An Infrastructure Modelling Tool for Cloud Provisioning

Julio Sandobalin[†*], Emilio Insfran[*], Silvia Abrahao[*]

[†]Departamento de Informática y Ciencias de la Computación
Escuela Politécnica Nacional
Quito, Ecuador
julio.sandobalin@epn.edu.ec

[*]Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Valencia, España
{jsandobalin, einsfran, sabrahao}@dsic.upv.es

Abstract— **Cloud computing offers computing, network, and storage capabilities through services that abstract the capabilities of the underlying hardware. Currently, a variety of tools exist that manage the infrastructure provisioning and use scripts to define the final state of the hardware to be deployed in the cloud. However, there are major challenges that need to be addressed to automate the infrastructure management so that they are effectively used in initiatives such as DevOps. In particular, the management of Infrastructure as a Code (IaC) is one of the most important technical challenges to support activities such as the integration, deployment, and continuous delivery of applications. To address this problem, we present a support for the management of DevOps tools, through the definition of a Domain Specific Language (DSL) based on the concept of Infrastructure as a Code, and a tool that supports this language allowing to model the final state of a provisioning infrastructure in the cloud and generating the provisioning scripts for the Amazon Web Services (AWS) platform. The proposed tool reduces the work for development and operations personnel and facilitates their communication.**

Keywords—Infrastructure as Code; DevOps; Infrastructure Provisioning; Cloud Services; Model Driven Development.

I. Introduction

Over the last years, an important technology change started, and we are leaving the age where to deploy a software application it was needed to physically install both hardware and operating systems with the whole necessary software. Nowadays, we are coming into a new *cloud age* where with a few clicks and short time, we can get a virtual machine with the whole necessary software for its proper operation. Cloud Computing [1] are hardware-based services offering computing, networking and storage capacity where hardware management is highly abstracted and infrastructure capacity is highly elastic. However, to deal with the high-demand and the very short time development cycles, cloud provisioning needs to be automated.

A new trend called DevOps [2] is encouraging continuous collaboration between developers and operation staff through of a set of principles and practices which optimize the delivery time of software, manage the Infrastructure as Code (IaC), and improve the user experience on the base of their feedback. Infrastructure as Code [3] is an approach to infrastructure automation based on software development practices that emphasize the use of consistent and repeatable routines for the hardware provisioning. This approach allows practitioners to align software development tools such as version control systems, automated testing libraries, and orchestration tools to manage the infrastructure. Commonly, companies use a number of interrelated tools to support their DevOps effort. The configuration and management of these tools are complex and time consuming.

As far as we know, there are no approaches that provide guidelines or automated support to manage a DevOps toolchain based on model-driven techniques. Model-driven techniques are mainly based on two well-known principles: abstraction and automation. For this reason, we propose an *infrastructure provisioning pipeline* based on DevOps practices where the models and the tool configuration scripts to be created for the infrastructure provisioning follow the model-driven principles.

The approach provides the necessary abstractions to deal with the complexity of using different tools to automate continuous delivery practices in cloud provisioning. Figure 1 presents an overview of the infrastructure provisioning pipeline.

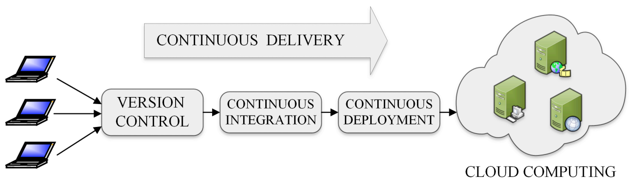


Fig. 1. Overview of infrastructure provisioning pipeline.

We take advantage of the Infrastructure as Code concept to apply DevOps practices by supporting the automatic generation of scripts to manage the tools that are used for provisioning in DevOps community.

First, we model the infrastructure provisioning to obtain an infrastructure model. Then, we take the infrastructure model and push it toward a *version control* system. We use a version control system in order to retain and provide access to every version of every infrastructure model that has ever been stored on it. Moreover, this approach allows teams that may have infrastructure models across different places to work collaboratively. The aim to use a version control system is to have everything that can possibly change at any point in the infrastructure model life cycle stored in a controlled manner.

Every infrastructure model must be checked into a single version control repository in order to begin the *continuous integration* stage. Continuous integration requires that every time developers and operation staff commits any change, the entire scripts for the provisioning tools are generated and a comprehensive set of automated tests are run against them. The objective is that all the scripts are up-to-date and ready to carry out the infrastructure provisioning successful. The *continuous deployment* stage take these scripts from the previous stage and automatically deploy them toward a cloud platform. The scripts are for DevOps community tools in order to carry out infrastructure provisioning in cloud platforms.

Finally, a *Continuous Delivery* takes advantage of previous stages to provide the capacity to release toward cloud platforms new scripts several times a day in order to carry out the infrastructure provisioning.

Currently, there are several tools to manage the infrastructure provisioning which use scripts to define the final state of infrastructure in the cloud. However, as previously mentioned, managing scripting languages of different tools from the DevOps community for infrastructure provisioning is a time-consuming and error-prone activity that practitioners are facing.

