Model-Based Cloud Resource Management with TOSCA and OCCI

Stéphanie Challita, Fabian Korte, Johannes Erbel, Faiez Zalila, Jens Grabowski, Philippe Merle

Noname manuscript No.
(will be inserted by the editor)

Model-Based Cloud Resource Management with TOSCA and OCCI

Stéphanie Challita · Fabian Korte · Johannes Erbel · Faiez Zalila · Jens Grabowski · Philippe Merle

Received: date / Accepted: date

Abstract With the advent of cloud computing, different cloud providers with heterogeneous cloud services (compute, storage, network, applications, etc.) and their related Application Programming Interfaces (APIs) have emerged. This heterogeneity complicates the implementation of an interoperable cloud system. Several standards have been proposed to address this challenge and provide a unified interface to cloud resources. The Open Cloud Computing Interface (OCCI) thereby focuses on the standardization of a common API for Infrastructure-as-a-Service (IaaS) providers while the Topology and Orchestration Specification for Cloud Applications (TOSCA) focuses on the standardization of a template language to enable the proper definition of the topology of cloud applications and their orchestrations on top of a cloud system. TOSCA thereby does not define how the application topologies are created on the cloud. Therefore, we analyse the conceptual similarities between the two approaches and we study how we can integrate them to obtain a complete standard-based approach to manage both Cloud Infrastructure and Cloud application layers. We propose an automated extensive mapping between the concepts of the two standards and we provide TOSCA Studio, a model-driven tool chain for TOSCA that conforms to OCCI. TOSCA Studio allows to graphically design cloud applications as well as to deploy and manage them at runtime using a fully model-driven cloud orchestrator based on the two standards. Our contribution is validated by successfully transforming and deploying three cloud applications: WordPress, Node Cellar and Multi-Tier.

Keywords Cloud Computing · Standards · OCCI · TOSCA · Model-Driven Engineering · Metamodels · Cloud Orchestrator · Models@run.time

Stéphanie Challita
University of Rennes 1 & IRISA/Inria, France.
E-mail: stephanie.challita@irisa.fr

Fabian Korte & Johannes Erbel & Jens Grabowski
University of Goettingen, Germany.
E-mail: firstname.lastame@cs.uni-goettingen.de

Faiez Zalila
CETIC, Belgium.
E-mail: faiez.zalila@cetic.be

Philippe Merle
Inria Lille - Nord Europe & University of Lille, France.
E-mail: philippe.merle@inria.fr

1 Introduction

With the growth of cloud computing, plenty of proprietary cloud APIs have emerged which made it hard for cloud costumers to switch between different cloud providers. To tackle the problem of this *cloud provider lock-in*, consortia have been formed to develop common standards for interfacing with cloud resources. The *Open Cloud Computing Interface* (OCCI) [1], developed by the *Open Grid Forum* (OGF)[1], [2], thereby aims to provide a standardized managing interface, enabling the customer to manage cloud resources. It has been initially published in 2010 and several open-source implementations have been developed since then supporting all major open-source cloud middleware frameworks, including OpenStack[3], OpenNebula[4] and CloudStack[5].

At a higher level of abstraction, the *Organization for the Advancement of Structured Information Stan-*

[1] `https://www.ogf.org/ogf/doku.php/start`
[2] All URLs have been last retrieved on January 27, 2021.
[3] `http://www.openstack.org`
[4] `http://opennebula.org`
[5] `https://cloudstack.apache.org`

dards (OASIS)[6] developed the *Topology and Orchestration Specification for Cloud Applications* (TOSCA), a template format that aims to standardize the definition of application topologies for cloud orchestration. As such, it enables the customer to define the topology of the cloud application in a reusable manner and to deploy it on TOSCA compliant clouds. TOSCA has been initially published in 2013 and major industrial cloud providers such as IBM Cloud are supporting it [2]. In contrast to OCCI, TOSCA does not define how the topologies are programmatically created on the cloud infrastructure and leaves the implementation to the cloud provider. The latter is a complex and error-prone task, it requires expertise in the technical details of the target cloud API.

While the approaches of TOSCA and OCCI are different, both define a model for cloud resources. The goal of this work is to identify the conceptual similarities and differences between the two models and provide a mapping between them where possible. Such a mapping is the first step for building a fully model-driven cloud-provider agnostic cloud orchestrator that leverages both TOSCA and OCCI for portable application and infrastructure provisioning and deployment.

An initial mapping of the two standards was introduced in [3]. In this article, we extend this mapping to support a complete coverage of both standards and we concretely implement our approach based on *Model-Driven Engineering* (MDE) techniques to conceive with a high level of abstraction, verify, deploy and adapt cloud applications. The similarities and differences between the two standards are defined via transformation rules between the concepts of their metamodels. These rules are outputted as a TOSCA model, called *TOSCA Extension*, that defines the necessary information about the characteristics and the management of cloud applications based on TOSCA. TOSCA Extension conforms to the OCCIware metamodel [4, 5] written in Ecore. In fact, the OCCIware approach [6] proposes an enhanced metamodel for OCCI and a whole tool chain for managing cloud resources. We leverage MDE in our approach since it has proven to be quite advantageous and is the mostly adopted methodology to rise in abstraction from the implementation level to the model level. It also reduces the cost of developing complex systems thanks to its ability of validation and artifacts generation.

Proposing a metamodel for a cloud API plays an important role to capture the expectations of this API and to a priori validate the correctness of its cloud configurations. These metamodels are manually designed so far, which is prohibitively labor intensive, time consuming and error-prone. To address this issue, we propose to generate a model-driven specification, i.e., TOSCA Extension, from the documentation of TOSCA written in YAML. This is a work of reverse engineering [7], which is the process of extracting knowledge from a man-made documentation and re-producing it based on the extracted information. TOSCA Extension is at the base of construction of *TOSCA Studio* which is a tool chain for TOSCA based on the OCCIware approach. TOSCA Studio is implemented in the form of a set of Eclipse plugins. It mainly contains a TOSCA Designer allowing users to design, edit and validate TOSCA-based cloud applications, as well as an OCCI Orchestrator allowing users to deploy these applications on IaaS Clouds and manage them, following a models@run.time approach [8]. TOSCA-Studio is publicly available online[7].

In other words, we propose a standard-based and model-driven orchestrator for cloud applications. We extend the features introduced in [3] in the following ways:

- we propose an automated, extensive and extensible approach for mapping TOSCA types towards OCCI types,
- we propose an automated, extensive and extensible approach for mapping predefined TOSCA topologies towards deployable OCCI configurations,
- we provide TOSCA Studio, a model-driven environment for graphically designing and verifying cloud applications using TOSCA concepts,
- and we provide an integrated plugin that ensures a concrete deployment and runtime management of these applications using an OCCI API.

Our contribution targets several audiences. It is useful for TOSCA users since we provide an additional tool for designing and deploying TOSCA topologies, and for the developers of orchestrators since we provide a technical contribution that could inspire them to build their own tool and easily map TOSCA topologies towards a uniform cloud API. Finally, our approach shows that the mapping between two cloud standards is real, which is interesting, in terms of knowledge, for researchers and educators working in this field.

The remainder of this paper is structured as follows. First, we briefly introduce the models of TOSCA and OCCI in Section 2. Then we provide a conceptual comparison, a mapping between the two models and preliminaries about model-driven orchestration in Section 3. In Section 4, we implement our approach and provide a model-driven environment, called TOSCA Studio where the cloud user can design, verify and deploy cloud configurations. These configurations are deployed and maintained via the OCCI Orchestrator. Three feasibility stud-

[6] `https://www.oasis-open.org`

[7] `https://github.com/occiware/TOSCA-Studio`

ies are discussed in Section 5. Section 6 presents the learned lessons from combining the two standards and providing a standard-based and model-driven environment for managing cloud applications. We compare our contribution to related work in Section 7. Finally, we draw our conclusions and give an outlook on future work in Section 8.

2 TOSCA and OCCI

Both TOSCA and OCCI define languages for modeling cloud resources. Since they hence provide a model for modeling they can be seen as *metamodels* [9]. We introduce these metamodels in the following.

2.1 TOSCA

According to its specification [10], TOSCA is "a language to describe service components and their relationships using a service topology, and it provides for describing the management procedures that create or modify services using orchestration processes". Therefore, it is able to describe both the service structure as well as the management processes. As the time of this writing, two versions of TOSCA exist. The first is based on XML [10], and the second is based on YAML [11]. While for TOSCA XML a *XML Schema Definition* (XSD) schema exists, the TOSCA YAML version lacks of a formal metamodel. A simplified metamodel of TOSCA is depicted in Fig. 1.

Service_ template captures the structure and the life cycle operations of the application. It consists of a *Topology_ template* and a *Plan*. Plans define how the cloud application is managed and deployed. Topology_templates contain *Entity_ templates*, which are *Node_ templates* that define e.g., the virtual machines or application components, *Relationship_ templates* that encode the relationships between the Node_templates, e.g., that a certain application component is deployed on a certain virtual machine, or *Group_ templates*[8] that allow to define groups of Node_templates, which e.g. should be scaled together. Additionally, TOSCA defines the Entity_templates *Capability* and *Requirement*. Capabilities are used to define that a Node_template has a certain ability, e.g., providing a container for running applications, and Requirements are used to define that a certain Node_template requires a certain Capability of another Node_template. All Entity_templates can have *Properties*, e.g., an IP address for a virtual machine, and a certain *type* that references an *Entity_ type*. The Entity_type defines the allowed Properties through *Property_ definitions*, and have *Interfaces*, which define the *Operations* that can be executed on instances implementing the type, e.g., the termination of a certain application component, or the restart of a virtual machine. Operations have *Parameters* that define their input and output. In addition to parameters for operations, TOSCA also allows to define input parameters for Plans. Many types inherit from *Entity_ type* such as *Node_ type* and *Relationship_ type*. The former is a reusable entity that defines the type of one or more Node Templates and the latter defines the connection between the node types. We provide more information in Section 3. Besides this abstract metamodel, the TOSCA YAML specification defines *normative types* that should be supported by each TOSCA conforming cloud orchestrator. These normative types include e.g., base types for cloud services and virtual machines. More details on the model elements can be found in [11] and [10].

2.2 OCCI

According to the OGF, "OCCI is a Protocol and API for all kinds of Management tasks. It was originally initiated to create a remote management API for IaaS model based services, allowing for the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring"[9]. The OCCI specification comprises several parts: OCCI Core model, OCCI Extensions, OCCI Renderings and OCCI Protocols. The OCCI Core Model [12] defines a model for cloud resources and their dependencies. In addition to the OCCI Core Model, OCCI Extensions define extensions of the core model to be used for a specific domain. Several extensions are already standardized, e.g., the OCCI Infrastructure Extension [13], which defines compute, network and storage resources for IaaS clouds, and the OCCI Platform Extension for the *Platform-as-a-Service* (PaaS) domain, that defines additional resources for the Platform Service level. The extensions also define the links that can be established between the resources, for instance *StorageLink*, which connects a compute resource to a storage resource and *NetworkInterface* and *IPNetworkInterface* which connects a compute resource to a network resource. Finally, OCCI Renderings define how the OCCI Core Model can be interacted with, e.g., the OCCI HTTP Protocol [14] that defines how OCCI resources can be managed over the

[8] Group_templates and Group_types are currently part of the TOSCA YAML rendering, but not part of the TOSCA XML specification.

