# MoDMaCAO: a model-driven framework for the design, validation and configuration management of cloud applications based on OCCI

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

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

# MoDMaCAO: A model-driven framework for the design, validation and configuration management of cloud applications based on OCCI

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

the date of receipt and acceptance should be inserted later

**Keywords** Cloud Computing, Open Cloud Computing Interface, OCCI, Models@run.time

**Abstract** To tackle the cloud-provider lock-in, the Open Grid Forum is developing the Open Cloud Computing Interface (OCCI), a standardized interface for managing any kind of cloud resources. Besides the OCCI Core model, which defines the basic modeling elements for cloud resources, further standardised extensions exist that reflect the requirements of different cloud service levels, such as infrastructure and platform elements. However, so far the OCCI platform extension is very coarse-grained and lacks supporting use cases and implementations. Especially, it does not define how the components of the application itself can be managed. In this paper, we discuss the features of MoDMaCAO, a model-driven framework that extends the OCCI platform extension. The users of the framework are able to design and validate cloud application topologies and subsequently deploy them on OCCI compliant clouds by using configuration management tools.

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

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

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

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

## 1 INTRODUCTION

With the broad proliferation of cloud computing in the industry and academia, many different cloud service providers have emerged, that offer different service levels and interfaces to the customer. This heterogeneity of cloud provider interfaces makes it hard to migrate applications between different cloud providers or combine different offerings. To tackle this problem, two different strategies can be identified in the literature: 1) the use of code libraries that provide a common software development kit for different cloud providers, e.g., Apache jclouds[1,2] or fog[3], or, 2) the resort to common standards, e.g., the *Topology and Orchestration Specification for Cloud Applications* (TOSCA)[4], and *Open Cloud Computing Interface* (OCCI) [1] integrated with model-driven techniques to decouple the cloud applications from the technical peculiarities of the different target platforms. In this context, multiple IDEs have emerged such as OCCIware [2], Cloudify[5] and Alien4Cloud[6]. In this paper, we focus on OCCI, which is developed by the *Open Grid Forum* (OGF) and aims to standardize an interface for the management of any kind of cloud resources. The OCCI standard comprises several parts, including the OCCI Core model and model extensions for the infrastructure and platform layer managing *Infrastructure-as-a-Service* (IaaS) and *Platform-as-a-Service* (PaaS) resources respectively. Several implementations and use cases for the infrastructure extension already exist, which demonstrate its feasibility. However, implementations and use cases for the platform extension are rare. This might be due to the fact that it only provides a

[1] All URLs have been last retrieved on 6th January, 2023.
[2] http://www.jclouds.org
[3] http://fog.io
[4] https://www.oasis-open.org/committees/tosca/
[5] https://cloudify.co/
[6] https://alien4cloud.github.io

very rough definition of cloud applications and their components, that does not include how these cloud applications can be configured and managed. Furthermore, it does not define how application components are connected to the hosting infrastructure, such as, which component gets deployed on which virtual machine. This kind of situation forces cloud developers to manually find the appropriate deployment plan for their applications which is a tedious task when the application has a considerable size and multiple components. To close these gaps, we introduced improvements to the OCCI platform extension to allow the deployment and management of modeled platform elements, e.g., application and components, on top of IaaS resources. We implemented and validated these extensions with the *Model-Driven Configuration Management of Cloud Applications with OCCI* (MoDMaCAO) framework [3]. The initial version of MoDMaCAO provides several improvements to the OCCI platform extension to complete its lifecycle model and allow the use of configuration management tools for managing cloud applications at runtime. In this paper, we provide an overview of the MoDMaCAO framework and extend its features in the following ways:

1. We provide visualization and design capabilities for cloud application topologies based on OCCI.
2. We integrate the definition of constraints on cloud resource types to allow for the verification of defined cloud application topologies at design time.
3. We introduce capabilities to generate configuration management artifact skeletons from the defined cloud application topologies to reduce the effort for implementing its lifecycle operations.

The remainder of this paper is structured as follows. We introduce OCCI and the OCCIware tool chain as a basis for our work in Section 2. Afterwards, in Section 3, we identify the problems we want to address and the contributions introduced in the paper. Subsequently, in Section 4, we give an overview of the MoDMaCAO framework and its extended features. Furthermore, we demonstrate how MoDMaCAO can be used to model the popular LAMP stack and a MongoDB cluster and how it integrates with configuration management tools in Section 5. Thereafter, in Section 6, we discuss our results and observations. Finally, we present related work in Section 7, and we conclude this paper and provide an overview on future work in Section 8.

## 2 BACKGROUND

In the following, we provide a brief overview of the OCCI standard, the extensions we made to the OCCI Platform extension and the general features of MoDMaCAO, our model-driven tool chain to the design, validation, and configuration management of cloud applications.

### 2.1 Open Cloud Computing Interface

The OCCI Core model [1] is composed of eight elements (grey boxes in Fig. 1). `Category` is the base type for all other classes and provides the necessary identification mechanisms. `Categories` can be uniquely identified by associated *Uniform Resource Identifier*s (URIs). They 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`. A `Kind` defines the type of a cloud entity, e.g., a compute resource, and `Mixins` define how an entity can be extended at runtime. Both have `Actions` that define which behaviours 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. In the remainder of this article we use the terms `OCCIware Extension` and `OCCIware Configuration` instead of OCCI extension and OCCI Configuration to refer to the extension and the configuration based on OCCIware metamodel.

The OCCI Core model is accompanied with several extensions. The OCCI Platform extension [4] defines the two specialized kinds of `Resource`: `Application` and `Component` and a new `Link` kind `ComponentLink` (see Fig. 2). The `Application` thereby represents the user accessible part of the overall cloud application. The `Application` itself is composed of several `Components`, that implement its functionality, e.g., through microservices. `Components` can be linked with help of `ComponentLinks` to establish a connections between them.
An `Application` or `Component` can be in the state `Active`, `Inactive` or `Error`. A transition from the `Inactive` to the `Active` state can be triggered by calling the `start` action on the specific `Application` or `Component`, and a transition from `Active` to `Inactive` can be triggered by calling the `stop` action. The `Error` state can be reached at any time, in case an error occurs in the `Application` or `Component`.