To mitigate this situation, we propose ARGON (An infRastructure modellinG tool for clOud provisioNing), which aims to abstract the complexity of working with different DevOps tools through a Domain Specific Language (DSL). It allows modelling an infrastructure mode with the final state of the infrastructure that need to be deployed into different cloud platforms. In addition, model-to-text transformations are performed to generate the corresponding scripts needed to manage the different DevOps provisioning tools used in the organization. Furthermore, we manage the Continuous Delivery stage shown in Figure 1 in order to use scripts generated by ARGON and to perform the infrastructure provisioning in cloud platforms. The main advantages of using ARGON are the following:

- It does not require advance knowledge on DevOps tools.
- It allows both development (Dev) and Operations (Ops) personnel to perform high-level modelling of cloud capabilities such as computing, storage, networking, and elasticity.
- It allows the automatic generation of infrastructure provisioning scripts for cloud provisioning tools.

Despite the fact that we propose an infrastructure provisioning pipeline where scripts are built in the *Continuous Integration* stage, we also developed an Eclipse plug-in to support the generation of scripts to show the usefulness of our infrastructure modelling tool.

The remainder of this paper is structured as follows: Section 2 discusses related works and identifies the needs to infrastructure provisioning in cloud computing. Section 3 presents the proposed DSL for infrastructure provisioning and the ARGON architecture. Section 4 introduces an illustrative case study that shows the proposed approach for modelling the infrastructure provisioning and an excerpt of the generated script. Finally, Section 5 presents our conclusions and future work.

II. Related Work

In recent years, there has been much interest in approaches and strategies to support cloud provisioning. Among these approaches, there are several infrastructure modelling approaches such as CloudFormation[1] and AWS OpsWork[2] in Amazon Web Services (AWS).

CloudFormation allows users to create template files which can be loaded into AWS to create stacks of resources. While the format that Amazon uses for the templates is easy to use, the structure and semantics of the template is not used by any other provider or cloud management tooling. On the other hand, OpsWorks is a configuration management service that helps users to configure and operate applications by using Chef[3]

TOSCA [4][5][6] is a standard for Topology and Orchestration Specification for Cloud Application which allows modelling nodes (virtual or physical machines) and orchestrates the deployment of cloud applications. TOSCA uses DevOps tools such as Chef to infrastructure provisioning and Juju[4] to deployment of cloud applications. In [7] is presented how TOSCA classifies DevOps tools in node-centric artefacts and environment-centric artefacts. The former are scripts that run on a server, virtual machine, or container for infrastructure provisioning. The latter are scripts that run on multiple nodes and support the deployment of cloud applications.

Several efforts aimed to offer support for designing, optimizing, and managing cloud applications. Specifically, several EU projects provided languages and methodologies to support the design of cloud applications (e.g., Cloud Application Modelling Language (CAML) [8], Cloud Application Modelling and Execution Language (CAMEL) [9]). However, to the best of our knowledge, none of them provide models that support the deployment and management of cloud applications.

Cloud WorkBench [10] is a framework based on Infrastructure as Code concept to foster simple definition, execution, and repetition of benchmarks over a wide array of cloud providers and configurations. The definition of client virtual machines and provisioning configurations follows the established notions of DevOps tool such as Vagrant[5] and Chef.

MORE [11] is a model-driven operation service for cloud-based IT systems that focuses on automating the initial deployment and the dynamic configuration of a system. MORE provides an online modelling environment to define a topology model to specify system structure and desired state. MORE transforms the topology model into executable code for Puppet[6] tool in order to get virtual machines, physical machines, and containers.

MODAClouds [12] is an European project undertaken to simplify the Cloud service usage process. One of its goals is delivering an Integrated Development

[1] https://aws.amazon.com/es/documentation/cloudformation/
[2] https://aws.amazon.com/opsworks/
[3] https://www.chef.io
[4] https://jujucharms.com
[5] https://www.vagrantup.com
[6] https://puppet.com

Environment (IDE) to support systems developers in building and deploying applications, together with related data, to multi-Clouds spanning across the full Cloud stack. Energizer 4Clouds is an executable platform from MODAClouds which includes automatic infrastructure provisioning using specially-designed Puppet modules, the ability to use existing infrastructure, and an API middleware for job control.

In summary, current approaches have focused most of their efforts in reusing the tools proposed by the DevOps community to solve gaps related to infrastructure provisioning and deployment of cloud applications. Differently, ARGON abstracts the complexity of working with the different DevOps provisioning tools through defining a DSL to model the final state of infrastructure in the Cloud and then automatically generates the scripts for particular provisioning tools.

III. ARGON

ARGON is a modelling tool for specifying the final state of the infrastructure provisioning of cloud resources and generating the scripts to manage the provisioning tools used in the DevOps community. In the following subsections, we explain the core elements of the DSL and the architecture of the modelling tool.

A. *DSL to model the infrastructure provisioning*

There is a wide range of cloud providers and tools that can be used to support the development of cloud applications. For example, Puppet, Chef, or Vagrant, which are very popular in the DevOps community, use each one a different scripting language to define their tasks and statements for infrastructure provisioning.

In order to mitigate the complexity of working with different scripting languages, and to facilitate a possible tool chaining among them, we developed a Domain Specific Language (DSL) that aims to abstract the specificities of the provisioning tools by modelling the infrastructure requirements that will need to be provisioned in a given cloud platform. We develop our DSL according to the guidelines described in [13].

1) Abstract syntax

The modelling concepts, relationships and their properties are defined through an *Infrastructure Metamodel* (IMM). Figure 2 shows an excerpt of the IMM which defines the valid models of our modelling language in ARGON.

