Information Systems 125 (2024) 102422

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

DOML: A new modeling approach to Infrastructure-as-Code[☆]

Michele Chiari [a,*], Bin Xiang [b], Sergio Canzoneri [c], Galia Novakova Nedeltcheva [c], Elisabetta Di Nitto [c,*], Lorenzo Blasi [d], Debora Benedetto [d], Laurentiu Niculut [d], Igor Škof [e]

[a] TU Wien, Treitlstraße 3, Vienna, 1040, Austria
[b] CNRS@CREATE, 1 Create Way, Singapore, 138602, Singapore
[c] Politecnico di Milano, Piazza Leonardo Da Vinci 32, Milano, 20133, Italy
[d] Hewlett Packard Italiana s.r.l., Via Giuseppe Vittorio, Cernusco sul Naviglio, 20063, MI, Italy
[e] Ministrstvo za Javno Upravo Republika Slovenija, Direktorat za informatiko, Tržaška 21, Ljubljana, 1000, Slovenija

ARTICLE INFO

Recommended by Matthias Weidlich

Keywords:
DOML
Infrastructure-as-Code
DevOps
IaC Modeling languages
Multi-layer modeling approach
Evaluation

ABSTRACT

One of the main DevOps practices is the automation of resource provisioning and deployment of complex software. This automation is enabled by the explicit definition of *Infrastructure-as-Code* (IaC), i.e., a set of scripts, often written in different modeling languages, which defines the infrastructure to be provisioned and applications to be deployed.

We introduce the DevOps Modeling Language (DOML), a new Cloud modeling language for infrastructure deployments. DOML is a modeling approach that can be mapped into multiple IaC languages, addressing infrastructure provisioning, application deployment and configuration.

The idea behind DOML is to use a single modeling paradigm which can help to reduce the need of deep technical expertise in using different specialized IaC languages.

We present the DOML's principles and discuss the related work on IaC languages. Furthermore, the advantages of the DOML for the end-user are demonstrated in comparison with some state-of-the-art IaC languages such as Ansible, Terraform, and Cloudify, and an evaluation of its effectiveness through several examples and a case study is provided.

1. Introduction

Employing Infrastructure-as-Code (IaC) means creating and managing an IT infrastructure, typically composed of computational resources and multiple software layers, by defining and executing code written in some special-purpose programming languages [1].

Defining an entire IT infrastructure deployment through IaC introduces several advantages in terms of repeatability of actions, reusability, and speed. However, it requires deep knowledge of multiple IaC languages and frameworks, since each specific framework covers a specific aspect of the whole problem [2]. This causes a steep learning curve for non-technical users and even for expert practitioners migrating from other technologies. Moreover, the selection of a specific set of IaC frameworks, given the peculiarities of each individual language, tends to foster vendor lock-in.

In this paper, we propose a low-code approach to IaC, which makes the creation of infrastructural code more accessible and friendly to designers, developers and operators. We introduce the DevOps Modeling Language (DOML), which hides the specificity and technicalities of the current IaC solutions.

The DOML allows for a complete specification of a deployment from its applications and software services to the infrastructural components and services supporting them. DOML models are mainly structured in three layers. Specifically, software components (e.g., web servers and databases) are described in the *application layer*, abstracting away from the infrastructure on which they are supposed to run. Infrastructure components are specified in the *abstract infrastructure layer*, and then linked to the applications they are supposed to host. This layer models infrastructural facilities, such as virtual machines, networks, and containers, without referring to their actual concretization in specific

☆ This work has been funded by the EU Commission in the Horizon 2020 research and innovation programme under grant agreement No. 101000162 (PIACERE project), partially funded in the Horizon Europe programme under grant agreement No. 101107303 (CORPORA project), and partially funded by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19018]. The authors acknowledge TU Wien Bibliothek for financial support through its TU Wien Open Access Funding Programme.

* Corresponding authors.
E-mail addresses: michele.chiari@tuwien.ac.at (M. Chiari), elisabetta.dinitto@polimi.it (E. Di Nitto).

https://doi.org/10.1016/j.is.2024.102422
Received 11 November 2023; Received in revised form 3 June 2024; Accepted 24 June 2024
Available online 6 July 2024

technologies (e.g., AWS or OpenStack VMs, Docker containers). This aspect is tackled by the *concrete infrastructure layer*, where the user specifies the infrastructure components offered by the Cloud Service Provider (CSP).

This modeling approach comes out from a careful analysis of related works concerning IaC languages, as well as other Cloud modeling approaches [2] and a critical review of the requirements for the DOML, provided by practitioners from several companies, including HP Enterprise, Ericsson, and Prodevelop.

Following the idea of generating code from an abstract model that is at the heart of Model Driven Development (MDD) [3], DOML models are turned into actual deployments by the Infrastructural Code Generator (ICG), which produces IaCs executable in the existing and well-supported frameworks. Our first target IaCs are Terraform, Ansible, and the configuration of Docker Compose. Nevertheless, the same ICG could be extended to generate other IaC languages to target more applications and CSPs.

In this paper, we present the DOML language and its advantages, and an overview of the extension mechanisms, called DOML-e. These allow expert users to introduce new elements in the DOML, for instance, new cloud resources, and to modify the existing ones. We evaluate our approach by comparing its usage with the direct use of IaC languages such as Terraform [4] and Cloudify [5], which is a TOSCA dialect [6]. We show that DOML is complete enough to model a whole deployment by itself, while the other approaches require the simultaneous use of more than one IaC language. Moreover, we show that DOML is generally more concise than the competing approaches.

The DOML language and the ICG were introduced in an article presented at CAiSE 2023 [7]. In that publication we have presented also their preliminary evaluation by answering two research questions, the first one, RQ1, concerning DOML's adequacy for representing deployments and the second, RQ2, concerning the DOML's ability to target multiple CSPs or deployment execution platforms. The evaluation was based on a Wordpress deployment example. The present paper extends [7] with the following contributions:

- We present the latest version of the DOML language and the ICG, which have been improved and consolidated.
- We present the requirements that led to the development of the DOML, illustrating the reasons for our design choices.
- We introduce a Model Checker (DMC) to support the verification of DOML models.
- We introduce the extension mechanisms that allow users to add support for new components and resources to the language.
- We extend and improve the DOML experimental evaluation. In particular, we split RQ1 into two sub-RQs concerning DOML's adequacy for representing deployments and its comparison with the state-of-the-art, respectively. RQ2 now evaluates our framework's ability to map to different existing cloud frameworks and technologies, not only in terms of execution platforms (RQ2 in [7], now renamed RQ2a), but also in terms of generated IaC languages (RQ2b). We answer RQ2b through a new example that employs a container group, which requires the configuration and usage of Docker Compose. We also add an assessment of the newly introduced verification approach (RQ3). Finally, we investigate how users perceive the adoption of the DOML (RQ4) by presenting the results of a new real-world case study conducted in collaboration with the IT Directorate of the Slovenian Ministry of Public Administration.

The DOML and all components of the software framework supporting it are open source and available in a public repository.[1] They have been developed as part of the European project PIACERE.[2]

[1] https://git.code.tecnalia.com/piacere/public
[2] https://piacere-project.eu/

Paper structure. In Section 2 we review some of the state-of-the-art IaC approaches, highlighting the motivation behind ours. Section 3 provides an overview of the DOML by presenting the requirements that have guided its development, a simple case study that will be used as running example, the principles behind the DOML, and the corresponding modeling abstractions; Section 4 contextualizes the usage of the DOML in a workflow, presents the correctness verification approach adopted for the DOML based on a model checker, and the IaC generation mechanism. Section 5 describes the mechanisms defined in DOML to enable the possibility to extend the language. Section 6 presents an evaluation of the DOML and compares it with a number of state-of-the-art IaC approaches, and Section 7 provides discussion of the evaluation part. Finally, Section 8 concludes the paper.

2. Related work

Choosing the right approach for automating the provisioning of computational resources and deployment of application components is not an easy task. In fact, each of the available IaC frameworks covers different parts of the whole problem. As a result, multiple frameworks must be combined, resulting in the need for the DevOps teams to understand all such frameworks. IaC frameworks can be divided into the following four categories.

- *Deployment and configuration management* frameworks focus on automating the installation, setup and life cycle of software applications deployed on top of an existing infrastructure. Examples of such tools are Chef [8], Puppet [9] and Ansible [10]. While they have similar purposes, they are quite different from each other in terms of the defined IaC language and the corresponding execution semantics.
- *Infrastructure provisioning* frameworks focus on describing the infrastructural topology, defining the virtual or physical infrastructural elements and their configurations, and providing automated means of managing their life cycles. For example, Terraform [4] is a proprietary language with an associated executor. It allows users to define an infrastructure configuration; it keeps track of the actual configuration of the managed infrastructure and, when needed, aligns it with the defined configuration. TOSCA [6], instead, is an OASIS standard modeling language that aims at allowing users to specify any type of IT system through powerful abstraction mechanisms, consisting of abstract *node templates* that can be combined through inheritance. The TOSCA language is adopted by a variety of executors that define its operational semantics in different ways [2,11].
- *Virtualization/Containerization* tools provide automation in building and managing VM or container images. An example of such tools is Docker [12] that has become the de-facto standard for running container-based applications on-premises, in public and private cloud providers [13]. Docker solves issues related to application portability, as containerized applications carry on their dependencies. It lets users define the recipe to build a container image using a custom, domain-specific language.
- *Runtime Orchestration* tools automate the whole life cycle of container-based deployments, including scaling and other management operations. An important representative of this category is Kubernetes [14], which also provides its own IaC language.

In general, when managing a complex application, multiple of the mentioned frameworks must be used. For instance, the infrastructure to be provisioned (e.g., VMs, network elements, and firewalls) could be modeled and then created with Terraform or TOSCA plus its executors. Ansible (or Chef/Puppet) playbooks could be executed to deploy and configure applications on top of the created infrastructure. Given that most of the application components rely on external preexisting software layers, it is typically advisable to embed all needed elements within some containers. This calls for the usage of Docker or of a similar approach. Finally, if the user wants to have a dynamic management of the application at runtime, an orchestration framework will have to be adopted.

This scenario clearly requires experienced users who are proficient with multiple IaC languages and tools, and that are able to take advantage of the ample and scattered offers.

An initial approach that aims at reducing the learning curve in adopting any DevOps-relevant platform is presented in [11]. The basic idea is to model DevOps processes, platforms and languages and to exploit these models within the context of low-code environments to let non-experienced users exploit the defined platforms and languages. Other approaches apply model-driven engineering in the specific context of IaC development. For example, DICER [15] focuses on deployment and operation of big data applications. It consists of a UML-based Domain-Specific Language (DSL) and a generator to derive TOSCA code from it. The limit of this approach is that it assumes the existence of additional low-level scripts (in Chef or Ansible) taking care of the configuration of applications. Such scripts work underneath and are not exposed to nor are modifiable by the DevOps team through the DICER modeling framework. SODALITE [16] is another framework based on TOSCA. Its aim is to offer support and guidance in the creation of TOSCA blueprints through the usage of its defined DSL. Additionally, it supports the creation of Ansible scripts for deployment and configuration, and exploits semantics reasoning to help users in the modeling task. Despite this, the SODALITE approach still requires users to be proficient in both Ansible and TOSCA.

A more sophisticated approach is EDMM [17], an Essential Deployment MetaModel. It defines the main concepts that are common to multiple deployment and configuration management frameworks and allows users to exploit such concepts to define application models. Then, through some transformations, EDMM supports the generation of code in various IaC languages.

In the DOML approach proposed in this paper, we follow the EDMM idea of targeting multiple IaC languages, but we try to extend the scope of the approach beyond deployment and configuration. Currently, we are also able to handle infrastructure provisioning as well as containerization and runtime orchestration of containers. We offer a single modeling language and a smart IaC generation approach allowing inexperienced DevOps teams to manage all aspects of deployment and operation on different types of infrastructures. So, from the same model, we are able to produce IaC code in multiple pre-existing languages.

Radius [18] is a brand new open-source project that appears to share the objectives of the PIACERE DOML for what concerns the definition of a software model and the needed resources and its usage for supporting proper deployment. Radius builds on the top of Kubernetes, focusing mainly on the containerized applications. Compared to it, DOML seems to be more flexible and advanced as it can model both containerized and normal applications on general cloud infrastructures, and supports model verification and IaC code generation. Besides, DOML has its own extension mechanism DOML-e that allows its users to extend the modeling language.

Recently, new endeavors aiming at producing infrastructural code from natural language by exploiting some kind of generative AI are under development [19]. Such approaches have a significant potential in terms of reducing the time and effort needed to develop IaC. At the moment, though, they are able to deal only with specific tasks, such as, “create a Terraform code for provisioning a VM on AWS”, but not with the more general task to create all it is needed to support the operation of a complex application.

Table 1
Requirements on the general characteristics of DOML.

Req ID	Description	Priority
REQ63	DOML must be unambiguous	High
REQ62	DOML must support different views	High
REQ70	DOML should allow users to state correctness properties in a suitable sub-language (possibly, formal logic)	Medium
REQ76	DOML should allow the user to model information needed for each of the four considered DevOps activities (provisioning, configuration, deployment, orchestration)	High
REQ57	The DOML core engine could enable both forward and backward translations from DOML to IaC and vice versa	Low
REQ58	DOML should offer the modeling abstractions to define the outcomes of the optimization component	High
REQ59	DOML should allow users to define rules and constraints for redeployment, reconfiguration, and other mitigation actions	High
REQ36	DOML should enable writing infrastructure tests	Low
REQ111	Users should be given the possibility to incorporate in DOML the external custom IaC	High

3. The DevOps modeling language (DOML)

3.1. Requirements

The DOML modeling language has been conceived starting from the requirements that have been identified and expressed by the PIACERE project[3] team. The team is composed of technical partners aiming at developing the whole PIACERE framework and of case study owners that apply the PIACERE results, and the DOML in particular, to their application domain.

The PIACERE framework is composed of the DOML itself, the modeling tools (the IDE, the DOML Model Checker, the optimizer, the IaC Generator (ICG), the IaC scan runner) and the tools supporting deployment orchestration, monitoring and self-healing. The PIACERE case studies include complex systems in three different application domains: public administration information systems, port and marine logistic, and public safety on IoT devices connected to a 5G network.

Both technical and case study partners have contributed to the formulation of requirements for the DOML, each from its own perspective. The process adopted for requirement gathering and analysis has been an iterative one, where each requirement has been formulated by one of the partners and assigned to a unique label, analyzed and discussed by the whole group and, if accepted, assigned to a specific PIACERE component together with a priority level.