[9] `http://occi-wg.org`

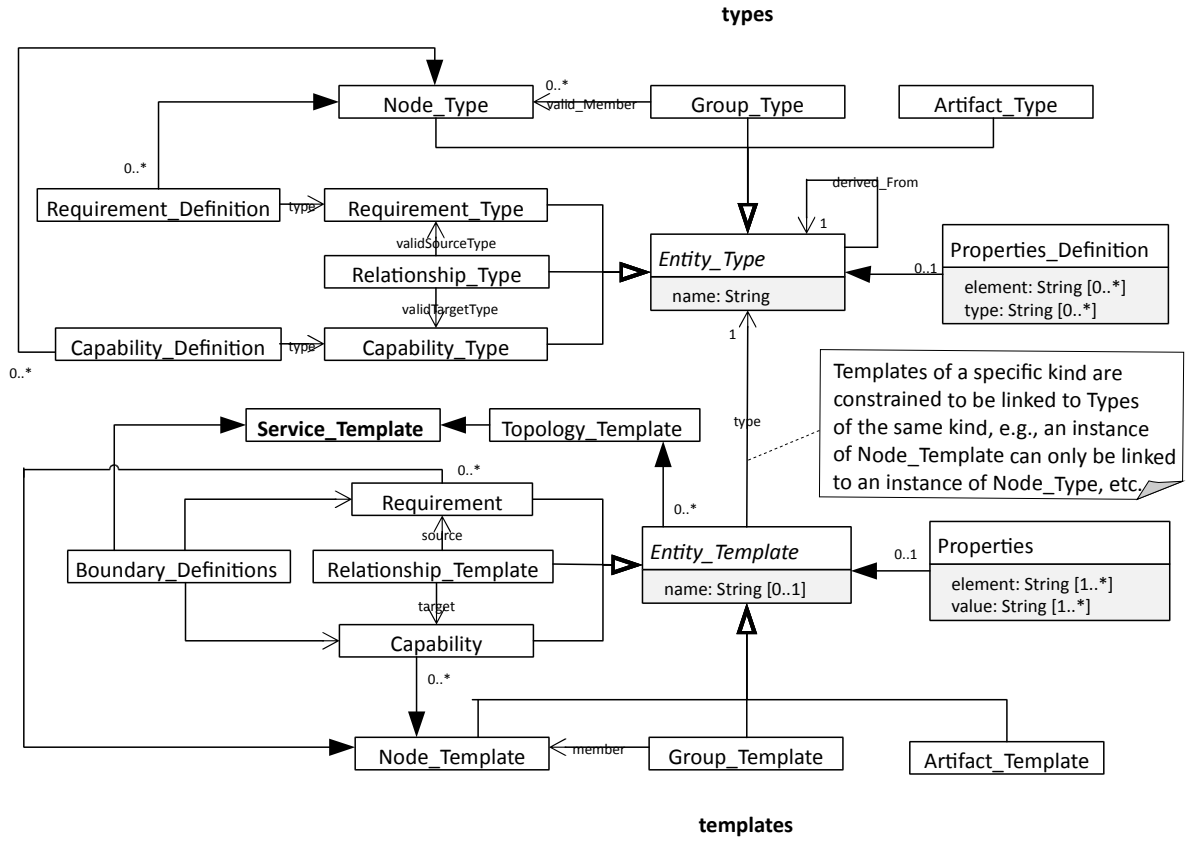

Fig. 1 TOSCA metamodel.

HTTP protocol. The OGF does not provide a formal metamodel for OCCI. This gap has been addressed by the OCCIware approach [5,6] and we adopt the OCCIware metamodel in the scope of this work. The OCCIware metamodel defines precise semantics of OCCI Core which is composed of eight elements that are represented in the grey boxes of Fig. 2. The *Category* is the base type for all other classes and provides the necessary identification mechanisms. Categories have *Attributes* that are used to define the properties of a certain class, e.g., the IP address of a virtual machine. Three classes are derived from Category: *Kind*, *Action*, and *Mixin*. Kind defines the type of a cloud entity, e.g., a compute resource and Mixins define how an entity can be extended. Both have Actions that define which actions can be executed on an entity. The cloud entities themselves are modeled by the class *Entity*, which provides the base class for cloud *Resources*, e.g., virtual machines, and *Links* that define how the resources are connected. Moreover, the OCCIware metamodel explicitly introduces, among others, the two key concepts: *Extension* and *Configuration* as represented in the blue boxes of Fig. 2. An OCCI *Extension* represents a specific domain such as infrastructure, platform, security, etc. and an OCCI *Configuration* defines a running system, i.e., an instance model . It represents an instantiation of one or several OCCI extensions. The yellow boxes of the metamodel represent the *DataType* concepts that define exact types of the attributes such as *StringType*, *RecordType*, *ArrayType*, etc. In addition, the OCCIware metamodel introduces the *Constraint* notion (pink box in Fig. 2) allowing the cloud architect to express business constraints related to each cloud computing domain. The constraints can be imposed on OCCI Kinds and Mixins.

We provide Fig. 3 to better understand the correspondence between the two standards. First, we provide a simple TOSCA topology example, presented as a YAML file (1). This topology is composed of a *tosca.nodes .Compute* node named *my_server* and a *tosca.nodes .BlockStorage* node named *my_storage*. There is a relationship *tosca.relationships.AttachesTo* that connects these two nodes. Later on, we use Winery[10] to model this topology (2). Winery implements the visualization concept specified by Vino4TOSCA [15]. Eventually, we model the same topology in an OCCI dialect (3). One can see that the *tosca.nodes.Compute* is translated into an OCCI Mixin that is applied on an OCCI *Compute*

[10] `https://winery.readthedocs.io/en/latest/user/getting-started.html`

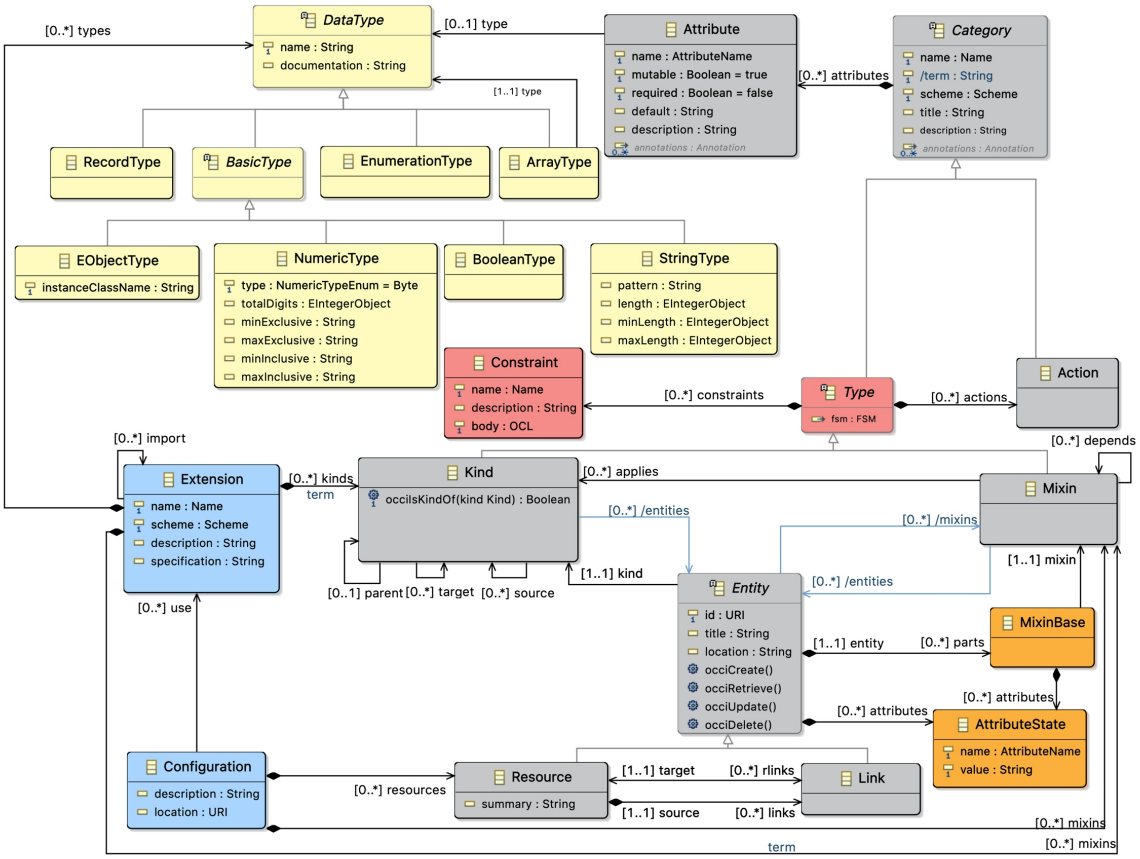

Fig. 2 A subset of OCCIware metamodel (adapted from [5]).

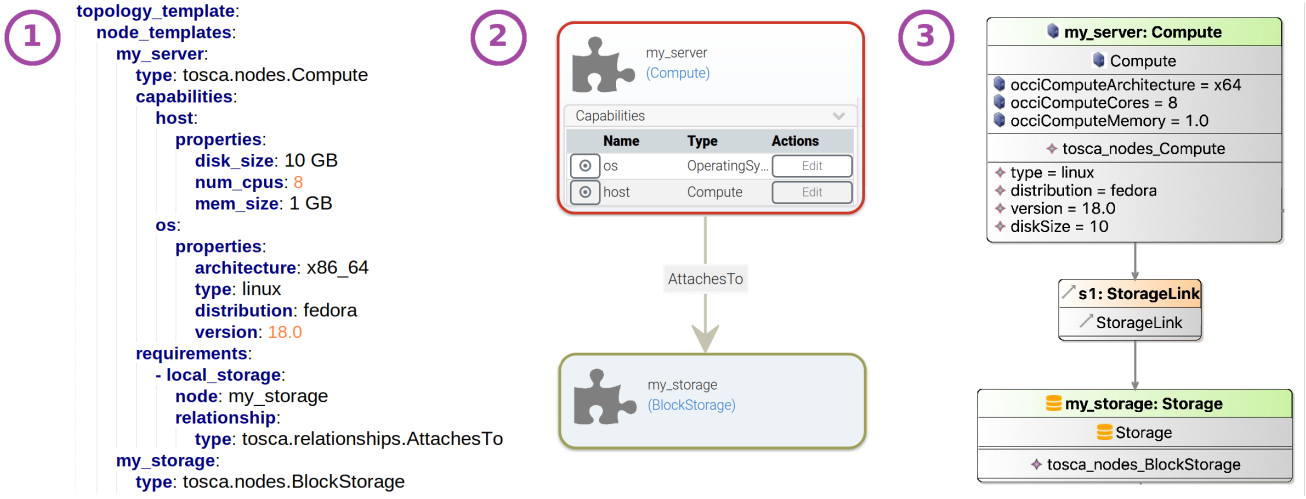

Fig. 3 TOSCA vs OCCI model.

resource. An OCCI Mixin can be seen as an interface that adds additional properties to a Resource, if required even at runtime. In this example, the *tosca_nodes_Compute* Mixin is applied to an OCCI Compute to add TOSCA specific properties like the *type*, *distribution*, *version* and *diskSize*. Due to these extending capabilities of OCCI Mixins, we choose them to represent TOSCA concepts.

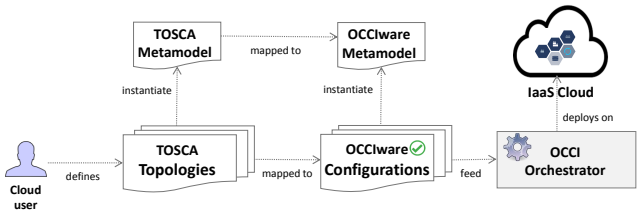

Fig. 4 Approach overview.

3 A Standard-based and Model-driven Approach for Managing Cloud Applications

In this section, we present our contribution that allows cloud application management by relying on TOSCA and OCCI. First, we give an overview of the proposed architecture and then we present how TOSCA concepts can be mapped to those of OCCI. We also describe how the automated generation of appropriate deployment artifacts can be achieved.

3.1 Overview

Our contribution ensures a standard-based approach to handle cloud applications in production environments. An overview of the proposed architecture is shown in Fig. 4. The architecture is composed of three parts: (1) The TOSCA Metamodel that is mapped to the OCCIware Metamodel, (2) the TOSCA Topologies that are mapped to OCCIware Configurations and (3) the OCCI Orchestrator. The TOSCA Metamodel provides an expressive model for cloud applications by relying on an appropriate formalization of TOSCA concepts. The concepts of this model are mapped to OCCIware concepts (cf. Sections 3.2.1 and 3.2.2 for more details). The TOSCA Topologies describe the structure of cloud applications. They instantiate the concepts of the TOSCA metamodel. These topologies are mapped to OCCI configurations that can be designed, edited, validated and deployed as cloud applications. Further information about TOSCA topologies can be found in Section 3.2.3. Finally, OCCI Orchestrator provides means to generate necessary OCCI artifacts and to deploy, via appropriate OCCI requests, the generated artifacts in the executing environment. Every artifact is handled in a seamless way thanks to the homogeneity provided by modeling principles.