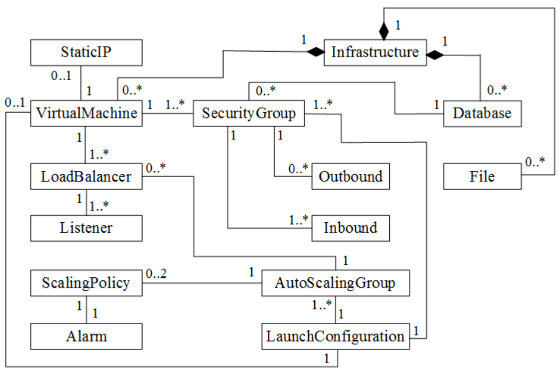


Fig. 2. Excerpt of the Infrastructure Metamodel.

To summarize the metamodeling process below, we describe the main steps.

- **Modelling domain analysis.** Since the specific domain in which we are working is cloud computing, we defined the generic infrastructure elements for the cloud platforms Amazon Web Services (AWS) and Microsoft Azure (MA). These two platforms were selected due to their education licenses and digital resources available. In further work we will also consider other platforms such as Google Compute Engine or RackSpace.
- **Modelling language design.** We focus on the cloud computing capabilities and the component elements of the cloud platforms (e.g., computing, storage, networking, elasticity) instead of the specific scripting languages of the DevOps tools. In this way, we would be able to later generate the scripts for any other tool that supports the cloud platform capabilities. The requirements to generate scripts for the DevOps community tools are also taken into account to be able to perform the model-to-text transformations.
- **Modelling language validation.** The IMM was instantiated for a specific infrastructure model that corresponds to the cloud platform and the tools of interest. In this way, we validated that the concepts abstracted in the metamodel allow to represent the infrastructure characteristics that we will use in a realistic cloud platform. Since we used the Eclipse Modeling Framework [14], the IMM is represented in the metamodeling language *Ecore*. We created several Dynamic Instances in order to verify that metaclasses and their associations are in accordance with the cloud capabilities and requirements for provisioning.

Figure 2 shows the metaclasses and their relationships of the IMM. We can distinguish some groups of metaclasses according to the cloud capacities:

- **Computing capability** allows the creation of *Virtual Machines* with one or more *Security Groups* that perform as a firewall. Each *Security Group* enables a *Virtual Machine* access through ports as *Inbound* rules and *Outbound* rules. *Load Balancer* allows distributing incoming application traffic between multiple *Virtual Machines* and with an input rule or *Listener* that checks the connection requests. In addition, we can assign a *Static IP* address to a *Virtual Machine*.
- **Storage capability** allows the creation of *Databases* and file servers named *Bucket*.
- **Elasticity capability** allows the creation of templates or *Launch Configuration* where characteristics of a *Virtual Machine* are specified. Templates are used to configure the creation of groups of virtual machines by means of *Auto Scaling Group*. Creation or elimination of *Virtual Machines* is done based on *Scaling Policy* which is executed by an *Alarm* that monitor a metric in a period of time.

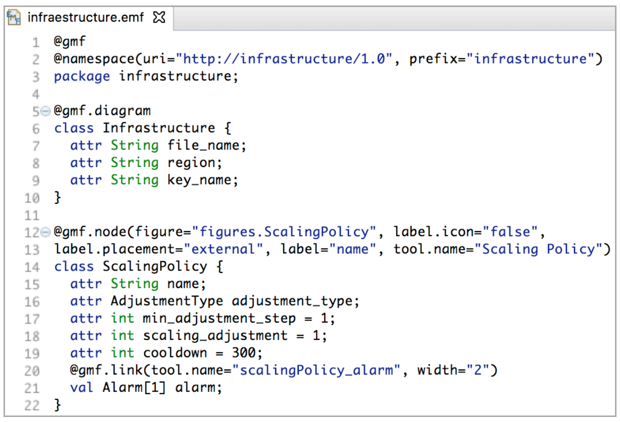
Networking capacity is implicitly represented by associations among metaclasses. In this way, we define a metamodel which represents a set of cloud elements and requirements that will be used to model the final state of the infrastructure provisioning in cloud platforms.

2) *Concrete syntax*

The IMM only defines the abstract syntax, but not a concrete notation of the graphical language in ARGON. In order to use graphical elements to render the model elements in the modelling editors, we use a Graphical Concrete Syntax. Although there are several powerful APIs and frameworks for the development of modelling editors such as Graphical Modelling Framework[7] (GMF) and Graphiti[8], we decided to use EuGENia [15]. EuGENia facilitates to generate the models needed to implement a GMF editor from a single annotated Ecore metamodel. EuGENia uses Emfatic[9] as a language designed to represent Ecore metamodels in a textual manner.

Figure 3 shows an excerpt of an *infrastructure.emf* file which depicts the textual form of the IMM. Taking advantage that the Emfatic language allows representing the IMM in a textual manner, we added annotations on the *infrastructure.emf* file in order to create a fully functional GMF editor. We specify the graphical concrete syntax as follows:

- **Graphical symbols:** `@gmf.node` annotation (line 12) indicates that `class ScalingPolicy` must appear on the diagram as a node.
- **Compositional rules:** `@gmf.link` annotation (line 20) indicate that `class ScalingPolicy` has a link with `class Alarm`, namely define how this graphical symbols are nested and combined.
- **Mapping:** EuGENia allows making a mapping between modelling concepts described in the IMM and their visual representation. For instance, `class ScalingPolicy` (line 14) is mapped with *ScalingPolicy* metaclass of the IMM.



infraestructure.emf

```
 1  @gmf
 2  @namespace(uri="http://infrastructure/1.0", prefix="infrastructure")
 3  package infrastructure;
 4
 5  @gmf.diagram
 6  class Infrastructure {
 7    attr String file_name;
 8    attr String region;
 9    attr String key_name;
10  }
11
12  @gmf.node(figure="figures.ScalingPolicy", label.icon="false",
13  label.placement="external", label="name", tool.name="Scaling Policy")
14  class ScalingPolicy {
15    attr String name;
16    attr AdjustmentType adjustment_type;
17    attr int min_adjustment_step = 1;
18    attr int scaling_adjustment = 1;
19    attr int cooldown = 300;
20    @gmf.link(tool.name="scalingPolicy_alarm", width="2")
21    val Alarm[1] alarm;
22  }
```

Fig. 3. Excerpt of infrastructure.emf file.

