Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 2473

Declarative Cloud Resource Provisioning Using OCCI Models

Johannes Erbel[1]

Abstract: Great competition in providing cloud services has built a huge diversity in ways to manage the rented computing resources. This diversity has led to the provider lock-in problem, i.e., binding the customer to one provider. Moreover, the increasing need for quick and easy infrastructure provisioning requires for an automated and parallel provisioning process, as it is a cumbersome process when done manually. To tackle these needs, we propose a standard conform cloud resource provisioning process which is able to deploy declarative models based on the Open Cloud Computing Interface (OCCI) standard. Therefore, a provisioning plan is generated, defining the resource provisioning order, followed by a systematic execution of it. In order to show the feasibility of this approach, we implemented a prototype and discuss it with the help of an example topology.

Keywords: Cloud Computing, Open Cloud Computing Interface, Model Driven Engineering.

1 Introduction

Cloud computing enables the user to dynamically rent computing resources on demand. Hereby, the way of managing such resources is provider dependent, leaving consumers vulnerable to a *provider lock-in* [Ar10], i.e., making it hard to switch the provider once a cloud application is build. Meanwhile, many different standards have been developed tackling this problem by focusing on different quality attributes. One of these standards is the *Open Cloud Computing Interface (OCCI)* [OG16a], developed by the *Open Grid Forum (OGF)*, which exposes an interface to easily manage cloud resources over *Representational State Transfer (REST)* calls. OCCI defines a metamodel capable of describing the infrastructure of cloud applications. By deriving the required REST calls directly from the model elements, the provisioning process can be automated saving a large amount of time, especially when it comes to large infrastructures. Nevertheless, these models only depict declarative states of cloud applications and do not consider the order in which the resources have to be provisioned and linked. To address this issue, we propose a model driven approach to implement a fully automatic and standard conform cloud resource provisioning process, able to extract the required provisioning order from the OCCI model. Therefore, we perform multiple model transformations to generate a model to describe the order in which the resources have to be provisioned. The proposed approach serves as the provisioning process

[1] University of Goettingen, Institute of Computer Science, Goldschmidtstraße 7, 37077 Goettingen, j.erbel@stud.uni-goettingen.de

doi:10.18420/in2017_251

used in scope of a model driven cloud orchestrator which we conceptually introduced in [GEG17]. Here we utilize the *Topology and Orchestration Specification for Cloud Applications (TOSCA)* [OA13], another cloud standard, and its benefits regarding the portability of topology models. The OCCI topology models used in the proposed provisioning process originate from these TOSCA models, which we transformed beforehand. In future work, this approach will be enhanced with the capability to dynamically adapt a cloud system according to changing environments and requirements. Therefore, we establish a feed back loop between the system and the OCCI topology model. In this paper we investigate how the dependencies of these topology models can be resolved in order to create a provisioning plan, and how this plan can be executed to allow for an automatic provisioning of cloud infrastructures.

In the following, the outline of this paper is given. Section 2 introduces the foundations required to understand the provisioning process. Hereby, the basics of modeling, OCCI and workflow languages are covered. Section 3 describes the cloud resource provisioning process, examining the generation of provisioning plans and their execution. Section 4 describes the provisioning process of an example infrastructure in order to test the feasibility of this approach. Section 5 covers related work delimiting our approach to similar ones. Finally, Section 6 sums up the outcome of the proposed provisioning process and gives a short outlook into future work.

2 Foundations

In this section, fundamental knowledge, required to understand the provisioning process, is introduced. Section 2.1 defines the foundations and benefits of Model Driven Engineering (MDE). Section 2.2 examines the basics of the OCCI standard covering a precise description of its metamodel. Section 2.3 provides a brief overview of workflow languages and their typical elements.