Table 1 lists the requirements concerning the general characteristics expected from DOML, together with their prioritization, while Table 2 presents those requirements that concern specific elements to be included in the modeling language, for instance, security groups and rules or non-functional requirements. Some requirements may appear cryptic because they contain terms that were agreed upon separately. In particular, in Table 1, REQ63 means that the semantics of DOML models should be defined so that each model describes a uniquely-identifiable set of deployed components. Moreover, REQ62 means that DOML models must describe a deployment by different abstraction layers, which results in the three layers described later in Section 3.4.

Industrial partners, such as HPE and 7Bulls, experts in cloud application deployment and in the use of IaC for customer projects, contributed requirements based on their experience, such as REQ36 and REQ76. Partners responsible for specific tools of the PIACERE framework, such as XLAB and TECNALIA, expressed requirements focused on modeling their tools’ input information, such as REQ58 and REQ59.

[3] https://piacere-project.eu/

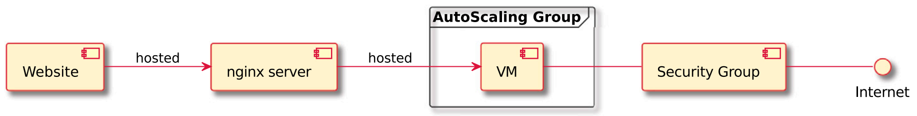


Fig. 1. Component diagram of the NGINX case study.

Table 2
Requirements on the specific elements to be modeled in DOML.

Req ID	Description	Priority
REQ01	DOML must be able to model infrastructural elements	High
REQ27	DOML should support modeling, provisioning, configuration, and usage of container engine execution technologies (e.g., docker-host)	High
REQ28	DOML should support modeling of containerized application deployment (e.g., pull/run/restart/stop docker containers)	High
REQ29	DOML should support modeling of VM provisioning for different platforms such as (OpenStack, AWS) for canary and production environments	High
REQ25	DOML should support modeling of security rules (e.g., by typing ingress/egress rules for network flow)	High
REQ26	DOML should support modeling of security groups (containers for security rules)	High
REQ60	DOML should support modeling of security metrics both at the level of infrastructure and application	High
REQ61	DOML must support modeling of NFRs and of SLOs	High
REQ30	DOML should enable support for policy definition constraints for QoS/NFR requirements	High

Academic partners, such as POLIMI, expert in modeling and formal logic, contributed more general requirements about fundamental DOML properties, such as REQ62, REQ63 and REQ70 that have been agreed upon during the project's work package meetings. Finally, use case partners, such as Prodevelop and Ericsson, formulated requirements linked to specific needs of their pilot (e.g., REQ111).

Some of the requirements, in particular, those to which a low priority has been assigned have not been implemented in the DOML yet. This is due to the incrementality of the process that led to the definition of the DOML language. We started from the structure, the basic elements and the higher priority requirements, then progressively adding more elements and refining the syntax. For each version of the language we implemented the related features in the PIACERE framework's tools: the ICG, translating DOML into executable IaC code, the Model Checker, to verify a DOML model, the Integrated Development Environment, with its syntax-directed DOML editor, and the DOML Optimizer. Adding new features took time and we tried to avoid defining a new version of the language until the previous one was implemented and supported by all the tools, possibly with tests from the use cases.

Lower priority requirements have been left for future versions of the language and the related tools. Anyhow, all the requirements in Tables 1 and 2 have been the basis for the analysis that conducted to the definition of the DOML design principles presented in Section 3.3.

3.2. Running example

To illustrate our approach, we use a simple deployment as a case study. It consists of a website hosted by an instance of the NGINX web server [20] deployed on a VM. More sophisticated examples, involving more components, will be demonstrated in Section 6. This example, though, is representative of typical deployments, because it contains some of the most common components (see Fig. 1 for a component diagram representation). The NGINX server instance is the execution environment for the website and runs on a VM with a GNU/Linux-based operating system (Ubuntu 20.04). To ensure the website scalability with respect to the number of connected users, multiple instances of the VM are spawned and managed by an auto-scaling group. The network interface that links the VMs to the Internet is managed by a security group, containing the security rules that enable HTTP, HTTPS and ICMP network traffic. The standard SSH port is enabled, giving direct access to the VMs, protected by an RSA key pair for authentication.

An infrastructure like this can be implemented by relying either on a private cloud or on public cloud providers, such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure. Initially, in our case study we choose to deploy the application on OpenStack [21], which is an open source industry standard. Further on, in Section 6 we show how we can change the underlying provider.

3.3. DOML design principles

In this section, we present the principles underneath the designing of DOML.

3.3.1. A single model for multiple IaC fragments

The DOML is defined to support the creation of models resulting in IaC codes written in different languages and dedicated to different operations. For instance, let us consider the system outlined in Fig. 1. The following steps must be performed to deploy the modeled system:

1. A VM with the correct OS must be retrieved if preexisting, or created;
2. The VM must be set up for access through SSH;
3. The NGINX server with the website sources must be installed on the VM;
4. The autoscaling group must be set up with the VM image;
5. The network must be configured with the required security rules;
6. The deployment process must be planned and executed.

To execute the above listed steps adopting the current technologies, we would need some Ansible playbooks or other scripts executing steps 2 and 3, together with a Terraform or TOSCA blueprint to orchestrate all other steps. Such scripts have their inherent complexities, and they are all written in different languages featuring different programming models. With the DOML approach, we aim to derive such scripts from a high-level model, and to reduce the need for the end users to work with the low-level target languages as much as possible.

3.3.2. Multiple modeling layers

Another objective we target is to support separate modeling of the application-level components from their execution environments (for instance, containers and VMs). In fact, we argue that different users, with different skills and roles, should focus on the specific aspects that fall within their expertise. Typically, the application designer will focus on the application structure definition in terms of software components and their connections, while an operations expert will oversee the allocation of software components within proper computing nodes.

Furthermore, multiple providers and technologies offering the same IaaS (Infrastructure-as-a-Service) and in some cases, compatible PaaS (Platform-as-a-Service) solutions are available. Thus, we want to offer the possibility to provide an abstract definition of the infrastructure to be used to run an application, and then to define different concretizations, so as to support deployment and execution of applications into multiple contexts.

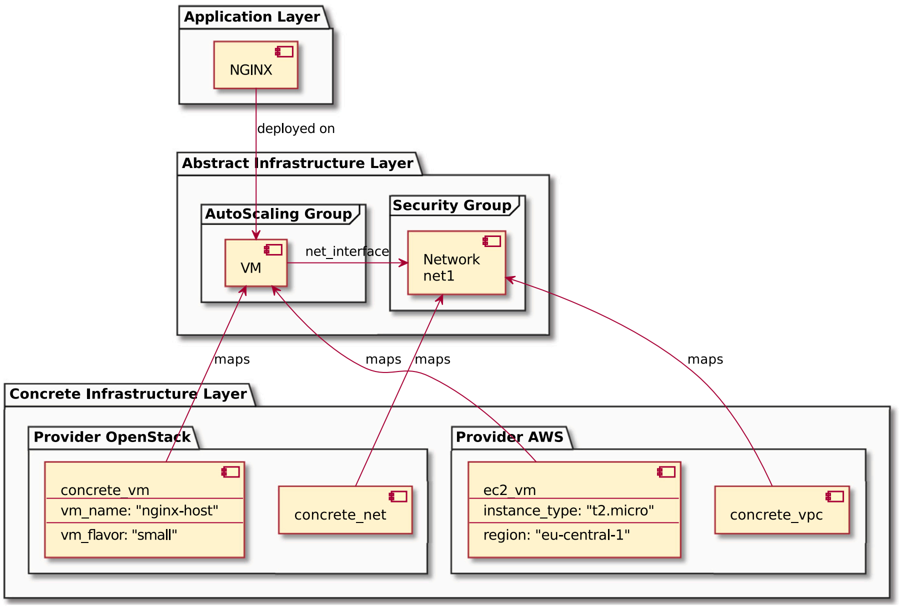


Fig. 2. The NGINX case study represented in different DOML layers.

Referring to the example of Figs. 1, 2 shows a distribution of components into three layers: one describes the application, and the remaining two layers describe the infrastructure at different levels. In particular, the same *abstract* infrastructure can be implemented by two different *concrete* infrastructures, respectively based on OpenStack and AWS.

3.4. Language overview

The DOML language implementation [22] consists of two parts: the DOML metamodel, described using Ecore from the Eclipse Modeling Framework (EMF) [23], and the textual syntax used to create models, based on Xtext [24].

We organize all modeled entities in the following layers, which aggregate the modeling abstractions in coherent groups:

- *Application Layer (AL)*: concepts required to define an application, e.g., software components, interfaces, connectors between components, services, and specific subcategories thereof.
- *Abstract Infrastructure Layer (AIL)*: concepts associated to the definition of the infrastructure (e.g., computing nodes) without referencing a specific provider.
- *Concrete Infrastructure Layer (CIL)*: concepts associated to the definition of infrastructure elements within a specific provider, e.g., a Docker container or an Amazon VM.

In Section 3.4.1, we provide an overview of the main concepts pertaining to each layer. We do not include the complete definition of the DOML language, which is available in [25,26], because it would be excessively long, but we illustrate it in Section 3.4.2 through an example.

3.4.1. Components of doml layers

Table 3 shows selected concepts of the DOML metamodel. The details are explained as follows.

A *DOMLModel* is composed of an AL, an AIL, one or more CILs, and a *Configuration*. A *Configuration* is a list of one or more *Deployment*s that consist of the associations between *ApplicationComponent*s (from the AL) and *InfrastructureElement*s (from the AIL) on which they are deployed. Only one *Deployment* can be active at a time. All other components derive from a base *DOMLElement* class, which gives them the ability to have custom *Properties* encoding some of their features.

Application layer (AL). The *ApplicationLayer* is composed of many *ApplicationComponent*s, which can be *SoftwareComponent*s, *SoftwareInterfaces*, or *SaaS* (Software-as-a-Service) components. A *SaaS* can be, e.g., a *SaaSDBMS* if it implements a database. *ApplicationComponent*s may expose or consume different *SoftwareInterfaces*, providing or requiring services from other *ApplicationComponent*s. For instance, a database *SoftwareComponent* can expose a SQL-based interface, and a web application component can consume it, meaning that the latter will communicate with the former to retrieve and write data.

Abstract infrastructure layer (AIL). The *InfrastructureLayer* is composed of *ComputingNodes*, *Networks*, *SecurityGroups*, and *AutoScalingGroups*. A *ComputingNode* models any infrastructure element that can run software: it can be a *Container*, a *PhysicalComputingNode* or a *VirtualMachine*. *ComputingNodes* can have multiple *NetworkInterfaces* that link them to a *Network*. A *Container* can be generated from a *ContainerImage*, and a *VirtualMachine* from a *VMImage*. A *Network* can have many *Subnet*s, and its configuration is represented by a *SecurityGroup* containing firewall rules. *Container*s can be organized in *ContainerGroups* that, at runtime, can be managed by the same controller, e.g., technologies such as Docker Compose or Kubernetes. This feature has been introduced in the latest version of the DOML language and will be extended to account for the fact that any set of *ComputingNodes* can be organized and managed as a group.

Concrete infrastructure layer (CIL). This layer provides the *concretizations* for the AIL, mapping the abstract infrastructure elements to the concrete ones from the supported cloud service providers. Most elements of the AIL have a corresponding "concrete" version with the same name. For instance, the *VirtualMachine* class appears both in the AIL and the CIL, but the two classes are distinguished by the containing package.

The CIL contains one or more *RuntimeProvider*s (which can be, for instance, Amazon AWS or OpenStack). Each *RuntimeProvider* contains the concrete elements which are linked to the AIL elements via the *maps* association. For instance, an OpenStack provider can provide *VirtualMachines*, *Networks*, and *Container*s.

3.4.2. Doml model of the running example

To illustrate the syntax of the DOML, we show and comment the DOML model of the case study of Section 3.2. The entire model can be found in [27].

Table 3
Classes of the DOML metamodel, grouped into packages (reported on the left).

	Class	Type	Extends class	Extends metaclass
Common	Configuration	Concrete	DOMLElement	None
	DeployableElement	Abstract	None	ecore.EObject
	Deployment	Concrete	None	ecore.EObject
	DOMLElement	Abstract	None	ecore.EObject
	DOMLModel	Concrete	DOMLElement	None
	Property	Abstract	None	ecore.EObject
	Source	Concrete	DOMLElement	None
Application L.	ApplicationComponent	Abstract	ApplicationLayer	None
	ApplicationLayer	Concrete	DOMLElement	None
	SaaS	Concrete	ApplicationComponent	None
	SaaSDBMS	Concrete	SaaS	None
	SoftwareComponent	Concrete	ApplicationComponent	None
	SoftwareInterface	Concrete	ApplicationComponent	None
Abstract Infrastructure Layer	AutoScalingGroup	Concrete	ComputingGroup	None
	ComputingGroup	Abstract	DOMLElement	None
	ComputingNode	Abstract	Node	None
	ComputingNodeGenerator	Abstract	DOMLElement	None
	Container	Concrete	ComputingNode	None
	ContainerGroup	Concrete	ComputingGroup	None
	ContainerImage	Concrete	ComputingNodeGenerator	None
	InfrastructureElement	Abstract	DOMLElement, DeployableElement	None
	InfrastructureLayer	Concrete	DOMLElement	None
	Network	Concrete	DOMLElement	None
	NetworkInterface	Concrete	InfrastructureElement	None
	Node	Abstract	InfrastructureElement	None
	PhysicalComputingNode	Concrete	ComputingNode	None
	SecurityGroup	Concrete	DOMLElement	None
	Subnet	Concrete	Network	None
	VirtualMachine	Concrete	ComputingNode	None
	VMImage	Concrete	ComputingNodeGenerator	None
Concrete Infra. Layer	AutoScalingGroup	Concrete	ConcreteElement	None
	ConcreteElement	Abstract	DOMLElement	None
	ConcreteInfrastructure	Concrete	DOMLElement	None
	ContainerImage	Concrete	ConcreteElement	None
	GenericResource	Concrete	ConcreteElement	None
	Network	Concrete	ConcreteElement	None
	RuntimeProvider	Abstract	DOMLElement	None
	VirtualMachine	Concrete	ConcreteElement	None
	VMImage	Concrete	ConcreteElement	None

Application layer. In Listing 1 we show the AL of the DOML model for the deployment of Fig. 1. It only contains a *SoftwareComponent* for the NGINX server, with a *Property* indicating the website's sources.