To sum up, our approach uses MDE techniques in order to design, verify and deploy cloud applications at a high level of abstraction. In fact, our TOSCA Model describes explicitly concerns of a cloud application that can be instantiated and deployed in an executing environment, i.e., a IaaS Cloud.

3.2 Mapping the two standards

While both standards define a metamodel for cloud resources, their focuses are different. The focus of OCCI is to provide a standardized API and it does not define concepts to address reusability, composability, and scalability. Instances of OCCI are not meant to be stored persistently and to be reused later on as it is the goal of TOSCA. TOSCA on the other side does not define how the defined topology is deployed by means of API calls to the cloud provider as it is done with the OCCI HTTP rendering. Hence, both approaches have their strengths and weaknesses and it is worthwhile to investigate how to integrate them. The mapping between the two standards is possible and is done through three stages: (1) mapping of TOSCA normative types to the OCCIware metamodel, (2) mapping of TOSCA custom types to OCCIware mixins and (3) mapping of TOSCA instantiation concepts to OCCIware instantiation concepts. This mapping is proposed after a deep reading and understanding of both TOSCA and OCCI specifications. Each of the mapping stages will be detailed in the following subsections.

3.2.1 TOSCA metamodel & normative types to OCCIware metamodel

We base our mapping on the TOSCA YAML specification [10] that defines the TOSCA normative types, and on the OCCIware metamodel [6]. We chose the TOSCA YAML specification since it has a more concise syntax, is easier to read and is widely adopted by the community comparing to the TOSCA XML specification. TOSCA normative types model several types of components, called nodes, that interact through relationships. In the following, we present the main concepts of TOSCA and how they can be related to OCCI concepts. The entirety of this mapping is presented in Table 1.

– Entity_type is an abstract concept used to define reusable elements in TOSCA, such as Node_type, Requirement_type, Relationship_type, Policy_type and Capability_type. This matches the purpose of attributes of OCCI Kinds or Mixins. Each Entity_type may have a *description* field that provides a description of the entity and a *derived_from* field that defines the parent this new entity derives from. They match the concepts of *description* and *parent* in OCCI, respectively. Each Entity_type may also have properties or attributes that define the properties that a certain entity is allowed to have. In our approach, we can map all the elements that inherit from Entity_types, namely

Table 1 TOSCA2OCCI mapping: metamodeling level

TOSCA metamodel & normative types	**OCCIware metamodel**
Entity_type	**Mixin**
description	description
derived_from	parent
Property & Attribute	**Attribute**
default	default
required	required
type	DataType
constraints	regular expressions
valid_values	EnumerationType
greater_or_equal	minInclusive
min_length	minLength
Node_type	**Mixin applied to Resource**
nodes.BlockStorage	Mixin applied to Storage Resource
nodes.ObjectStorage	Mixin applied to Storage Resource
nodes.Compute	Mixin applied to Compute Resource
nodes.SoftwareComponent	Mixin applied to Component Resource
nodes.WebServer	Mixin that depends on nodes.SoftwareComponent Mixin
nodes.WebApplication	Mixin applied to Component Resource
nodes.DBMS	Mixin that depends on nodes.SoftwareComponent Mixin and on Database Mixin
nodes.Database	Mixin applied to Component Resource
nodes.LoadBalancer	Mixin applied to Resource
nodes.container.Runtime	Mixin that depends on nodes.SoftwareComponent Mixin
nodes.container.Application	Mixin applied to Component Resource
Requirement_type	**OCL Constraint**
Relationship_type	**Mixin applied to Link**
relationships.AttachesTo	Mixin applied to StorageLink Link
relationships.ConnectsTo	Mixin applied to ComponentLink Link
relationships.DependsOn	Mixin applied to ComponentLink Link
relationships.HostedOn	Mixin applied to ComponentLink Link
relationships.RoutesTo	Mixin that depends on relationships.ConnectsTo Mixin
Datatype_type	**RecordType**
datatypes.Credential	CredentialRecordType
datatypes.network.NetworkInfo	NetworkInfoRecordType
datatypes.network.PortDef	PortDefRecordType
datatypes.network.PortInfo	PortInfoRecordType
datatypes.network.PortSpec	SHORT
Interface_type	**Mixin (with 0 attribute and only actions)**
interfaces.node.lifecycle.Standard	Mixin applied to Resource
interfaces.node.lifecycle.Standard/start()	Component/start() or Storage/online() or Compute/start()
interfaces.node.lifecycle.Standard/stop()	Component/stop() or Storage/offline() or Compute/stop()
interfaces.relationship.Configure	Configure Mixin applied to Link
Operation	Action
Capability_type	**Mixin applied to Resource or Link**

Node_types and Relationship_types to OCCI mixins. Their properties become attributes in OCCI.

– Property & Attribute define the properties that a certain Entity_type is allowed to have. This matches the purpose of OCCI Attributes. A property definition should have a *type*, which matches the *DataType* concept in OCCI. Constraints can be applied to the attribute type, like the *valid_values* constraint that limits the property value to values declared in a list, and the *greater_or_equal* constraint indicating the number of an attribute. For example, CPU that represents a characteristic of an entity, e.g., Compute, is *greater_or_equal* than 2. These two constraints (*valid_values* and *greater_or_equal*) become an *EnumerationType* declaration and a *minInclusive* value in OCCI, respectively. A property may have several optional fields, for example the *required* field that indicates if a property is required or not can be matched to the *required* concept in OCCI.

– NODE_TYPE defines the possible types of building blocks for constructing a cloud application, e.g., virtual machines, network, middleware, etc.. The node types are separately defined for reusability purpose. In fact, the defined node types can be reused when a developer or an application architect wants to define the topology of a cloud application. Further information about the instantiation of the node types are given in Section 3.2.3. The NODE_TYPE concept matches a MIXIN applied to a KIND RESOURCE in OCCI. For example, *tosca.nodes.BlockStorage* becomes a mixin applied to *Storage* kind in OCCI. *tosca.nodes.WebServer* becomes a mixin that depends on *tosca.nodes.SoftwareComponent* mixin. The latter is in turn applied to *Component* kind in OCCI.

– REQUIREMENT_TYPE defines that a certain NODE_TYPE requires a certain capability of another NODE_TYPE. This is encoded as *Object Constraint Language* (OCL) [16] constraints in OCCI mixins.

– CAPABILITY_TYPE extends an ENTITY_TYPE with a certain ability, e.g., providing an operating system for a processor or a container for a server. This concept complies to the concept of an OCCI MIXIN. For example, *tosca.capabilities.OperatingSystem* becomes a mixin that represents the operating system of a certain node. It defines information regarding of the operating system such as its *type*, *distribution* and *version*.

– RELATIONSHIP_TYPE encodes the relationships between the NODE_TYPES, e.g., it encodes that a specific application component is deployed on a specific virtual machine. This becomes a MIXIN applied to a KIND LINK in OCCI. For example, *tosca.relationships.AttachesTo* becomes a mixin applied to *StorageLink* in the OCCI Infrastructure extension.

– DATA_TYPE defines complex data types of TOSCA properties. This concept matches the OCCIware RECORDTYPE concept which is used to define structures. For example, *NetworkInfo* becomes a record type which contains data about the *network* attribute. Each RECORDTYPE has at least one RECORDFIELD which represents a field of the record. In our example, *networkid* and *networkname* are record fields of the *network* attribute and expect a string type value.

– INTERFACE_TYPE defines the allowed OPERATIONS that can be executed on NODE_TYPE or on a RELATIONSHIP_TYPE. This becomes a MIXIN that contains only ACTIONS in OCCI. For example, *tosca.interfaces.node.lifecycle.Standard* becomes a mixin that contains information which operations can be performed on a node type, e.g., *create, configure* and *delete*.

We can map all the elements that inherit from Entity_types, namely Node_types, Requirement_types, Relationship_types, , Datatype_types, Interface_types and Capability_types to OCCI mixins. However, TOSCA introduces some additional concepts, such as Group_types, Policy_types, and Artifact_types, that have no one-to-one correspondents in OCCI. This issue is out of the scope of this article.

3.2.2 TOSCA custom types to OCCIware mixins

The TOSCA specification defines basic root types called TOSCA normative types. These are default types for describing the cloud infrastructure and application. We showed in Section 3.2.1 how we mapped these concepts to the OCCIware metamodel. However, most of the application components are not part of the normative types but extend the TOSCA normative types. These are the custom types. They are defined in several YAML files that are scattered over the internet in GitHub repositories. In fact, during our study, we observed that the community around TOSCA is quite active. Several projects, such as Alien4Cloud[11], Cloudify[12], CELAR[13], SeaClouds[14], DICER[15] and INDIGO-DataCloud[16] have raised. Each project has been defining new TOSCA custom types or modifying existing types according to its need. In our approach, we parse these projects and we automatically map TOSCA custom types to OCCIware mixins, since custom types inherit from TOSCA normatives types. However, some of these custom types appear to be duplicated but with different names. In fact, there is no centralized repository for all these types, which leads to an inconsistent use of TOSCA types across organizations. For example, *tosca.nodes.Mysql*[17] and *tosca.nodes.Database.MySQL*[18],

[11] http://alien4cloud.github.io

[12] https://cloudify.co

[13] https://github.com/CELAR/c-Eclipse

[14] http://www.seaclouds-project.eu/

[15] https://github.com/dice-project/DICER

[16] https://www.indigo-datacloud.eu

[17] https://github.com/alien4cloud/samples/blob/master/mysql/mysql-type.yml#L24

[18] http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html

tosca.Rsyslog[19] and *tosca.nodes.SoftwareComponent .Rsyslog*[20] are two couples of redundant node types that are semantically equivalent but differently defined. In our approach, we collected 30 custom TOSCA types defined in TOSCA projects and mapped them automatically and exhaustively to OCCIware mixins. We show in Table 2 a subset of the TOSCA custom types automatically mapped to OCCIware mixins. In order to add new custom types, a Cloud developer has to provide a YAML file that contains the definition of his/her own types. These types must inherit from the TOSCA normative types using the clause "derived_from". Then, within TOSCA Studio, he/she can parse the YAML file to automatically generate the corresponding OCCIware mixins that are usable by TOSCA-Studio to design and deploy cloud applications.

By mapping TOSCA normative types, defined in the YAML specification, and the diverse added custom types, to those of OCCIware metamodel, we designed a TOSCA Model which conforms to the OCCIware metamodel.

3.2.3 TOSCA instantiation concepts to OCCIware instantiation concepts

The TOSCA specification allows the definition of a cloud application by reusing a set of nodes that are connected to other nodes using relationships. For the definition of provisionable elements, TOSCA defines some ready-to-use topologies that represent popular cloud applications and describe their deployment. A topology is a composition of multiple nodes that may be connected through relationships. Hence, topologies use the concepts of Entity_templates, namely Node_templates, Relationship_templates and Group_templates. We explain in the following, how TOSCA topologies can be mapped to OCCIware configurations and we provide a summary in Table 3.

- A Topology_template defines a reusable and portable representation of the structure of an application to facilitate understanding of its functional components and eliminating unnecessary details. It consists of a set of Node_templates and Relationship_templates. Each Topology_template is mapped into a Configuration in OCCI.

- A Node_template specifies the occurrence of a node in a topology template. Each Node_template refers to a Node_type and instantiates the semantics of its properties, attributes, requirements, capabilities and interfaces. It gets to be transformed into a Resource with a MixinBase in an OCCI Configuration. A MixinBase refers to a Mixin and instantiates the attributes of the referenced mixin outside the owner entity in order to separate the entity attributes from the mixin ones.