2.1 Model Driven Engineering

Compared to the widely used object oriented development paradigm, where “everything is an object” [Bé04], the *Model Driven Engineering (MDE)* paradigm assumes that “everything is a *model*” [Bé04]. These models are defined by Kühne as “an abstraction of a (real or language-based) system allowing predictions or inferences to be made” [Kü06]. Overall, models represent not only a way to provide an abstract view on a complex system, but also store valuable system information which can be processed. *Metamodels* define the semantics and structure of a model by specifying elements a model can instantiate [OM11b]. Each model is “an instance of a metamodel” [OM11b], which can be seen as “a model of models” [OM03]. For example, a metamodel able to instantiate multiple kinds of cars has the elements seat, motor and wheels. Hereby, the metamodel does not only describe the

attributes of the element, for example a seat's color, but also the relationships of the elements, for example a car has two to six seats. Summed up, metamodels define a language for models. Based upon metamodels, model transformations are defined to "develop, maintain and evolve software" [MVG06], building the fundamentals of MDE [Kü06]. A complete definition of model transformation is given by Kleppe et al.: "A *transformation* is the automatic generation of a target model from a source model, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language" [KWB03]. In the following, we discuss the OCCI standard and its metamodel, which defines the language of the models used as input for the provisioning process.

2.2 Open Cloud Computing Interface

OCCI is a standard developed to uniformly access and manage cloud resources [OG16a]. Even though OCCI was originally created for an *Infrastructure as a Service (IaaS)* purpose, it has evolved to an *Application Programming Interface (API)* able to handle *Platform as a Service (PaaS)* and *Software as a Service (SaaS)* as well. The OCCI standard focuses on interoperability and extendability. To perform management activities, OCCI provides a RESTful protocol and API. Hereby, the information required to perform management REST calls can be derived from the elements in the OCCI model. These models instantiate the OCCI core metamodel [OG16a], which is depicted in Figure 1. The OCCI standard only provides the specification of the metamodel which got implemented by Merle et al. [Me15]. This metamodel can be separated into two different parts describing the *core base types* and the *classification and identification mechanisms*.

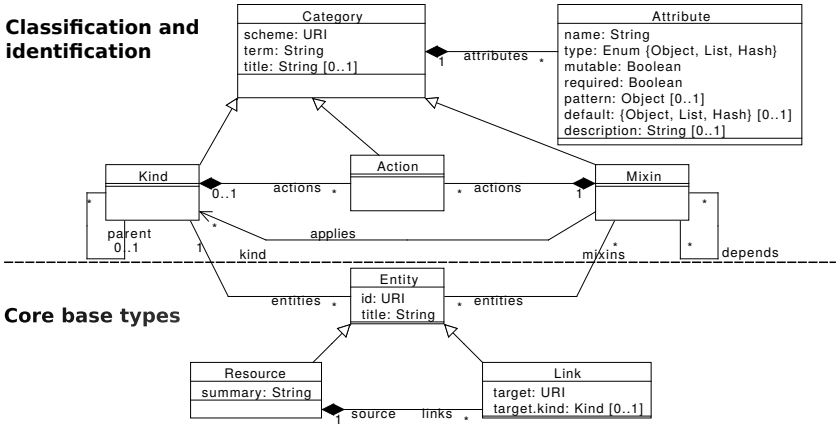

Fig. 1: Overview of the OCCI core metamodel adapted from [OG16a].

Elements considering the core base types are Entity, Resource and Link. The element Resource represent resources from the real world, for example a *virtual machine (VM)*, storage or network. The connection between those is modeled via the Link element storing information about the source and the target Resource. Both of these elements inherit from the abstract Entity type defining the id of the elements, which serves as *Uniform Resource Identifier (URI)*. Furthermore, the Entity element binds each Entity sub type to a unique Kind and stores information about related Mixins.

The classification and identification mechanisms are formed by the elements Category, Attribute, Kind, Action and Mixin. Category serves as a non abstract type identifying Kinds, Actions and Mixins, whereby the *term* and *scheme* attribute uniquely identifies them. Moreover, every instantiation of Category is able to define Attribute elements representing client readable attributes. Kinds represent the classifying element for Entity sub-types, indicating available Actions and Attributes to be filled by the Entity. Kinds are able to inherit other Kinds, whereby each Kind has to refer to one of the three core Kinds, equally named to the core base types Entity, Resource and Link. Mixins provide additional Attributes that can be added to instances at creation or run-time to completely classify Entity sub-types, complementing the Kind element. Summed up, Mixins represent an extension mechanism to model Entity sub-types in more detail. Finally, the Action element identifies operations that can be invoked on Entity instances of the corresponding Kind or Mixin.