Listing 1: DOML Application Layer
```
application app {
  software_component nginx {
    properties {
        ↪ source_code="/.../html/index.html"; }
  }
}
```

Abstract infrastructure layer. The AIL is partially shown in Listing 2. It defines the infrastructure topology that supports the execution of application components. We define the VM that hosts the NGINX instance in the autoscaling group that manages it. We declare its guest operating system, its credentials, and its network interface, which is linked to a network called `net1` and controlled by a security group called `sg` (we do not show all components here for the sake of brevity, but they are defined in this layer too [27]).

The *deployment configuration*, at the bottom of Listing 2, provides the link between the AL and AIL: it assigns the NGINX instance to the VM.

Listing 2: Autoscaling group in the AIL, and deployment configuration.
```
infrastructure infra {
  ...
  autoscale_group ag {
    vm vm1 {
      os "ubuntu-20.04.3"
      iface i1 {
        address "10.0.0.1"
        belongs_to net1
        security sg
      }
      credentials ssh_key
    }
  }
}
deployment config {
  nginx -> vm1
}
```

Listing 3: Part of the concrete infrastructure layer.
```
concretizations {
  concrete_infrastructure con_infra {
    provider openstack {
      autoscale_group concrete_ag {
        properties {
          vm_key_name = "user1"
        }
        resource_name "nginx-host"
        vm_type "small"
        maps ag
      }
      ...
    }
  }
  ...
```

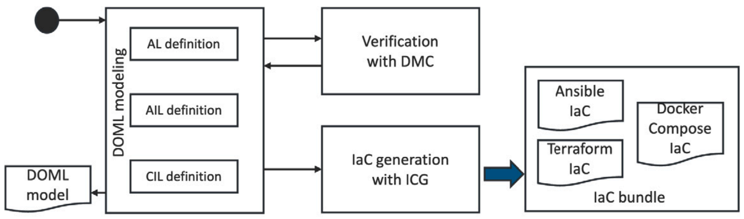


Fig. 3. The modeling workflow with DOML.

```
  active con_infra
}
```

Concrete infrastructure layer. The last step needed to make this DOML model functional is to assign all components in the AIL to a cloud service provider. This is done by defining one or more CILs (only one of which will be active at a time). To simplify the presentation, in Listing 3 we show only one of such layers for the autoscaling group.

We create a concrete infrastructure configuration called `con_infra`. We could assign different components of the AIL to different providers. For the sake of space, in this section we use only OpenStack. Thus, we create a block for the OpenStack provider containing a component for each of the abstract infrastructure elements. In Listing 3 we show the concretization of autoscaling group. Its `maps` attribute links it to the corresponding one in the AIL. Moreover, in the CIL we can customize aspects that cannot be described in the AIL because they are provider-specific. Here we choose the name and size of the VM with the autoscaling group.

The information in this model is enough for the ICG to produce IaC scripts that can create VMs on OpenStack, the autoscaling group and all other required features and install NGINX on top of such infrastructure.

4. Modeling workflow and supporting tools

4.1. Modeling workflow

The DOML is used by DevOps teams within the context of the iterative workflow shown in Fig. 3, where models are created starting from the definition of the system to be deployed in the AI, mapping it to an abstract infrastructure in the AIL, and then introducing the instantiation of one or more concrete infrastructures in the CIL.

At any stage of this process, the internal consistency of the model can be verified with the DOML Model Checker (DMC) (see Section 4.2), which can be executed multiple times in a continuous process where the DMC may highlight possible problems in the model and the user can fix or add elements till model completion. At this point, the DOML model can be submitted to the ICG component, which generates Infrastructure as Code in the languages adopted to define and execute the operations of provisioning, deployment, configuration, and orchestration (see Section 4.3). The resulting code can be executed with the standard executors of the selected languages or with the PIACERE project-specific runtime environment (this environment is not the focus of this paper; interested readers can refer to [28]). In the following we describe in further details the characteristics of both DMC and ICG.

This process is supported by an Integrated Development Environment (IDE) [29], which consists of a graphical user interface including an editor with syntax highlighting, and allows users to easily invoke the DMC and the ICG. The IDE parses DOML models and serializes them in an XML-based format called DOMLX [29], before sending them to other components.

4.2. Verifying the correctness of DOML models

Like all code-based artifacts, deployment configurations defined through an IaC language such as DOML can contain errors and bugs. Such defects can introduce very serious issues in deployments, possibly undermining availability, or introducing security vulnerabilities. In order to help DOML users write higher-quality models, we offer the DOML Model Checker (DMC), a tool aimed at detecting infrastructural inconsistencies and other issues that may be present in DOML models. Such issues are thus caught before attempting the deployment, preventing them from reaching the testing – or worse, production – environment, and therefore reducing the overall time needed to develop the deployment configuration.

The main aspects of a DOML model that the DMC verifies are its topology and architecture. The layered structure of DOML models itself acts as a specification of the deployment to obtain. In fact, the application layer describes the software components that are supposed to be run on the infrastructure. Thus, the DMC checks several aspects of the infrastructure specified in the AIL to make sure it is adequate to run the software components. Similar consistency checks are performed on the CIL to ensure consistency with the AIL.

In more detail, the following default requirements are always checked by the DMC on all DOML models:

MC1. All VMs are linked to a network (to prevent them from being locked-in).
MC2. All software components that share interfaces in the AL must be deployed to infrastructure nodes that can communicate among each other through a network.
MC3. There are no duplicated network interfaces (e.g., no two nodes are assigned the same IP address in the same network).
MC4. All software components in the AL have been deployed to some components in the active AIL.
MC5. All components in the AIL are mapped to a concrete element assigned to a cloud service provider in the CIL.
MC6. All security groups in the AIL protect at least one network interface.

The DMC also supports a requirement specification language based on First-Order Logic. This is used to encode the requirements listed above and also to allow users to specify custom requirements to be checked automatically. To demonstrate the usage of this language, we show how it can express requirement number MC1 from the list above:

```
forall vm (
  vm is class abstract.VirtualMachine
  implies
  exists iface (
    vm has abstract.ComputingNode.ifaces iface
  )
)
```

First, we introduce a universally quantified variable `vm`. Through the `implies` keyword, we state that if it is a `VirtualMachine` node from the AIL (`abstract`), then there exists an interface `iface` such that `vm` is linked to `iface`. If the model contains a VM that has no

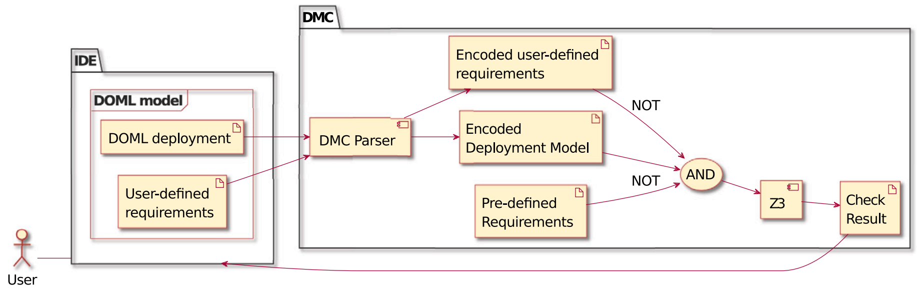


Fig. 4. Architecture of the DOML Model Checker.

interface, then this requirement is false, and the DMC detects this fault and shows it to the user.

The internal architecture of the DMC is shown in Fig. 4. First, the DOML model is encoded as a first-order formula. This formula contains relations representing attributes and relationships between different components. For instance, the formula may state that a VM component is linked to a specific interface which, in turn, belongs to a certain network. The requirements, also expressed as a first-order logic formula as shown above, contain the same relations. The DMC then checks for satisfiability of the conjunction of the formula encoding the DOML model and the one encoding the *negation* of the requirements. If the check succeeds – i.e., the conjunction of the two formulas is found to be satisfiable – then there is a mistake in the model. For instance, for requirement MC1, this means that there is a VM without network interfaces. Otherwise, the requirement is satisfied.

The satisfiability check is performed by means of the Z3 Theorem Prover [30,31], which can check the satisfiability of quantified first-order logic formulas containing only terms from decidable theories, in our case abstract data types and uninterpreted functions [32]. Solvers like Z3, called *SMT solvers*, are widely used in formal methods and program verification, in which they are exploited to solve proof obligations [33]. Giving a detailed explanation of the inner workings of SMT solvers is out of the scope of this paper: we refer the interested reader to automated theorem proving and formal methods literature [33].

The implementation of the DMC is available at [34]. We demonstrate the use of the DMC through a case study in Section 6.4.

4.3. IaC generation mechanism

To generate executable IaC code from DOML, we have built a tool named Infrastructural Code Generator (ICG) [35]. The ICG receives a DOML model as input and generates IaC code targeting Terraform, Ansible, or a similar framework, as output.

The generation of code from an abstract model is one of the main advantages of MDD [3], but the benefit is real when the generated code is complete and executable, i.e., not just a skeleton or something that needs manual editing. Generating code from a model can be done using Template-Based Code Generation (TBCG), a technique that transforms input data into structured text by using templates [36]. The process is simple: our template engine uses templates that contain code in the target language and substitutes values taken from the input for placeholders. Each template has a static part, that is transferred as-is in the output, and a dynamic part whose result depends on the input. Most template engines support control structures in the dynamic part of their templates, which allows part of the transformation logic to be embedded in the template itself.

Template engines can be classified according to the input they rely on: according to Luhunu [37], there are model-based engines, such as Acceleo [38], that are based on an input metamodel, and code-based tools, such as Velocity [39], which rely on a DSL to express the dynamic part of their templates. The ICG uses the code-based Jinjia2 library [40], which is a simple but powerful template engine

Listing 4: ICG template for OpenStack VMs.

```
resource "openstack_compute_instance_v2"
    ↪ "{{infra_element_name}}"
{
  name        = "{{name}}"
  image_name  = "{{os}}"
  flavor_name = "{{vm_flavor}}"
  key_pair    =
      ↪ openstack_compute_keypair_v2.{{credentials}}.name
  network { ... }
}
```

that supports plain placeholder substitution, and also several control structures, such as loops, conditionals and functions, to build dynamic templates (see Listing 4).

We implemented the ICG in Python. Its internal architecture, represented in Fig. 5, is inspired by the classic structure of a compiler (see e.g. [41]) and consists of separate modules for parsing the input and for generating the output, with an Intermediate Representation (IR) in between. The parser reads the DOML model using the PyEcore library [42] and generates an IR as a JSON document. Then, different Code Generator plug-ins, one for each language to be generated, read the input data from the IR and substitute values in the templates. The whole flow is driven by the Controller, that selects the right templates and activates the corresponding plug-in, depending on the information included in the IR itself.

The IR created by the Parser is structured as a sequence of steps, representing the main code blocks to be generated. Each step includes general information, such as the target language, the target cloud provider, the type of DOML object for which code should be generated, and attributes specific to the target DOML object, in the form of key/value pairs, to be substituted in the template for the corresponding placeholder.

The Controller selects the template to be used depending on the information indicated above (target language, cloud provider, and type of DOML object), and then activates the Code Generator plug-in specific to the desired target language. Template selection and plug-in activation are repeated for each one of the steps in the IR, therefore the ICG can generate code in multiple IaC languages at the same time. The ICG starts creating IaC code for the concrete DOML elements (CIL layer) and navigates up in the model to find other resources for which there is enough information for generating the related code.

When generating code for the example of Fig. 1, the ICG first learns from the CIL of Listing 3 that the VM should be deployed on OpenStack. Then, it selects the template shown in Listing 4, and populates the fields it finds in the CIL (e.g., `name` with the value of `vm_name`). The remaining fields (network and key-pair) are populated by looking at the AIL.

The generated IaC code is intended to create an infrastructure in the selected provider or to configure it in some way, in short, to modify

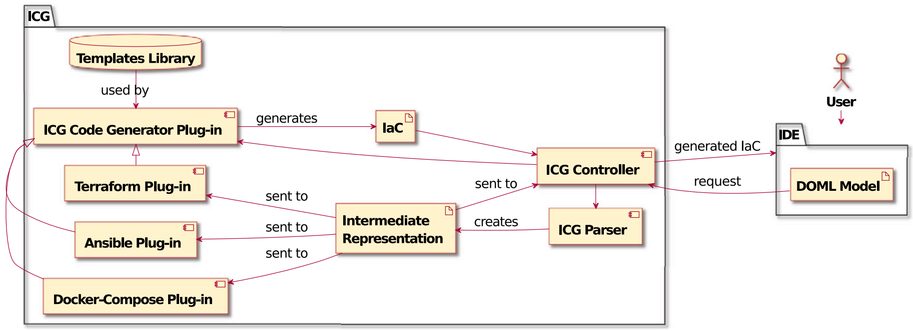


Fig. 5. Infrastructural Code Generator Architecture.

the target environment. Our aim is to create a stateless code generator, i.e., one that does not take into consideration the current status of the target environment. This works well when the target language is declarative, such as Terraform, but it is not the case when generating code for an imperative language. In the latter case, the target code (better, its template) should be idempotent, i.e. such that the status of the target system does not change even if the code is run multiple times.

ICG can generate both Terraform and Ansible code, depending on the specific activity that is intended to be performed. We use Terraform for provisioning and Ansible for configuration. The aim is to have a code generator that can be configured to produce code also for other IaC tools, and can be extended also to support new model abstractions. For example, in the latest ICG version, Docker Compose code generation has been added for orchestrating containers, defined in the AIL and grouped in container groups. An example of this will be shown in Section 6.3.2. ICG can also get as input a folder (we refer to it as *asset folder*) containing user-provided preexisting IaC code, that ICG integrates with the IaC code generated from DOML in a single multi-language zipped package. This feature has been required by the partners experimenting with DOML, as described in Section 6.5.

Lastly, the templates currently available with the ICG have some parameters that are definable using DOML, while other parameters are set to defaults given by the service provider or based on the most common use for the specific service. DOML-definable parameters are generally optional, and the ICG chooses for them appropriate defaults in case they are not specified. This helps inexperienced users to achieve functional deployments easily. We will see an example of this aspect of the ICG in the evaluation, in Section 6.3.1.

To account for the limitations posed by unsupported parameters, the ICG allows for easy editing, change or addition of the templates, where the only requirement for the integration of a new template in the ICG is the editing of a properties file. This functionality integrated with the DOML extension mechanism, which will be presented in Section 5, allows the user to define his/her own set of parameters and to adapt the code generation to specific use cases if necessary.

5. Extension mechanisms for the DOML (DOML-e)

DOML-e is the set of extension mechanisms defined for the DOML. It includes the language elements that allow users to personalize a DOML model with specific characteristics, presented in Section 5.1, and also a more sophisticated approach that allows users to extend the DOML metamodel and to automatically modify the DOML syntax and the ICG templates to account for such changes. This will be presented in Section 5.2.