Finally, EuGENia allows automatic transformations in order to generate the models needed to accomplish the Graphical Concrete Syntax in GMF. *ARGON* uses this DSL to create an *Infrastructure Model* (IM) representing the infrastructure with its provisioning requirements. The infrastructure model is composed of two files: *i) infrastructure_diagram* that represents the infrastructure through a graphical notation (Figure 4); and *ii) infrastructure* that represents the infrastructure through a hierarchical tree view (Figure 5).

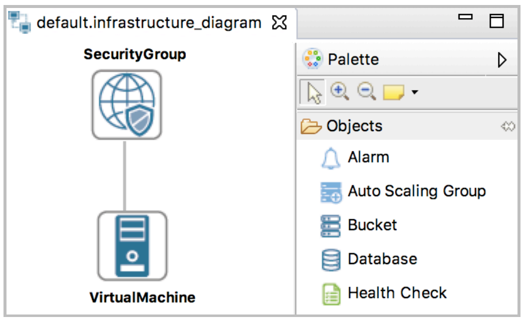
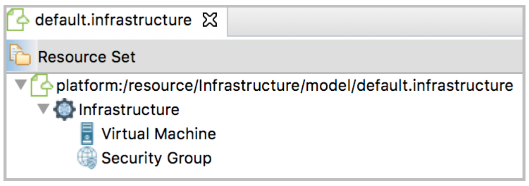


Fig. 4. **ARGON**: Graphical Notation.



default.infrastructure

Resource Set

- platform:/resource/Infrastructure/model/default.infrastructure
  - Infrastructure
    - Virtual Machine
    - Security Group

Fig. 5. **ARGON:** Hierarchical Tree View.

B. *ARGON Architecture*

The definition of the ARGON tool follows the model-driven engineering principles. We decided to use a model-driven approach to develop a generic tool that embrace the majority of cloud platforms and DevOps community tools. In addition, we defined a layered architecture that help us to work at different levels of abstractions allowing to model the infrastructure provisioning (independently of the platform and tools) and to automatically generate the scripts for specific platforms and tools. Figure 6 shows the layered ARGON architecture, where:

- **Requirements** represent the needs for infrastructure provisioning and the solution context that give the knowledge to understand and guide the definition of the Infrastructure as Code.
- **Platform-Independent Model (PIM)** describes the structure and behaviour using the graphical notation defined in ARGON, regardless of the implementation platform. It allows to model the cloud platforms elements and their associations in order to specify an independent and generic provisioning model.
- **Platform-Specific Model (PSM)** contains all required information regarding the structure and behaviour of a specific cloud platform.
- **Transformations** which define the set of model-to-model (M2M) and model-to-text (M2T) transformations. *M2M transformations* are performed to obtain a specific Infrastructure Model for each cloud platform, for instance AWS or Microsoft Azure. We use ATL[10] as the M2M transformation tool. *M2T transformations* are performed to obtain the specific scripts (model instance) required for the DevOps tools. We use Acceleo[11] as the M2T transformation tool.

[7] https://www.eclipse.org/gmf

[8] http://www.eclipse.org/graphiti/

[9] https://www.eclipse.org/emfatic/

[10] https://www.eclipse.org/atl/

[11] https://www.eclipse.org/acceleo/

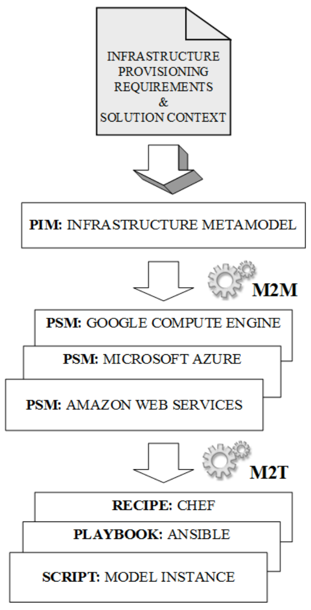
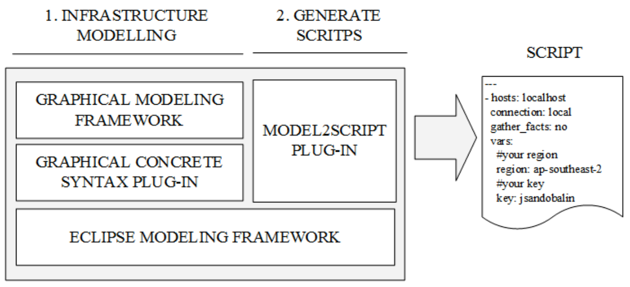

INFRASTRUCTURE PROVISIONING REQUIREMENTS & SOLUTION CONTEXT
PIM: INFRASTRUCTURE METAMODEL
M2M
PSM: GOOGLE COMPUTE ENGINE
PSM: MICROSOFT AZURE
PSM: AMAZON WEB SERVICES
M2T
RECIPE: CHEF
PLAYBOOK: ANSIBLE
SCRIPT: MODEL INSTANCE

Fig. 6. ARGON Architecture.

