proposes a model-driven translation technique that transforms TOSCA artefacts into native DevOps-specific artefacts, by using model-to-model and model-to-text transformations for transforming TOSCA models into Compose and Dockerfile models stored in YAML format. A similar model-driven approach is implemented in [6], which involves in addition to TOSCA, the Open Cloud Computing Interface (OCCI), a standardised interface for managing cloud resources that aims to avoid cloud provider

lock-in. In this work, the YamlBeans [40] library is used for translating YAML documents into TOSCA models, which are used in a model-driven cloud orchestration process involving OCCI models.

EMC Drivers. Similar research efforts have been carried out in [14, 22, 35, 41], by implementing EMC drivers to extend Eclipse Epsilon to support a wider range of metamodelling technologies. The methodology behind these works is similar to ours, as they essentially enable model management over models captured in a different representation format.

In the work from [22], an EMC driver has been implemented to enable the interaction with schema-less XML documents. As a result, Epsilon-based programs can perform model management operations on top of plain XML documents. The driver has comparable capabilities to the EMC YAML driver, and it uses a similar convention for accessing and querying nodes, by using prefixes followed by names.

An EMC driver has been implemented in [14], with the aim of treating spreadsheets as models in MDE processes. Therefore, Epsilon-based programs can access, query and modify spreadsheets as if they were models. The driver treats worksheets as model element types, columns as their properties and rows as concrete model elements.

Additional support has been added to Epsilon for managing MATLAB Simulink models [35] and PTC Integrity Modeller (IM) models [41], to avoid the need to first transform them into EMF-compatible representations.

6 CONCLUSIONS AND FUTURE WORK

We advocated in this paper for the importance of adding support for YAML documents to the MDE toolkit, as a means of lowering the entrance barrier for newcomers in MDE. We proposed an approach to add first-class support for YAML to the Eclipse Epsilon framework such that schema-less YAML documents can seamlessly be used in model management operations. Our approach was evaluated in an industrial case study on cloud infrastructure automation, by implementing a forward engineering process that generates YAML models from EMF models, to produce Ansible Playbooks.

In future work, we plan to implement the reverse and round-trip engineering processes, in the context of the case study, which will involve tackling challenges related to synchronisation and constant model element identifiers. In addition, we plan to evaluate our approach with other MDE tasks such as model validation, comparison and code generation.

ACKNOWLEDGMENT

REFERENCES

[1] Ansible. 2024. *Website.* [Online]. Available: https://www.ansible.com, (Last Accessed: 2024-08-15).

[2] Artur Boronat. 2023. Asymmetric and Directed Bidirectional Transformation for Container Orchestrations with YAMTL and EMF-Syncer. In *15th Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF).*

[3] Hayet Brabra, Achraf Mtibaa, Walid Gaaloul, Boualem Benatallah, and Faiez Gargouri. 2019. Model-Driven Orchestration for Cloud Resources. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD).* IEEE, 422–429.

[4] Thomas Buchmann. 2023. A BXtendDSL Solution to the TTC2023 Asymmetric and Directed Bidirectional Transformation for Container Orchestrations Case. In *15th Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF).*

[5] Domenico Calcaterra, Vincenzo Cartelli, Giuseppe Di Modica, and Orazio Tomarchio. 2018. A Framework for the Orchestration and Provision of Cloud Services Based on TOSCA and BPMN. In *International Conference on Cloud Computing and Services Science (CLOSER).* Springer, 262–285.

[6] Stéphanie Challita, Fabian Korte, Johannes Erbel, Faiez Zalila, Jens Grabowski, and Philippe Merle. 2021. Model-based cloud resource management with TOSCA and OCCI. *Software and Systems Modeling* (2021), 1–23.

[7] Alberto Hernández Chillón, Diego Sevilla Ruiz, Jesus García Molina, and Severino Feliciano Morales. 2019. A Model-Driven Approach to Generate Schemas for Object-Document Mappers. *IEEE Access* 7 (2019), 59126–59142.

[8] Robert A DeLine. 2021. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* 1–11.

[9] Amirhossein Deljouyi and Raman Ramsin. 2022. MDD4REST: Model-Driven Methodology for Developing RESTful Web Services. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,.* INSTICC, SciTePress, 93–104.

[10] Eclipse Epsilon. 2024. *EMC Layer.* [Online]. Available: https://eclipse.dev/epsilon/doc/emc, (Last Accessed: 2024-08-15).

[11] Eclipse Epsilon. 2024. *Website.* [Online]. Available: https://eclipse.dev/epsilon, (Last Accessed: 2024-08-15).

[12] EMC YAML Driver. 2022. *Repository.* [Online]. Available: https://github.com/epsilonlabs/emc-yaml, (Last Accessed: 2024-08-15).

[13] Vincenzo Ferme and Cesare Pautasso. 2017. Towards Holistic Continuous Software Performance Assessment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion.* 159–164.

[14] Mārtiņš Francis, Dimitrios S Kolovos, Nicholas Matragkas, and Richard F Paige. 2013. Adding Spreadsheets to the MDE Toolkit. In *Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS).* Springer, 35–51.