One of the OCCI core metamodel extensions we use, is the OCCI infrastructure extension [OG16b], which extends the OCCI core metamodel for managing IaaS resources. Overall, this extension is composed of three Kinds describing Resources and two Kinds describing Links. The Kinds for Resources specify a Resource as Compute, Network or Storage, whereas NetworkInterface and StorageLink, the Link types, allow for additional attributes to specify connections between VMs, storages and networks. It should be noted that OCCI defines further extensions, for example to address the PaaS layer. Nevertheless, these are not further discussed due to missing implementations. In the following section, we introduce the concepts of workflow models, which we use to depict provisioning plans.

2.3 Workflow Modelling

In order to create an automated provisioning process based on a declarative model, a workflow model is required to specify the needed steps to be performed. Hereby, the workflow model does not only define the order in which the different operations have to be performed, but also which operations can be done in parallel [Br14]. Thus, we use workflow models to instantiate provisioning plans, whereby the low-level operations represent provisioning tasks and the control flow the order in which these operations are performed. Two of the more commonly known modeling languages used to create workflows are the *Business Process Model and Notation (BPMN)* [OM11a] and the *Unified Modeling Language (UML)* [OM11b] with its *activity diagrams*. To ease the understanding of workflows the activity diagram depicted in Figure 5 can be used as reference. Workflows

provide the capability to start complex tasks by modeling an initializing element, which starts a control flow. This flow can be directed to *control nodes*, manipulating the control flow, or *actions* performing a task of low complexity. Control nodes are able to *fork* incoming control flows into multiple ones, allowing for a parallel execution. Later in the process, every opened control flow is merged by a *join* control node. Moreover, control nodes are able to check if specific conditions are met to send the control flow onto one out of multiple possible paths. Actions are able to define input values, which store additional information required by the action. In the following section we discuss the cloud resource provisioning process making use of such workflow models as provisioning plan.

3 Resource Provisioning Process

To provide an automatic provision process, the dependencies between the elements, hidden in the OCCI model, must be resolved. To identify those, we separated the provisioning process into a plan generation and a resource provisioning step, as depicted in Figure 2. At first, in the plan generation step, multiple *model to model transformations (M2M)* are performed on the OCCI model to identify dependencies between the elements in the infrastructure. To perform these transformations, we adapted an approach by Breitenbücher et al. [Br14]. The first transformation transforms the OCCI model into a *Provisioning Order Graph (POG)*, describing the dependencies of the topology elements. Thereafter, this POG is transformed into a workflow model specifying a provisioning plan. In the second step, the resource provisioning, this workflow is traversed element by element by the Provisioner entity. The Provisioner is connected to an Extractor entity, able to gather topology model information, and an Executor entity, which performs the service calls to the OCCI API to actually provision resources. Section 3.1 contains an explanation of the performed transformations, while Section 3.2 investigates the processing of the provisioning plan.

3.1 Transformations

To generate a provisioning plan, we adapt an approach by Breitenbücher et al. [Br14], who combined declarative and imperative models to generate executable provisioning plans. In our approach, an OCCI model serves as the input topology model starting the transformation process. The POG is an instance of a simple directed graph metamodel comprising *vertices*, able to store an id of the corresponding OCCI element, and *directed edges*, linking vertices together. The used transformation rules are depicted in Table 1. For each Resource and Link in the OCCI model a vertex, storing their id, is created in the POG. Moreover, for every Link two edges are created in the POG. Based on the Kind of the Link the edges are connected to the vertices, representing the relationship partners of the Link, to describe a specific dependency. Overall, Breitenbücher et al. define two dependency patterns, which are mapped onto the different Kinds. The depends-on pattern and the uses pattern, which are depicted in Figure 3. A depends-on pattern describes a dependency, which requires the