In order to perform a first proof of concept, we decided to use Ansible[12] as infrastructure provisioning tool since this tool do not require to install any agent unlike Chef or Puppet. On the other hand, we decided to use AWS as cloud platform due to its availability for academic research. AWS provides one-year-free tier unlike Microsoft Azure that offers only one-month-free tier. Thus, the tool Ansible is used to orchestrate the infrastructure provisioning in AWS showing the viability and usefulness of ARGON.

C. *ARGON in Eclipse*

EuGENia allows developing the needed models to use the DSL as GMF editor in Eclipse. Figure 7 show an overview of the plug-ins used in ARGON.


1. INFRASTRUCTURE MODELLING
2. GENERATE SCRITPS
SCRIPT
GRAPHICAL MODELING FRAMEWORK
GRAPHICAL CONCRETE SYNTAX PLUG-IN
MODEL2SCRIPT PLUG-IN
ECLIPSE MODELING FRAMEWORK

```
---
- hosts: localhost
  connection: local
  gather_facts: no
  vars:
    #your region
    region: ap-southeast-2
    #your key
    key: jsandobalin
```

Fig. 7. Overview of plug-ins in ARGON.

The following models were generated before creating the *Graphical Concrete Syntax plug-in* (see Figure 7):

- **gmfgraph** model defines the graphical elements used to visualize model elements.
- **gmftool** model specifies the tool palette which is used to depict the Infrastructure Model elements.
- **gmfmap** model defines the mapping between elements in the Infrastructure Model and the graphical elements defined in the gmfgraph model.

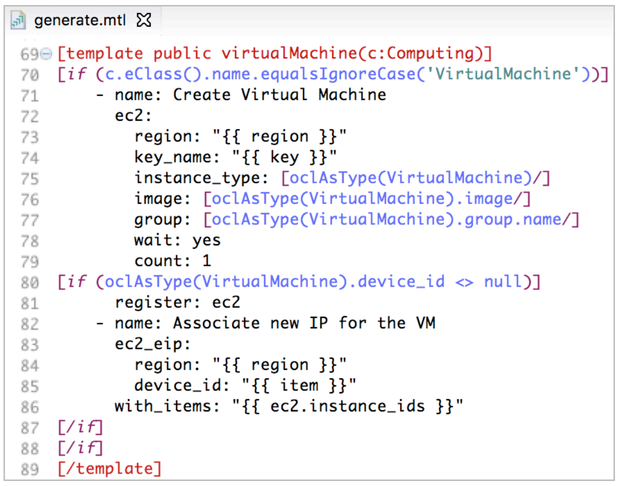
[12] https://www.ansible.com

Figure 4 shows an excerpt of the graphical elements, the tool palette, and the *Virtual Machine* graphical element which represent the *Virtual Machine* metaclass from the Infrastructure Metamodel from Figure 2.

On the other hand, we also define the transformations rules to generate playbooks or Ansible scripts in the *Model2Script plug-in*.

Figure 8 shows an excerpt of the *generate.mtl* file where the transformation rules automatically generate a playbook file or infrastructure provisioning script for Ansible.

Transformation rules corresponding to create a *Virtual Machine* (lines 71-79) have the following statements: ***name*** of the task (line 71), ***ec2*** is Ansible module (line 72) to create a *Virtual Machine*, ***region*** is a parameter (line 73) for the location of infrastructure, ***key_name*** is a parameter (line 74) for the name of the key pair that is necessary to access to AWS, ***instance_type*** is a parameter (line 75) for the hardware features of a *Virtual Machine*, ***image*** is a parameter (line 76) for the operating system image that will be used by the *Virtual Machine*, ***group*** is a parameter (line 77) for the *Securty Group*, ***wait*** is a parameter (line 78) that indicates to Ansible that continues the installation process after *Virtual Machine* instance is started up, and ***count*** is a parameter (line 79) that indicates the number of *Virtual Machines* that will be launched. Moreover, transformation rules corresponfing to create a *Static IP* address (lines 81-86) have the following statements: ***name*** of the task (line 82), ***ec2_eip*** is an Ansible module (line 83) to create a *Static IP* address, ***region*** is a parameter (line 84) for the location of the infrastructure, and ***device_id*** is a parameter (line 85) in which is registered the *Virtual Machine* instance stored in ***ec2*** variable (line 81).



```
69 [template public virtualMachine(c:Computing)]
70 [if (c.eClass().name.equalsIgnoreCase('VirtualMachine'))]
71     - name: Create Virtual Machine
72       ec2:
73         region: "{{ region }}"
74         key_name: "{{ key }}"
75         instance_type: [oclAsType(VirtualMachine)/]
76         image: [oclAsType(VirtualMachine).image/]
77         group: [oclAsType(VirtualMachine).group.name/]
78         wait: yes
79         count: 1
80 [if (oclAsType(VirtualMachine).device_id <> null)]
81       register: ec2
82     - name: Associate new IP for the VM
83       ec2_eip:
84         region: "{{ region }}"
85         device_id: "{{ item }}"
86       with_items: "{{ ec2.instance_ids }}"
87 [/if]
88 [/if]
89 [/template]
```

Fig. 8. Excerpt of generate.mtl file.