[15] Antonio Garcia-Dominguez. 2023. Asymmetric and Directed Bidirectional Transformation for Container Orchestrations. In *15th Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF).*

[16] Georg Hinkel. 2023. An NMF Solutions to the TTC2023 Containers to MiniYAML Case. In *15th Transformation Tool Contest (TTC) part of the Software Technologies: Applications and Foundations (STAF).*

[17] Pierre Kelsen, Qin Ma, and Christian Glodt. 2020. A Lightweight Modeling Approach Based on Functional Decomposition. *Journal of Object Technology* 19, 2 (2020).

[18] Dimitrios Kolovos. 2008. *An Extensible Platform for Specification of Integrated Languages for Model Management.* Ph. D. Dissertation. University of York.

[19] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2006. The Epsilon Object Language (EOL). In *Model Driven Architecture – Foundations and Applications (ECMDA-FA).* Springer, 128–142.

[20] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations: First International Conference, ICMT 2008. Proceedings 1.* 46–60.

[21] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2009. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday.* 204–218.

[22] Dimitrios S Kolovos, Louis M Rose, James Williams, Nicholas Matragkas, and Richard F Paige. 2012. A Lightweight Approach for Managing XML Documents with MDE Languages. In *Modelling Foundations and Applications: 8th European Conference, ECMFA 2012. Proceedings 8.* 118–132.

[23] YA Mensah, M Agbaje, A Izang, OF Ajayi, O Bamidele, and AI Amusa. 2023. Reactive Code Generation for Modular Web Engineering (MWE) Framework. *International Journal of Scientific Research and Engineering Development* 6, 5 (2023).

[24] Kief Morris. 2016. *Infrastructure as Code: Managing Servers in the Cloud, Part I. Foundations.* "O'Reilly Media".

[25] NetApp. 2024. *Ansible Playbooks Collection.* [Online]. Available: https://docs.ansible.com/ansible/latest/collections/netapp, (Last Accessed: 2024-08-15).

[26] NetApp. 2024. *BlueXP.* [Online]. Available: https://www.netapp.com/bluexp, (Last Accessed: 2024-08-15).

[27] NetApp. 2024. *Cloud Volumes ONTAP.* [Online]. Available: https://bluexp.netapp.com/ontap-cloud, (Last Accessed: 2024-08-15).

[28] NetApp. 2024. *ONTAP.* [Online]. Available: https://www.netapp.com/data-management/ontap-data-management-software, (Last Accessed: 2024-08-15).

[29] NetApp. 2024. *Website.* [Online]. Available: https://www.netapp.com, (Last Accessed: 2024-08-15).

[30] Bruno Piedade, João Pedro Dias, and Filipe F Correia. 2020. An empirical study on visual programming docker compose configurations. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS-C)*. 1–10.

[31] Ionut Predoaia. 2023. Towards Systematic Engineering of Hybrid Graphical-Textual Domain-Specific Languages. In *Proceedings of the 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS-C)*. IEEE, 153–158.

[32] Ionut Predoaia, Dimitris Kolovos, Matthias Lenk, and Antonio García-Domínguez. 2023. Streamlining the Development of Hybrid Graphical-Textual Model Editors for Domain-Specific Languages. *Journal of Object Technology* 22, 2 (2023), 1–14.

[33] Zahra Rajaei, Shekoufeh Kolahdouz-Rahimi, Massimo Tisi, and Frédéric Jouault. 2021. A DSL for Encoding Models for Graph-Learning Processes. In *20th International Workshop on OCL and Textual Modeling*.

[34] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. 2008. The Epsilon Generation Language. In *Model Driven Architecture–Foundations and Applications: 4th European Conference, ECMDA-FA 2008. Proceedings 4*. 1–16.

[35] Beatriz Sánchez, Athanasios Zolotas, Horacio Hoyos Rodriguez, Dimitris Kolovos, and Richard Paige. 2019. On-the-fly Translation and Execution of OCL-like

[36] Anthony Savidis and Constantine Stephanidis. 2006. From requirements to source code: a Model-Driven Engineering approach for RESTful web services. *Automated Software Engineering* 13, 2 (2006), 303–339.

[37] Snake YAML. 2024. *Repository*. [Online]. Available: https://bitbucket.org/snakeyaml/snakeyaml, (Last Accessed: 2024-08-15).

[38] Sesto Vincent. 2020. *Practical Ansible: Configuration Management from Start to Finish*. Apress.

[39] YAML. 2021. *Specification v1.2.2*. [Online]. Available: https://yaml.org/spec/1.2.2, (Last Accessed: 2024-08-15).

[40] YamlBeans. 2024. *Repository*. [Online]. Available: https://github.com/EsotericSoftware/yamlbeans, (Last Accessed: 2024-08-15).

[41] Athanasios Zolotas, Horacio Hoyos Rodriguez, Stuart Hutchesson, Beatriz Sanchez Pina, Alan Grigg, Mole Li, Dimitrios S Kolovos, and Richard F Paige. 2020. Bridging proprietary modelling and open-source model management tools: the case of PTC Integrity Modeller and Epsilon. *Software and Systems Modeling* 19 (2020), 17–38.

Queries on Simulink Models. In *Proceedings of the 22nd IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 205–215.