Model-Based Cloud Resource Provisioning 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 designing and deploying two cloud applications: WordPress\footnote{https://wordpress.org} and Node Cellar\footnote{http://nodecellar.coenraets.org}.

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

1 Introduction

With the growth of cloud computing, plenty of proprietary cloud APIs have emerged which made it hard for cloud customers 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) \footnote{https://www.ogf.org/ogf/doku.php/start} thereby aims to provide a standardized managing interface, enabling the customer to manage cloud resources. The Open Cloud Computing Interface (OCCI) \footnote{http://www.openstack.org} developed by the Open Grid Forum (OGF) \footnote{http://opennebula.org} 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\footnote{http://www.openstack.org}, OpenNebula\footnote{http://opennebula.org} and CloudStack\footnote{https://cloudstack.apache.org}.

At a higher level of abstraction, the Organization for the Advancement of Structured Information Standards (OASIS)\footnote{https://www.oasis-open.org} developed the Topology and Orchestration Specification for Cloud Applications (TOSCA)\footnote{https://www.oasis-open.org/tc شيء} which 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 designing and deploying two cloud applications: WordPress\footnote{https://wordpress.org} and Node Cellar\footnote{http://nodecellar.coenraets.org}.

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

1 Introduction

With the growth of cloud computing, plenty of proprietary cloud APIs have emerged which made it hard for cloud customers 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) \footnote{https://www.ogf.org/ogf/doku.php/start} thereby aims to provide a standardized managing interface, enabling the customer to manage cloud resources. The Open Cloud Computing Interface (OCCI) \footnote{http://www.openstack.org} developed by the Open Grid Forum (OGF) \footnote{http://opennebula.org} 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\footnote{http://www.openstack.org}, OpenNebula\footnote{http://opennebula.org} and CloudStack\footnote{https://cloudstack.apache.org}.

At a higher level of abstraction, the Organization for the Advancement of Structured Information Standards (OASIS)\footnote{https://www.oasis-open.org} developed the Topology and Orchestration Specification for Cloud Applications (TOSCA)\footnote{https://www.oasis-open.org/tc شيء} which 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 designing and deploying two cloud applications: WordPress\footnote{https://wordpress.org} and Node Cellar\footnote{http://nodecellar.coenraets.org}.

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

1 Introduction

With the growth of cloud computing, plenty of proprietary cloud APIs have emerged which made it hard for cloud customers 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) \footnote{https://www.ogf.org/ogf/doku.php/start} thereby aims to provide a standardized managing interface, enabling the customer to manage cloud resources. The Open Cloud Computing Interface (OCCI) \footnote{http://www.openstack.org} developed by the Open Grid Forum (OGF) \footnote{http://opennebula.org} 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\footnote{http://www.openstack.org}, OpenNebula\footnote{http://opennebula.org} and CloudStack\footnote{https://cloudstack.apache.org}.

At a higher level of abstraction, the Organization for the Advancement of Structured Information Standards (OASIS)\footnote{https://www.oasis-open.org} developed the Topology and Orchestration Specification for Cloud Applications (TOSCA)\footnote{https://www.oasis-open.org/TC_OrchSpec} which 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 designing and deploying two cloud applications: WordPress\footnote{https://wordpress.org} and Node Cellar\footnote{http://nodecellar.coenraets.org}.

1 Introduction

With the growth of cloud computing, plenty of proprietary cloud APIs have emerged which made it hard for cloud customers 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) \footnote{https://www.ogf.org/ogf/doku.php/start} thereby aims to provide a standardized managing interface, enabling the customer to manage cloud resources. The Open Cloud Computing Interface (OCCI) \footnote{http://www.openstack.org} developed by the Open Grid Forum (OGF) \footnote{http://opennebula.org} 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\footnote{http://www.openstack.org}, OpenNebula\footnote{http://opennebula.org} and CloudStack\footnote{https://cloudstack.apache.org}.