Finally, also from Fig. 7, the *Graphical Concrete Syntax plug-in* and *Model2Script plug-in* need the *Graphical Modelig Framework* to show the graphical notation and also the *Eclipse Modeling Framework* to use their core libraries.

IV. Case Study Description

In order to illustrate our approach, we use a case study based on a MOOC (Massively Open Online Courses) cloud application. CEC University (CEC for short) offers massive and free courses which are accessible through the

Internet. In the last few years, the demand of online courses has increased because MOOC has represented a revolution in the field of education, especially in the university education. This leads to courses with hundreds of students accessing video lessons and other multimedia materials, as well as online applications to assess their knowledge, among other academic activities. The courses are hosted on servers located in Virginia, USA.

The main problem is the high demand that exists for certain courses that cause a work overload in the servers. In addition, students have difficulty accessing video lessons and other multimedia materials. CEC has decided to solve this problem by purchasing new servers in order to create a cluster. However, these new servers will be idle when there is no demand for courses. To solve this dilemma, CEC has decided to migrate their infrastructure toward cloud computing. Amazon Web Services (AWS) has been selected as cloud platform.

In order to solve the problems indicated above, the operation staff have decided to design a scalable architecture that works on cloud platforms. Due to the fact that two servers are enough to provide access to courses when there is low demand, they have decided that two servers must be working continuously. The hardware features must be 4 CPU and 16 GB RAM memory for each server. In the case of a work overload on the server that exceeds 80% of CPU usage the cloud platform must create a new server in order to share the workload among servers. In the opposite case, if a server runs on less than 20% of the CPU usage, this server must be removed. However, two servers must always be working and the maximum number of servers to be created is eight. Finally, requests for access to the courses must be distributed evenly among servers to avoid overloading one of them.

Before starting the modelling of the infrastructure with ARGON, the requirements must be specified:

- **Req. 1:** Hardware features of each server must be of 4 CPU and 16 GB RAM memory.
- **Req. 2:** Two servers must always be working and the maximum number of servers to be created is eight.
- **Req. 3:** If a server exceeds 80% of CPU usage the cloud platform must create a new server.
- **Req. 4:** If a server runs on less than 20% of the CPU usage, this server must be removed.
- **Req. 5:** Requests for access to the courses must be distributed evenly among servers.
- **Req. 6:** The place where CEC is located is Virginia, USA.

Table 1 shows the mapping between the requirements of the architecture and the ARGON elements.

Firstly, requirement 6 must be set up. The region in which the infrastructure will be deployed is Virginia. In the AWS, the region code of Virginia, USA is `us-east-1`.

Requirements 1 and 2 use a *Launch Configuration* element which is a template to define the hardware of a *Virtual Machine*. In this case, we use `m4.xlarge` as hardware instance because it has 4 CPU and 16 GB RAM memory. In addition, a *Security Group* element is necessary to provide an access control to the *Virtual Machines*. An *Inbound* rule is mandatory to provide access through port 80 (Http).

TABLE I. MAPPING BETWEEN THE REQUIREMENTS AND THE ARGON ELEMENTS.

Req.	**Element**	**Symbol**
1, 2	Launch Configuration	⚙
	Security Group	🌐
	Inbound	🔓
3, 4	Auto Scaling Group	⊕
	Scaling Policy	⚙
	Alarm	🔔
5	Load Balancer	●
	Listener	🔓
6	Diagram Attribute	

Requirements 3 and 4 use an *Auto Scaling Group* element to set up the minimum number of *Virtual Machines* that must always be working and the maximum number of *Virtual Machines* to be created. A *Scaling Policy* specifies whether the *Auto Scaling Group* scale in or out, namely each *Scaling Policy* define whether a *Virtual Machine* is created or removed. An *Alarm* watches the usage CPU metric over a time period specified and performs the creation or remove of *Virtual Machines*. Finally, requirement 5 uses a *Load Balancer* which automatically distributes incoming application traffic across multiple *Virtual Machines*. Moreover, a *Listener* is linked to the *Load Balancer* in order to check for connection requests. *Listener* is configured with Http protocol and port 80 for connections.

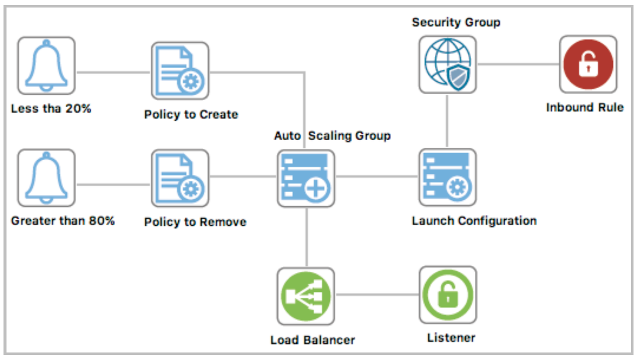
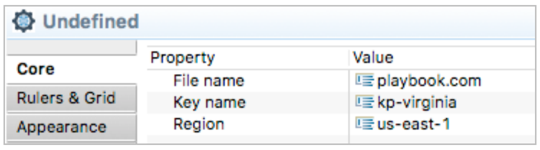


Fig. 9. ARGON: Infrastructure Model.