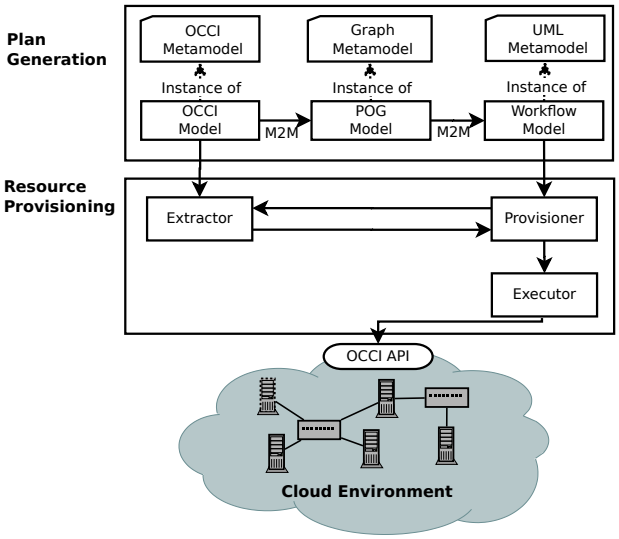

Fig. 2: Overview of the cloud resource provisioning process.

Source Resource of the Link to be instantiated before the Target Resource can be created, for example an execution environment which depends on a running VM. In this case the dependency resolves in a Target Resource → Link → Source Resource pattern. A use pattern characterizes a dependency, which requires the Source Resource as well as the Target Resource to be instantiated before they can be connected. For example when a VM is connected to a network both need to be provisioned beforehand. This dependency resolves in a Target Resource → Link and Source Resource → Link pattern.

OCCI	POG
Resource	Vertex
Link	Vertex
Link Kind	Use/Depends-on pattern

Tab. 1: Mapping of OCCI to POG elements.

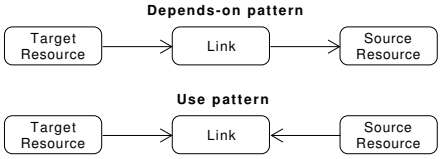

Fig. 3: Dependency pattern.

Based on this POG a second transformation is used to generate a workflow model in form of an UML activity diagram. This results in a provisioning plan which can be processed to provision and link the cloud resources in the correct order. At first, an initial node is created serving as starting point for the processing later on. Then, for each vertex in the POG an empty action is created only storing the id of the vertex, e.g. the id of the OCCI Entity sub-type. Every vertex without an incoming edge indicates that the Resource has no dependencies, allowing for an immediate and parallel provisioning. This immediate

parallelization is represented by a fork node, which splits the control flow into multiple ones. During the transformation process every vertex that is able to be provisioned is marked, so we can check whether the dependencies of other vertices are met. If an action has only one predecessor it is directly linked to it. Otherwise, a join is created combining the control flows of the required vertices before continuing the control flow. Finally, in a post processing step, every action without an outgoing edge is connected to a final join merging the control flow into the end node closing the workflow. In the following, we introduce the processing of provisioning plans, discussing how this workflow is traversed and how actual service calls are performed.

3.2 Provisioning Process

To automatically provision cloud resources, the generated provisioning plan is processed. At the beginning of the provisioning process, an authentication token is requested from the cloud to start a session, which is required to perform provisioning tasks. This token does not only indicate the user's credentials, but also the project in which the resources have to be provisioned. After the token is received, the Provisioner traverses the workflow starting at the initial node. From there on the control flow is followed to the next node, indicating a specific operation to be performed. The different nodes and operations are shown in Table 2.

Workflow Element	**Provisioning Action**
Initial	Starting point
Action	Extract OCCI Element Execute REST call Wait for active state
Fork	Start threads for outgoing control flows
Join	Wait for incoming control flows to join

Tab. 2: Processing of workflow elements.