1 Introduction

With the growth of cloud computing, plenty of proprietary cloud APIs have emerged which made it hard for cloud customers 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) \footnote{https://www.ogf.org/ogf/doku.php/start} thereby aims to provide a standardized managing interface, enabling the customer to manage cloud resources. The Open Cloud Computing Interface (OCCI) \footnote{http://www.openstack.org} developed by the Open Grid Forum (OGF) \footnote{http://opennebula.org} 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\footnote{http://www.openstack.org}, OpenNebula\footnote{http://opennebula.org} and CloudStack\footnote{https://cloudstack.apache.org}.
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 IaaS clouds. TOSCA has been initially published in 2013 and major industrial cloud providers such as IBM Cloud are supporting it \[7\]. In contrast to OCCI, TOSCA does not define how the topologies are programatically 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 \[13\]. In this article, we extend this mapping to support a complete coverage of both standards and we concretely implement our approach based on \textit{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 model, called \textit{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 written in Ecore. In fact, the OCCIware approach \[34\] 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.

Cloud models play an important role to capture the expectations of a cloud API and to a priori validate the correctness of its cloud configurations. These models are manually designed so far, which is prohibitively labor intensive, time consuming and error-prone. To address this issue, we propose to infer a model-driven specification, i.e., TOSCA Extension, from the documentation of TOSCA written in YAML. This is a work of reverse engineering \[31\], 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 \textit{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 and manage these applications, following a \textit{models@run.time} approach \[5\]. TOSCA-Studio is publicly available online \[9\].

In other words, we propose a standard-based and model-driven orchestrator for cloud applications. We extend the features introduced in \[13\] 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.

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. Two feasibility studies 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 [29]. We introduce these metamodels in the following.

2.1 TOSCA

According to its specification [28], 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 processes that can be executed on this structure. As the time of this writing, two versions of TOSCA exist. The first is based on XML [28], and the second is based on YAML [27]. 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.

Fig. 1 TOSCA metamodel.

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[10] 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

---

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

[10] Group_templates and Group_types are currently part of the TOSCA YAML rendering, but not part of the TOSCA XML specification.
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. 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 [27] and [28].

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”. The OCCI specification comprises several parts: OCCI Core model, OCCI Extensions, OCCI Renderings and OCCI Protocols. The OCCI Core Model [25] 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 [22], 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 [23] that defines how OCCI resources can be managed over the HTTP protocol. The OGF does not provide a formal metamodel for OCCI. This gap has been addressed by the OCCIware metamodel [26], which 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 application domain, and an OCCI Configuration defines a running system. 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.

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. 3. 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 con-
cepts. 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 normative types to OCCIware metamodel

We base our mapping on the TOSCA YAML specification [27] that defines the TOSCA normative types, and on the OCCIware metamodel [34]. 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 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 min_Inclusive 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 virtual machines or application components. 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)] 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.

- **POLICY_TYPE** defines non-functional behavior or quality-of-services in TOSCA. This becomes a **MIXIN** to a **KIND RESOURCE** from the OCCIware SLA EXTENSION [17]. For example, tosca.policies.Scaling becomes a mixin that depends on the Agreement.Term mixin in OCCI, which in turn is applied to the Agreement kind resource in the SLA extension.

- **DATA_TYPE** defines complex data types of TOSCA properties. This concept matches the OCCIware RECORDTYPE concept which is used to define struc-
Table 1  TOSCA2OCCI mapping: metamodeling level

| TOSCA 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      |
| 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 |
| **Policy_type**       | Mixin applied to SLA Resource |
| policies.Update       | Mixin that depends on Agreement_term Mixin which is applied to Agreement Resource |
| policies.Placement    | Mixin that depends on Agreement_term Mixin which is applied to Agreement Resource |
| policies.Scaling      | Mixin that depends on Agreement_term Mixin which is applied to Agreement Resource |
| policies.Performance  | Mixin that depends on Agreement_term Mixin which is applied to Agreement Resource |
| **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 |