5.1. Internal extension mechanisms

The extension mechanisms already embedded in the language include the elements *Properties*, *Source*, and *GenericResource* that we explain below.

Properties. DOML allows for the definition of different types of properties that can be added to the definition of any *DOMLElement*, thus increasing the DOML's expressive power. The following *Properties* subclasses are defined: *IProperty*, *FProperty*, *SProperty*, and *BProperty* represent respectively integer-, float-, string-, and Boolean-valued features; *ListProperty* allows to define custom, multi-valued properties, including sub-fields of any *Properties* type.

The example in Listing 5 demonstrates how properties can be used to extend the characteristics of a general *SoftwareComponent*. In the example, to completely characterize the *ElasticSearch*[4] component, we added six properties, four of type *SProperty* and two of type *IProperty*, thus being able to incorporate in the model the necessary information to properly personalize the configuration of the component.

Listing 5: DOML Properties Example
```
application app {
  software_component ElasticSearch {
        properties {
            es_cluster_name = "custom-cluster"
            es_data_dirs =
                ↪ "/opt/elasticsearch/data"
            es_log_dir =
                ↪ "/opt/elasticsearch/logs"
            es_http_port = 9201
            es_transport_port = 9301
            es_discovery_seed_hosts =
                ↪ "localhost:9301"
        }
        provides { search }
        consumes { dbAccess }
    }
}
```

Source. Another internal extension mechanism accounts for the possibility to reuse as part of DOML IaC fragments written in other languages. This need has been expressed by the PIACERE end users as REQ111 (see Section 3.1). The solution we propose is the introduction in the DOML of the *Source* element, which enables the import of external IaC to configure a software component. Listing 12 shows an example of usage of such element where the software component `nio3` is defined as having source a `mail.yml` file which is interpreted by the `backend` Ansible. This example will be contextualized and described in further details as part of the SI-MPA case study in Section 6.5.

GenericResource. A third internal extension mechanism is offered by the concept of *GenericResource* defined within the CIL. It is used to exploit in a specific DOML model resources that are already available on some infrastructure, i.e., an already running VM or an operating datastore. To address this case, a special class, named *GenericResource*,

[4] https://www.elastic.co/elasticsearch

has been introduced. In the DOML syntax, *GenericResource* are identified by the field `preexisting true`. An example can be seen in Listing 15 in Section 6.5, where `template` and `database` are two generic resources representing, respectively, a VM image and a storage system.

5.2. Metamodel extension

Even though the extension mechanisms described in Section 5.1 may satisfy most of users' needs, they may not be sufficient to achieve the full extensibility of the language. Therefore, we have analyzed the problem of how to support users in extending the DOML metamodel to incorporate new concepts in the language. In the following, we focus, in Section 5.2.1, on presenting the specific requirements that should be fulfilled by the offered mechanism to make it usable in the context we are considering. Then in Section 5.2.2 we discuss the way we have adapted and used an existing library, Eclipse Edapt, to allow users to keep existing DOML models consistent with the possible evolution of the corresponding metamodel and, finally, we focus on the automatic extension of the DOML syntax in Section 5.2.3.

5.2.1. Requirements

The DOML consists of both an EMF Ecore-based metamodel and an Xtext-based syntax, allowing users to easily create models in a programmatic way.

Introducing a new element in the DOML – for instance, a special-purpose application-level component such as a specific SaaS or even a new type of computing environment or of network – means extending the metamodel but also updating the DOML syntax to offer new keywords that identify the new type of element together with all its specific characteristics. The update of these two elements has to be perfectly synchronized to prevent the language from becoming quickly useless.

Another important aspect concerns ensuring the compatibility between DOML models developed with a previous version of the language and the new versions resulting from the metamodel evolution approach.

In the following, we present the approach we adopted for the metamodel extension and the automatic creation of the corresponding syntax that takes these two points into account.

5.2.2. Metamodel extension through eclipse edapt

Over the years, various approaches have been developed to update Ecore-based metamodels [43–45], but only a few of them offer the capability to automatically migrate existing models to align them with the updated metamodel, which, as discussed in the previous section, is an important requirement to keep the compatibility between the evolved language and the models defined before the evolution. The approaches offering support to this co-evolution can be classified in two main categories [46]: *difference-based approaches*, which capture, *a posteriori*, explicit changes at the metamodel level and apply specific corresponding changes at the model level, and *operator-based approaches* that use predefined transformation operations to evolve the metamodel and propagate these changes to the models.

A popular tool within the EMF community that adopts an operator-based approach is Eclipse Edapt [47]. This framework gives users the flexibility to select from a wide range of operators for co-evolution of metamodels and models.

Although operator-based approaches limit the range of changes to metamodels only to those defined by their operators, they provide standardized migration strategies, minimizing the need for human intervention to the handling of exceptional cases. The Edapt Operation Browser offers an extensive set of operators [48] for co-evolution, providing sufficient options to perform necessary modifications to the DOML metamodel.

Among the provided operators, some of the simplest ones include *Create Class* and *Create Attribute*, shown in Fig. 6 and Fig. 7 respectively. It is worth to mention that even some of these implement migration strategies for existing models: for example, *Rename* changes all the occurrences of the involved element. Edapt also provides users with complex operators, such as *Fold Super Class*, through which a number of features are replaced by features of a new super class. The advantage of an approach employing coupled operators [46] as Edapt mainly resides in the possibility to automatically migrate existing models when using complex operators. In this case, for example, the values of the affected features are moved to the new features based on a mapping.

To adopt the Edapt approach within the DOML approach, we had to define special behavior for some specific cases. One of them concerns the need, in Edapt, to change the namespace URI to distinguish between different releases of a single metamodel and, while migrating models, allowing them to apply operators to evolve metamodels. Since the DOML URI is intended to remain unchanged, this has led to the need to develop a specific mechanism: the Edapt framework has been modified to handle DOML metamodel as a special case, using the version attribute within the *DOMLModel* class, instead of the URI, to distinguish between different releases of the language, both at the metamodel and at the model level.

Another issue concerned the creation of the DOML-X, that is, the XML serialization to be used for communication with the DMC and ICG. Normally, the XML resource saving process in EMF discards default contents, which could result in the loss of important values during model migration. Since in DOML we use default value literals to store key values, including the version name, to prevent them from being discarded, an option to include such values in the resulting XML file during the saving process has been added.

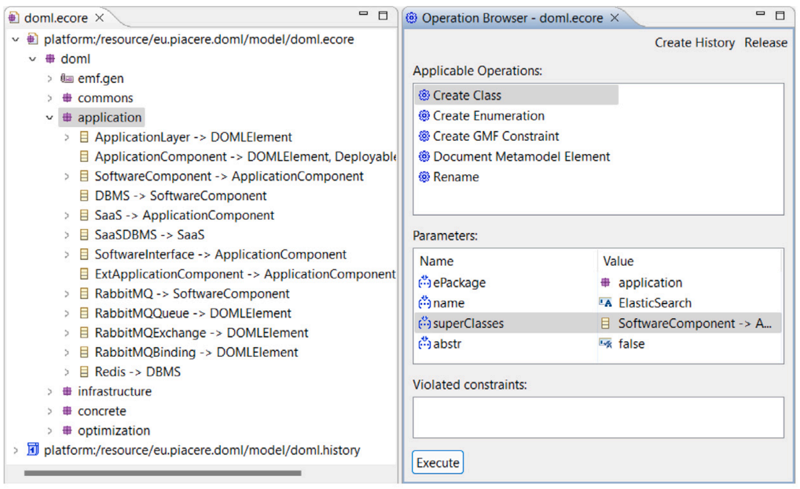
To see how the approach works, let us consider the case of extending the metamodel to introduce a new element representing the ElasticSearch software that we have already mentioned in Section 5.1. In this case, the user would interact with Edapt as shown in Fig. 6, where the Create Class operator is used and the *ElasticSearch* class is created, extending the *SoftwareComponent*. The user would then create specific attributes for this class. For instance, in Fig. 7, the *Create Attribute* operation of Edapt is used for creating a new *httpPort* attribute. Analogously, both containment and non-containment references can be added through the *Create Reference* operation. Fig. 8 shows the result of creating the class by adopting this approach.

After defining the new concept, two steps are needed. One, discussed in Section 5.2.3, concerns the generation of the syntax to let the end user exploit the ElasticSearch element in a DOML model, the other concerns the creation of a proper ICG template, as presented in Section 4.3, to enable the generation of proper IaC for the new element. In Listing 6, the ElasticSearch template is presented. Being this a SoftwareComponent, it will be based on Ansible, which is the language generated by the ICG for software configuration.

Listing 6: Ansible template used to model the ElasticSearch component

```
---

- name: Elasticsearch with custom
    ↪ configuration
  hosts: servers_for_elasticsearch
  roles:
    - role: elastic.elasticsearch
  vars:
    es_data_dirs:
      - "{{␣dataDirs␣}}"
    es_log_dir: "{{␣logDir␣}}"
    es_config:
      node.name: "{{␣nodeName␣}}"
      cluster.name: "{{␣clusterName␣}}"
      discovery.seed_hosts:
          ↪ "{{␣discoverySeedHosts␣}}"
      http.port: {{ httpPort }}
      transport.port: {{ transportPort }}
      node.data: false
      node.master: true
```

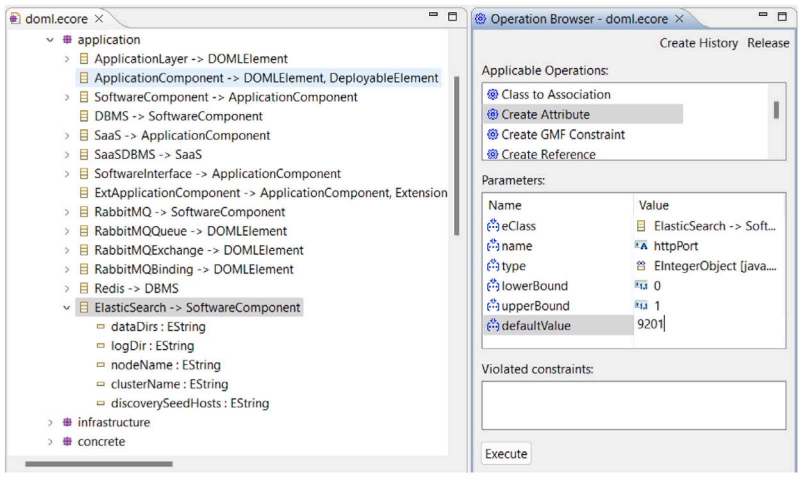

doml.ecore
platform:/resource/eu.piacere.doml/model/doml.ecore
doml
emf.gen
commons
application
ApplicationLayer -> DOMLElement
ApplicationComponent -> DOMLElement, Deployable
SoftwareComponent -> ApplicationComponent
DBMS -> SoftwareComponent
SaaS -> ApplicationComponent
SaaSDBMS -> SaaS
SoftwareInterface -> ApplicationComponent
ExtApplicationComponent -> ApplicationComponent
RabbitMQ -> SoftwareComponent
RabbitMQQueue -> DOMLElement
RabbitMQExchange -> DOMLElement
RabbitMQBinding -> DOMLElement
Redis -> DBMS
infrastructure
concrete
optimization
platform:/resource/eu.piacere.doml/model/doml.history

Operation Browser - doml.ecore
Create History Release
Applicable Operations:
Create Class
Create Enumeration
Create GMF Constraint
Document Metamodel Element
Rename
Parameters:

Name	Value
ePackage	application
name	ElasticSearch
superClasses	SoftwareComponent -> A...
abstr	false

Violated constraints:
Execute

Fig. 6. Using Edapt to create a new class.


doml.ecore
application
ApplicationLayer -> DOMLElement
ApplicationComponent -> DOMLElement, DeployableElement
SoftwareComponent -> ApplicationComponent
DBMS -> SoftwareComponent
SaaS -> ApplicationComponent
SaaSDBMS -> SaaS
SoftwareInterface -> ApplicationComponent
ExtApplicationComponent -> ApplicationComponent, Extension
RabbitMQ -> SoftwareComponent
RabbitMQQueue -> DOMLElement
RabbitMQExchange -> DOMLElement
RabbitMQBinding -> DOMLElement
Redis -> DBMS
ElasticSearch -> SoftwareComponent
dataDirs : EString
logDir : EString
nodeName : EString
clusterName : EString
discoverySeedHosts : EString
infrastructure
concrete

Operation Browser - doml.ecore
Create History Release
Applicable Operations:
Class to Association
Create Attribute
Create GMF Constraint
Create Reference
Parameters:

Name	Value
eClass	ElasticSearch -> Soft...
name	httpPort
type	EIntegerObject [java....
lowerBound	0
upperBound	1
defaultValue	9201

Violated constraints:
Execute

Fig. 7. Creating a new *httpPort* attribute for the *ElasticSearch* class.

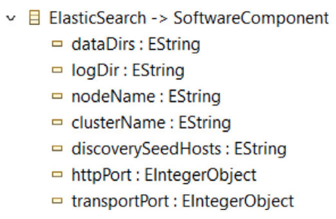

ElasticSearch -> SoftwareComponent
dataDirs : EString
logDir : EString
nodeName : EString
clusterName : EString
discoverySeedHosts : EString
httpPort : EIntegerObject
transportPort : EIntegerObject

Fig. 8. *ElasticSearch* class in the Ecore metamodel.

```
      bootstrap.memory_lock: true
    es_heap_size: 1g
    es_api_port: {{ httpPort }}
```

In the listing, some parameters, e.g., es_heap_size, assume default values while the others assume the values end users assign to the corresponding attributes enclosed in the double brackets and defined in the metamodel.

Note that the main advantage of introducing the ElasticSearch concept in the metamodel as opposed to the usage of the properties to specify its application-specific parameters is mainly in the possibility to reuse the new definition in other models and to have the development environment highlighting the syntax to be used. These points make the new concept usable also by newcomers not fully knowledgeable about the specifics of the ElasticSearch technology.

5.2.3. An ad hoc solution to update concrete syntax

The consistent update of the DOML metamodel and the corresponding syntax is not straightforward due to an unwise design choice we adopted at the beginning of the project. In fact, not being fully proficient with metamodeling and Xtext, we decided to proceed with the development of these two parts in a manually coordinated way. In other terms, the DOML syntax was not automatically generated from the metamodel, but the grammar rules used to generate it were created manually. To overcome this problem, we experimented with the existing generation mechanisms offered by Xtext framework, but the resulting syntax often differs significantly from the actual DOML syntax. This can affect both the presentation of content and the overall operation of DOML, such as the correct functioning of the conversion mechanism from DOML to DOMLX (XML-based representation) and vice versa. Therefore, adopting this approach would necessitate restructuring the language and several existing tools within the PIACERE framework.