Overall, the Provisioner considers actions and two different kinds of control flow nodes, which are forks allowing for a parallel execution of the workflow and join nodes merging these later on. When the Provisioner reaches a fork node, a thread is created for every outgoing edge of the fork. Each thread starts another instance of a Provisioner responsible for processing one control flow of the fork, starting with one of the following nodes as initial start point. Hereby, the threads handle every upcoming node on its control flow path until a join is reached. When a control flow reaches a join, it is checked if all incoming control flows already reached it. The last incoming thread continues to process the control flow after the join. It should be noted, that not only actions, but also fork nodes can be directly connected to a join. When the Provisioner reaches an action, operations to provision cloud resources are performed. Therefore, a REST request is send to the OCCI API of the cloud,

which contains the information about the OCCI element to be provisioned. To gain the information about this element, the Provisioner passes the id stored within the action to the Extractor. The Extractor searches the OCCI topology model for the element with the corresponding id and returns it to the Provisioner. This element in turn is handed over to the Executor, which establishes a connection to the OCCI API and performs the REST request. In case of a cloud resource provisioning task, a POST request is performed. Depending on the Kind of the OCCI element, the connection is established to a specific URI. For example, if a Compute instance has to be provisioned, the POST request is send to the cloud's address followed by /occi1.1/compute/. Depending on the rendering used by the OCCI API, the information of the OCCI element are either added as header or as input to the REST request. When the text rendering [OG16c] is used, the information is passed over request headers. These are separated into a *Category header* and an *Attribute header*. The Category header contains the OCCI Category elements connected to the element to be provisioned, e.g. Kind and Mixins of the element. The Attribute header stores the element's attributes and their values. After a successful provisioning request, the actual id of the provisioned element is received. Before another action on the same control flow is allowed to be performed, the Provisioner waits until the provisioned resource is in an active or online state. This ensures that the resources are ready to be further adjusted and linked. In the following section the feasibility of this approach is discussed.

4 Feasability Study

To discuss the feasibility of the proposed cloud provisioning process, we implemented a prototype, which is discussed in this section with the help of an example OCCI topology model. Due to a missing implementation of the OCCI PaaS extension, we provide an example only considering the IaaS layer. In Section 4.1, the utilized tools and systems are discussed. Section 4.2 describes the example topology model, the generated plans and the corresponding REST calls. Finally Section 4.3, gives a short discussion about the feasibility of the proposed approach.

4.1 Tooling

As for the plan generation, we use the *Eclipse Modeling Framework (EMF)* [St08] and the *Epsilon Transformation Language (ETL)* [Ep14]. To instantiate the OCCI model, we use the EMF based OCCI metamodel proposed by Merle et al. [Me15]. For the POG, we created a simple directed graph metamodel. Finally, the UML [OM11b] metamodel is used to instantiate activity diagrams as workflow model. The provisioning process itself is implemented in Java. The infrastructure is provisioned on an *OpenStack* [Op16] cloud of our research group, which uses the *OpenStack OCCI Interface (OOI)* [In15], an implementation of OCCI for OpenStack.

4.2 Example Topology

In the following we investigate the process of provisioning a simple cloud infrastructure described by an OCCI model. The infrastructure is composed of two VMs, which are both connected to a network. Furthermore, one of these VMs is attached to a storage device. This infrastructure is translated into an OCCI model, which is depicted in Figure 4 by an UML object diagram.

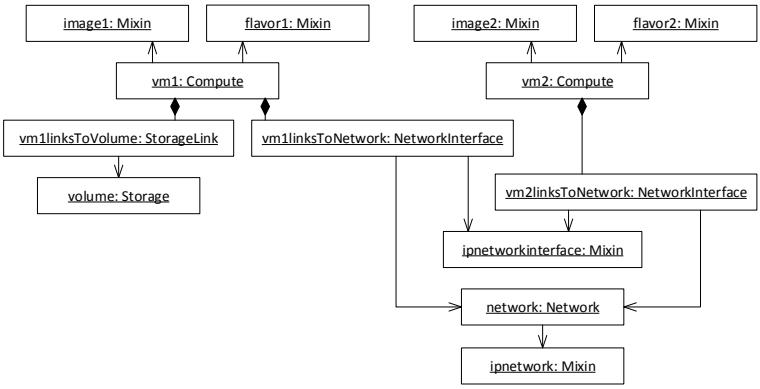

Fig. 4: Example OCCI cloud infrastructure model [GEG17].