- A Relationship_template specifies the occurrence of a relationship between nodes in a topology template. Each Relationship_template refers to a Relationship_type and instantiates the semantics of its properties, attributes, interfaces, etc. It can be transformed into a Link with a MixinBase in an OCCI Configuration.

- A Group_template defines a group of nodes that share some semantics, e.g. an autoscaling group is a group of *Virtual Machine*s (VMs) that would be scaled together. It can be transformed into a number of Resources and Links in OCCI.

The mapping detailed above is used to propose a fully model-driven cloud orchestrator which we define in the next subsection.

3.3 Model-driven cloud orchestration

To deploy the cloud application specified in the TOSCA topology, we use a combined approach of TOSCA and OCCI, as shown in Fig. 4. By utilizing both standards we benefit from their individual advantages while neglecting their drawbacks. Namely the matured design time capabilities of TOSCA topologies combined with the uniform interface provided by OCCI allowing to instantiate the desired cloud application using the process depicted in Fig. 5. Based on the modeled TOSCA topology, an OCCI *Platform Independent Model* (PIM) is generated, i.e., a (PIM) OCCIware Configuration. The resulting configuration, containing the modeled TOSCA resources as OCCI elements, then serves as input for the OCCI Orchestration process initially described in [17]. In general, the concept resembles a models@run.time approach [8] which derives imperative steps from a declarative description in order to adapt the running system. In the following the transformation, as well as the derivation of required OCCI requests to reach the state of the designed cloud configuration are described in more detail.

[19] https://github.com/openstack/tosca-parser/blob/master/toscaparser/tests/data/custom_types/nested_rsyslog.yaml

[20] https://github.com/openstack/tosca-parser/blob/master/toscaparser/tests/data/custom_types/rsyslog.yaml

Table 2 Subset of TOSCA2OCCI mapping: metamodeling level

TOSCA custom types	OCCIware mixins
nodes.Apache	Mixin that depends on nodes.WebServer Mixin
nodes.SoftwareComponent.Collectd	Mixin that depends on nodes.SoftwareComponent Mixin
nodes.HACompute	Mixin that depends on nodes.Compute Mixin
nodes.Database.Mysql	Mixin that depends on nodes.Database Mixin
nodes.DBMS.MySQL	Mixin that depends on nodes.DBMS Mixin
nodes.Container.Application.Docker	Mixin that depends on nodes.container.Application Mixin
nodes.SoftwareComponent.Elasticsearch	Mixin that depends on nodes.SoftwareComponent Mixin
nodes.SoftwareComponent.Logstash	Mixin that depends on nodes.SoftwareComponent Mixin
nodes.SoftwareComponent.Kibana	Mixin that depends on nodes.SoftwareComponent Mixin
nodes.AbstractMysql	Mixin that depends on nodes.Database Mixin
nodes.network.Network	Mixin applied to Network Resource
nodes.network.Port	Mixin applied to Network Resource
nodes.Nodejs	Mixin that depends on nodes.WebServer Mixin
nodes.WebApplication.PayPalPizzaStore	Mixin that depends on nodes.WebApplication Mixin
nodes.PHP	Mixin that depends on nodes.SoftwareComponent Mixin
nodes.SoftwareComponent.Rsyslog	Mixin that depends on nodes.SoftwareComponent Mixin
nodes.Wordpress	Mixin that depends on nodes.WebApplication Mixin
nodes.Nodecellar	Mixin that depends on nodes.WebApplication Mixin
nodes.MongoD	Mixin that depends on nodes.DBMS Mixin

Table 3 TOSCA2OCCI mapping: modeling level

TOSCA instantiation concepts	OCCIware metamodel
Topology_template	Configuration
Node_template	Resource with a MixinBase
Relationship_template	Link with a MixinBase

3.3.1 PIM to PSM Transformation

To allow for an automated deployment of the OCCI configuration on the target environment, a *Platform Specific Model* (PSM) is generated from the (PIM) OCCIware Configuration that contains all information required for the deployment. To generate the (PSM) OCCIware Configuration, a *Model-to-Model transformation* (M2M) transformation is applied that can be configured to add cloud provider specific information. The information to be added largely depends on the elements introduced within the OCCI extensions used to handle cloud provider specifics. For example, in the extension of Paraiso et al. [18] specialized Compute entities are designed offering attributes that can be filled to provision VMs on individual cloud providers. In the presented approach, the transformation is mainly used to ensure a correct functionality of the orchestration process. This comprises the addition of a mixin to each infrastructural resource which relates the OCCI id to the id assigned by the cloud provider allowing for a concrete mapping between the modeled and actual resource running in the cloud. Additionally, the transformation adds a management network to the OCCI configuration that serves two purposes. Firstly, it ensures a connection between the OCCI interface and each modeled VM which is needed to deploy the modeled OCCI components via configuration management scripts. Secondly, it fulfills the requirement of specific cloud providers, such as Openstack, to declare a network to which a newly provisioned VM gets connected. It should be noted that the transformation may be used to incorporate further provider specifics, e.g., by adding OCCI Resource and Operating system templates [13] describing available sizes and images for VMs by the target environment over OCCI terms and schemes. Finally, the resulting OCCI configuration serves as input for the orchestration process as described in the following.

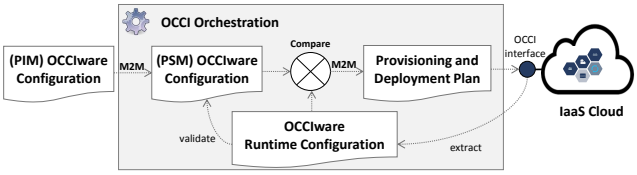

Fig. 5 Model-driven cloud orchestration process.

After the transformation, the orchestration process extracts the *current* cloud deployment in form of an OCCI Runtime Configuration, and compares it to the *desired* cloud deployment. Based on this comparison a M2M is triggered that generates a Provisioning and Deployment Plan (see e.g., [17], [19] or [20] for automatic deployment and provisioning workflow generation). Within this plan the OCCI requests required to manage cloud resources are sequenced in order to bring the cloud resources into the desired state. It should be noted that the OCCI requests are gener-

ated from the elements contained within the (PSM) OCCIware Configuration. Finally, the provisioned and deployed OCCI model can be again extracted and synchronized to validate whether the design time concepts and constraints, specified in the TOSCA model, are met. We exemplify the proposed orchestration process in Section 5.

3.3.2 Orchestration Process

To derive required adaptive steps, the current and desired cloud deployment need to be investigated. As a first step the orchestration process extracts the current cloud deployment in form of an OCCI Runtime Configuration, and compares it to the desired one. During the comparison each individual resource of the new and actual deployment are matched to each other based on their identity, i.e. their id and kind. Based on the match, each entity is identified as already existing, to be deleted, updated or added resulting in the corresponding instructions that need to be send to the OCCI interface. While entities marked as to be updated get their values adjusted in the runtime model and resources marked as to be deleted get undeployed and deprovisioned, the resources to be added need to be provisioned in the correct order.
To sequence the provisioning requests, a M2M is performed on the input model generating a Provisioning and Deployment Plan. As a first step within this transformation, a *provisioning order graph* [21] is generated. This graph describes the dependencies between modeled cloud resources based on the kind of links between them. Thereafter, we remove the updated and existing resources from the graph, as they are already present in the currently deployed runtime model [17]. As a final step, this graph is transformed into an UML activity diagram containing a sequence of provisioning actions to follow. Hereby, each action in the diagram refers to an OCCI entity from which the requests are formed. The resulting activity diagram is interpreted by the orchestration process which is responsible for sending the provisioning requests to the OCCI interface. After the activity diagram has been interpreted successfully, the runtime model has reached the new desired state with the infrastructural resources being started. Then the deploy, configure and start action is triggered on each application within the runtime model resulting in the deployment of the individual application components. The provisioned and deployed OCCI model can be subsequently extracted and synchronized to validate whether the design time concepts and constraints, specified in the TOSCA model, are met. We exemplify the proposed orchestration process in Section 5.

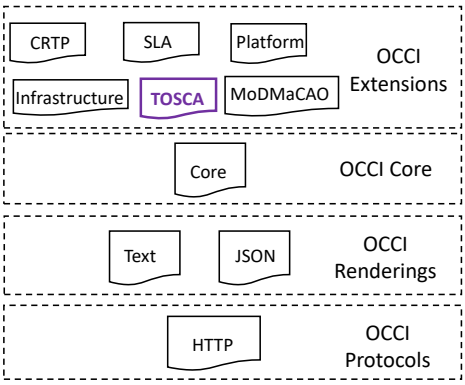

Fig. 6 OCCI specifications.

4 Implementation: TOSCA Studio

Our approach for managing cloud applications by relying on TOSCA topologies and the OCCIware framework and API is implemented through **TOSCA Studio**. TOSCA Studio is a model-driven tool chain for modeling and deploying cloud application topologies encoded by the TOSCA standard based on the OCCIware approach. It relies on a metamodel, called **TOSCA Extension**, defining the static semantics for the TOSCA standard in Ecore and OCL [16] and conforming to the OCCIware Metamodel. More specifically, TOSCA Studio is implemented as a set of Eclipse plugins that are publicly available[21]. It mainly contains a **TOSCA Designer** that provides users facilities for designing, editing, validating TOSCA-based cloud applications, and an **OCCI Orchestrator** that allows users to deploy and manage these applications. In this section, we detail each of our solution main components: **TOSCA Extension**, **TOSCA Designer**, and **OCCI Orchestrator**.

4.1 TOSCA Extension

The mapping between the original TOSCA metamodel (in YAML) and OCCIware metamodel, detailed in Section 3.2, is encoded as an OCCI extension for TOSCA (called the **TOSCA Extension**), as depicted in the purple box in Fig. 6. TOSCA Extension captures all the necessary information related to the characteristics and management of TOSCA-based cloud applications. TOSCA Extension is represented in the form of a diagram in Fig. 7. This diagram was designed with OCCIware Studio, our open source model-driven development environment dedicated to OCCI [5,6]. TOSCA Extension is conceptually divided into three levels.

[21] https://github.com/occiware/TOSCA-Studio

The top level represents the OCCI Core model, encoded with the *Eclipse Modeling Framework* (EMF). The middle level contains OCCI standardized extensions, which are OCCI Infrastructure, OCCI Platform and OCCI SLA. It also contains *Model-Driven Configuration Management of cloud Applications with OCCI* (MoDMaCAO) [22], which is an enhanced version of the standardized OCCI Platform extension. The bottom level represents TOSCA concepts, which extend the Infrastructure, MoDMaCAO and SLA extensions, in the form of mixins. TOSCA Extension is quite rich in concepts. It contains 68 mixins, 10 of which extend the Infrastructure extension, 33 extend the MoDMaCAO extension and 4 extend the SLA extension. The remaining concepts extend the generic Resource and Link types from the OCCI Core model. Fig. 7 illustrates how TOSCA mixins extend already existing OCCI extensions, and shows the graphical output of a subset of the TOSCA Extension. For example, *tosca_ nodes_ Compute* extends OCCI Infrastructure. It contains an OCL constraint SourceMustBeSoftwareComponent which enforces that the *Compute* instance cannot run if it is not linked to a *SoftwareComponent* instance. It also depends on three other mixins *tosca_ capabilities_ Container*, *tosca_ capabilities_ OperatingSystem* and *tosca_ capabilities_ Endpoint*, and therefore inherits their attributes. TOSCA Extension also defines exact types thanks to the DataType system provided by the OCCIware metamodel. The EMF validator then checks the type constraints that are attached to the attribute. For example, scalarSizeMinOneMB is translated into a NumericType, especially an Integer, containing the following constraint: *minInclusive = 1*.