tures. 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, Policy_types, Datatype_types, Interface_types and Capability_types to OCCI mixins. However, TOSCA introduces some additional concepts, such as Group_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\textsuperscript{12} Cloudify\textsuperscript{13} CELAR\textsuperscript{14} SeaClouds\textsuperscript{15} DICER\textsuperscript{16} and INDIGO-DataCloud\textsuperscript{17} have raised. Each project has been defining new TOSCA types or modifying existing types according to its need. However, 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\textsuperscript{18} and tosca.nodes.Database\textsuperscript{19}, MySQL\textsuperscript{20} tosca.Rsyslog\textsuperscript{21} and tosca.nodes. SoftwareComponent.Rsyslog\textsuperscript{22} 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 exhaustively to OCCIware mixins, as listed in Table 2.

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 aMixinBase in an OCCI Configuration. AMixinBase 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 aMixinBase in an OCCI Configuration.

- A Group_template defines a group of nodes that share some semantics, e.g., an autoscaling group is

\textsuperscript{12} http://alien4cloud.github.io
\textsuperscript{13} https://cloudify.co
\textsuperscript{14} https://github.com/CELAR/c-Eclipse
\textsuperscript{15} https://www.seaclouds-project.eu/
\textsuperscript{16} https://github.com/dice-project/DICER
\textsuperscript{17} https://www.indigo-datacloud.eu
\textsuperscript{18} https://github.com/alien4cloud/samples/blob/master/tosca.nodes.Mysql\textsuperscript{19}
\textsuperscript{19} https://github.com/CELAR/c-Eclipse
\textsuperscript{20} https://cloudify.co
\textsuperscript{21} https://github.com/alien4cloud/samples/blob/master/tosca.Rsyslog
\textsuperscript{22} https://github.com/alien4cloud/samples/blob/master/tosca.nodes.Database
\textsuperscript{23} https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/
\textsuperscript{24} https://github.com/openstack/tosca-parser/blob/master/toscaparser/tests/data/custom_types/nested_rsyslog.yaml
\textsuperscript{25} https://github.com/openstack/tosca-parser/blob/master/toscaparser/tests/data/custom_types/MySQL
\textsuperscript{26} http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/
\textsuperscript{27} https://github.com/openstack/tosca-parser/blob/master/toscaparser/tests/data/custom_types/MySQL
Table 2: 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.ComputeWithAttrList | Mixin that depends on nodes.Compute Mixin |
| nodes.ComputeWithCapWithAttr | Mixin that depends on nodes.Compute Mixin |
| nodes.SoftwareComponent.Collectd | Mixin that depends on nodes.SoftwareComponent Mixin |
| nodes.SoftwareComponent.Logstash | Mixin that depends on nodes.SoftwareComponent Mixin |
| nodes.Docker | Mixin that depends on nodes.container.Application Mixin |
| nodes.Elasticsearch | Mixin that depends on nodes.SoftwareComponent Mixin |
| nodes.Logstash | Mixin that depends on nodes.SoftwareComponent Mixin |
| nodes.Kibana | Mixin that depends on nodes.SoftwareComponent Mixin |
| nodes.AbstractMysql | Mixin that depends on nodes.Database Mixin |
| nodes.HACompute | Mixin that depends on nodes.SoftwareComponent Mixin |
| nodes.Database.Mysql | Mixin that depends on nodes.Database Mixin |
| nodes.DatabaseWithListParam | Mixin that depends on nodes.Database Mixin |
| nodes.DBMS.MySQL | Mixin that depends on nodes.DBMS Mixin |
| nodes.DatabaseNetwork | Mixin that depends on nodes.Database Mixin |
| 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.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 |

In the design phase, as described in the overview (see Fig. 3), cloud resources are modeled according to the TOSCA Metamodel comprising the desired infrastructure and applications to be deployed. 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 [11]. In general, the concept resembles a models@run.time approach [5] which derives imperative steps from a declarative description in order to adapt the running system.