This diagram shows that the two VMs, vm1 and vm2, are represented by two Compute type Resources, the network network by a Network type Resource and the storage volume by a Storage type Resource. The Mixins bound to the Compute types define the *flavor* and the *image* of the VM. The flavor encodes the cores, memory and storage capacity of the VM, while the image defines its operating system. Both VMs are connected to the network over NetworkInterface type Links, further defined by a Mixin called Ipnetworkinterface, which allows for a specific ip address to be defined [OG16b]. The network itself is connected to a Mixin called Ipnetwork to define an address range and gateway for the network [OG16b]. Finally, vm1 is connected to volume over a StorageLink, defining the device id and mountpoint of the Storage [OG16b]. This model serves as input topology for the provisioning process, which is discussed in the following.

Figure 5 shows the results of the plan generation phase, with the POG on the left and the resulting activity diagram on the right. The POG shows that vm1, vm2, volume and network need to be provisioned before corresponding connections can take place. These kinds of relationships result from the use pattern, which is associated with all the IaaS Kinds of OCCI. Therefore, the depends-on pattern is not observed in this example, because it can only be mapped to Kinds of the OCCI PaaS extension. Based on the POG, the provisioning plan, depicted on the right side of Figure 5, is generated. The initial point starts the control flow, which is immediately followed by a fork parallelizing the provisioning process of the vertices without incoming edges in the POG, e.g. vm1, vm2, network and volume. Because

vm1 and network are required for multiple actions, a fork splits up their control flow to mark the action as finished for both of the following joins. In the POG, vm2 and volume only posses one edge, resulting in a direct connection to the following action. Because the following actions have multiple incoming edges, a join is created to merge the incoming control flows before the next action is performed. Finally, a join merges every activity, which has no outgoing edge at the end of the transformation. This join is connected to the final node of the provisioning plan closing the workflow.

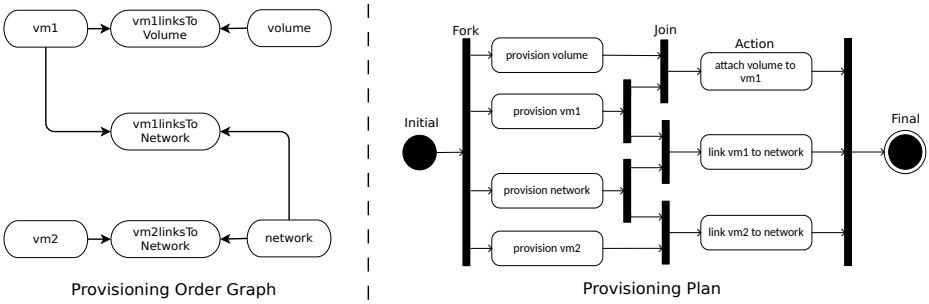

Fig. 5: Example Provisioning Order Graph and Workflow Model.

To provision cloud resources, REST requests are send to the OOI interface of the OpenStack cloud. The required information for this requests is gathered from the elements in the OCCI model. To provide an in depth view of how an OCCI element can be provisioned, Figure 6 shows the vm1 element in all of its details and the REST call build upon it. To create a VM, a POST request to the address of the provider, using the port of the OOI interface, is send. The session used for this request is stated by an authentication token in the *X-Auth-Token header*. Furthermore, the content type of the request must be set to text/occi. The information about the element are added to the Category- and Attribute header as defined in the OCCI text rendering documentation [OG16c]. In this case, vm1 is a Compute type with two Mixins resource and os. These Mixins, uniquely identified by their term, describe the image and flavor of the VM. Therefore, the element vm1 is created as an Ubuntu 16.04 server of size m1.medium. The attributes of the element are inserted into the *X-OCCI-Attributes header* giving additional information about the VM, for example the title, id or state. Summed up, in order to provision a cloud resource over an OCCI REST request, the information about the model element is simply added to the correct header, highlighting the simplicity of OCCI when it comes to resource provisioning.