Figure 9 shows the infrastructure architecture modeled in ARGON. Every element explained in Table 1 is used to model the scalable architecture which will be deployed in AWS. Furthermore, every element has its own attributes which can be modified to indicate the specific characteristics. Figure 10 shows the Diagram attributes which have a ***File name*** attribute to indicate the playbook name (script for Ansible), a ***Key name*** attribute to indicate the name of the key pair file which Ansible needs to access the AWS, and a ***Region*** attribute to indicate the region code that in this case corresponds to Virginia, USA.

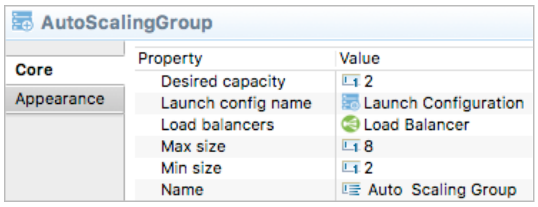
Figure 11 shows the *Auto Scaling Group* attributes that have a ***Desired capacity*** attribute to indicate the number of *Virtual Machines* which must be always working, a ***Launch config name*** attribute to select the *Launch Configuration* element, a ***Load balancers*** attribute to select one or more *Load Balancers* elements, a ***Max size*** attribute to indicate the maximum number of Virtual Machines that must be created, a ***Min size*** attribute to indicate the minimum number of *Virtual Machines* that must be working, and a ***Name*** attribute that is the name of the Auto Scaling Group element.


Undefined

Core | Rulers & Grid | Appearance

Property	Value
File name	playbook.com
Key name	kp-virginia
Region	us-east-1

Fig. 10. ARGON: Diagram Attributes.


AutoScalingGroup

Core | Appearance

Property	Value
Desired capacity	2
Launch config name	Launch Configuration
Load balancers	Load Balancer
Max size	8
Min size	2
Name	Auto Scaling Group

Fig. 11. ARGON: AutoScalingGroup Attributes.

Finally, we use the *Model2Script plug-in* to automatically generate a playbook or script from the infrastructure model. Figure 12 shows an excerpt of the playbook which will be used in the Ansible tool to make the infrastructure provisioning in AWS. This playbook shows the creation statements of a *Launch Configuration* element that have a ***name*** of the task (line 35), ***ec2_lc*** is the Ansible module (line 36) to create a *Launch Configuration* element, ***region*** is the location (line 37) where the infrastructure will be deployment, ***name*** is the element name (line 38), ***image_id*** is the operating system image (line 39), ***key_name*** is the key pair to access in AWS (line 40), ***instance_type*** is the hardware features (line 41) of the *Virtual Machine*, and ***security_groups*** is the access control (line 42) of the *Virtual Machine*.

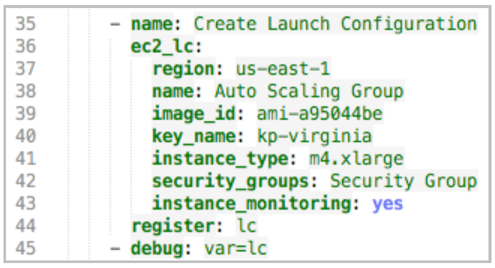

```
35      - name: Create Launch Configuration
36        ec2_lc:
37          region: us-east-1
38          name: Auto Scaling Group
39          image_id: ami-a95044be
40          key_name: kp-virginia
41          instance_type: m4.xlarge
42          security_groups: Security Group
43          instance_monitoring: yes
44        register: lc
45      - debug: var=lc
```

Fig. 12. Excerpt of an Ansible playbook.

Although, we are aware that the size and complexity of the MOCC example shown in this section is not representative of a large cloud applications, we believe that it may be enough to illustrate the main characteristics and the potential use of the ARGON tool to model and automatically generate the infrastructure provisioning for cloud platforms.

A. Discussion

In this work, we introduced two main contributions: a Domain Specific Language (DSL), that allows us to abstract the complexity of tools used in DevOps settings (e.g., Ansible, Chef, or Vagrant), and the supporting tool (ARGON), which uses this language to allow us: i) modelling the final state of the infrastructure provisioning, and ii) automatically generating the scripts for specific DevOps tools needed to provisioning the infrastructure where the cloud applications are to be deployed.

Regarding the first contribution, our DSL is not limited to a single cloud platform since we model the cloud platforms elements and their characteristics instead of the specific scripting features of DevOps tools. Although, at this time, we focus only on Amazon Web Services and Microsoft Azure to define the Infrastructure Metamodel, we propose a generic infrastructure metamodel that could be extended to also represent other cloud platforms such as Google Compute Engine or RackSpace.

We believe that managing the technological aspects of the different cloud platforms at a higher level of abstraction helps software developers to prevent the problem of *cloud vendor lock-in* and to be more prepared to effectively deal with changes.

Regarding the automatic generation of scripts for DevOps tools, our proposal is extensible to cover future tools and also prepared to manage change updating the required scripts. This is possible since we specify the scripting language features of the tools in the transformation rules meaning that we are potentially able to generate the scripts for any tool as far as its concepts are represented in the Infrastructure Model. Additionally, new transformation rules may be added in case a new feature is provided by a tool or if a new version of the tool is available. In this way, we achieve to manage the constellation of tools which are available for different cloud platforms in a very homogeneous way.

In our experience, it is a far better option to create transformation rules for the scripts of the tools rather than writing the specific scripts for each one of them. Since there is no standard scripting language, it is better to raise the level of abstraction to manage the scripting language of all these tools. In this way, we support the concept of *Infrastructure as Code*, by generating from the provisioning model the specific provisioning scripts for managing the infrastructure.