One potential solution is to update only the parts of the syntax related to the changed content and affected classes. However, linking metamodel operations in the Edapt framework with Xtext rules requires significant effort and may still result in specific solutions that struggle with misalignment between components.

Given these limitations, an ad hoc solution has been developed to generate fully compliant syntax based on Eclipse Xtext directly from the updated Ecore metamodel. This solution, implemented as an

Eclipse plugin, is integrated into the PIACERE IDE. The plugin, visible when selecting Ecore files, includes a command to generate the Xtext grammar file from the metamodel.

The code structure follows the Eclipse command schema, with a default handler managing utility classes and executing the workflow for generating the Xtext syntax file. A single class is responsible for building various sections of the grammar, which are then concatenated into a single string. The handler directly manages the writing process.

Content generation relies on rules to handle different types of objects in the metamodel, creating corresponding Xtext code fragments. These rules were empirically developed to align elements of the DOML Ecore metamodel with existing DOML syntax rules. This heuristic approach highlights the challenge of managing exceptions, as the DOML's syntax was not designed for automated content generation and became highly specific over time. To create a generic solution capable of handling multiple exceptions and special cases, the syntax structure was deeply analyzed to find common patterns and group similar cases.

However, some aspects still require human intervention. For example, most component classes are identified in the DOML syntax by a fixed keyword similar to the class name. The definition of security rules in a security group is, however, different: each rule is introduced by either the keyword `egress` or `ingress`, which are the possible values of the *RuleKind* enum. Both keywords denote an instance of the *Rule* class, differing only in the value of their *RuleKind* field.

Moreover, some rules represent an exception for both their structure and content. For instance, while most classes require the specification of a user-chosen name after the introductory keyword, the syntax for the *ContainerHostConfig* class requires a reference to an existing hosting computing node.

To address the potential mismatch between class names in the metamodel and the resulting elements in the syntax and to allow users to generate rules such as the one mentioned above for the *RuleKind* class, a simple JSON configuration file must be defined. This file contains:

- Keywords for classes, attributes, and references;
- Enumeration attributes to replace specific keywords;
- Symbols for lists of multiple values;
- Fixed content including language declaration, datatype and terminal rules, and special unchangeable rules.

In Listing 7, an example of JSON configuration for ElasticSearch keywords is shown. The keywords on the left-hand side correspond to the attribute names defined in the metamodel while those on the right-hand side are the ones that will be used in the DOML syntax.

The syntax generation process will generate the Xtext definition shown in Listing 8. For instance, the keyword `cost` at line 5 in the listing is generated considering the definition at line 2 in the JSON configuration, while the keyword `is_persistent` at line 4 in Listing 8 is generated according to the standard rules defined by the generation mechanism. In both cases, the types defined for the attributes within the metamodel (FLOAT and Boolean, respectively) are properly reported in the Xtext definition and can be used to support syntax checks within a DOML-specific editor.

Listing 7: ElasticSearch JSON configuration for the Syntax Generator
```
1 "ElasticSearch" : {
2     "licenseCost": "cost",
3     "exposedInterfaces": "provides",
4     "consumedInterfaces": "consumes",
5     "clusterName": "cluster"
6 }
```

Listing 8: Xtext definition of the new DOML syntax for ElasticSearch components
```
1  ElasticSearch returns app::ElasticSearch:
2   'elastic_search' name=ID '{' DOMLElement
3    (
4     ('is_persistent' isPersistent=Boolean)? &
5     ('cost' licenseCost=FLOAT)? &
6     (src=Source)? &
7     ('provides' '{'
            ↪ exposedInterfaces+=InterfaceDefinition+
            ↪ '}')? &
8     ('consumes' '{'
            ↪ consumedInterfaces+=[app::SoftwareInterface]+
            ↪ '}')? &
9     ('data_dirs' dataDirs=STRING)? &
10    ('log_dir' logDir=STRING)? &
11    ('node_name' nodeName=STRING) &
12    ('cluster' clusterName=STRING)? &
13    ('discovery_seed_hosts' discoverySeedHosts=STRING)?
            ↪ &
14    ('http_port' httpPort=INT)? &
15    ('transport_port' transportPort=INT)?
16   )
17  '}'
18 ;
```

This approach enhances customizability and durability, enabling users to update the syntax as needed while maintaining the necessary fixed contents for the PIACERE environment.

6. Evaluation

We evaluate the DOML by using it in a concrete case study and against other state-of-the-art IaC approaches, with the objective to demonstrate its capabilities and shortcomings. We provide a discussion of the evaluation in Section 7. We identify the following research questions (RQs):

- RQ1: How effective is DOML at modeling deployments?
  We answer this question by splitting it into two sub-RQs evaluating DOML from two different points of view:
  - RQ1a: Can a DOML model represent the information required to generate executable IaC tackling both provisioning and configuration?
  - RQ1b: How does DOML compare with state-of-the-art approaches?
- RQ2: How effective are DOML and the ICG at targeting different existing cloud frameworks and technologies?
  We also answer this RQ by splitting it into two more specific RQs:
  - RQ2a: Is a DOML model able to target multiple execution platforms?
  - RQ2b: Is the ICG able to generate code in different IaC languages required to implement different components of the deployment?
- RQ3: Can the model checking component (the DMC) detect frequent mistakes during the definition of a DOML model?
- RQ4: Is the DOML approach perceived as useful in a real-world case?

We answer the RQs in separate subsections Section 6.2–6.5. First, however, in Section 6.1 we present a cloud deployment that we use while answering the RQs in the next sections.

6.1. A wordpress deployment

This example is taken from existing literature [49], and allows us to compare the DOML to state-of-the-art IaC languages. We use it to answer RQ1, RQ2 and RQ3 in Sections 6.2, 6.3 and 6.4, respectively. This example is available in [50] and is described in the component diagram of Fig. 9.

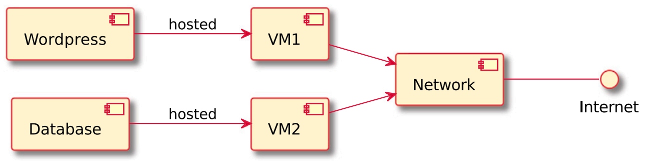


Fig. 9. Component diagram of the Wordpress application deployment from [49].

The application to be deployed is the Wordpress content management system, which runs on a VM. Wordpress depends on a database hosted on a separate VM. The two VMs communicate with each other as well as with the Internet through a common network.

6.2. RQ1: How effective is DOML at modeling deployments?

We answer RQ1a and RQ1b separately, both based on the Wordpress deployment introduced in Section 6.1.

6.2.1. RQ1a: Can a DOML model represent the information required to generate executable IaC tackling both provisioning and configuration?

We answer RQ1 through the Wordpress example presented in Section 6.1, which is larger in scope and more similar to real-world deployments than the NGINX case presented in Section 3.

We write a DOML specification of the Wordpress deployment targeting OpenStack as the CSP and successfully running the corresponding IaC code generated by the ICG.

We do not show the whole DOML model for the sake of brevity, but it is available in [27]. Similarly to the NGINX example, this DOML model closely resembles the diagram of Fig. 9. The model is small enough to be represented in a single file, and its subdivision into components is very natural: two components in the application layer for Wordpress and the database, and two corresponding VMs, one key-pair for the VM credentials, and one network in the abstract infrastructure layer.

An advantage brought by the DOML is that the ICG is capable of automatically generating certain common components from its templates, so that they need to be specified in the DOML only if a non-default configuration is required. One of such components is the Security Group: in the Wordpress example, the ICG creates it automatically thanks to the information included in the AL, so it is not necessary to include the Security Group in the DOML model. In fact, the Wordpress component has to be exposed to the Internet by default, and the AL states that it needs to be able to communicate with the database: the security group is created accordingly.

This DOML specification leads to a successful deployment of Wordpress on the selected infrastructure. This constitutes evidence that we can positively answer RQ1a, by means of this particular case. In Sections 6.5 and 6.3.2, we show two more deployments that we successfully encode as DOML models, further confirming this answer.

6.2.2. RQ1b: How does DOML compare with the state-of-the-art approaches?

We compare the DOML specification of the Wordpress example with two equivalent specifications in the state-of-the-art languages: the Terraform and Cloudify implementations written by the authors of [49].

We first answer RQ1b from a qualitative point of view, by highlighting the main differences in the design philosophy of each language. Next, we compare the three approaches in terms of conciseness.

Table 4
Metrics on the IaC in Section 6.2.2.

Approach	#LOC	# Files	# Languages	Available at
DOML	125	1	1	[27]
Terraform + Shell	305	3	2	[50]
Cloudify + Ansible	506	9	2	[50]

Terraform [49]. The Terraform definition of the deployment is centered around the provisioning of VMs. First, the provider is set, in this case AWS. This enables the use of AWS-specific components, such as `aws_instance` to define the VMs. The VMs are defined in this way, with their features set as properties (e.g., size, image to be run); the network to which VMs are attached is defined similarly.

The applications (Wordpress and the database) need to be deployed to their respective VMs. Terraform does not support application deployment directly, but a configuration language – Bash scripts in this case – is needed. The two Bash scripts, one for Wordpress and another one for the database, are stored as templates, that are instantiated, sent to the VMs through SSH, and automatically executed during deployment.

Cloudify [49]. The Cloudify definition of the deployment starts by importing plugins related to the targeted cloud platform. These allow to use provider-specific node types to define the VMs and the networks. Again, the Ansible configuration language is needed to deploy applications to the VMs.

Qualitative analysis. The main difference between the analyzed approaches is that Terraform and Cloudify are centered around infrastructure components, while in DOML applications and computational environments running on top of VMs (e.g., containers) have a central role. In IaC languages other than DOML, the fact that Wordpress is deployed on the VM can only be inferred by reading an Ansible or shell script. In DOML, Wordpress is a first-class component, and it is a task of the ICG to ensure that it is correctly deployed on a VM, instead of the user. This makes the whole deployment more robust.

Conciseness. In Table 4 we provide a summary of the metrics we have collected. The DOML appears to be more concise than other languages, with a total of 125 lines of code used for the complete definition of the deployment model and a single language used. The use of one single language also influences the number of files in which the model is split, which is much lower for DOML.

6.3. RQ2: How effective are DOML and the ICG at targeting different existing cloud frameworks and technologies?

We answer this RQ by considering two orthogonal aspects: the ability of the ICG to generate code in different IaC languages that tackle different components of the same deployment, within the same target CSPs (RQ2a), and its ability to generate code targeting different CSPs for the same deployment (RQ2b).

6.3.1. RQ2a: Is a DOML model able to target multiple execution platforms?

DOML supports multi-platform deployment and operation. We show this by modifying the OpenStack-based Wordpress DOML model introduced in Section 6.1 to incorporate a new deployment on AWS EC2 [51] other than the one on OpenStack we have presented in Section 6.2.

This can be accomplished by introducing in the CIL a new concrete infrastructure definition, including the specification of the new VM and network. In the CIL, the VM image name is specified by the Amazon Machine Images (AMI) format, e.g., `name = ‘‘ami-xxx’’`, which provides the information required to launch a VM instance. In the VM block, we specify other vendor-specific settings, such as the location information (e.g., `region = ‘‘eu-central-1’’`) and the `instance_type`. These parameters are optional, and the ICG chooses default values if they are not specified. Another issue caused by switching to AWS concerns the network. Amazon provides Virtual Private Cloud (VPC) for controlling the virtual networking environment, which requires defining a subnet into any network. If no subnet has been defined in the AIL, the ICG creates a default one. This way, a user not familiar with AWS requirements can still create a working deployment.

Listing 9 shows the new CIL, while everything else is unchanged. Different concrete infrastructures can coexist in the same DOML model and the currently selected one is identified by the `active` keyword.

Listing 9: VM definition in the concrete infrastructure layer.

```
concrete_infrastructure con_aws_infra {
  provider aws {
    properties { }
    vm concrete_vm1 {
      properties {
        vm_flavor = "t2.micro";
      }
      maps vm1
    }
    vm concrete_vm2 {
      properties {
        vm_flavor = "t2.micro";
      }
      maps vm2
    }
    net concrete_net {
      properties {}
      cidr "10.10.10.0/24"
      subnet concrete_subnet {
        cidr "10.100.1.0/24"
        maps subnet1
      }
      maps net1
    }
  }
  active con_aws_infra
}
```

The code generated by the ICG for the AWS provider successfully deploys the application. This provides evidence supporting a positive answer for RQ2b.

Moreover, this example also answers RQ1b from a different point of view, comparing the ability of IaC languages to support different CSPs for the same deployment. The same DOML model can define how a deployment is run on different CSPs, collecting settings specific to each CSP in a separate, well-marked section of the code. Aspects of the deployment that are common to all providers are abstracted and factored away in the AL and AIL. Thus, DOML minimizes redundancy which, in turn, results in better consistency between the actual deployment on the different providers, and better readability. Conversely, Terraform and Cloudify do not offer the same abstraction mechanisms. Each supported provider offers a different set of components, that are syntactically different, even if they represent the same kind of deployment node (e.g., VM, network, or application). To port a deployment to a different CSP, its whole description has to be changed, by translating components from one provider into corresponding (or similar) components of the other.

6.3.2. *RQ2b: Is the ICG able to generate code in different IaC languages required to implement different components of the deployment?*

The focus of this example is to show how we can use DOML to model a containerized MongoDB deployment and to generate, besides the usual Terraform code for VM provisioning, Ansible and Docker Compose code to orchestrate the execution of the whole system.

Fig. 10 shows the component diagram for this example: a MongoDB server and the corresponding client are running within containers `cont_mongodb` and `cont_hello_mongo`, respectively. The two containers are hosted on a single VM and communicate among each other and with the Internet through a network, which is protected by a security group. The firewall rules in the security group allow for access to the VM from the Internet through the HTTP, HTTPS, SSH and ICMP protocols. The target cloud service provider is OpenStack in this case.

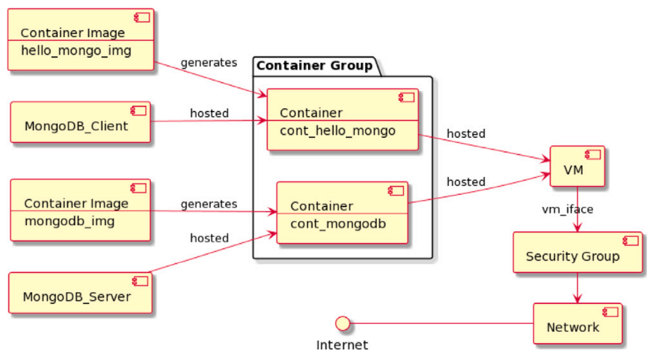


Fig. 10. Component diagram of the MongoDB application deployment.