4.3 Discussion

Our feasibility study indicates that we are able to automatically provision OCCI topology models with our approach. Even though the proposed example represents only a small example infrastructure, we want to emphasize that this approach is also feasible when it

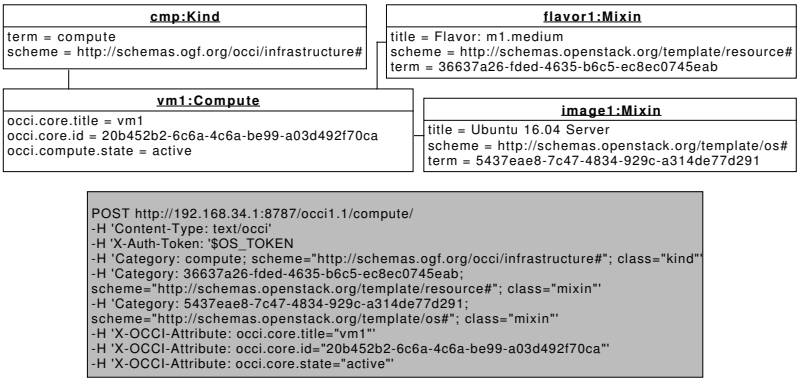

Fig. 6: Compute resource vm1 with the corresponding REST request.

comes to larger infrastructures, because every provisioning plan generated from an OCCI model, only respecting the current IaaS extension, results in the same pattern. This pattern consists of two parts. The resource provisioning part, where each Compute, Storage and Network instance is provisioned in parallel and the part in which these Resources are linked. Other patterns can currently not occur, because every Resource to be provisioned is completely independent from one another. This is because all Kinds for Links of the OCCI IaaS extension have to be mapped to the use pattern of Breitenbücher et al. [Br14]. Therefore, no Resource can have any incoming edges in the POG, which results in the parallel resource provisioning part of the provisioning plan. The linking part of the provisioning plan arises, because every Link depends on the two Resources it connects, which are already build up to this point. A detailed evaluation of this approach is necessary when the PaaS extension of OCCI is taken into account or the IaaS extension is modified, due to the occurrence of the depends-on pattern.

5 Related Work

In addition to OCCI, other standards defining metamodels for cloud topologies exist. One of them is TOSCA [OA13], a standard by the *Organization for the Advancement of Structured Information Standards (OASIS)*, which is able to provider-independently describe cloud applications. In the approach by Breitenbücher et al. [Br14] TOSCA topology models are used to generate provisioning plans. Compared to OCCI, TOSCA models are meant to be stored and reused later on, to increase the portability and reusability of cloud applications in order to prevent the provider lock-in. Nevertheless, TOSCA does not define how actual resources can be provisioned over API calls, making it unsuitable for actual resource provisioning. In a conference paper [GEG17] we investigated, how TOSCA models can be transformed into OCCI models to address this issue and to utilize the benefits of both

standards. Another approach to perform model driven resource provisioning is proposed by Ferry et al. [Fe14], who introduce a models at runtime approach, able to not only provision resources, but also adapt existing cloud applications. Therefore, they define a domain-specific modeling language called *CloudML*. In contrast to our approach, their approach is not standard driven and hence does not address the provider lock-in problem.

6 Conclusion

In this paper we proposed a model driven approach to automatically provision cloud infrastructures based on OCCI topology models. To this aim, we generated a provisioning plan to indicate the order in which the resources have to be provisioned and linked. Based on the actions determined in this plan, we performed REST requests to provision and link the corresponding OCCI elements on the cloud. Hereby, we extracted the required information directly from the topology model. To test the feasibility of this approach, we implemented a prototype able to provision cloud infrastructures, whereby we discussed one example in more detail. In scope of future work, this provisioning engine represents one of the first steps to build a complete model driven cloud orchestrator. Therefore, we will extend this engine by adding the capability to adapt cloud infrastructures at runtime by establishing a feedback loop between the system and the model. Herewith, we introduce the possibility to react to changing requirements or environments. Furthermore, we will evaluate the feasibility of this approach to handle OCCI models using the OCCI PaaS extension. Summed up, we proposed an automatic and standard conform cloud resource provisioning process, based upon a declarative OCCI model, which is able to provision complete infrastructures and offers the possibility to be enhanced to handle complete cloud applications.