ARGON is a tool developed for the Eclipse platform as a proposal to an open and extensible framework for modelling the infrastructure provisioning for cloud applications. The tool provides facilities to developers or operation staff to focus their efforts on modelling the tasks of the infrastructure requirements instead of learning the different scripting languages for defining the infrastructure. Nevertheless, it is necessary to know and understand the concepts of cloud computing, for instance, the elasticity in order to interpret the requirements and to specify the desired behaviour of the virtual machines that can scale in and out (horizontal scaling).

Throughout the development of the ARGON tool, we have learned several lessons, especially technical lessons since we propose a generic and extensible tool that is able to deal with different cloud platforms and provisioning tools. The most remarkable experiences are:

- To run a playbook on Amazon Web Services is advisable to run the playbook on a Virtual Machine created in Amazon Web Services. In the case of running the playbook out of the cloud platform some abnormal behaviours during the infrastructure provisioning might occur.
- It is possible to model idempotent scripts. We take advantage of the idempotency of Ansible to run a playbook (script) many times and only the changes made in the infrastructure model are done in the cloud platform, the rest of elements do not suffer any change.
- In the continuous integration stage, we are only working with the *infrastructure* file to generate the scripts to infrastructure provisioning. This is due to the version control systems to merge well the infrastructure files (ARGON's diagram elements) and to work fine with continuous integration servers. However, the *infrastructure_diagram* file (ARGON's diagram layout) specifies the color, position, font, etc. of the elements and control version systems may recognize a position change of an element as a change in the model.

As far as we know, the research and industry efforts are mainly focusing on the reuse of DevOps tools to improve the infrastructure provisioning or deployment of applications in cloud platforms. ARGON instead focus on supporting the tasks of infrastructure modelling in line with other research projects such as TOSCA [6] or MODAClouds [10].

V. Conclusion and Future Work

In this paper, we have presented ARGON, a modelling infrastructure tool for cloud provisioning towards to automate the management of infrastructure provisioning. The purpose is to facilitate the adoption of initiatives such as DevOps where one of its pillars is the automation through the concept of the Infrastructure as Code (IaC). IaC allows to manage all the infrastructure as a code, that is, to save it in a code control system, to test it, etc. The feasibility of the ARGON tool and it usefulness to model the final state of provisioning of infrastructure for a Massively Open Online Courses (MOOC) was also introduced.

As a future work, we want to extend the concept of Infrastructure as Code to manage scripts for version control systems, libraries for automated tests, and tools for integration and deployment. This will require to define the concepts of the Infrastructure Metamodel as well as the definition of the corresponding transformation rules.

Finally, we also plan to run experiments with students and practitioners with experience in cloud computing development and in particular with knowledge of provisioning resources in the cloud. This will help us to validate the effectiveness of the proposed solution with DevOps tools such as Puppet or Chef in terms of provisioning infrastructure on different platforms in the cloud.

Acknowledgment

This research is supported by the Value@Cloud project (TIN2013-46300-R).

References

[1] R. Buyya, J. Broberg, and A. Gościński, *Cloud computing : principles and paradigms*, Wiley, 2011.

[2] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional, 2010.

[3] K. Morris, *Infrastructure As Code: Managing Servers in the Cloud*, O'Reilly Media, 2016.

[4] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, and M. Zimmermann, "Unified Invocation of Scripts and Services for Provisioning, Deployment , and Management of Cloud Applications Based on TOSCA," in *International Conf. on Cloud Computing and Service Science*, CLOSER, 2014, pp. 559–568.

[5] J. Wettinger, U. Breitenbücher, and F. Leymann, "DevOpSlang - Bridging the gap between development and operations," in *The European Conf. on Service-Oriented and Cloud Computing*, ESOCC, 2014, vol. 8745, pp. 108–122.

[6] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel," in *Future Generation Computer Systems*, 2015, vol. 56, pp. 317–332.

[7] J. Wettinger, U. Breitenbucher, and F. Leymann, "Standards-based DevOps automation and integration using TOSCA," in *International Conference on Utility and Cloud Computing*, UCC, 2014, pp. 59–68.

[8] A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, and G. Kappel, "UML-based cloud application modeling with libraries, profiles, and templates," in *Model-Driven Engineering on and for the Cloud*, 2014, vol. 1242, pp. 56–65.

[9] A. Rossini, "Cloud application modelling and execution language (CAMEL) and the PaaSage workflow," in *European Conference on Service-Oriented and Cloud Computing*, ESOCC, 2016, vol. 567, pp. 437–439.

[10] J. Scheuner, P. Leitner, J. Cito, and H. Gall, "Cloud work bench - Infrastructure-as-code based cloud benchmarking," in *Cloud Computing Technology and Science*, CloudCom, 2014, pp. 246–253.

[11] W. Chen *et al.*, "MORE: A model-driven operation service for cloud-based IT systems," in *Proc. IEEE International Conference on Services Computing,* SCC, 2016, pp. 633–640.

[12] E. Di Nitto, P. Matthews, D. Petcu, and A. Solberg, *Model-Driven Development and Operation of Multi-Cloud Applications*, Springer International Publishing, 2017.

[13] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, Morgan & Claypool, 2012.

[14] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. 2008.

[15] D. S. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige, "Eugenia: towards disciplined and automated development of GMF-based graphical model editors," in *Software and System Modeling*, 2015, vol. 16, pp. 229–255.