### 2.2 OCCIware Tool Chain

OCCI has been proposed as a generic model and an interface for managing any kind of cloud computing resources. However, OCCI suffers from the lack of a precise definition of its concepts and a modeling framework to model, verify, validate, document, deploy and manage OCCI artifacts. To resolve the first issue, a metamodel from OCCI, named OCCIware metamodel (see Fig. 1), has been proposed in [5] and enhanced in [6]. It defines a precise semantics of OCCI concepts and introduces, among others, two key concepts: `Extension` and `Configuration`. An OCCI `Extension` represents a specific application domain, e.g., inter-cloud networking extension [7], infrastructure extension [8], platform

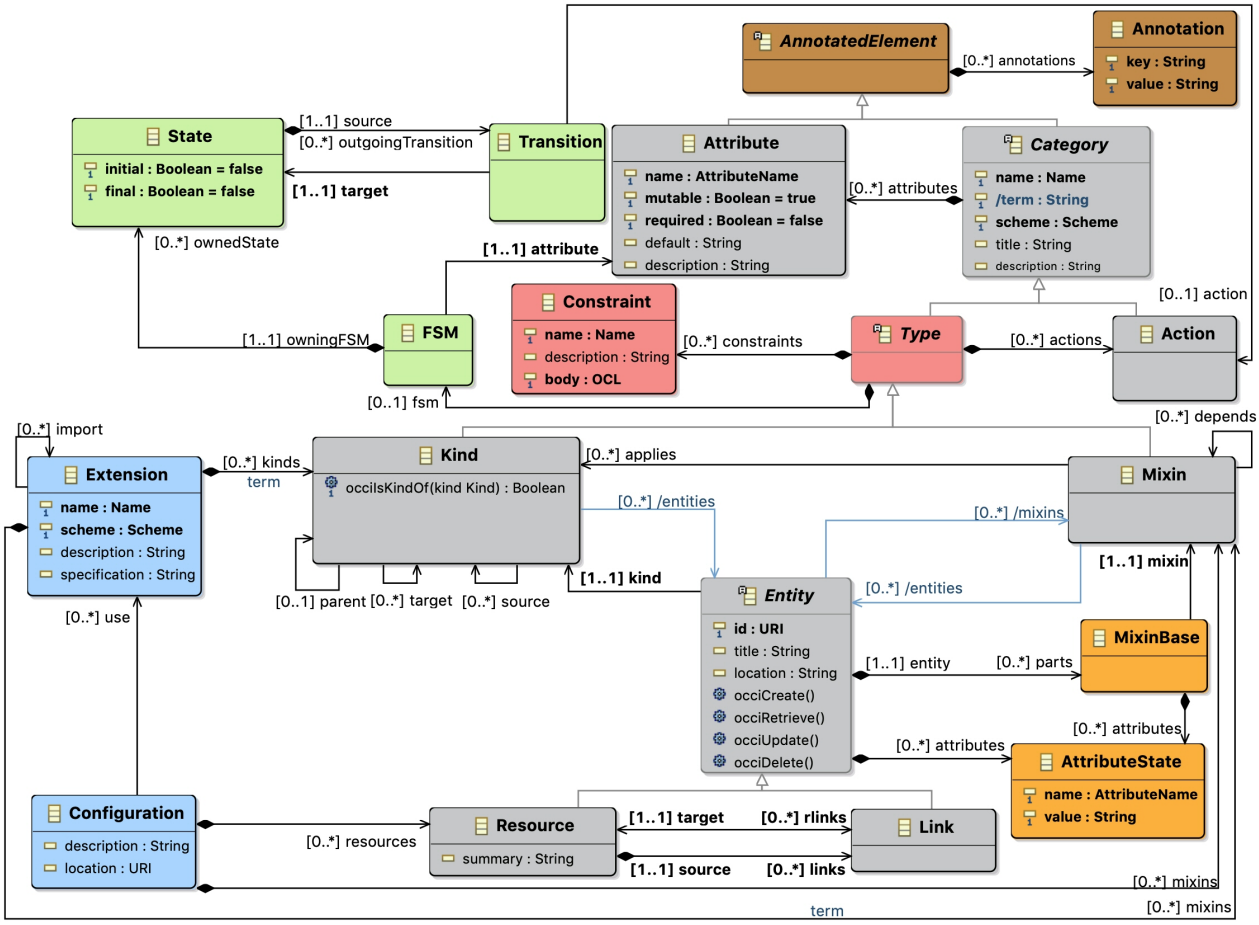


**Fig. 1** A subset of OCCIWARE METAMODEL.

extension [9, 10, 4], application extension [10], etc. An OCCI `Configuration` defines a running system. It represents an instantiation of one or several OCCI extensions. In addition, the OCCIWARE METAMODEL introduces the `Constraint` notion allowing the cloud architect to express business constraints related to each cloud computing domain. The constraints can be expressed on OCCI kinds and mixins. In addition, the OCCIWARE METAMODEL integrates the *Finite State Machine* (FSM) model. This mechanism allows to describe the behavior of each OCCI kind/mixin. Finally, `AnnotatedElement` and `Annotation` allow to design non-OCCI information to deal with non-functional needs such as visualization and documentation.

To resolve the second issue, a model-driven tool chain for OCCI, named OCCIWARE STUDIO, has been proposed [6]. It is built based on the OCCIWARE METAMODEL and proposed as a set of plugins for the Eclipse IDE. OCCIWARE STUDIO allows both cloud architects and users to encode OCCI extensions and configurations, respectively, graphically via the **OCCI Designer** tool, and textually via the **OCCI Editor** tool. They can also automatically verify the consistency of these extensions and configurations via the **OCCI Validator** tool. In addition, OCCIWARE STUDIO provides a tool, named **Connector Generator**, that generates the Java code associated to an OCCI extension. This connector code must be completed by cloud developers to implement concretely how OCCI CRUD operations and actions must be executed on a real cloud infrastructure. Later on, this generated connector is deployed on the OCCIware Runtime[7].

## 3 PROBLEM STATEMENT

As stated above, there are several use cases and implementations of the OCCI Infrastructure extension available, while the OCCI Platform extension has not reached a widespread adoption yet. We identify the following reasons for this situation:

– **No precise modeling framework for OCCI (P1)**: The current version of OCCI lacks of formality and concepts

[7] `https://github.com/occiware/MartServer`

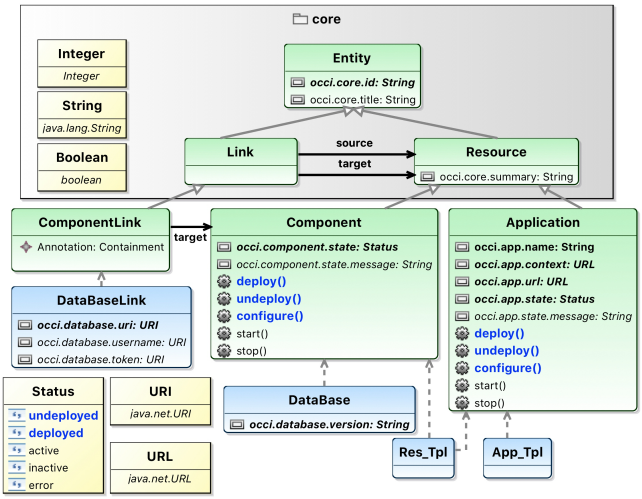


**Fig. 2** Enhanced OCCI Platform kinds.

for the design of cloud applications. Subsequently, no tooling is available that allows to graphically design cloud applications, their lifecycles, and their underlying infrastructure based on OCCI models. One issue is that no connection between infrastructure and platform models is defined. In fact, the OGF provides two separate OCCI extensions for the infrastructure and platform layers, but it misses to define the connection between them. According to the specification it is hence not possible to connect a `Component` or `Application` to a `Compute` resource of the OCCI Infrastructure extension. In addition, a generic interface is currently missing from the standard that allows to couple state of the art configuration management tools with OCCI. Moreover, the lifecycle for the `Component` and `Application` resources as defined in the OCCI specification is incomplete. `Components` can either be inactive or active, but the specification does not allow to model information about the installation or configuration states. This incomplete lifecycle information inhibits cloud developers to finely observe the execution of their cloud resource.