Listing 10 shows an excerpt of the corresponding DOML model where, as part of the abstract infrastructure layer, a *ContainerGroup* is defined, which includes the two aforementioned containers. The components shown in Fig. 10 are part of a larger deployment; the corresponding completed DOML model can be found in [52].

During the IaC generation phase, the ICG detects the presence of the container group and produces the Docker Compose configuration shown in Listing 11, together with the Terraform code to provision the VM and the Ansible code to install the Docker Compose layers and then launch the generated configuration.

Listing 10: Container group definition in DOML.

```
cont_group vm1_cont_group {
 services {
  container cont_hello_mongo {
   host vm1 {
    cont_config {
     cont_port 5003
     vm_port 5002
     iface vm1_iface
    }
   }
  }
  container cont_mongodb {
   host vm1 {
    cont_config {
     cont_port 85
     vm_port 8085
     iface vm1_iface
    }
   }
  }
 }
}
```

Listing 11: Docker Compose file generated by the ICG.

```
version: '3'
services:
  cont_mongodb:
    image: .../mongo:4.2
    restart: on-failure
    ports:
    - "127.0.0.1:85:vm1_iface:8085"

  cont_hello_mongo:
    image: .../hello-mongo:latest
    restart: on-failure
    ports:
    - "127.0.0.1:5002:vm1_iface:5003"
```

The ICG is thus capable of automatically determining which target framework should be used for each component. Thus, it generates IaC

Table 5
Results of the DMC on the Wordpress deployment model. The numbers in the first column refer to requirements from Section 4.2.

#	Original		Defective	
	Time (s)	Satisfied	Time (s)	Satisfied
MC1	0.016	Yes	0.207	No
MC2	0.118	Yes	0.227	No
MC3	0.001	Yes	0.001	Yes
MC4	0.016	Yes	0.015	Yes
MC5	0.089	Yes	0.203	No
MC6	0.001	Yes	0.001	Yes

files in the main target language (Terraform in this case), referencing files in other languages needed to deploy and configure specific components (Ansible and Docker Compose). This example shows that RQ2b can be answered positively, at least for the given case.

Moreover, the DOML model representing the deployment of Fig. 10 further substantiates the positive answer to RQ1a.

6.4. RQ3: Can the DMC detect frequent mistakes during the definition of a DOML model?

We illustrate the DMC's capabilities on the DOML model of the Wordpress deployment described in Section 6.1. We ran the DMC on two versions of the Wordpress example: the original one, which we used in Section 6.2.1, and another one where we introduced some errors on purpose, to test the DMC's ability to detect them. All tests were run on a PC equipped with an AMD Ryzen 7 7700X CPU and 64 GiB of RAM running Ubuntu 23.10. We display the results in Table 5, where we reference, in the first column, requirements from Section 4.2 by their numbers.

The original version of the model is functional, as we saw in Section 6.2, and thus satisfies all requirements in Section 4.2.

In the defective version, we made the following modifications:

- we removed the network interface of the VM hosting Wordpress;
- we removed the `maps` attribute of the concrete network.

These modifications lead to three requirements from Section 4.2 not being satisfied:

- REq. MC1, because without its network interface, the Wordpress VM is linked to no network;
- REq. MC2, because according to the AL, the Wordpress VM should communicate with the database; but without being linked to the same network as the VM hosting the database, this is impossible;
- REq. MC5, because after removing the `maps` attribute of the network in the CIL, the network specified in the AIL is not associated to any element in the CIL.

The DMC also outputs helpful error messages identifying the offending components: for instance, for Req. 2 it outputs:

> Software components 'mysql' and 'wordpress' are supposed to communicate through interface 'DB_interface', but they are deployed to nodes that cannot communicate through a common network.

All requirements in Section 4.2 are checked in less than one second on the Wordpress example, which makes the DMC suitable for performing frequent checks, anytime the user wishes to check that a modification made to the model does not introduce other issues.

One of the common features of requirements MC1–MC6 is that they involve multiple DOML layers, checking that the lower ones are compatible with the upper ones. In fact, the different levels of abstraction of DOML layers allow them to act both as deployment descriptions and requirements for layers below. More specifically, the AL describes the overall application focusing on the roles and features of its components. Thus, we can check that the AIL and CIL are adequate to fulfill the requirements implicitly declared in the AL. This partially solves the problem of requirement specification by the user, one of the most relevant usability issues of verification solutions. The user can simply describe the application without thinking about verification: the requirements can be elicited from the AL by means of pre-defined properties.

6.5. RQ4: Is the DOML approach perceived as useful in a real-world case?

Recently, DOML has been used by the PIACERE project case study owners to model their systems deployment. In this section we provide an overview of the experiment carried out by the IT Directorate of the Slovenian Ministry of Public Administration (SI-MPA).

6.5.1. Case study overview

SI-MPA is hosting information systems on a centralized IT infrastructure based on WebSphere [53]. Resource provisioning, infrastructure configuration and deployment of new systems on such infrastructure are the Directorate's main challenges from an operational and security perspective. In this context, SI-MPA has experimented with the usage of DOML, providing feedback for its further extension. The focus of the experiments has rotated around NIO — National Interoperability Framework Portal system [54]. The challenging aspects have concerned, in particular, the microservice-oriented architecture adopted for NIO and the DevOps delivery style, which targets different types of infrastructures for different phases of the software lifecycle. The purpose was, in fact, to define a DOML model able to support continuous local deployment during the development process, continuous deployment on the private cloud during the staging process and, eventually, deployment on the production environment (private cloud or public cloud). NIO as such requires constant delivery and/or upgrading (due to new features, modules, and bug fixes). Having automated modeling and verification of IaC is expected to provide the following advantages:

- moving from a traditional to an agile way of delivering information systems, with short delivery times and less human resources for deploying, maintaining, and managing information systems;
- being able to reduce the cost of deploying IT solutions on various infrastructure platforms;
- greatly alleviating the delivery processes and reducing the number of errors;
- accurately documenting through models the main aspects of the deployment process, thus improving compliance with the requirements from IT management standards, guidelines, and good practices.

The team that experimented with the DOML usage was composed of five members: the team leader, who is also one of the authors of this paper, three system engineers, two of which experts in VMWare and one with expertise in LAMP webhosting, integration and maintenance of PHP open-source CMS systems, and an enterprise architect. All team members had a basic knowledge of Terraform and Ansible and between 10 and 40 years experience in their respective roles. The team exploited the DOML approximately for a one year period and had weekly meetings with the DOML developers to discuss encountered issues and new requirements. The objectives of the experiments were to model the case study, check its correctness with the DMC, generate the corresponding IaC code with the ICG, test the resulting code and verify that the system deployed as expected. This activity has proceeded incrementally, starting from simple modeling tasks, fixing bugs in the models, in the DOML as well as in the various components of the system, and adding new details at every iteration. The meetings have been highly beneficial for both parties and have resulted, on the one side, in the development of two versions of the SI-MPA DOML model presented in the following sections, and, on the other side, in new improvements in the language itself.

6.5.2. The SI-MPA case DOML model: First version

The DOML model developed by the SI-MPA team includes at the application level the components pertaining to the NIO system. Since each component was already associated to a corresponding Ansible code managing its configuration, SI-MPA decided, initially, to keep these pieces of code and link them from the DOML model as artifacts belonging to the corresponding component. In fact, Listing 12 shows that component `nio3` is associated to some codebase, `r1`, whose starting element is `main.yml` (the Ansible code) and that should be executed by an Ansible engine using `inventory.j2` as a source of information about the resources available. The `r1` codebase is packaged in the same project including the DOML model, in a special purpose directory called `assets`.

Listing 12: `nio3` software component.

```
software_component nio3 {
  source r1 {
    entry "main.yml"
    backend "ansible"
    inventory "inventory.j2"
  }
}
```

Listing 13: Part of the NIO abstract infrastructure layer.

```
vm vm1 {
    os "centos7_64Guest"
    cpu_count 4
    mem_mb 16384.00
    iface i1 {
      address "10.83.18.81"
      belongs_to net1
    }
    credentials ssh_key
}
storage disk0 {
    label "disk0"
    size_gb 100
}
```

In general, this same approach in DOML can be used in all cases we want to associate to a software component a specific piece of code and the corresponding executor.

In the abstract infrastructure, `nio3` is mapped into a virtual machine having 4 CPUs and 16 GB of memory that is accessed through an SSH key (see Listing 13).

In the DOML concrete layer, SI-MPA abstract infrastructure is mapped into resources available in the WebSphere datacenter owned by SI-MPA itself. Listing 14 shows the definition of the concrete VM used for the execution of `nio3` on WebShpere. It includes as DOML properties some parameters specific of WebSphere VMs and refers to three resources that are preexisting in the datacenter. Listing 15 shows, in particular, the definition of `template`, which is the preexisting image used to create the VM, and `datastore`, which is preexisting as well.

Listing 14: VM definition in the NIO concrete infrastructure layer.

```
vm con_vm1 {
  properties {
    host_name = "piac-0"
    domain    = "ad.sigov.si"
    disk      = "disk0"
    disk_size = "100"
    guest_id  = "centos7_64Guest"
  }
  refs_to {
    pool
    datastore
    template
  }
  maps vm1
}
```

Listing 15: Definition of the `template` and `datastore` preexisting resource.

```
vm_image template {
 preexisting true
 refs_to { dc }
 image_name "c7tmp"
 maps img
}

storage datastore {
 properties {
  vsphere_datastore_name = 'NFSShare01'
 }
 preexisting true
 refs_to { dc }
 maps disk0
}
```

In general, explicitly identifying preexisting resources in DOML is important to let the ICG know that provisioning is not required in that case.

The DOML model includes also a second concrete infrastructure running within an OpenStack cloud provider. The two can coexist in the same model, but only one at a time can be defined as `active`.

The created model can be verified successfully and can then be transformed into IaC organized in a .zip file which, in this case, includes the Terraform outcome of the model together with the original Ansible files associated to each individual NIO component.

This DOML model also corroborates the positive answer to RQ1a and demonstrates the usefulness of the internal extension mechanisms presented in Section 5.1.

6.5.3. The SI-MPA case DOML model: Second version

The development of the DOML model described in Section 6.5.2 was satisfactory from the viewpoint of the SI-MPA team who has addressed the problem of handling provisioning of the needed resources and, at the same time, has been able to reuse the preexisting Ansible scripts to manage deployment of components. It, however, opened up a debate with the DOML developers that were arguing about the importance of incorporating in the DOML model the information about the application structure without hiding it in the Ansible scripts. To demonstrate to the professionals the usefulness of this approach, as a further step in the experiment, we hired a Master's student, who had never collaborated with the DOML development team, and trained him on the DOML framework. He was given the task of employing the DOML-e extension mechanism to extend the DOML with new components required by the NIO platform deployment. Then, it took for him two days to understand the involved frameworks and to write a new DOML model in which several application and infrastructure components have been extracted from the Ansible file, and are now explicitly represented as DOML elements. Fig. 11 shows the component diagram of the new model, which is available at [27].

Starting from the AL, compared to the base model described in Section 6.5.2, in the new one the `nio3` software component was broken down into several components, increasing the degree of explicitness of the model. The first introduced element is *RabbitMQ* [55] (Listing 16), a message broker software that facilitates communication between distributed systems by enabling asynchronous messaging. *RabbitMQ* revolves around the concepts of *queue*, *exchange* and *binding*. A queue is a buffer storing messages awaiting processing by consumers. An exchange is a routing mechanism that directs messages from producers to queues based on specified rules. A binding defines the relationship between exchanges and queues, determining how messages are routed and delivered. This model contains an instance of each one of them, highlighting their most significant attributes. In this deployment, the role of RabbitMQ is to support message exchange between the front-end and back-end components of the application. The front-end application is still setup by an external Ansible file. An important application component in the back-end is *Redis* [56], an open-source, in-memory

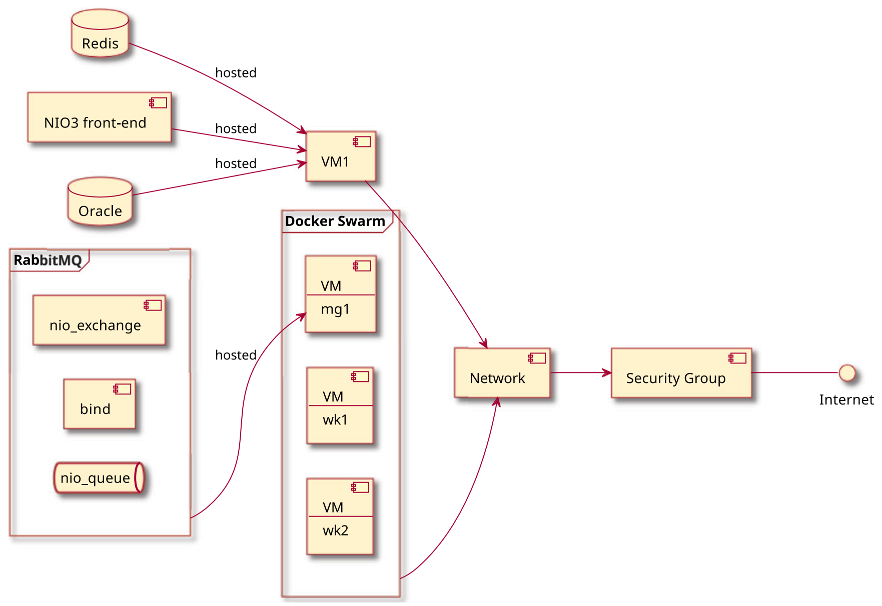


Fig. 11. Component diagram of the deployment of the NIO system.

data structure store, used as a database, cache, and message broker, known for its high performance and versatility. Here some of its main configuration parameters are specified, including replication mode and storage engine. Finally, an *Oracle* DBMS providing a software interface named dbAccess is defined.