To implement TOSCA Extension, we implemented a YAML parser in Java, using yamlbeans[22] library. In fact, we provide an algorithm that infers TOSCA Extension from YAML specifications. This automated extraction allows a better modeling of TOSCA concepts. So far, existing models are manually designed, which is prohibitively labor intensive, time consuming and error-prone. To address this issue, we propose a novel approach to automatically infer model-driven specification from YAML specification files of TOSCA standard. First, using the OCCI API, we create a model, i.e., an OCCI extension. Then, the algorithm loads the content of the YAML specification files using yamlbeans library. The types in these YAML file are grouped semantically: nodes, relationships, capabilities, data, etc. The algorithm runs through each of this group, then for each TOSCA type it builds the corresponding OCCI mixin. For each type, the algorithm matches its information (*derived_from*, *description*, *attributes*, *requirements*, etc.) to corresponding OCCI concepts. If an attribute requires a type that has not been defined yet, the algorithm keeps the name of this type in memory. When defined, the latter will be assigned to this attribute. Eventually, we add the new mixin to the model under construction and so on. This model represents our TOSCA Extension.

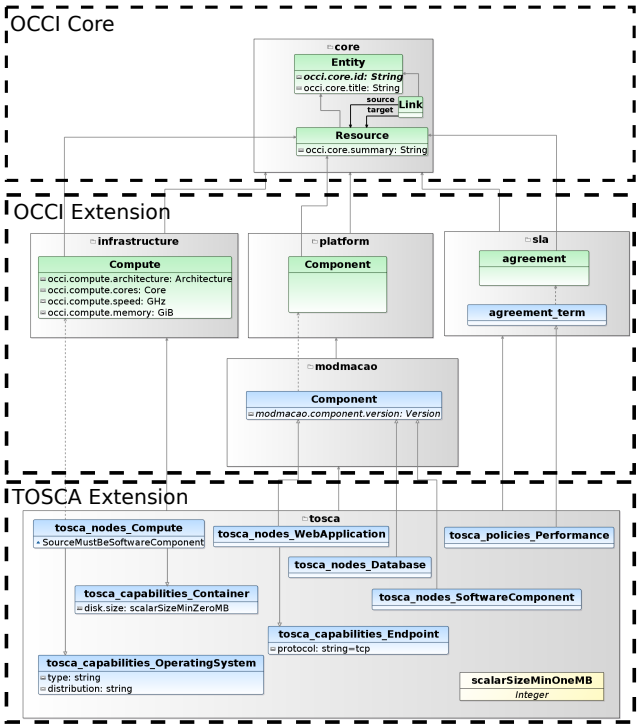

Fig. 7 A subset of TOSCA Extension.

Readers can find the parser code as well as our precise TOSCA Extension on GitHub[23].

4.2 TOSCA Designer

Our approach allows cloud architects to visualize, edit and verify configured instances of cloud applications using TOSCA types defined in TOSCA Extension. To do so, we provide **TOSCA Designer**, which is a specific graphical modeler of both OCCI extensions and configurations for TOSCA. This tool is implemented on top of the Eclipse Sirius framework. A screenshot of our TOSCA Designer is depicted in Fig. 8. Frame (1) shows the Eclipse Model Explorer used to navigate through a TOSCA project containing a TOSCA Model. Frame (2) gives a perspective or a global view of the modeled topologies. Frame (3) contains the topology elements. Frame (4) contains the Eclipse properties editor for visualizing and modifying attributes of

[22] https://github.com/EsotericSoftware/yamlbeans

[23] https://github.com/occiware/TOSCA-Studio

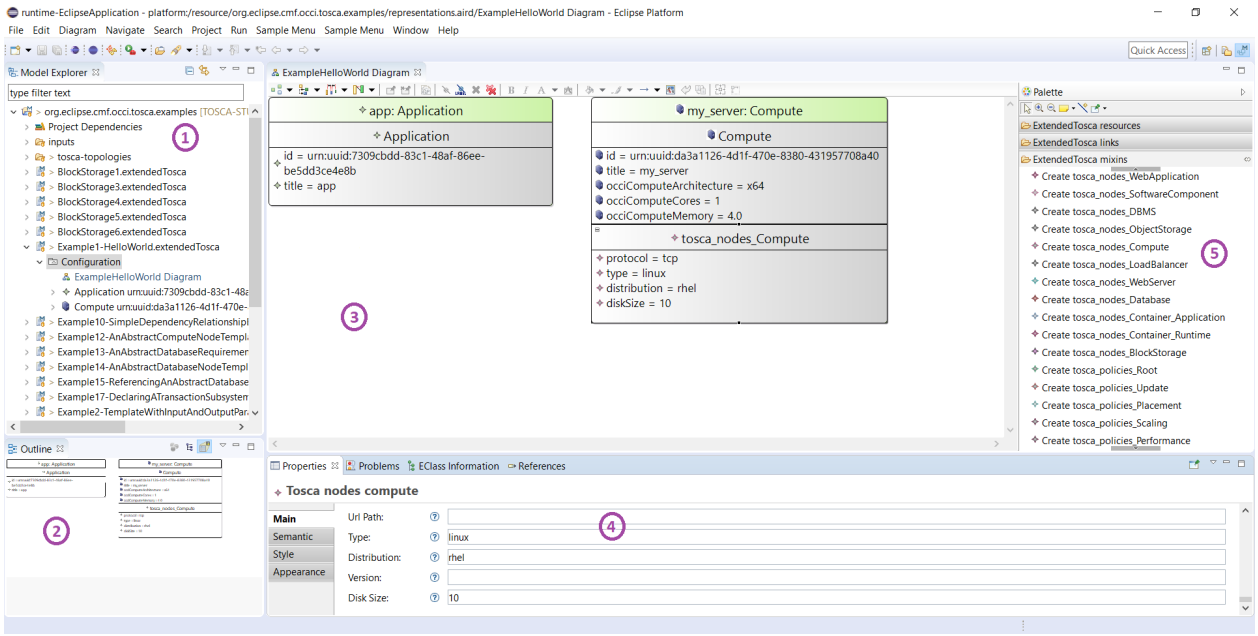

Fig. 8 TOSCA Designer.

a selected modeling element. All TOSCA elements displayed in Frame (3) can be set through their properties. Frame (5) displays the configuration pallet that represents the TOSCA types (normative and custom) such as: *tosca_nodes_WebApplication, tosca_nodes_SoftwareComponent, tosca_nodes_WebServer, tosca_nodes_Apache*, etc.

This tool can be used in two ways:

1. It can take as input any existing TOSCA topology, translate it into an OCCI configuration for TOSCA and graphically represent it. To do so, we implemented a **config-generator**, which parses any existing TOSCA topology_template written in YAML and transforms it into an OCCI configuration that conforms to the TOSCA Extension. More specifically, OCCI configuration instantiates the normative and custom types defined in TOSCA Extension.
2. It can design cloud applications from scratch using TOSCA types from the palette of TOSCA Designer.

Finally, TOSCA Designer checks the validity of cloud application configurations by checking all the constraints defined by used TOSCA mixins. If a constraint is false, the cloud architect must correct its cloud application configuration. When all the constraints are true, the TOSCA-based cloud application can be deployed using OCCI Orchestrator.

4.3 OCCI Orchestrator

To provision and deploy the transformed TOSCA application over an OCCI interface, either the required OCCI requests can be generated using TOSCA Studio and manually sequenced or the presented OCCI Orchestration process can be used. An implementation of the orchestration process is publicly available [24]. While the presented orchestration process can be generally applied on any kind of OCCI API, such as the *OpenStack OCCI Interface* (OOI) used in [17], the implementation got enhanced to focus on the OCCI API provided by the *OCCIware Runtime*[25]. The OCCIware Runtime is a server that maintains a runtime model of the currently deployed cloud system which is utilized by the orchestrator. Moreover, the OCCIware Runtime server follows a plugin-based architecture for OCCI extensions modeled with OCCIware, such as the TOSCA extension. Based on these extensions, connector skeletons can be generated that can be filled to interpret incoming OCCI requests. Among others, this mechanism is used to translate incoming OCCI infrastructure requests to the proprietary interface of the cloud provider. While our implementation provides a connector to an OpenStack Cloud, connector skeletons to further cloud providers can be easily modeled and generated within TOSCA Studio. Moreover, to address multi-cloud deployments, extensions such as the one presented in [18] can be created to model on which specific cloud provider a VM should be provisioned including required attributes.

To perform the deployment of modeled components, the MoDMaCAO framework [22] is used, providing a connector which implements lifecycle operations for OCCI

[24] `https://gitlab.gwdg.de/rwm/de.ugoe.cs.rwm.docci`
[25] `https://github.com/occiware/MartServer`

Applications and Components. These operations trigger the execution of configuration management scripts to deploy applications on top of VMs as specified within the generated OCCI configuration. Therefore, the framework provides a generic component mixin that can be extended. Each specialized component mixin is linked to the configuration management script to be executed which is deposited on the OCCIWare Runtime server making them reusable for multiple cloud deployments. In addition to typical configuration management features, the framework allows to use modeled attributes as well as runtime information, e.g., attributes of linked components and machines, within the configuration management scripts. It should be noted that the specialized component mixins, such as the ones shown in Table 2 are automatically generated during the TOSCA to OCCI transformation. To perform the tasks specified within the configuration management scripts, the OCCIware Runtime server needs to be connected to the VMs to be configured which is ensured by the PIM to PSM transformation. In general, for the application deployment process the orchestrator only sends a request to the OCCI API triggering the start action of the applications to be deployed. The execution of the management scripts to deploy the modeled application is then handled by the MoDMaCAO framework. To ensure that all infrastructure requirements of the cloud topology to be deployed are met, the application deployment is only performed when the provisioned infrastructure, reflected in the runtime model, conforms to the designed state of the OCCIware configuration to be deployed.

Within TOSCA-Studio, the transformation, as well as the deployment process can be directly enacted on top of modeled or generated OCCI Configurations which allows to easily model, manage and deploy cloud resources.

5 Case Studies

To evaluate the proposed approach, we selected three case studies that represent popular distributed cloud applications: a WordPress application[26], a Node Cellar application[27] and a Multi-Tier application with Elasticsearch Logstash Kibana (ELK) stack[28]. We chose the first two case studies because they are medium size which proves that our approach is able to handle real applications and allow us to present them in a clear manner. Moreover, these two case studies are widely used in the community. WordPress is one of the most adopted *Content Management System* (CMS) in industry and it is used as an example in TOSCA official specification [10], and NodeCellar is a prominent and interactive LAMP stack application that is used in Alien4Cloud and Cloudify projects. In order to prove that our approach can support larger systems, we chose the Multi-Tier use case which represents a complex system composed of ELK stack together with a NodeCellar application. We relied on existing TOSCA YAML topologies for WordPress[29] and Node Cellar[30]. For our third use case, we adapted the Multi-Tier example presented in TOSCA official specification [10]. We only replaced the PaypalPizzaStore web application by the NodeCellar application since the former is not available anymore. We demonstrate how our approach can design, validate and deploy these applications, and we provide a configuration model for each.

To provision and deploy the modeled cloud configurations, we used the presented model-driven cloud orchestration process. Hereby, we connected the OCCI Orchestrator to a private Openstack cloud to provision modeled infrastructure resources. To perform the deployment of the individual components, the MoDMaCAO framework is used. The latter executes scripts of Ansible[31],for which we utilized and updated already existing deployment artifacts used in the TOSCA topologies to deploy, configure, and start modeled application components.

5.1 WordPress

WordPress is an open source CMS that allows to build custom web applications based on Apache as the web server, MySQL as the relational database management system and PHP as the object-oriented scripting language. TOSCA allows to define such types, and therefore it allows to define a WordPress application. A screenshot of the WordPress topology as defined in TOSCA YAML file is depicted in Fig. 9.

To validate our approach, the config-generator reuses the mixin types defined by the TOSCA Extension to be able to model a WordPress system. It is done with the help of the following mixins:

- The **tosca.nodes.WordPress** mixin type abstracts the notion of a WordPress CMS and depends on the tosca.nodes.WebApplication mixin type, which depends on the MoDMaCAO Component mixin type.