– **Lack of verification for designed cloud applications (P2)**: Currently, the only manner to be sure that a cloud application will run correctly is to provision and deploy it in the cloud. Thus, when errors occur, a correction is made and the deployment task must be repeated several times before the application becomes operational. This process is cumbersome and supporting tooling is necessary to spot errors as early as possible.
– **Lack of IDEs for Infrastructure as Code (P3)**: With cloud orchestration and configuration management tools it is possible to encode the configuration of whole datacenters inside reusable artifacts. Thereby, lightweight and human readable serialization formats based on YAML or JSON are commonly used. However, there is a lack of supportive tooling to create and edit these artifacts.

To overcome **P1**, we proposed the MoDMaCAO framework [3]. The MoDMaCAO framework is based on an improved version of the OCCI Platform extension which provides an extended lifecyle model. Furthermore, we introduced a connection between both OCCI Infrastructure and Platform extensions and offered an integration mechanism for configuration management tools. By using a generic interface, several configuration management tools can be coupled with the MoDMaCAO framework. Hereby, we make use of model-driven techniques to support the development and runtime management of cloud applications. For example, configuration management script skeletons and variable files that reflect information about the runtime state of the cloud can be generated. To further improve our solution for **P1** and provide solutions for **P2** and **P3**, we extend the MoDMaCAO framework in the following way:

– **Deployment model visualization and design (C1)**: We extend the MoDMaCAO modeling framework by adding facilities to generate customized graphical designers for defined cloud resource types. In addition, we revisit the different runtime elements (actions, states, and state machines) by completing the missing ones and propose a complete lifecycle for each cloud resource type (cf. **P1**).
– **Deployment model verification (C2)**: The custom cloud resource types can be annotated with constraints at design time. When designing configurations based on the defined resource types, the MoDMaCAO framework can subsequently be used to check if the defined constraints are fulfilled. This helps to already spot errors during the design process and thus addresses **P2**.
– **Configuration management artifact generation (C3)**: MoDMaCAO defines an integration mechanism for configuration management tools. We extend this approach by supporting the generation of configuration management artifact skeletons from the modeled cloud resource types which can then be further manually extended to implement the management of the resources at runtime. Thereby, we reduce the effort necessary to edit these artifacts. The generation process is integrated as part of the MoDMaCAO modeling framework and thus provides a step towards an IDE for Infrastructure as Code (cf. **P3**).

## 4 MODMACAO

In the following, we will introduce the building blocks of the MoDMaCAO framework [3] and how it addresses the shortcomings of the OCCI platform extension.


**Fig. 3** Overall Architecture.

### 4.1 Overall Architecture

The overall architecture of the proposed MoDMaCAO framework and its contributions are depicted in Fig. 3. The features, we discuss in this paper are numbered. Our first contribution (**1a**) , initially presented in [3], is to address **P1** by enhancing the OCCI Platform extension via additional lifecycle states and actions, introducing a new OCCI `Link` kind to be able to connect `Component`s of the OCCI Platform extension to `Compute` resources of the OCCI Infrastructure extension, and defining a new OCCI extension to be able to model application components that are managed with help of a configuration management tool. For this, we form a configuration management interface that is based on the introduced extensions lifecycle actions. This interface allows to plug-in state of the art implementations of current configuration management tools like Ansible with minmal effort. In this improved version of MoDMaCAO framework, we introduce the following features: At first, we propose an approach to ease the visualization of MoDMaCAO configurations. It consists in annotating the MoDMaCAO extensions with visualization annotations (for example, show a resource inside another, hide an attribute information, etc.). Then, we extended the **Designer Generator** of the OCCIware Studio (**1b**) to support these annotations and generate pre-customized graphical designers. Furthermore, we introduce a verification mechanism based on the *Object Constraint Language* (OCL)[8] to assess the wellformedness of application configurations (**2**). This feature allows to define domain-specific invariants related to a particular MoDMaCAO domain and to verify whether conforming configurations respect these invariants (addressing **P2**). Finally, we demonstrate the feasibility of the defined extension by modeling two different distributed cloud applications with MoDMaCAO and provide a framework for implementing model-driven configuration management with different configuration management tools (**3**), thereby addressing **P3**.

[8] `https://www.omg.org/spec/OCL/`

### 4.2 MoDMaCAO Modeling Framework

Experimenting with the OCCI Platform extension in real use cases shows several hidden lacks. The OCCI Platform extension provides only `inactive`, `active`, and `error` states with two actions: `start` and `stop`. This design assumes that a component is already installed and configured which might not be the case. For instance, an application component, e.g., a software component, like a database or an application server, will first be installed ("deployed"), and configured, prior to managing it (start/stop etc.). Therefore, we argue that the lifecycle of the Component and Application kinds is not expressive enough and does not define all possible states of a resource (compare **P1**). To resolve this issue, we propose an enhancement of the OCCI Platform extension as shown in Fig. 2.

The different improvements are colored in blue. We propose to add two additional states in the `Status` enumeration type: `undeployed` and `deployed`. In addition, we define three new actions for each kind: `configure`, `deploy`, and `undeploy`. Finally, we enhance the FSMs of both kinds by integrating the new provided states and actions, and adding eleven new transitions. Fig. 4 shows the enhanced FSM for `Component` and `Application` kinds. Therefore, a `Component` or `Application` resource is initially `undeployed`. Once the `deploy` action is triggered, the resource is deployed. By triggering the `configure` action, the resource is configured. We treat this configuration as a rather intermediate state which directly transfers to the `inactive` state originally defined by the standard. As a result, a distinct configured state is not covered in the FSM. Finally, a `Component` or `Application` can reach the `active` state by triggering the `start` action.