Fig. 4: Model-driven cloud orchestration process.
First, a Platform Specific Model (PSM) is generated from the (PIM) OCIIWARE Configuration that contains all information required for an actual cloud deployment. To generate a (PSM) OCIIWARE Configuration, a Model-to-Model transformation (M2M) transformation is applied that can be configured to add OCCI elements that are cloud provider specific. Amongst others this comprises OCCI Operating System and Resource templates describing available sizes and images of VMs. Moreover, this transformation adds orchestration specific cloud resources required for configuration management. Thereafter, the 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 Model-to-Model transformation (M2M) is triggered generating a Provisioning and Deployment Plan (see e.g., [11], [6] 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 generated from the elements contained within the (PSM) OCIIWARE 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 will exemplify the proposed orchestration process in Section 4.

4 Implementation: TOSCA Studio

Our approach for managing cloud applications by relying on TOSCA topologies and the OCIIware 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 OCIIware approach. It relies on a metamodel, called TOSCA Extension, defining the static semantics for the TOSCA standard in Ecore and OCL [32] and conforming to the OCIIware Metamodel. More specifically, TOSCA Studio is implemented as a set of Eclipse plugins that are publicly available [22]. 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.

https://github.com/occiware/TOSCA-Studio

4.1 TOSCA Extension

The mapping between the original TOSCA metamodel (in YAML) and OCIIware 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. 5. 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. 5. This diagram was designed with TOSCA Studio, our open source model-driven development environment dedicated to OCCI [33,34]. TOSCA Extension is conceptually divided into three levels.

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) [18], 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. 6 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 OCCI-ware metamodel. The validation 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 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 files 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.

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

### 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. 7. 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 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_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. While the presented concept can be generally applied on any kind of OCCI API, such as the OpenStack OCCI Interface (OII) used in [11], the implementation got enhanced to focus on the OCCI API provided by the OCCIware Runtime. 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 translate incoming OCCI requests, e.g., to the API of the desired cloud provider. Thus, any kind of cloud provider can be used to provision, deploy and maintain cloud applications using the presented approach. To perform the deployment of modeled components, the MoDMaCAO framework is used, providing a connector which implements lifecycle operations for OCCI 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. 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 two case studies that represent popular distributed cloud
applications: a WordPress application and a Node Cellar application. We relied on existing TOSCA YAML topologies for WordPress and Node Cellar. 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 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, a configuration management tool, to deploy, configure, and start the modeled application components.

5.1 WordPress

WordPress is an open source Content Management System (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. 8.

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. 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. 9 shows the model of a WordPress application that corresponds to the topology in Fig. 8. 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. 8, 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 of Fig. 10.

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. 11 shows the model of a Node Cellar application that corresponds to the topology in Fig. 10. It is composed of three components (Nodecellar, Nodejs and MongoDB) deployed on two VMs (NodejsHost and MongoDB). 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. 10 have been correctly automatically set in the model.

5.3 Orchestration

Both 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 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. Thus, in both use cases the infrastructure required to connect the web server component of (WordPress and NodeCellar) to its database component (mySql and Monkey) is present. 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.

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 [11] 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 a 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 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 de-
Compatibility between TOSCA & OCCI. Based on our experience with TOSCA and OCCI, we identified two major feedback:

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 two 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 build 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 two existing applications WordPress and Node Cellar. We believe it can handle every existing TOSCA topology, even it may require sometimes to enrich TOSCA Extension by adding new TOSCA types.