[26] `https://fr.wordpress.com`

[27] `http://nodecellar.coenraets.org`

[28] `https://www.elastic.co/fr/elastic-stack`

[29] `https://github.com/alien4cloud/samples/tree/master/topology-wordpress`

[30] `https://github.com/alien4cloud/samples/tree/master/topology-nodecellar`

[31] `https://www.ansible.com/`

```yaml
topology_template:
  description: Wordpress deployment template
  inputs:
    os_distribution:
      type: string
      constraints:
        - valid_values: [ debian, ubuntu, knoppix ]
      description: The host Operating System (OS) architecture.
      default: ubuntu
  node_templates:
    wordpress:
      type: tosca.nodes.Wordpress
      requirements:
        - host: apache
        - database:
            node: mysql
            capability: tosca.capabilities.Endpoint.Database
        - php:
            node: php
            capability: tosca.capabilities.Root
    php:
      type: tosca.nodes.PHP
      requirements:
        - host: computeWww
    computeDb:
      type: tosca.nodes.Compute
      capabilities:
    # omitted for brevity
    computeWww:
      type: tosca.nodes.Compute
      capabilities:
        os:
          properties:
            type: linux
            architecture: x86_64
            distribution: { get_input: os_distribution }
        host:
          properties:
            num_cpus: 1
            disk_size: 1 GB
            mem_size: 1024 MB
            cpu_frequency: 1 GHz
    apache:
      type: tosca.nodes.Apache
      properties:
      requirements:
        - host: computeWww
    mysql:
      type: tosca.nodes.Mysql
      requirements:
        - host: computeDb
```

Fig. 9 WordPress YAML Topology.

It is hosted on an Apache WebServer and connected to a MySQL database and a PHP SoftwareComponent.

- The **tosca.nodes.Apache** mixin type abstracts the notion of an Apache server. It depends on the tosca.nodes.WebServer mixin type, which depends on the tosca.nodes.SoftwareComponent mixin type and therefore also on the MoDMaCAO Component mixin type.
- The **tosca.nodes.Mysql** mixin type abstracts the notion of a Mysql database. It depends on the tosca.nodes.Database mixin type, which depends on the MoDMaCAO Component mixin type.
- The **tosca.nodes.PHP** mixin type abstracts the notion of PHP scripting language used to develop a WordPress application. It depends on the tosca.nodes.SoftwareComponent mixin type, which depends on the MoDMaCAO Component mixin type.
- The **tosca.nodes.Compute** mixin type abstracts the notion of real or abstract processors of software applications such as VMs. It is applied on the Compute resource type.

Fig. 10 shows the model of a WordPress application that corresponds to the topology in Fig. 9. It is composed of four components (WORDPRESS, PHP, APACHE and MYSQL) deployed on two VMs (COMPUTEWWW and COMPUTEDB). OCCI resources and links are represented by boxes in green and orange color, respectively. The application resource is connected to the four Component resources via COMPONENTLINKS (C1 to C4). The WordPress component is connected to the PhP and MySQL components via CONNECTSTO links (C5 and C7). The WordPress is hosted on the APACHE component via a HOSTEDON link (C6). Each component is placed on one VM via a PLACEMENTLINK (P1 to P4). Finally, the properties of all the components and VMs are configured. For the sake of brevity, we omit the depiction of ATTRIBUTES of the components in this model. For illustration, we only keep the attributes of COMPUTEWWW. We can notice that its properties declared in the YAML file of Fig. 9, i.e., the architecture, the number of cores, the speed, the memory, the protocol, the type, the distribution and the disk size, have been correctly automatically set in the model.

5.2 Node Cellar

The Node Cellar application is a sample JavaScript application that allows to manage (retrieve, create, update, delete) the wines in a wine cellar database. A Node Cellar application can be described using TOSCA types, as depicted in the TOSCA YAML file is depicted of Fig. 11.

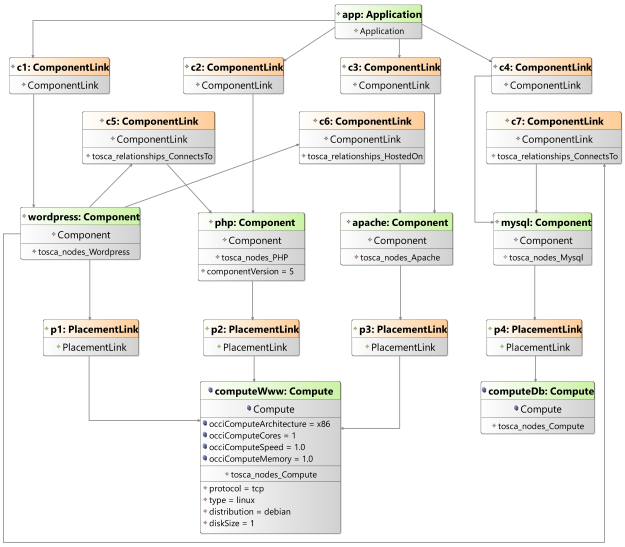

Fig. 10 WordPress Configuration.

This topology is automatically transformed into a Node Cellar Configuration using the following mixins defined in TOSCA Extension:

- The **tosca.nodes.Nodecellar** mixin type abstracts the notion of a Node Cellar application and depends on the tosca.nodes.WebApplication mixin type which depends on the MoDMaCAO Component mixin type. It is hosted on a Nodejs server and connected to a MongoDB database.
- The **tosca.nodes.MongoDB** mixin type abstracts the notion of a MongoDB database. It depends on the tosca.nodes.DBMS mixin type, which depends on the MoDMaCAO Component mixin type.
- The **tosca.nodes.Nodejs** mixin type abstracts the notion of a JavaScript running environment. It depends on the tosca.nodes.WebServer mixin type, which depends on the MoDMaCAO Component mixin type.
- The **tosca.nodes.Compute** mixin type abstracts the notion of real or abstract processors of software applications such as VMs. It is applied on the Compute resource type.

Fig. 12 shows the model of a Node Cellar application that corresponds to the topology in Fig. 11. It is composed of three components (NODECELLAR, NODEJS and MONGODB) deployed on two VMs (NODEJSHOST and MONGOHOST). OCCI resources and links are represented by boxes in green and orange color, respectively. The application resource is connected to the three component resources via COMPONENTLINKS (C1 to C3). The Nodecellar component is connected to the MongoDB component via a CONNECTSTO link (C4). The NodeCellar is hosted on the NODEJS component via a HOSTEDON link (C5). Each component is placed on one VM via a PLACEMENTLINK (P1 to P3). Finally, the properties of all the components and VMs are configured. For the sake of brevity, we omit the depiction of ATTRIBUTES of the components in this model. For illustration, we only keep the attributes regarding the ports used by MONGOD and NODECELLAR. We can notice that the ports values declared in the YAML file of Fig. 11 have been correctly automatically set in the model.

5.3 Multi-Tier

This use case shows the ELK stack being used in a typical manner to collect, search and monitor/visualize data from a running application. This use case builds upon our NodeCellar application (cf. Section 5.2) as the one being monitored. We successfully describe a Multi-Tier application using TOSCA types, as depicted in the TOSCA YAML file of Fig. 13.

Besides the mixins used to describe a NodeCellar Configuration (cf. Section 5.2), the Multi-Tier Configuration encompasses the following mixins:

- The **tosca.nodes.SoftwareComponent.Elasticsearch** mixin type abstracts the notion of an Elasticsearch search and analytics engine.
- The **tosca.nodes.SoftwareComponent.Logstash** mixin type abstracts the notion of a Logstash data collection engine.
- The **tosca.nodes.SoftwareComponent.Kibana** mixin type abstracts the notion of a Kibana data visualization dashboard.
- The **tosca.nodes.SoftwareComponent.Collectd** mixin type abstracts the notion of a Collectd daemon which collects system and application performance stats.
- The **tosca.nodes.SoftwareComponent.Rsyslog** mixin type abstracts the notion of a an Rsyslog program which transfers log messages over an IP network.

All the five mixins defined above depend on the tosca.nodes.SoftwareComponent mixin type which depends on the MoDMaCAO Component mixin type.

We show in Fig. 14 the Multi-Tier configuration that corresponds to the YAML topology in Fig. 13. It is composed of nine components (NODECELLAR, ELASTICSEARCH, LOGSTASH, KIBANA, APP_COLLECTD, APP_RSYSLOG, NODEJS, MONGO_DB, MONGO_DBMS) deployed on six VMs (ELASTIC_SERVER, LOGSTASH_SERVER, KIBANA_SERVER, NODEJSHOST, APP_SERVER, MONGO_SERVER). Similarly to the two previous use cases, OCCI

```yaml
topology_template:
  description: A Javascript Sample Application with Node.js and MongoDB
  inputs:
    mongo_port:
      type: integer
      required: true
      default: 27017
    nocecellar_port:
      type: integer
      required: true
      default: 8088
  node_templates:
    NodejsHost:
      type: tosca.nodes.Compute
      capabilities:
    # omitted for brevity
    Nodejs:
      type: tosca.nodes.Nodejs
      requirements:
        - host:
            node: NodejsHost
            capability: tosca.capabilities.Container
            relationship: tosca.relationships.HostedOn
    Nodecellar:
      type: tosca.nodes.Nodecellar
      properties:
        port: { get_input: nocecellar_port }
      requirements:
        - host: Nodejs
        - database:
            node: Mongod
            capability: tosca.capabilities.Endpoint.Database
            relationship: tosca.relationships.ConnectTo
    MongoHost:
      type: tosca.nodes.Compute
      capabilities:
    # omitted for brevity
    Mongod:
      type: tosca.nodes.Mongod
      properties:
        port: { get_input: mongo_port }
      requirements:
        - host: MongoHost
      capabilities:
    # omitted for brevity
```

Fig. 11 Node Cellar YAML Topology.

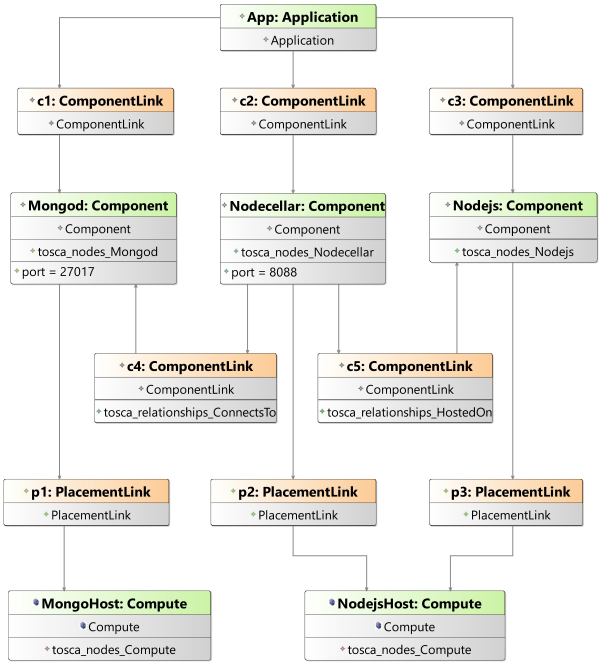

Fig. 12 Node Cellar Configuration.

resources and links are represented by boxes in green and orange color, respectively. The application resource is connected to the nine component resources via ComponentLinks (c1 to c9). Each component is placed on one VM via a PlacementLink (p1 to p9). app_collectd and app_rsyslog share the same VM, as well as mongo_db and mongo_dbms. For brevity, we omit the Attributes of the components in this model.