Fig. 5 depicts the definition of a new link kind named `PlacementLink` addressing the missing connection between the OCCI Platform extension and the OCCI Infrastructure extension. As a specialization of the generic `Link` kind, the `PlacementLink` provides the user with an additional constraint to restrict the selection of the source and target resources. Now, thanks to the `PlacementLink`, we can connect a `Component` resource (from the Platform extension) as its source, to a `Compute` resource (from the Infrastructure extension) as its target, and hence allows us to model the placement of an application component on a virtual machine. In addition, the `PlacementLink` type allows us to easily query the model, using the uniform and standardized

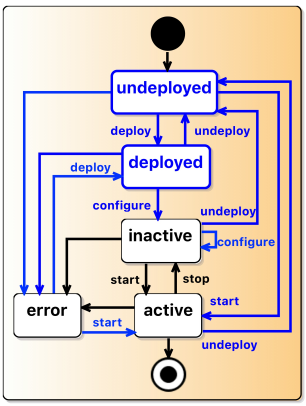


**Fig. 4** Enhanced OCCI Platform FSMs.

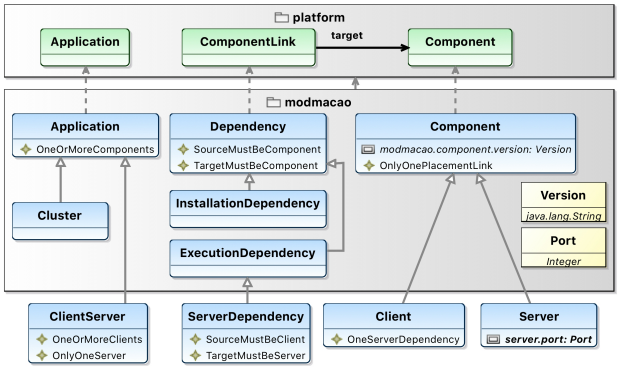


**Fig. 6** The MoDMaCAO Modeling Framework.

OCCI interface, based on the link type instead of deducing it from the type of source and target resources.

The MoDMaCAO modeling framework is based on the OCCIware tool chain presented in Section 2.2 and allows cloud architects to:

1. design abstract types modeling cloud applications and their components,
2. model configured instances of cloud applications, and,
3. check the validity of instances of cloud applications.

Firstly, as shown in Fig. 6, the MoDMaCAO modeling framework defines the following set of abstract types:

– The **Application** mixin type abstracts the notion of a cloud application. This mixin applies to OCCI Platform `Application` resources. A cloud application is composed of one or more cloud application components as enforced by the `OneOrMoreComponents` constraint. Then, modeling specific cloud applications requires to design new mixin types inheriting from `Application`, e.g., `Cluster` and `ClientServer` types. These new types could define their own attributes and constraints. For instance, a client-server application has only one server component (i.e., `OnlyOneServer` constraint) and some client components (i.e., `OneOrMoreClients` constraint).

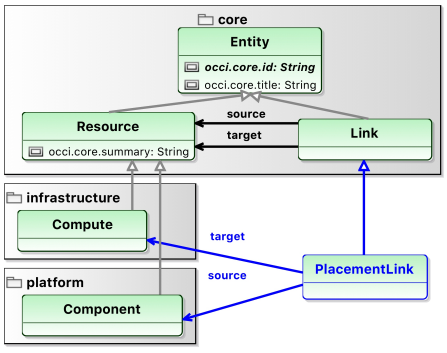


**Fig. 5** New OCCI Placement Extension.

– The **Cluster** mixin type abstracts the notion of a clustered cloud application.
– The **Component** mixin type abstracts the notion of a cloud application component. This mixin applies to OCCI Platform `Component` resources. Each component has an optional immutable `modmacao.component.version` attribute representing the version of the component used at runtime, and must be placed on only one OCCI `Compute` resource (i.e., `OnlyOnePlacementLink` constraint). Then, modeling specific cloud application components requires to define new mixin types inheriting from `Component`, e.g., `Client` and `Server` types. These new component types can define their own attributes and constraints. For instance, a server component has a network port on which it listens to client requests (i.e., `server.port` immutable attribute) and a client component must be connected to a server component (i.e., `OneServerDependency` constraint).
– The **Version** data type defines the valid string pattern for version values, i.e., `<major>.<minor>`.
– The **Port** data type defines the valid network port values, i.e., range from 0 to 65535.
– The **Dependency** mixin type abstracts the notion of a dependency between two cloud application components. This mixin applies to OCCI Platform `ComponentLink` links. Both `SourceMustBeComponent` and `TargetMustBeComponent` constraints enforce that a dependency link connects two `Component` instances. Then, modeling specific dependencies requires to define new mixin types inheriting from `Dependency`, e.g., `InstallationDependency`, `ExecutionDependency`, or `ServerDependency`. These sub-mixins can be added as needed to capture other types of dependency while defining their own attributes and constraints. For instance, `ServerDependency` defines two constraints enforcing the dependency source to be a client component and the dependency target to be a server component.

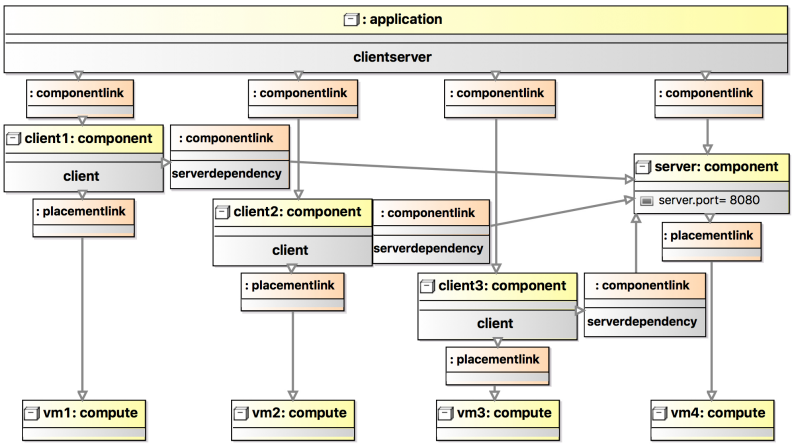


**Fig. 7** Modeling a Client/Server Application with MoDMaCAO.

– The **InstallationDependency** mixin type abstracts an installation dependency, i.e., the deploy action can only be successfully executed on the source component when the target component is already in the deployed state.
– The **ExecutionDependency** mixin type abstracts an execution dependency, i.e., the source component can only be started when the target component is already in the active state. For instance, the `ServerDependency` type abstracts the execution dependency from a client and a server component, i.e., the client component can not start until the server component is active.

Secondly, the MoDMaCAO modeling framework allows architects to model configured instances of cloud applications and their components. As illustration, Fig. 7 shows the model of a client-server application composed of three client components (`client1` to `client3`) and one server component (`server`) deployed on four virtual machines (`vm1` to `vm4`). OCCI resources and links are represented by boxes in yellow and orange color, respectively. The `application` resource is connected to the four `component` resources via `componentlink`s. Each `client` component is connected to the `server` component via a `componentlink` associated with a `serverdependency` mixin. The network port of the `server` component is set to 8080. Each component is placed on one virtual machine via a `placementlink`. Finally, the architecture, the number of cores, the host name, and the memory of each virtual machine are configured.