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 (CAMP) standard [26]. Similar to OCCI, CAMP provides a common API for managing cloud providers. We have seen that both 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. CARRASCO ET AL. [8] present an automated approach for the migration of cloud applications components. This approach relies on the trans-cloud framework [7], which is based on the TOSCA topology descriptions and the API of Cloud Application Management for Platforms (CAMP) standard [26].
suitable for provisioning IaaS, PaaS and SaaS resources. In contrast to [3], 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. Andrikopoulos et al. define the GEneralized Topology Language (GENTL) [1] 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 [2]. Cloudify [36] 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. GENTL and Cloudify do not consider OCCI that allows models to be executable inside a Models@run.time interpreter framework. With the Eclipse Incubation Project Cloud Application Management Framework (CAMF) [10], Loulouides 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., [4], [16], [14]. In addition, Bergmayr et al. [3] 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. Ferry et al. [12] define a models at runtime approach for the management of cloud applications. Their approach is based on a modeling language called CloudML and is not based on standards.

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 two feasibility studies and showed how Word-Press and Node Cellar applications can be modeled and concretely deployed using our approach.

For future work, we aim to provide a formal verification of TOSCA Extension by using formal specification languages such as Alloy [15]. Alloy allows to specify TOSCA Extension using first order-logic and to reason about this specification in order to verify desired properties [10]. 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.