Listing 16: RabbitMQ definition in the AL.

```
rabbit_mq rb {
  rabbit_mq_queue nio_queue {
    arguments {
      message_ttl = "6000"
    }
    durable true
  }

  rabbit_mq_exchange nio_exchange {
    exchange_type "direct"
    durable true
  }

  rabbit_mq_binding bind {
    queue nio_queue
    exchange nio_exchange
    routing_key "#"
  }
}
```

Listing 17: Definition of the Docker Swarm hosting RabbitMQ in the AIL.

```
swarm docker_swarm {
  manager mg {
    vm mg1 {
      os "centos7_64Guest"
      cpu_count 8
      mem_mb 16384.00
      iface iface_mg1 {
        address "10.83.18.131"
        belongs_to sub_swarm
      }
    }
  }

  worker wk {
    vm wk1 {
      os "centos7_64Guest"
      cpu_count 4
      mem_mb 8192.0
      iface iface_wk1 {
        address "10.83.18.132"
        belongs_to sub_swarm
      }
    }

    vm wk2 {
      os "centos7_64Guest"
      cpu_count 4
      mem_mb 8192.0
      iface iface_wk2 {
        address "10.83.18.133"
        belongs_to sub_swarm
      }
    }
  }

  network_mode "overlay"
}
```

Moving to the AIL, some new concepts were introduced, while some minor changes were performed on some elements already present in the original version of the model. First, a *Docker Swarm* was modeled to allow the execution of the *RabbitMQ* software (Listing 17). The swarm consists of one manager node and two worker nodes, which are VMs having different characteristics. The network mode configured for this swarm is an overlay network, the most common configuration for *Docker Swarms* [57]. To isolate network traffic in the swarm from the database, the existing network was divided into two subnets. Moreover, a security group was introduced to define traffic rules for the *Oracle* database [58] service, including some standard rules for both ingress and egress.

While the *RabbitMQ* component is deployed onto the swarm, all the other application components are deployed onto a single virtual machine, defined in the same way as in the original model, only having one more network interface.

In the CIL, the three vSphere resources previously modeled as preexisting generic resources were defined as first-class elements, introducing the *Datacenter*, *ComputeCluster* and *ResourcePool* classes at the metamodel level. The concrete network mapping the one defined at the AIL was then updated to be coherent with the new abstract structure, defining two subnets. Finally, the virtual machines defined for the *Docker Swarm* were concretized in an analogous way to the one used for the already existing virtual machine.

Table 6
Results of the DMC on the second version of the SI-MPA case DOML model. The numbers in the first column refer to requirements from Section 4.2.

#	Defective		Final	
	Time (s)	Satisfied	Time (s)	Satisfied
MC1	6.031	No	0.199	Yes
MC2	1.352	Yes	1.846	Yes
MC3	0.010	Yes	0.021	Yes
MC4	0.152	Yes	0.200	Yes
MC5	6.172	No	1.590	Yes
MC6	0.153	Yes	0.201	Yes

After writing a preliminary version of the model, the student checked it with the DMC. The check highlighted two issues: the VMs in the swarm were not linked to any network (rEq. MC1 from Section 4.2), and no corresponding concrete VMs had been defined in the CIL (rEq. MC5 from Section 4.2). After fixing these issues, the final version of the model was ready.

Table 6 reports the time taken by the DMC to check the two models on the same machine used in Section 6.4 (the student did not record check timings: those reported here have been obtained by one of the authors by repeating the experiment on the student's models).

6.5.4. Results

The adoption of DOML has helped the SI-MPA DevOps team in creating a model that fulfills their need to accurately document the main aspects of the deployment process. The team agreed that this has confirmed their hypothesis concerning the possibility to keep compliance with IT management standards, guidelines, and good practices under control, through the adoption of the DOML.

The team has also experimented with the possibility to deploy their system both on WebSphere and OpenStack by simply changing the `active` infrastructure field in the DOML model, running again the IaC generation and then executing the resulting code.

We could not measure the time saving introduced by the usage of the DOML in this case study as the SI-MPA team collaborated with the DOML development team through multiple iterations during the experiment. This has been necessary to align the language and the generation mechanism to the team's needs.

At the end of all iterations, we interviewed the SI-MPA team to acquire their feedback about the usability of the DOML approach. They acknowledged its utility, especially to easily deploy on different providers, and usability, but highlighted that, having them already acquired a significant experience in the development of Ansible and Terraform IaC, they were not in real need of adopting the DOML, which could be, instead, suitable for less experienced teams.

After having shown the student's model of their deployment (cf. Section 6.5.3), they were favorably impressed by the possibility to express, as part of the DOML and by exploiting the extension mechanisms, the many components of their system in an explicit manner. They realized that the obtained model could have been considered also as an architectural documentation and that it would be beneficial to help newcomers in the team understanding their system. Moreover, the experiment with the student highlighted the benefits of analyzing models and checking their consistency through the DMC.

The development of this case study has been also beneficial to the improvement of the DOML that has been extended to incorporate the following aspects:

- The possibility to associate components to a codebase of any complexity and to specify the corresponding executor (see Section 5.1). The initial version of the DOML, in fact, was working under the assumption of simple codebases constituted by a single file leaving the executor completely unspecified.
- The possibility to map abstract infrastructure elements on top of preexisting ones. In fact, in the SI-MPA WebSphere infrastructure some resources, i.e., the VM images to use, network, datastore, and some clusters were already up and running, while VMs needed to be provisioned as part of the deployment process. To accommodate this aspect, we have realized the importance of explicitly specifying as *preexisting* those resources for which we do not need to generate the corresponding provisioning code, and, to this end, we have introduced the *GenericResource* concept as explained in Section 5.1 and we have tested also the possibility to express them as proper extensions of metamodel elements. Such resources, of course, must to be properly correlated with the ones to be provisioned through the execution of the IaC resulting from the model, as shown in Listing 14 where VM `con_vm1` is not preexisting, but, when created, refers to the `pool`, `datastore` and `template` other resources.

7. Discussion

The evaluation in Section 6 has demonstrated the capabilities and shortcomings of DOML by means of two examples, a concrete case study, and a benchmark set of IaC languages.

7.1. Answers to the research questions

We summarize the answers to the four research questions RQ1–RQ4 below:

- RQ1: *How effective is DOML at modeling deployments?*
  We divided RQ1 into two sub-RQs (RQ1a and RQ1b). Their answers are based on the Wordpress deployment introduced in Section 6.1 and also on deployments introduced in the rest of Section 6.
  RQ1a: *Can a DOML model represent the information required to generate executable IaC tackling both provisioning and configuration?* To answer RQ1a, we wrote a DOML specification of the Wordpress deployment targeting OpenStack as a CSP (Section 6.2.1). The ICG is able to successfully deploy Wordpress on the infrastructure described by the DOML model, suggesting a positive answer to RQ1a. Moreover, in Section 6.3.1 we show that DOML can represent the information needed to deploy on different CSPs. Through this example we can also observe that the ICG can automatically generate commonly used components from its templates when needed, even when they are not explicitly defined in the DOML. This is a feature that can be of considerable help to inexperienced users. For instance, in the Wordpress deployment, the ICG creates automatically the SecurityGroup, so it is not necessary to include it in the DOML model.
  The experiment in Section 6.3.2 (i.e., the MongoDB application deployment) demonstrates that DOML can represent a deployment containing an application running on a container group, while the case study in Section 6.5 shows that it can represent a real-world cloud application with multiple VMs (the NIO platform of SI-MPA) and that it can enable the reuse of existing IaC fragments if needed.
  RQ1b: *How does DOML compare with the state-of-the-art approaches?* We answered RQ1b from a qualitative and a conciseness point of view. We compared the DOML specification of the Wordpress example with two equivalent specifications in Terraform and Cloudify. The comparison shows that, in DOML, the structure of the application and its mapping to the infrastructure is explicit and clearer, taking advantage of the modeling abstractions offered by the language. On the contrary, the Terraform and Cloudify scripts alone do not provide complete information and a complete overview can be obtained only by analyzing also the other Shell and Ansible scripts.

According to the metrics collected in Table 4, the DOML model for the Wordpress deployment is smaller than the ones in Terraform and Cloudify in terms of number of lines of codes and files. Moreover, both alternative approaches require using an additional language for application configuration.

- RQ2: *How effective are DOML and the ICG at targeting different existing cloud frameworks and technologies?*
  We answered RQ2 along two lines: the ICG's ability to generate code targeting different CSPs (RQ2a), and to integrate different IaC languages (RQ2b).
  RQ2a: *Is a DOML model able to target multiple execution platforms?* In Section 6.3.1 we show that the abstraction mechanisms offered by DOML allow for having the AIL decoupled from the CSPs targeted by the deployment, which makes it easier to switch between different CSPs by just adding and activating a new concrete infrastructure. On the contrary, switching CSPs requires extensive rewriting in Terraform and Cloudify. Moreover, DOML provides better consistency between the actual deployments on the different providers and better readability.
  RQ2b: *Is the ICG able to generate code in different IaC languages, required to implement different components of the deployment?* The experiment in Section 6.3.2 shows that the ICG can generate and integrate code in different IaC languages. In particular, to deploy the infrastructure hosting a MongoDB application, the ICG generates Terraform, Ansible and Docker Compose code, thus supporting the steps of provisioning, configuration, deployment and orchestration.
- RQ3: *Can the DMC detect frequent mistakes during the definition of a DOML model?*
  In Section 6.4, we check an erroneous version of the DOML model introduced in Section 6.2.1 with the DMC. Moreover, in Section 6.5.3 we show the benefit of the DMC when it is used in an iterative mode during the development of a DOML model. This shows how the DMC can help users ensure the correctness of their DOML models by detecting frequent mistakes during its definition. This gives a positive answer to RQ3.
- RQ4: *Is the DOML approach perceived as useful in a real-world case?*
  In Section 6.5, DOML has been used by PIACERE use case owners to model a real-world system deployment. The SI-MPA system has been deployed both on Websphere and OpenStack by changing only the active infrastructure field in the DOML model, then running the IaC generation and executing the resulting code. The SI-MPA team has acknowledged the utility and usability of the DOML approach, stating that it would be particularly useful for less experienced users.
  Moreover, we show how the DOML was extended to better model the deployment of this case study. The resulting DOML model shows how DOML allows even relatively inexperienced users to define more complex real-world deployments.

7.2. Comparison of DOML with state-of-the-art approaches

To compare DOML with the other state-of-the-art approaches, we have tried to analyze to what extent they are able to fulfill the requirements we have defined for the DOML. Table 7 provides a summary of our analysis. Besides Terraform and Cloudify, which we have compared directly with the DOML in Section 6, we consider here also Ansible, which focuses on configuration and deployment; Docker Compose and Kubernetes, which focus on orchestration; and Radius that, as mentioned in Section 2, appears to share the same goals as DOML in terms of ability to abstract from the details of the individual IaC languages. Of course, when referring to other approaches, in each requirement definition, DOML is meant to be replaced by the name of the considered approach. We note the complete fulfillment of the requirement by a check-mark while the symbol ±stands for a partial fulfillment. In the case of *REQ76—DOML should allow the user to model information needed for each of the four considered DevOps activities (provisioning, configuration, deployment, orchestration)*—we specify the supported activities as follows: P is for provisioning, C for configuration, D for deployment and O for orchestration.

Table 7
Fulfillment of requirements by DOML and the most well-known tools (T/C stands for TOSCA/Cloudify, DC for Docker Compose, and K8s for Kubernetes).

ReqID	Terraform	T/C	Ansible	DC	K8s	Radius	DOML
REQ63	✓	✓	✓	✓	✓	✓	✓
REQ62							✓
REQ70	±		±				✓
REQ76	PD	PD	DC	O	DO	PDO	PDCO
REQ57							
REQ36	✓		✓				
REQ111	✓	✓	✓	✓	✓	✓	✓
REQ01	✓	✓					✓
REQ27	✓	✓	±		±	✓	✓
REQ28		✓	±	✓	✓	✓	✓
REQ29	✓	±	±			±	✓
REQ25	✓	✓	✓	✓	✓	✓	✓
REQ26	✓	✓	✓	✓	✓	✓	✓
REQ60	✓	✓	✓	✓	✓	✓	✓

From the table, we can see that DOML is the only one supporting different views on a deployment (REQ62), as it offers the possibility to model it at different abstraction levels. Moreover, DOML is the only one offering a complete model checking approach (REQ70), while Terraform and Ansible offer static analysis and linting tools helping with syntactical and code smells checks [59]. As for REQ76, DOML is the only approach that is able to address all the four DevOps activities, while the others tend to specialize to a subset of them. REQ57 is not fulfilled by the current version of the DOML but is planned for a future iteration; to our knowledge, it is not fulfilled by Radius, while it is not relevant for the other approaches are they do not offer mechanisms for translating from a high-level representation to a lower-level one. REQ36, concerning the ability to write infrastructure tests, is addressed only by Terraform and Ansible, for which testing frameworks have been defined. REQ111, that is, the ability of an approach to incorporate other IaC, is – to a certain extent – fulfilled by all approaches, while REQ01 is fulfilled only by DOML, Terraform, and Cloudify, as they describe infrastructural elements. REQ27 and REQ28 are fulfilled by all approaches, because they all allow for containers to be used and containerized applications to be modeled. Concerning the ability to model VM provisioning for different platforms (REQ29), we argue that the fulfillment is partial for Cloudify since the target provider cannot be specified in a script, even if the Cloudify executor can be configured to run Compute nodes on different providers; it is also partial for Ansible since anything can be done procedurally, but provisioning is not the main task for which Ansible has been developed; similar considerations apply to Radius, which can exploit Terraform to this purpose. Security rules (REQ25), security groups (REQ26), and security metrics (REQ60) can be modeled by all approaches, even though in different manners and at different levels of detail, considering the specific focus of each approach.

To sum up, with reference to the requirements identified within the PIACERE project, DOML seems to be the most complete approach, except for requirements REQ36 and REQ57 which could not be addressed so far. The DOML has been developed with the purpose of simplifying the definition of cloud deployments, both to help inexperienced users, and to give professionals a technology that offers powerful abstraction mechanisms that can help in creating and maintaining even complex deployments. The cloud computing landscape is, however, much wider than what is currently supported by our framework (according to Section 3.4), and commercially-supported tools of course offer a much wider array of features and components—although, as we saw in the previous sections, multiple languages and tools must often be combined together. Indeed, the DOML implementation presented in this paper is a proof of concept, developed to the extent that was possible with the limited resources available for the project. Its purpose is to demonstrate that the ideas behind the approach, and in particular the organization of the language in three abstraction layers, are beneficial for specifying deployments.