### 4.3 MoDMaCAO Verification

Thanks to the OCCIware metamodel, we can now define business constraints related to a cloud domain. These constraints are defined in the OCCIware extension and must be later respected by the conforming configurations. For the MoDMaCAO approach, we have defined a set of generic OCL constraints in the MoDMaCAO extension. Listing 1 shows the different implemented OCL constraints for the MoDMACAO framework. These constraints can be extended with additional ones that are specific to a MoDMaCAO use case.

```
context Application
inv OneOrMoreComponents : self.entity.oclAsType(occi::
    Resource).links->collect(l:occi::Link|l.target)->
    select(rs:occi::Resource|rs.oclIsTypeOf(platform::
    Component))->size()>=1

context Component
inv OnlyOnePlacementLink : self.entity.oclAsType(occi::
    Resource).links->select(l:occi::Link|l.oclIsTypeOf(
    placement::PlacementLink))->size()=1

context Dependency
inv SourceMustBeComponent : self.entity.oclAsType(occi::
    Link).source.oclIsTypeOf(platform::Component)

context Dependency
inv TargetMustBeComponent : self.entity.oclAsType(occi::
    Link).target.oclIsTypeOf(platform::Component)
```

**Listing 1** OCL constraints of MoDMaCAO framework

MoDMaCAO checks the validity of cloud application configurations by evaluating the constraints defined by used abstract types. For the client-server application, MoDMaCAO evaluates that the `Application` resource is connected to some `Component` resources (`OneOrMoreComponents` constraint), some client components (`OneOrMoreClients`), and only one server component (`OnlyOneServer`), all `Component` resources are placed on only one `Compute` resource (`OnlyOnePlacementLink`), each `client` is connected to one `server` (`OneServerDependency`), the value of the network port of the `server` component is in the valid range (0 to 65535), each `componentlink` must connect two `Component` resources (`SourceMustBeComponent` and `TargetMustBeComponent`), and each `componentlink` associated with `serverdependency` mixin must connect a `client` to a `server` component (`SourceMustBeClient` and `TargetMustBeServer`).

As long as a constraint is false, the architect must correct its cloud application configuration. When all the constraints are fulfilled, the cloud application can be deployed by the MoDMaCAO implementation framework.

### 4.4 MoDMaCAO Designers

The approach of MoDMaCAO aims to ease the design of cloud applications using the OCCIware toolchain. It allows to design multiple types of cloud applications and to instantiate them inside configurations which represent the running applications. Currently, the design of these running applications is done inside the generic OCCIware Designer as shown for example in Fig. 7. The elements shown in this kind of configurations are the instances of OCCI `Resource` and `Link` concepts. Therefore, the graphical representation does not reflect the concrete running system in the cloud. So, it is necessary to have a designer which (1) allows to instantiate the extension concepts and (2) can be customized to be as near as possible to the visual aspect of the designed application. To deal with this issue, the **Designer Generator** of the OCCIware toolchain allows us to generate a graphical designer for each OCCIware extension. However, the generated designer must be customized. To deal with this issue, we proceeded as following. At first, we have referred to the `Annotation` mechanism defined in the extended version of the OCCIware metamodel [11]. This feature allows us to design non-functional information such as visualization and documentation. The `Annotation` mechanism consists of key-value pairs. Several information can be modeled using this mechanism, such as the containment between resources, the ability to show/hide an attribute and highlight an edge. Then, we have extended the **Designer Generator** in order to support the defined annotations and generate a pre-configured designer implementing the specified annotations. For example, in the `ComponentLink` kind of the `platform` extension, we can associate a `Containment` annotation claiming that an "`Application` contains `Component`" (see Fig. 8). Based on this annotation, the generated designer will allow the architect to draw a `Component` resource inside an `Application` resource.

Originally, the **Designer Generator** of OCCIware Studio allows us to automatically obtain multiple graphical designers, and each one is specific to the domain of the application to design. The drawback of these generated graphical

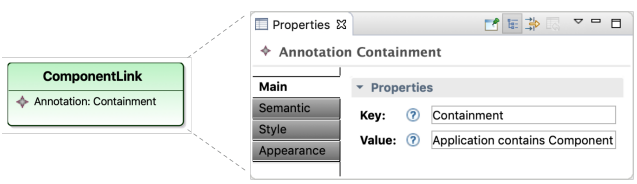


**Fig. 8** Annotation mechanism.

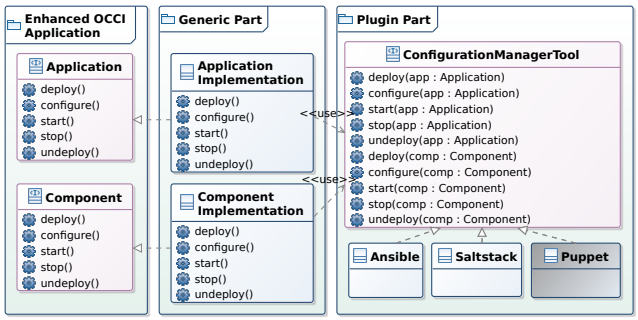


**Fig. 9** MoDMaCAO Implementation Class Diagram.

designers is they allow to create flat diagrams, i.e., an application is linked to a component, a component is linked to a compute, etc. Thanks to our annotation mechanism proposed in the MoDMaCAO framework, we can now, for each cloud domain, generate multiple graphical designers, and each one shows the system from a different point of view with a hierarchically manner. For example, we can visualize a system from a "Compute-viewpoint" where we show the different components deployed in a `Compute` resource. Or, we can visualize the system from an "Application-viewpoint" where the different components are shown inside the constituting `Application`.

### 4.5 The MoDMaCAO Implementation Framework

The MoDMaCAO implementation framework achieves the whole provisioning – i.e., installation, configuration then execution – of model-based cloud application instances on top of diverse configuration management tools such as Ansible, Saltstack and Puppet by using model interpretation. As illustrated in Fig. 9, this framework is split into two main parts: a generic part independent of any configuration management tool and a plugin part specific to each supported configuration management tool.

For the generic part, we used OCCIware Studio to automatically generate the skeleton of the framework from our three proposed OCCI extensions – enhanced OCCI Platform, Placement, and MoDMaCAO – and only implemented the five lifecycle actions – `deploy`, `undeploy`, `configure`, `start`, and `stop` – of both `Application` and `Component` kinds respecting their finite state machine. The following paragraphs describe the key behaviour we implemented.

The implementation of `Application` orchestrates the provisioning of all the components linked to an application. When the state of an application is `undeployed`, the implementation of `deploy` computes the order in which all the application components must be deployed according to their dependency links. Components connected by `Installation Dependency` links will be deployed sequentially, otherwise, components will be deployed in parallel. For instance, the

four components of the client-server application shown in Fig. 7 are deployed in parallel because they have no installation dependencies. When the state is `deployed`, the implementation of `configure` consists of configuring all the application components in parallel. When the state is `inactive`, the implementation of `start` computes the order on which all the application components must be started according to their `ExecutionDependency` links. For instance in the client-server application, the `server` component is started before the three client components can be started in parallel. When the state is `active`, the implementation of `stop` consists of stopping all the application components in the reverse order of their starting. For instance, client components are stopped before the server component is stopped. When the state is `inactive`, the implementation of `undeploy` consists of uninstalling all the application components in the reverse order of their deployment.