References

1. Andrikopoulos, V., Reuter, A., Sáez, S.G., Leymann, F.: A GENTL Approach for Cloud Application Topologies. In: European Conference on Service-Oriented and Cloud Computing, pp. 148–159. Springer (2014)
2. Andrikopoulos, V., Reuter, A., Xiu, M., Leymann, F.: Design Support for Cost-Efficient Application Distribu-
tion in the Cloud. In: 2014 IEEE 7th International Conference on Cloud Computing, pp. 697–704. IEEE (2014)
3. Bergmair, A., Breitenbücher, U., Kopp, O., Wimmer, M., Kappel, G., Leymann, F.: 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)
4. Bergmair, A., Troya, J., Neubauer, P., Wimmer, M., Kappel, G.: UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. In: 3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE), pp. 56–65 (2014)
5. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer 42(10) (2009)
6. Breitenbücher, U., Binz, T., Kepes, K., Kopp, O., Leymann, F., Wettig, J.: Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA. In: IC2E, pp. 87–96. IEEE Computer Society (2014)
7. Breiter, G., Behrendt, M., Gupta, M., Moser, S.D., Schulze, R., Sippi, I., Spatzier, T.: Software Defined Environments based on TOSCA in IBM Cloud Implementations. IBM Journal of Research and Development 58(2/3), 9–1 (2014)
8. Carrasco, J., Cubo, J., Pimentel, E., Durán, F.: Deployment over Heterogeneous Clouds with TOSCA and CAMP. In: CLOSER (1), pp. 170–177 (2016)
9. Carrasco, J., Durán, F., Pimentel, E.: Trans-cloud: CAMP/TOSCA-based Bidimensional Cross-Cloud. Computer Standards & Interfaces 58, 167–179 (2018)
10. Chailлина, S., Zalila, F., Merle, P.: Specifying Semantic Interoperability between Heterogeneous Cloud Resources with the FCLouds Formal Language. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 367–374. IEEE (2018)
11. Erbel, J., Korte, F., Grabowski, J.: 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)
12. Ferry, N., Brataas, G., Rossini, A., Chauvel, F., Solberg, A.: Towards Bridging the Gap Between Scalability and Elasticity. In: 4th International Conference on Cloud Computing and Services Science (CLOSER), pp. 746–751 (2014)
13. Glaser, F., Erbel, J.M., Grabowski, J.: Model Driven Cloud Orchestration by Combining TOSCA and OCCI. In: 7th International Conference on Cloud Computing and Services Science (CLOSER), pp. 644–650. SciTePress (2017)
14. Guillén, J., Miranda, J., Murillo, J.M., Canal, C.: A UML Profile for Modeling Multicloud Applications. In: Service-Oriented and Cloud Computing, pp. 180–187. Springer (2013)
15. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
16. Kamali, A., Mohammadi, S., Barforoush, A.A.: UCC: UML profile to cloud computing modeling: Using stereotypes and tag values. In: 7th International Symposium on Telecommunications (IST), pp. 689–694. IEEE (2014)
17. Katsaros, G.: Open Cloud Computing Interface – Service Level Agreements. Recommendation GFD-R-P.228, Open Grid Forum (2016)
18. Korte, F., Chailлина, S., Zalila, F., Merle, P., Grabowski, J.: Model-Driven Configuration Management of Cloud Applications with OCCI. In: 8th International Conference on Cloud Computing and Services Science (CLOSER), pp. 100–111 (2018)
19. Louloudes, N., Sofokleous, C., Trihinas, D., Dikaiakos, M.D., Fallis, G.: Enabling Interoperable Cloud Application Management through an Open Source Ecosystem. IEEE Internet Computing 19(3), 54–59 (2015). DOI 10.1109/MIC.2015.60
20. Lushpenko, M., Ferry, N., Song, H., Chauvel, F., Solberg, A.: Using Adaptation Plans to Control the Behavior at Runtime. In: N. Bencomo, S. Götz, H. Song (eds.) CEUR Workshop Proceedings, vol. 1474. CEUR (2015)
21. Merle, P., Barais, O., Parpaillon, J., Plouzeau, N., Tata, S.: A Precise Metamodel for Open Cloud Computing Interface. In: 8th IEEE International Conference on Cloud Computing (CLOUD), pp. 852–859. IEEE (2015)
22. Metsch, T., Edmonds, A., Parák, B.: Open Cloud Computing Interface - Infrastructure (2016). [Available online: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf]
23. Nyrén, R., Edmonds, A., Metsch, T., Parák, B.: Open Cloud Computing Interface - HTTP Protocol (2016). [Available online: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/csprd01/tosca/TOSCA-Simple-Profile-YAML-v1.2.csprd01.pdf]
24. Nyrén, R., Edmonds, A., Papapyyrou, A., Metsch, T., Parák, B.: Open Cloud Computing Interface - Core (2016). [Available online: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01]
25. Nyrén, R., Edmonds, A., Papapyyrou, A., Metsch, T., Parák, B.: Open Cloud Computing Interface - Core. Specification Document GFD.221, Open Grid Forum (2016)
26. OASIS: Cloud Application Management for Platforms (CAMP) 1.1 (2014). [Available online: http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.pdf]
27. OASIS: TOSCA Simple Profile in YAML Version 1.0 (2016). [Available online: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html]
28. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) 1.2 (2017). [Available online: http://docs.oasis-open.org/tosca/tosca/TOSCA-Simple-Profile-YAML/v1.2/csprd01/TOSCA-Simple-Profile-YAML-v1.2.csprd01.pdf]
29. OMG: MDA Guide rev. 2.0 (2014). OMG Document ormg/2014-06-01 [Available Online: http://www.omg.org/cgi-bin/doc?ormg/2014-06-01]
30. Paraiso, F., Chailлина, S., Al-Dhuraibi, Y., Merle, P.: Model-Driven Management of Docker Containers. In: 9th IEEE International Conference on Cloud Computing (CLOUD), pp. 718–725. IEEE (2016)
31. Rugaber, S., Stirewalt, K.: Model-Driven Reverse Engineering. IEEE software 21(4), 45–53 (2004)
32. Warner, J.B., Kleppe, A.G.: The Object Constraint Language: getting your models ready for MDA. Addison-Wesley Professional (2003)
33. Zalila, F., Chailлина, S., Merle, P.: A Model-Driven Tool Chain for OCCI. In: 25th International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS), pp. 389–409. Springer, Cham (2017)
34. Zalila, F., Chailлина, S., Merle, P.: Model-driven Cloud Resource Management with OCCIware. Future Generation Computer Systems 99, 260–277 (2019)