To account for the need to support more components in more complex deployments, we have introduced the DOML extension mechanism (Section 5), and demonstrate its use with an extension of the case study in Section 6.5.3. This extension mechanism is made possible by the modularity of the ICG, as we explain in Section 4.3. Thus, experienced users can use DOML to model complex deployments by extending it, and they can access all configuration options offered by CSPs by adding attributes to the *concrete infrastructure layer (CIL)*. It is true that this requires considerable knowledge of technicalities specific to these CSPs, but we argue that the DOML still offers benefits in terms of robustness and especially a more structured way of describing a deployment, which constitutes an architectural documentation useful for helping newcomers understand the deployment and learning incrementally how to extend it. This was acknowledged by the SI-MPA team involved in the case study of Section 6.5. However, less experienced users are still able to create less complex deployments, but with considerably less expertise than competing approaches. This is due, in particular, to the ICG's ability to automatically fill-in CSP settings with sensible defaults when they are not specified by the user.

7.3. Limitations and threats to validity

The experimental evaluation conducted in Section 6 has some limitations that will be addressed through further research.

In particular, we compared the definition of the Wordpress deployment in DOML with some specific definitions in other IaC languages. The main conclusion we inferred from the evaluation is that the DOML-based definition is more concise than the benchmark approaches. This represents a threat to internal validity, because the analysis of a few specifications does not allow us to exclude the possibility of making more concise ones. Nonetheless, such specifications were written by experts, and we argue that it is unlikely that such a large conciseness gap can be recovered.

Another threat to internal validity is that we have only compared metrics concerning code size as a measure of both ease of writing and maintaining code. In part, the ease of use has been demonstrated through the SI-MPA case study presented in Section 6.5. However, a more precise evaluation of such features would require a complete empirical study.

The last – but not least – threat to external validity is that the evaluation is performed on a few typical application deployments, so the claims we make may not generalize to other more complex cloud applications. One of our next research steps will be the evaluation on a larger benchmark suite.

8. Conclusion

We have presented the DOML, a novel approach for cloud deployment modeling and it has been shown that the approach works for relatively simple but complete systems with practical significance.

This paper shows that the DOML can decrease the expertise required to create cloud application deployments, and its abstraction mechanisms can be helpful even in large deployments that inherently require greater expertise. For the most complex deployments, a certain domain knowledge is still required, but we believe that the model-based approach and the possibility to extend the language bring definite benefits also in this case, and are a tangible step in the direction of improving the user experience of DevOps teams.

The proposed approach offers the functionality to write the DOML model only once and then use it to deploy the same complex system on different cloud service providers or physical machines. This has resulted in the definition of the DOML as a multi-layer modeling language (see Section 3), where the application and abstract infrastructure layers include a platform-independent specification of the application and its underlying infrastructure, while the concrete infrastructure layer specifies the details associated to the actual deployment on a specific platform. By means of the examples presented in Section 6, we demonstrated that DOML is complete enough to model a whole deployment by itself. Moreover, the DOML has been developed keeping in mind the need for extensibility, and includes an extension mechanism called DOML-e (see Section 5), which will be analyzed in more detail in a future work.

Ultimately, our next challenge is to check whether the approach is usable and works with other case studies different from the ones that have guided its development and that have used application and infrastructure-level components that we have not considered so far.

What is presented in this paper is only part of the PIACERE framework, which also includes components that handle performance optimization, monitoring, and runtime adjustment of deployments. Their presentation is available in the project deliverables[5] and will be detailed in separate publications.

CRediT authorship contribution statement

Michele Chiari: Conceptualization, Formal analysis, Methodology, Software, Writing – original draft, Writing – review & editing. **Bin Xiang:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Visualization, Writing – original draft, Writing – review & editing. **Sergio Canzoneri:** Conceptualization, Formal analysis, Methodology, Visualization, Writing – original draft, Writing – review & editing. **Galia Novakova Nedeltcheva:** Writing – original draft, Writing – review & editing. **Elisabetta Di Nitto:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Supervision, Visualization, Writing – original draft, Writing – review & editing. **Lorenzo Blasi:** Conceptualization, Data curation, Methodology, Project administration, Software, Supervision, Writing – original draft, Writing – review & editing. **Debora Benedetto:** Software. **Laurentiu Niculut:** Conceptualization, Methodology, Software. **Igor Škof:** Software.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Michele Chiari, Bin Xiang, Galia Novakova Nedeltcheva reports financial support was provided by Horizon Europe. Michele Chiari reports financial support was provided by Vienna Science and Technology Fund. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data has been made available in a GitHub repository referenced in the article.

[5] See the project website https://piacere-project.eu/public-deliverables/.

References

[1] K. Morris, Infrastructure as Code, O'Reilly Media, 2016.
[2] A. Bergmayr, U. Breitenbucher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann, A systematic review of cloud modeling languages, ACM Comput. Surv. 51 (1) (2018) http://dx.doi.org/10.1145/3150227.
[3] B. Selic, The pragmatics of model-driven development, IEEE Softw. 20 (5) (2003) 19–25, http://dx.doi.org/10.1109/MS.2003.1231146.
[4] HashiCorp, Inc, Terraform documentation, 2022, URL https://www.terraform.io/docs.
[5] Cloudify Platform Ltd., Cloudify DevOps automation and orchestration platform, multi cloud, 2023, URL https://cloudify.co/.
[6] OASIS Standard, Topology and Orchestration Specification for Cloud Applications Version 1.0, Tech. Rep., OASIS Standard, 2013, URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html.
[7] M. Chiari, B. Xiang, G.N. Nedeltcheva, E. Di Nitto, L. Blasi, D. Benedetto, L. Niculut, DOML: A new modelling approach to infrastructure-as-code, in: Advanced Information Systems Engineering - 35th International Conference, CAiSE 2023, Zaragoza, Spain, June 12-16, 2023, Proceedings, in: LNCS, vol. 13901, Springer, 2023, pp. 297–313, http://dx.doi.org/10.1007/978-3-031-34560-9_18.
[8] Chef, Chef infra: Powerful policy-based configuration management system software, 2022, URL https://www.chef.io/products/chef-infra.
[9] Puppet, Inc., Puppet, 2022, URL https://puppet.com/.
[10] Red Hat, Inc, Ansible documentation, 2022, URL https://docs.ansible.com/.
[11] A. Colantoni, L. Berardinelli, M. Wimmer, DevOpsML: Towards modeling DevOps processes and platforms, in: Proc. MODELS'20, ACM, 2020, pp. 69:1–69:10, http://dx.doi.org/10.1145/3417990.3420203.
[12] Docker, Docker, 2022, URL https://www.docker.com/.
[13] I. Miell, A. Sayers, Docker in Practice, Simon and Schuster, 2019.
[14] CNCF, Kubernetes, 2022, URL https://kubernetes.io/.
[15] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin, D.A. Tamburri, Infrastructure-as-code for data-intensive architectures: A model-driven development approach, in: Proc. ICSA'18, 2018, pp. 156–165, http://dx.doi.org/10.1109/ICSA.2018.00025.
[16] L. Baresi, E. Di Nitto, D. Vladušič, The SODALITE approach: An overview, in: E. Di Nitto, J. Gorroñogoitia Cruz, I. Kumara, D. Radolović, K. Tokmakov, Z. Vasileiou (Eds.), Deployment and Operation of Complex Software in Heterogeneous Execution Environments: The SODALITE Approach, Springer International Publishing, Cham, 2022, pp. 9–21, http://dx.doi.org/10.1007/978-3-031-04961-3_2.
[17] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov, The EDMM modeling and transformation system, in: Proc. ICSOC 2019 Workshops, LNCS 2019, Springer, 2020, pp. 294–298, http://dx.doi.org/10.1007/978-3-030-45989-5_26.
[18] Azure Incubations Team, Radius, 2023, URL https://radapp.io/.
[19] Firefly, Artificial Intelligence Infrastructure-as-Code Generator, 2023, URL https://aiac.dev/. (Accessed: 09 November 2023).
[20] F5, Inc, NGINX, 2022, URL https://www.nginx.com/.
[21] The OpenStack Project, OpenStack, 2022, URL https://www.openstack.org/.
[22] The PIACERE Project Partners, DOML, 2023, URL https://git.code.tecnalia.com/piacere/public/the-platform/doml.
[23] Eclipse Foundation, Eclipse Modeling Framework (EMF), 2022, URL https://www.eclipse.org/modeling/emf/.
[24] Eclipse Foundation, Xtext, 2022, URL https://www.eclipse.org/Xtext/.
[25] B. Xiang, E. Di Nitto, G.N. Nedeltcheva, Deliverable D3.2 PIACERE Abstractions, DOML and DOML-E - v2, 2023, http://dx.doi.org/10.5281/zenodo.7645687, Zenodo.
[26] The PIACERE Project, PIACERE DevSecOps Modelling Language (DOML), 2022, URL https://www.piacere-doml.deib.polimi.it/.
[27] The PIACERE Project, IaC artefacts repository, 2022, URL https://github.com/michiari/DOML-case-study.
[28] J. Díaz de Arcaya, Deliverable D5.3 - IaC Execution Manager Prototype - v3, Zenodo, PIACERE Consortium, 2023, http://dx.doi.org/10.5281/zenodo.8299722.
[29] E. Villanueva, I. Torres, E. Osaba, S. Canzoneri, A. Franchini, L. Blasi, PIACERE integrated development environment, in: 3rd Eclipse Security, AI, Architecture and Modelling Conference: On Cloud to Edge Continuum, ESAAM 2023, Ludwigsburg, Germany, 17 October 2023, ACM, 2023, pp. 62–66, http://dx.doi.org/10.1145/3624486.3624507.
[30] L.M. de Moura, N.S. Bjørner, Z3: An efficient SMT solver, in: Proc. 14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08, in: LNCS, vol. 4963, Springer, 2008, pp. 337–340, http://dx.doi.org/10.1007/978-3-540-78800-3_24.
[31] Microsoft Research, The Z3 theorem prover, 2022, URL https://github.com/Z3Prover/z3. (Accessed: 02 May 2022).
[32] J.R. Burch, D.L. Dill, Automatic verification of pipelined microprocessor control, in: Proc. 6th Int. Conf. Computer Aided Verification, CAV'94, in: LNCS, vol. 818, Springer, 1994, pp. 68–80, http://dx.doi.org/10.1007/3-540-58179-0_44.
[33] C.W. Barrett, C. Tinelli, Satisfiability modulo theories, in: E.M. Clarke, T.A. Henzinger, H. Veith, R. Bloem (Eds.), Handbook of Model Checking, Springer, 2018, pp. 305–343, http://dx.doi.org/10.1007/978-3-319-10575-8_11.
[34] The PIACERE Project Partners, DOML model checker, 2023, URL https://git.code.tecnalia.com/piacere/public/the-platform/doml-model-checker.
[35] The PIACERE Project Partners, ICG – Infrastucture as code generator, 2023, URL https://git.code.tecnalia.com/piacere/public/the-platform/icg.
[36] Z. Chared, S.S. Tyszberowicz, Projective template-based code generation, in: CAiSE Forum, 2013, pp. 81–87, URL https://ceur-ws.org/Vol-998/Paper11.pdf.
[37] L. Luhunu, E. Syriani, Comparison of the expressiveness and performance of template-based code generation tools, in: Proc. SLE'17, ACM, 2017, pp. 206–216, http://dx.doi.org/10.1145/3136014.3136021.
[38] Eclipse Foundation, Acceleo home page, 2022, URL https://www.eclipse.org/acceleo/.
[39] Apache Software Foundation, The Apache velocity project, 2022, URL https://velocity.apache.org/.
[40] The Pallets Project, Jinja, 2022, URL https://jinja.palletsprojects.com/.
[41] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, & Tools, Pearson Education India, 2007.
[42] M. Pagel, et al., Pyecore/pyecore: A Python(nic) implementation of EMF/Ecore (Eclipse Modeling Framework), 2022, URL https://github.com/pyecore/pyecore.
[43] M. Eysholdt, EMF Ecore Based Meta Model Evolution and Model Co-Evolution (Master's thesis), University of Oldenburg, 2009.
[44] L. Bettini, D. Di Ruscio, L. Iovino, A. Pierantonio, Supporting safe metamodel evolution with edelta, Int. J. Softw. Tools Technol. Transf. 24 (2) (2022) 247–260, http://dx.doi.org/10.1007/S10009-022-00646-2.
[45] W. Kessentini, H.A. Sahraoui, M. Wimmer, Automated metamodel/model co-evolution: A search-based approach, Inf. Softw. Technol. 106 (2019) 49–67, http://dx.doi.org/10.1016/J.INFSOF.2018.09.003.
[46] M. Herrmannsdoerfer, S.D. Vermolen, G. Wachsmuth, An extensive catalog of operators for the coupled evolution of metamodels and models, in: B. Malloy, S. Staab, M. van den Brand (Eds.), Software Language Engineering, Springer, Berlin, Heidelberg, 2011, pp. 163–182.
[47] Eclipse Foundation, Eclipse Edapt, 2014, URL https://eclipse.dev/edapt/.
[48] Eclipse Foundation, Eclipse Edapt - Operations, 2014, URL https://eclipse.dev/edapt/operations.php.
[49] L. Rebouças de Carvalho, A.P. Favacho de Araújo, Performance comparison of terraform and cloudify as multicloud orchestrators, in: Proc. CCGRID'20, IEEE, 2020, pp. 380–389, http://dx.doi.org/10.1109/CCGrid49817.2020.00-55.
[50] L. Rebouças de Carvalho, Cloud orchestrators exam, 2019, GitHub.com, URL https://github.com/leonardoreboucas/cloud-orchestrators-exam.
[51] Amazon Web Services, Inc., AWS EC2, 2022, URL https://aws.amazon.com/ec2/.
[52] The PIACERE Project Partners, DOML model of the Ericsson use case (Ericsson_DOML_Latest.doml), 2024, https://git.code.tecnalia.com/piacere/public/the-platform/doml/-/blob/main/case-studies-DOML-models/Ericsson_DOML_Latest.doml. (Accessed: 17 May 2024).
[53] IBM, WebSphere application server, 2023, URL https://www.ibm.com/products/websphere-application-server.
[54] SI-MPA, National interoperability framework portal system (NIO), 2023, URL https://nio.gov.si/nio/vstopna.nio?lang=en.
[55] Rabbit Technologies Ltd., RabbitMQ, 2006, URL https://www.rabbitmq.com/.
[56] Redis Labs, Redis, 2011, URL https://redis.io/.
[57] Docker, Docker Swarm, 2015, URL https://docs.docker.com/engine/swarm/.
[58] Oracle Corporation, Oracle DB, 2024, URL https://www.oracle.com/database/.
[59] M. Chiari, M. De Pascalis, M. Pradella, Static analysis of infrastructure as code: a survey, in: IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022, IEEE, 2022, pp. 218–225, http://dx.doi.org/10.1109/ICSA-C54293.2022.00049.