The implementation of `Component` implements the FSM of the `Component` kind and checks that the `Compute` resource where the component is placed, is already started before orchestrating the provisioning of the component.

Finally, the generic part delegates the calls to the plugin part specific to the used configuration management tool. Each plugin must implement the `ConfigurationManagerTool` interface shown in Fig. 9. For instance, the implementation of `start(Application)` called by the generic part must finalize the starting of a given application after all its components have been started. This implementation is specific to the used configuration management tool.

To provision the infrastructure resources to be configured by MoDMaCAO, a connector is required to translate incoming OCCI infrastructure requests to the interface of the cloud provider. While this translation can be done for arbitrary infrastructures, we implemented a connector[9] for a private Openstack cloud to perform our case studies.
In the following, we briefly discuss the implementation for the configuration management tool, Ansible, with help of the MoDMaCAO implementation framework. We also discuss how MoDMaCAO can be used to generate skeletons for Ansible artifacts.

We implemented an Ansible-specific plugin that implements the `ConfigurationManagementTool` interface. For each of the defined `Mixins`, an Ansible role[10] is created. It bundles the steps and files that are necessary to manage the corresponding software component on a specific machine. MoDMaCAO can assist with developing these roles by the generation of role skeletons from OCCI Extensions. For the prototypical implementation, we assume that these roles are already accessible from the OCCIware Runtime. When executing an action, the corresponding Ansible-tasks are executed on the machine. The latter is target of the `PlacementLink` of the corresponding `Component` instance. The `AttributeStates` defined in the `Components` and from all connected `Components` are used to generate Ansible variable files, that can be consumed by the Ansible roles to make their values available to the configuration management at runtime. Furthermore, we defined an additional OCCI Extension that allows to tag specific instances of the OCCI `NetworkInterface` as endpoints to be used by Ansible.

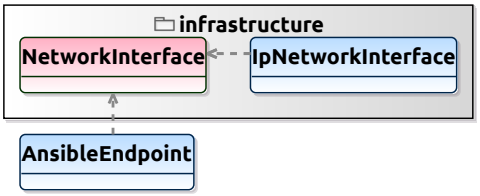


**Fig. 10** Ansible-specific OCCI Extension.

The Extension is depicted in Fig. 10. It proved to be useful in environments where virtual machines are connected to several networks. Based on the Mixins defined with help of the MoDMaCAO modeling framework, we can generate skeletons for configuration management scripts that can be later on extended and executed with Ansible. Listing 2 shows an excerpt from an Ansible playbook generated for the general `Component` Kind.

```
- name: Deploy Component
  block:
    - debug: msg="Operation deploy not implemented."
  when: task=="DEPLOY"
  become: yes

- name: Configure Component
  block:
    - debug: msg="Operation configure not implemented."
  when: task=="CONFIGURE"
  become: yes
...
```

**Listing 2** Excerpt of Ansible playbook skeleton generated for the Component Mixin.

For each of the Actions defined for the lifecycle of the `Component` Kind, a code block is generated that can be subsequently executed at runtime. This skeleton must be manually refined to actually implement the desired behavior.
Furthermore, to pass the model information at runtime to the configuration management tool, files that contain the information as variables can be generated. We provide examples for generated and extended artifacts in Section 5.

Currently, we have started to extend our approach by supporting the Saltstack[11] configuration management tool.

[9] `https://github.com/occiware/MoDMaCAO/tree/master/plugins/org.modmacao.openstack.connector`

[10] `https://docs.ansible.com/ansible/2.4/playbooks_reuse_roles.html`

[11] `https://www.saltstack.com`

In the future, we plan to extend our approach by supporting an additional tools such as Puppet[12].

## 5 USE CASES

In the following, we present two use cases to demonstrate the capabilities of the MoDMaCAO framework: a distributed MongoDB database[13] and the popular LAMP Web application stack[14]. While complementary work exists that allows to deploy OCCI configurations using several cloud providers [12], we performed our case studies on a private Openstack cloud. For this we used an OCCI deployment engine [13] which compares the desired cloud deployment model with the cloud runtime state to derive and send OCCI requests to adapt the cloud. As the OCCI interface, to which the requests of the engine are send, an OCCIWare runtime server instance is used. This server serves as a middleware that translates the incoming OCCI infrastructure request to the interface of the Openstack cloud. To manage the lifecycle of modeled components we use the configuration management capabilities provided by MoDMaCAO, focusing on the ansible implementation. After the presentation of both use cases, we provide a discussion about the capabilities provided by the enhanced OCCI platform extension and the use of the MoDMaCAO framework for application design, verification and deployment.

### 5.1 MongoDB

MongoDB is a NoSQL database that can be highly scaled and is often used in cloud environments. To achieve scalability, it supports the concept of *sharding*, i.e., the decomposition and distributed storage of a data collection to several machines. Furthermore, *replication sets* can be used, to provide redundancy and high availability in case a machine experiences a failure.

#### *5.1.1 Design*

Fig. 11 depicts how we specialize the mixin types defined by the MoDMaCAO framework to be able to model MongoDB clusters:

- The **MongoDBComponent** mixin type is the base type for all other MongoDB-specific `Component` mixin types. It defines the `mongodb.bindip` and `mongodb.port` attributes that specify the IP address and port on which the MongoDB service should be listening.

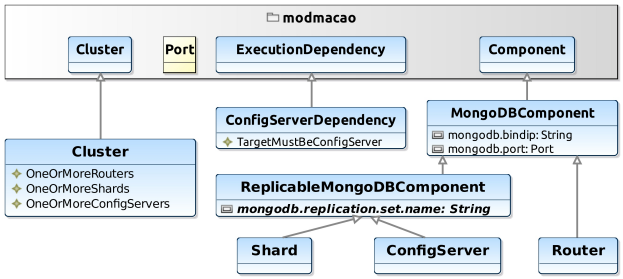


**Fig. 11** Modeling MongoDB with MoDMaCAO.

- The **ReplicableMongoDBComponent** mixin type defines the base type for components that can be replicated. It defines the `mongodb.replication.set.name` attribute that is used to assign a component to a certain replication set. MongoDB components belonging to the same replication set are synchronized copies of each other.
- The **Router** mixin type abstracts the notion of a *router* in the MongoDB cluster. A router implements the component to which the user connects. It forwards the requests of the user to the machines that actually hold the data.
- The **ConfigServer** mixin type abstracts the notion of a *config server* of a MongoDB cluster. A config server stores the metadata, including the state and organization of the data. It is also responsible to store authentication configuration information.
- The **Shard** mixin type abstracts the notion of a *shard* in the MongoDB cluster. The shards are used to store the actual data of the database. Each shard holds a subset of the overall data.
- The **Cluster** mixin type defines constraints for a MongoDB cluster: A cluster must contain at least one router (i.e., `OneOrMoreRouters`), at least one shard (i.e., `OneOrMoreShards`), and at least one config server (i.e., `OneOrMoreConfigServer`).
- The **ConfigServerDependency** mixin type abstracts the execution dependency between `MongoDBComponents` and a `ConfigServer`, to ensure that the `ConfigServer` is started, before the other components get started.