5.4 Orchestration

The presented use case topologies are deployed in the cloud using the model-driven cloud orchestration process. Before the service requests for the individual OCCI resources and links are send to the OCCIware Runtime, the PIM to PSM transformation is performed on the input configuration, i.e, the WordPress , Multi-Tier or Node Cellar configuration. This transformation ensures that the requirements of the MoDMaCAO framework are fulfilled by adding a management network resource to the OCCI configuration. This network ensures, that the MoDMaCAO framework has access to each individual Compute node to manage the lifecycle of each modeled component placed on them. Moreover, in case of the WordPress example, this network also connects the Compute nodes computeWww and computeDb to each other, while in the Node Cellar topology the MongoHost and the NodejsHost are linked. Also in the Multi-Tier-Deployment the individual Compute nodes are linked to each other. Thus, in each use case the infrastructure required to connect the web server component of (WordPress and Nodecellar) to its database component (mysql and Mongod) is present, as well the connection between the ELK components. It should be noted, that instead of the management network a designated network may be modeled that connects the individual Compute nodes. In addition to the management network, the transformation adds general information to the model, e.g., default SSH keys, user data, flavor and images to be used by the VMs to be spawned, which eases the modeling process.

```yaml
topology_template:
  description: A Multi-Tier application with ELK stack
  node_templates:
    nodejs_host: # omitted for brevity
    nodejs: # omitted for brevity
    nodecellar: # omitted for brevity
    mongo_db:
      type: tosca.nodes.Database
      requirements:
        - host: mongo_dbms
      interfaces: # omitted for brevity
    mongo_dbms:
      type: tosca.nodes.DBMS
      requirements:
        - host: mongo_server
      interfaces: # omitted for brevity
    elasticsearch:
      type: tosca.nodes.SoftwareComponent.Elasticsearch
      requirements:
        - host: elasticsearch_server
      interfaces: # omitted for brevity
    logstash:
      type: tosca.nodes.SoftwareComponent.Logstash
      requirements:
        - host: logstash_server
        - search_endpoint: elasticsearch
      interfaces: # omitted for brevity
    kibana:
      type: tosca.nodes.SoftwareComponent.Kibana
      requirements:
        - host: kibana_server
        - search_endpoint: elasticsearch
      interfaces: # omitted for brevity
    app_collectd:
      type: tosca.nodes.SoftwareComponent.Collectd
      requirements:
        - host: app_server
        - collectd_endpoint: logstash
      interfaces: # omitted for brevity
    app_rsyslog:
      type: tosca.nodes.SoftwareComponent.Rsyslog
      requirements:
        - host: app_server
        - rsyslog_endpoint: logstash
      interfaces: # omitted for brevity
    app_server:
      type: tosca.nodes.Compute
      capabilities: # omitted for brevity
    mongo_server:
      type: tosca.nodes.Compute
      capabilities: # omitted for brevity
    elasticsearch_server:
      type: tosca.nodes.Compute
      capabilities: # omitted for brevity
    logstash_server:
      type: tosca.nodes.Compute
      capabilities: # omitted for brevity
    kibana_server:
      type: tosca.nodes.Compute
      capabilities: # omitted for brevity
```

Fig. 13 Multi-Tier YAML Topology.

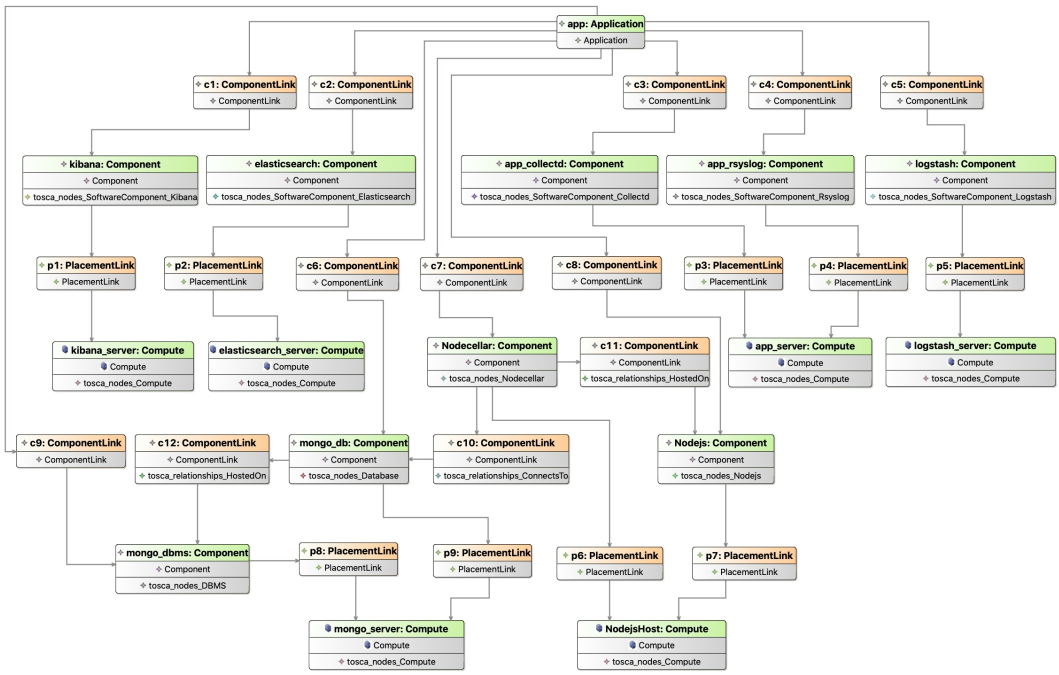

Fig. 14 Multi-Tier Configuration.

After the transformation, the current cloud deployment is extracted in form of an OCCI model from the OCCIware runtime. Based on the current and desired topology, a provisioning plan is generated [17] that sequences the OCCI requests required to provision and deploy the depicted model. Hereby, the requests are sequenced in such a manner that each Resource is provisioned first, i.e., Compute, Network, Application, and Component. Thereafter, link requests are performed connecting the individual resources with each other. While resources of the platform layer can be immediately linked, Compute nodes have to be in an active state before they can be connected to networks or storage. These states, amongst others, are reflected in the runtime model and used by the orchestration process. Once the infrastructure has been completely provisioned, i.e., every Compute node being active and connected to the management network, the modeled Applications are deployed. At this point in time, each application including its components are in an undeployed state. Then, depending on the use case, the orchestration process triggers the start action on the WordPress, Multi-Tier or Node Cellar application. This lifecycle management action is implemented by the MoDMaCAO framework and triggers the execution of a set of configuration management scripts using the management network provided by the PIM to PSM transformation. Within these scripts, it is described how to deploy, configure, and start the individual configuration management scripts of the WordPress, and Node Cellar components. In case of the WordPress use case this comprises WORDPRESS, PHP, APACHE, and MYSQL, while in the Node Cellar use case, configuration management scripts describing the management of MONGOD, NODECELLAR and NODEJS are used. Additionally, in the Multi-Tier use case, the components ELASTICSEARCH, LOGSTASH, and KIBANA are employed. Within this deployment process, the installation and execution dependency are respected as modeled within the OCCI model describing the dependencies of the individual components to be deployed, and thus the order in which they get started. A visual documentation of the deployment process of each use case is available in the GitHub repository[32].

6 Lessons Learned

Based on our experience with TOSCA and OCCI, we identified two major feedback:

Compatibility between TOSCA & OCCI. Relying a cloud solution on standards is quite advantageous since the latter result of a collective agreement, which means they are accepted in the community, and they also are good in defining the key actions and characteristics of cloud providers. With the implementation of TOSCA Studio and our three case studies we have successfully demonstrated that both standards can be used orthogonally to implement a model-driven cloud orchestration process. We have seen that both provide a similar extension strategy, which can be exploited to achieve their compatibility. The two standards have a different focus: TOSCA provides higher level concepts such as the grouping of elements, the definition of policies, and capabilities and requirements, while OCCI provides concepts that mimic runtime behavior, e.g., Mixins that allow to adapt model elements at runtime and a uniform API that allows to create the defined elements on the target cloud infrastructure. TOSCA provides a richer set of modeling elements, while OCCI is built around a core model which is easier to understand and extend. In this work, we have successfully demonstrated that the two standards can complement each other, using the strength of TOSCA at design time to model cloud applications and the strengths of OCCI to actually render API calls from the model to actually provision and deploy the defined cloud resources in a cloud environment.

Model-driven design and orchestration of existing cloud applications. Using MDE principles, we provided TOSCA Studio, a complete standard-based framework for modeling cloud applications as resources and then concretely provisioning these resources from the cloud. For this, we exploited several assets of MDE such as *model transformation* when we map TOSCA to OCCI and when we transform the PIM to PSM, *model verification* when we define structural constraints on TOSCA Extension, *tooling* when we provide TOSCA Studio to have a graphical support of the configurations, and *artifacts generation* when we generate scripts that provision the necessary resources from the cloud. The cherry on the top is the ability of our approach to reuse existing TOSCA topologies and seamlessly ensure their deployment using OCCI API, without any required changes. This does prove the compatibility of our approach with TOSCA and OCCI. This framework was successfully tested on three existing applications WordPress, Node Cellar and Multi-Tier. We believe it can handle every existing TOSCA topology, even it may require sometimes to enrich TOSCA Extension by adding new TOSCA types.

[32] `https://github.com/occiware/TOSCA-Studio`

7 Related Work

Besides TOSCA, several other orchestration template formats exist, which have been developed by different cloud providers or communities, e.g., OpenStacks Heat Orchestration Template Language[33] and the Amazon's CloudFormation template format[34]. They are not considered in this paper, since our focus is on interoperability of TOSCA and OCCI. We detail in the following the state-of-the-art of the works around TOSCA and those around OCCI, as well as the works that tackle the integration of standards.

Around TOSCA Andrikopoulos et al. define the GEneralized Topology Language (GENTL) [23] with the aim to provide a generic modeling language that can easily be mapped to other concrete, e.g., provider-specific modeling languages that subsequently allow for automated provisioning of the defined resources including TOSCA. They use this language to support the cost-efficient design of application distribution across different cloud provider offerings [24]. Also Wurster et al. [25] propose an Essential Deployment Metamodel (EDMM) which is inspired by TOSCA to provide a generic language for declarative cloud deployment models. Cloudify[35] is an open source orchestration and management framework for cloud applications lifecycle. It is also based on TOSCA and provides a commercial Web Interface that enables the developer to create deployments and execute workflows. Furthermore, web based modeling tools for TOSCA like Winery [26] exist that visualizes topology models using the Vino4TOSCA language [15] that can be provisioned and deployed using OpenTOSCA [27]. Within OpenTOSCA, a uniform interface is defined providing an invocation mechanism for management operations offered by node types [28]. While this approach addresses the issue of handling a multitude of proprietary interfaces in TOSCA, OCCI provides a uniform interface by design. Hirmer et al. [29] proposed an approach that completes automatically partial TOSCA topologies in order to make them deployable. The goal is to let the user to be focus on the business-logic and not on technical details. The automatic completion of an uncomplete TOSCA topology is done in two steps: First, it fullfills the requirements. Then it checks the completness of the topology by trying to automatically provision it. If it is not provisionable, the TOSCA runtime returns a TOSCA topology with missing templates. These missing templates are added to the topology and the first step is repeated, since there are new requirements. Brabra et al. [30] propose a model-driven approach based on TOSCA to design resource-related artifacts regardless of specific DevOps tools. The approach enables a new model-driven translation technique that serves to translate the designed artifacts using TOSCA into DevOps specific artifacts and provides connectors that intend to establish the bridge between DevOps-specific artifacts and the DevOps tools. While a multitude of approaches are based on TOSCA, non of the named approaches consider a connection to OCCI that allows models to be executable inside a Models@run.time interpreter framework. With the Eclipse Incubation Project *Cloud Application Management Framework* (CAMF) [31], Loulloudes et al. attempt to build a whole IDE to manage cloud applications with the help of TOSCA. In the scope of the project different adapters have been developed to deploy the defined TOSCA topology on multiple clouds. However, no model-driven mapping and interaction with OCCI is provided. Regarding the modeling of cloud applications, several extensions to UML have been developed to capture cloud application specifics, e.g., [32], [33], [34]. In addition, Bergmayr et al. [35] show how to convert refined UML models to TOSCA templates. Their approach is also based on an Ecore metamodel generated from the TOSCA XSD. These works consider the modeling of cloud applications, but do not take the mapping to certain API calls into account.

Around OCCI A metamodel for OCCI was defined with help of EMF by [4], and enhanced by [6], to provide a common basis for the generation and conformance testing of OCCI tools. This metamodel is used by [18] to model the deployment of applications with help of containers. It is also applied to define a unified metamodel to manage elasticity in the cloud [36]. These works have been published in scope of the OCCIware[36] project, that aims to provide a fully integrated IDE to support the whole cloud application management life cycle on multiple clouds based on OCCI. Apart from the OCCIware project, approaches exist that utilize OCCI which, e.g, focus on the PaaS layer [37]. Still, interoperability with TOSCA is not considered.