References

[Ar10] Armbrust, Michael; Fox, Armando; Griffith, Rean; Joseph, Anthony D; Katz, Randy; Konwinski, Andy; Lee, Gunho; Patterson, David; Rabkin, Ariel; Stoica, Ion et al.: A view of cloud computing. Communications of the ACM, 53(4):50–58, 2010.

[Bé04] Bézivin, Jean: In search of a basic principle for model driven engineering. Novatica Journal, Special Issue, 5(2):21–24, 2004.

[Br14] Breitenbücher, Uwe; Binz, Tobias; Képes, Kálmán; Kopp, Oliver; Leymann, Frank; Wettinger, Johannes: Combining declarative and imperative cloud application provisioning based on TOSCA. In: Cloud Engineering (IC2E), 2014 IEEE International Conference on. IEEE, pp. 87–96, 2014.

[Ep14] Epsilon Transformation Language Documentation, Available online: `http://www.eclipse.org/epsilon/doc/etl/`, 15.06.2017.

[Fe14] Ferry, Nicolas; Brataas, Gunnar; Rossini, Alessandro; Chauvel, Franck; Solberg, Arnor: Towards Bridging the Gap Between Scalability and Elasticity. In: CLOSER. pp. 746–751, 2014.

[GEG17] Glaser, F.; Erbel, J.; Grabowski, J.: Standard and Model Driven Cloud Orchestration by Combining TOSCA and OCCI. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science. CLOSER, 2017.

[In15] OpenStack OCCI Interface, Available online: http://ooi.readthedocs.io/en/stable/, 15.06.2017.

[Kü06] Kühne, Thomas: Matters of (meta-) modeling. Software & Systems Modeling, 5(4):369–385, 2006.

[KWB03] Kleppe, Anneke G; Warmer, Jos B; Bast, Wim: MDA explained: the model driven architecture: practice and promise. Addison-Wesley Professional, 2003.

[Me15] Merle, Philippe; Barais, Olivier; Parpaillon, Jean; Plouzeau, Noël; Tata, Samir: A precise metamodel for open cloud computing interface. In: Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on. IEEE, pp. 852–859, 2015.

[MVG06] Mens, Tom; Van Gorp, Pieter: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science, 152:125–142, 2006.

[OA13] OASIS: Topology and Orchestration Specification for Cloud Applications Version 1.0, Available online: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html, 15.06.2017.

[OG16a] OGF: Open Cloud Computing Interface - Core, Available online: https://www.ogf.org/documents/GFD.221.pdf, 15.06.2017.

[OG16b] OGF: Open Cloud Computing Interface - Infrastructure. OGF, Available online: https://www.ogf.org/documents/GFD.224.pdf, 15.06.2017.

[OG16c] OGF: Open Cloud Computing Interface - Text Rendering. OGF, Available online: https://www.ogf.org/documents/GFD.224.pdf, 15.06.2017.

[OM03] OMG: MDA Guide Version 1.0.1, Available online: http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf, 15.06.2017.

[OM11a] OMG: Business Process Model and Notation, Available online: http://www.omg.org/spec/BPMN/2.0/PDF, 15.06.2017.

[OM11b] OMG: Unified Modeling Language Infrastructure Specification, Available online: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF, 15.06.2017.

[Op16] OpenStack Newton, Available online: https://releases.openstack.org/newton/, 15.06.2017.

[St08] Steinberg, Dave; Budinsky, Frank; Merks, Ed; Paternostro, Marcelo: EMF: eclipse modeling framework. Pearson Education, 2008.