A model for a MongoDB cluster with three shards and no replication is depicted in Fig. 12. For the sake of brevity, we omit the depiction of `Attributes`. The MongoDB `cluster` consists of the components `router`, `configserver`, and the three shards, `shard1` to `shard3`. The `router` and `shard1` to `shard3` have an execution dependency to the `configserver`. Moreover, the `router` has an execution dependency to each `shard`. The components are placed on five different virtual machines, `vm1` to `vm5`, using `PlacementLinks`, which are connected to a network using `NetworkInterfaces`.

Fig. 12 shows a MongoDB cluster designed with the MongoDB designer, generated using the **Designer Gener-**

[12] `https://puppet.com`
[13] `https://www.mongodb.com/`
[14] `https://help.ubuntu.com/community/ApacheMySQLPHP`

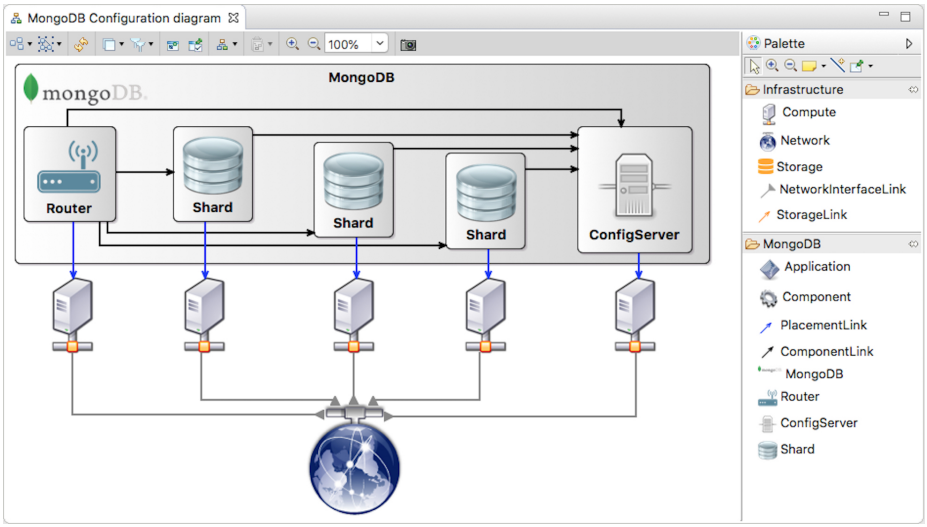


**Fig. 12** Modeling a MongoDB Cluster with MoDMaCAO.

**ator**. As we can see, the designed configuration follows the containment annotation defined in Section 4.4. So, different components are shown inside the MongoDB cluster application.

### *5.1.2 Verification*

To ensure the well-formedness of MongoDB configurations, MoDMaCAO approach allows to extend the generic OCL constraints by ones specific to the business of the application to design. Listing 3 shows the different implemented constraints for the MongoDB use case. So, a MongoDB application must have at least one router (`OneOrMoreRouters`), one shard (`OneOrMoreShards`), and one configuration server (`OneOrMoreConfigServers`).

```
context Cluster
inv OneOrMoreRouters :
self.entity.oclAsType(occi::Resource).links->collect(l:
    occi::Link|l.target)->collect(c:occi::Resource|c.
    parts)->select(mb:occi::MixinBase|mb.oclIsTypeOf(
    mongodb::Router))->size()>=1

inv OneOrMoreShards :
self.entity.oclAsType(occi::Resource).links->collect(l:
    occi::Link|l.target)->collect(c:occi::Resource|c.
    parts)->select(mb:occi::MixinBase|mb.oclIsTypeOf(
    mongodb::Shard))->size()>=1

inv OneOrMoreConfigServers : self.entity.oclAsType(occi::
    Resource).links->collect(l:occi::Link|l.target)->
    collect(c:occi::Resource|c.parts)->select(mb:occi::
    MixinBase|mb.oclIsTypeOf(mongodb::Configserver))->
    size()>=1
```

**Listing 3** OCL constraints of MongoDB use case

### *5.1.3 Configuration Management Artifact Generation*

The MongoDB OCCI extension defined above can be used as a basis for the generation of configuration management artifact skeletons. Thereby, for each of the Mixin types that can be applied to Resources of Kind Component, a skeleton for a configuration management script is generated. Listing 4 shows a configuration management skeleton generated (the blue parts) and extended for the MongoDB Router Mixin of the MongoDB extension. Hereby, the block including its name, e.g., `Deploy Router`, as well as when the block should be executed, `task == "DEPLOY"`, is generated. The individual modules describing the logic of what to deploy, e.g., `apt`, has to be manually set by the user which in this case deploys a router component of a MongoDB.

```
- name: Deploy Router
  block:
  - apt_key:
      keyserver: hkp://keyserver.ubuntu.com:80
      id: 9DA31620334BD75D9DCB49F368818C72E52529D4
      state: present
  - apt_repository:
      repo: deb [ arch=amd64 ] https://repo.mongodb.org/apt
          /ubuntu bionic/mongodb-org/4.0 multiverse
      state: present
  - apt:
      name: mongodb-org
      update_cache: yes
      state: present
  when: task == "DEPLOY"
  become: yes
- name: Configure Router
  block:
  - name: Copy startup script
    template: src=mongos_init.j2 dest=/etc/init/mongos.
        conf owner=mongodb
  - name: Copy configuration file template
    template: src=mongos.conf.j2 dest=/etc/mongos.conf
        owner=mongodb
```

```
when: task == "CONFIGURE"
become: yes
```

**Listing 4** Excerpt from Ansible playbook generated and extended for MongoDB router.

Listing 4 shows a variables file that is generated at runtime and can be consumed by Ansible when configuring the software component router of the defined MongoDB cluster. From this variable file, the IP address of the configuration server can be read by Ansible for the configuration of the router.

```
id: b6fc880a-0571-46ba-86db-a206c0d13675
kind: component
ipaddresses:
 - 10.0.0.31
mixins:
 - name: Router
   ...
attributes:
  occi.core.id: b6fc880a-0571-46ba-86db-a206c0d13675
  occi.core.title: router
  occi.core.summary: MongoDB cluster router
  occi.component.state: undeployed

links:
  - id: 4b9a6567-7cc8-4643-98a6-533068062b55
    kind: componentlink
    target:
      id: b6fc880a-0571-46ba-86db-a206c0d13679
      kind: component
      mixins:
        - name: ConfigServer
          ...
      ipaddresses:
        - 10.0.0.24
```

**Listing 5** Excerpt from Ansible variables file generated for MongoDB router at runtime.

Once the Ansible artifacts have been generated, they are later executed on our Openstack cloud. A demonstration, available here[15], shows the deployment process of the MongoDB cluster.