Around standards integration Carrasco et al [38] aim at improving the management and the deployment of multi-cloud applications by combining two standards: TOSCA and CAMP. Their approach is based on model transformations and allows users to describe their cloud applications according to the distribution of modules and deploy these modules over different clouds. Their

[33] `https://wiki.openstack.org/wiki/Heat`

[34] `https://aws.amazon.com/cloudformation`

[35] `https://cloudify.co/`

[36] `http://occiware.org`

approach is done in two phases: describing the multi-cloud application using Winery and then they transform the TOSCA topology into a CAMP-compliant YAML file in order to deploy the application. Later on, Carrasco et al. [39] present an automated approach for the migration of cloud applications components. This approach relies on the trans-cloud framework [40], which is based on the TOSCA topology descriptions and the API of Cloud Application Management for Platforms (CAMP) standard [41]. The main difference between TOSCA-Studio and their approach is the used standards. Similar to OCCI, CAMP provides a common API for managing cloud providers. However, CAMP targets the deployment of cloud applications on top of PaaS resources, whereas OCCI is suitable for provisioning IaaS, PaaS and SaaS resources. In contrast to [39], we focus on the deployment and runtime reconfiguration of cloud applications and not on the migration of these applications. Moreover, we propose a resource-based approach, whereas Carrasco et al. propose a component-based approach.

8 Conclusion

Many cloud standards have emerged to cope with the diversity of cloud providers and the heterogeneity encountered in the cloud ecosystem. These standards have different focus work at different levels. In this article, we argued that TOSCA and OCCI standards are complementary and we presented an approach to combine TOSCA and OCCI for model and standard driven cloud orchestration. We defined an exhaustive and automated mapping between the metamodel elements of TOSCA and OCCI and we adopted this mapping for generating a model for TOSCA that conforms to the OCCIware metamodel (TOSCA Extension). We also proposed TOSCA Studio, a dedicated model-driven environment for designing applications with TOSCA using TOSCA Designer, and for deploying these modeled applications in production environments and adapting them at runtime using OCCI Orchestrator. Furthermore, we used this approach to support the adaptation of models at runtime to keep the model of the infrastructure and the application deployment consistent with its actual state in the cloud. This will also allow us to react to changes in the model or in the cloud. We also provided three feasibility studies and showed how WordPress, Node Cellar and Multi-Tier applications can be modeled and concretely deployed using our approach.

For future work, we plan to support deployment on PaaS by modifying the OCCI orchestrator in order to make requests on a PaaS provider such as Force.com or Cloud Foundry, and to connect to existing SaaS offerings. We also aim to provide a formal verification of TOSCA Extension by using formal specification languages such as Alloy [42]. Alloy allows to specify TOSCA Extension using first order-logic and to reason about this specification in order to verify desired properties [43]. Moreover, by adopting a model-driven approach and automation in our mapping process, it is possible to incorporate changes to both evolving standards and to provide an extensible playground for new concepts. Hence, we aim to extend our catalog of transformation rules by continuously parsing new emerging TOSCA types and adding them to TOSCA Extension. We also aim to support more automated transformation of predefined TOSCA topologies into OCCI configurations. Finally, we plan to conduct a round-trip validation of the deployed application against the designed model, i.e., the configuration.

Availability Readers can find TOSCA Studio including TOSCA Extension, the model-driven designer and orchestrator at: `https://github.com/occiware/TOSCA-Studio`.

Acknowledgements This work is supported by the OCCIware research and development project funded by French Programme d'Investissements d'Avenir (PIA). We also thank the Simulationswissenschaftliches Zentrum Clausthal-Göttingen (SWZ) for financial support.

References

1. Ralf Nyrén, Andy Edmonds, Alexander Papaspyrou, Thijs Metsch, and Boris Parák. Open Cloud Computing Interface - Core, September 2016. [Available online: `http://ogf.org/documents/GFD.221.pdf`].
2. Gerd Breiter, Michael Behrendt, M Gupta, Simon Daniel Moser, R Schulze, I Sippli, and Thomas Spatzier. Software Defined Environments based on TOSCA in IBM Cloud Implementations. *IBM Journal of Research and Development*, 58(2/3):9–1, 2014.
3. Fabian Glaser, Johannes Martin Erbel, and Jens Grabowski. Model Driven Cloud Orchestration by Combining TOSCA and OCCI. In *7th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 644–650. SciTePress, 2017.
4. Philippe Merle, Olivier Barais, Jean Parpaillon, Noël Plouzeau, and Samir Tata. A Precise Metamodel for Open Cloud Computing Interface. In *8th IEEE International Conference on Cloud Computing (CLOUD)*, pages 852–859. IEEE, 2015.
5. Faiez Zalila, Stéphanie Challita, and Philippe Merle. A Model-Driven Tool Chain for OCCI. In *25th International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS)*, pages 389–409. Springer, Cham, 2017.
6. Faiez Zalila, Stéphanie Challita, and Philippe Merle. Model-driven Cloud Resource Management with OCCIware. *Future Generation Computer Systems*, 99:260–277, 2019.
7. Spencer Rugaber and Kurt Stirewalt. Model-Driven Reverse Engineering. *IEEE software*, 21(4):45–53, 2004.

8. Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run.time. *Computer*, 42(10), 2009.
9. OMG. MDA Guide rev. 2.0, 2014. OMG Document ormsc/2014-06-01 [Available Online: `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf`].
10. OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) 1.2, December 2017. [Available online: `http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/csprd01/TOSCA-Simple-Profile-YAML-v1.2-csprd01.pdf`].
11. OASIS. TOSCA Simple Profile in YAML Version 1.0, February 2016. [Available online: `http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html`].
12. Ralf Nyrén, Andy Edmonds, Alexander Papaspyrou, Thijs Metsch, and Boris Parák. Open Cloud Computing Interface - Core. Specification Document GFD.221, Open Grid Forum, February 2016.
13. Thijs Metsch, Andy Edmonds, and Boris Parák. Open Cloud Computing Interface - Infrastructure, September 2016. [Available online: `http://ogf.org/documents/GFD.224.pdf`].
14. Ralf Nyrén, Andy Edmonds, Thijs Metsch, and Boris Parák. Open Cloud Computing Interface - HTTP Protocol, September 2016. [Available online: `http://ogf.org/documents/GFD.223.pdf`].
15. Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and David Schumm. Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 416–424. Springer, 2012.
16. Jos B Warmer and Anneke G Kleppe. *The Object Constraint Language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.
17. Johannes Erbel, Fabian Korte, and Jens Grabowski. Comparison and Runtime Adaptation of Cloud Application Topologies based on OCCI. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, 2018.
18. Fawaz Paraiso, Stéphanie Challita, Yahya Al-Dhuraibi, and Philippe Merle. Model-Driven Management of Docker Containers. In *9th IEEE International Conference on Cloud Computing (CLOUD)*, pages 718–725. IEEE, 2016.
19. Uwe Breitenbücher, Tobias Binz, Kalman Kepes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA. In *IC2E*, pages 87–96. IEEE Computer Society, 2014.
20. Maksym Lushpenko, Nicolas Ferry, Hui Song, Franck Chauvel, and Arnor Solberg. Using Adaptation Plans to Control the Behavior at Runtime. In Nelly Bencomo, Sebastian Götz, and Hui Song, editors, *CEUR Workshop Proceedings*, volume 1474. CEUR, 2015.
21. U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger. Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA. In *2014 IEEE International Conference on Cloud Engineering*, pages 87–96, 2014.
22. Fabian Korte, Stéphanie Challita, Faiez Zalila, Philippe Merle, and Jens Grabowski. Model-Driven Configuration Management of Cloud Applications with OCCI. In *8th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 100–111, 2018.
23. Vasilios Andrikopoulos, Anja Reuter, Santiago Gómez Sáez, and Frank Leymann. A GENTL Approach for Cloud Application Topologies. In *European Conference on Service-Oriented and Cloud Computing*, pages 148–159. Springer, 2014.
24. Vasilios Andrikopoulos, Anja Reuter, Mingzhu Xiu, and Frank Leymann. Design Support for Cost-Efficient Application Distribution in the Cloud. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 697–704. IEEE, 2014.
25. Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, Frank Leymann, Karoline Saatkamp, and Jacopo Soldani. The Essential Deployment Metamodel: a Systematic Review of Deployment Automation Technologies. *SICS Software-Intensive Cyber-Physical Systems*, pages 1–13, 2019.
26. Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. Winery–a modeling tool for tosca-based cloud applications. In *International Conference on Service-Oriented Computing*, pages 700–704. Springer, 2013.
27. Uwe Breitenbücherand Christian Endresand Kálmán Képes, Oliver Kopp, Frank Leymann, Sebastian Wagner, and Johannes Wettingerand Michael Zimmermann. The OpenTOSCA Ecosystem –Concepts & Tools. *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges -Volume 1: EPS Rome 2016*, pages 112–130, 2016.
28. Johannes Wettinger, Tobias Binz, Uwe Breitenbücher, Oliver Kopp, Frank Leymann, and Michael Zimmermann. Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA. In *CLOSER*, pages 559–568, 2014.
29. Pascal Hirmer, Uwe Breitenbücher, Tobias Binz, Frank Leymann, et al. Automatic Topology Completion of TOSCA-based Cloud Applications. In *GI-Jahrestagung*, pages 247–258, 2014.
30. H. Brabra, A. Mtibaa, W. Gaaloul, B. Benatallah, and F. Gargouri. Model-driven orchestration for cloud resources. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 422–429, 2019.
31. N. Loulloudes, C. Sofokleous, D. Trihinas, M. D. Dikaiakos, and G. Pallis. Enabling Interoperable Cloud Application Management through an Open Source Ecosystem. *IEEE Internet Computing*, 19(3):54–59, May 2015.
32. Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. In *3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 56–65, 2014.
33. Ali Kamali, Soheil Mohammadi, and Ahmad Abdollahzadeh Barforoush. UCC: UML profile to cloud computing modeling: Using stereotypes and tag values. In *7th International Symposium on Telecommunications (IST)*, pages 689–694. IEEE, 2014.
34. Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A UML Profile for Modeling Multicloud Applications. In *Service-Oriented and Cloud Computing*, pages 180–187. Springer, 2013.
35. Alexander Bergmayr, Uwe Breitenbücher, Oliver Kopp, Manuel Wimmer, Gerti Kappel, and Frank Leymann. From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA. In *6th International Conference on Cloud Computing and Services Science (CLOSER)*, 2016.
36. Y. Al-Dhuraibi, F. Zalila, N. Djarallah, and P. Merle. Model-driven elasticity management with occi. *IEEE Transactions on Cloud Computing*, pages 1–1, 2019.
37. Sami Yangui and Samir Tata. An OCCI Compliant Model for PaaS Resources Description and Provisioning. *The Computer Journal*, 59(3):308–324, 2014.
38. Jose Carrasco, Javier Cubo, and Ernesto Pimentel. Towards a Flexible Deployment of Multi-Cloud Applications Based on

TOSCA and CAMP. In *European Conference on Service-Oriented and Cloud Computing*, pages 278–286. Springer, 2014.
39. Jose Carrasco, Javier Cubo, Ernesto Pimentel, and Francisco Durán. Deployment over Heterogeneous Clouds with TOSCA and CAMP. In *CLOSER (1)*, pages 170–177, 2016.
40. Jose Carrasco, Francisco Durán, and Ernesto Pimentel. Trans-cloud: CAMP/TOSCA-based Bidimensional Cross-Cloud. *Computer Standards & Interfaces*, 58:167–179, 2018.
41. OASIS. Cloud Application Management for Platforms (CAMP) 1.1, November 2014. [Available online: `http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.pdf`].
42. Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
43. Stéphanie Challita, Faiez Zalila, and Philippe Merle. Specifying Semantic Interoperability between Heterogeneous Cloud Resources with the FCLOUDS Formal Language. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 367–374. IEEE, 2018.