## 5.2 LAMP

This second use case addresses LAMP, which is an open source Web development platform that uses **L**inux as the operating system, **A**pache as the Web server, **M**ySQL as the relational database management system and **P**HP, **P**erl or **P**ython as the object-oriented scripting language.

### *5.2.1 Design*

The LAMP Web application can be modeled with help of the following mixins:

[15] https://github.com/occiware/MoDMaCAO/blob/master/videos/MongoDB-Cluster

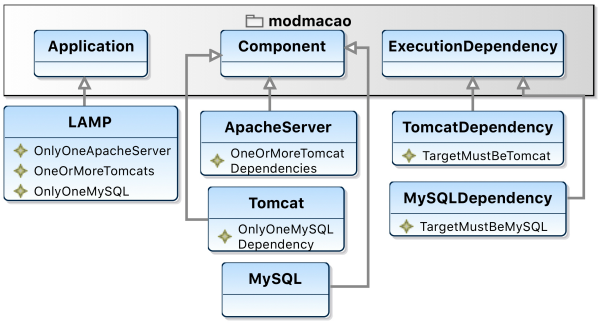


**Fig. 13** Modeling LAMP with MoDMaCAO.

- The **LAMP** mixin type abstracts the notion of a LAMP application and depends on MoDMaCAO `Application` mixin. A LAMP application is accessible via only one `ApacheServer` as enforced by the `OnlyOneApacheServer` constraint. It is deployed using one or more `Tomcat` container (i.e., `OneOrMoreTomcats` constraint). Moreover, the persistent data of a LAMP application are stored in only one MySQL database (i.e., `OnlyOneMySQL` constraint).
- The **ApacheServer** mixin type abstracts the notion of a LAMP Web server. It inherits from the `Component` mixin of the MoDMaCAO modeling framework. It defines `OneOrMoreTomcatDependencies` constraint enforcing that the `ApacheServer` instance cannot run if it is not linked to at least one `Tomcat` instance.
- The **Tomcat** mixin type abstracts the notion of a LAMP application container. It inherits from MoDMaCAO `Component` mixin. Each `Tomcat` instance is executed if it is connected to only one `MySQL` instance (i.e., `OnlyOneMySQLDependency` constraint).
- The **MySQL** mixin type abstracts the notion of a LAMP MySQL database and also inherits from MoDMaCAO `Component` mixin.
- The **TomcatDependency** mixin type abstracts a LAMP execution dependency by always connecting a `Component` instance to a `Tomcat` instance (`TargetMustBeTomcat`).
- The **MySQLDependency** mixin type abstracts a LAMP execution dependency by always connecting a `Component` instance to a `MySQL` instance (`TargetMustBeMySQL`).

Fig. 14 shows a LAMP stack designed with the LAMP designer, generated using the **Designer Generator**. As we can see, the designed configuration follows the containment annotation defined in Section 4.4. So, different components are shown inside the LAMP application.

### *5.2.2 Verification*

Before the deployment of the LAMP stack, it is necessary to verify the defined configuration. The approach of MoDMaCAO allows to extend the generic OCL constraints by

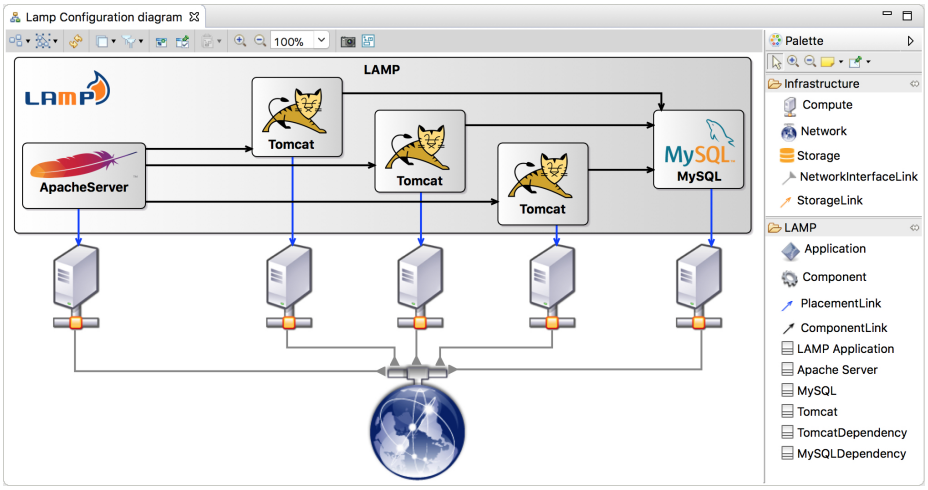


**Fig. 14** Modeling a LAMP stack with MoDMaCAO.

ones specific to the business of the application to design. Listing 6 shows the different implemented constraints for the LAMP use case. For example, a LAMP application must have exactly one Apache server (`OnlyOneApacheServer`), one MySQL database (`OnlyOneMySQL`) and at least one Tomcat server (`OneOrMoreTomcats`).

```
context LAMP
inv OnlyOneApacheServer :
self.entity.oclAsType(occi::Resource).links->collect(l:
    occi::Link|l.target)->collect(c:occi::Resource|c.
    parts)->select(mb:occi::MixinBase|mb.oclIsTypeOf(lamp
    ::Apacheserver))->size()=1

inv OnlyOneMySQL :
self.entity.oclAsType(occi::Resource).links->collect(l:
    occi::Link|l.target)->collect(c:occi::Resource|c.
    parts)->select(mb:occi::MixinBase|mb.oclIsTypeOf(lamp
    ::Mysql))->size()=1

inv OneOrMoreTomcats :
self.entity.oclAsType(occi::Resource).links->collect(l:
    occi::Link|l.target)->collect(c:occi::Resource|c.
    parts)->select(mb:occi::MixinBase|mb.oclIsTypeOf(lamp
    ::Tomcat))->size()>=1

context Tomcat
inv OnlyOneMySQLDependency :
self.entity.oclAsType(occi::Resource).links->select(l:occi
    ::Link|l.oclIsTypeOf(platform::Componentlink))->
    collect(l:occi::Link|l.parts)->select(mb:occi::
    MixinBase|mb.oclIsTypeOf(lamp::Mysqldependency))->
    size()>=1

context ApacheServer
inv OneOrMoreTomcatDependencies :
self.entity.oclAsType(occi::Resource).links->select(l:occi
    ::Link|l.oclIsTypeOf(platform::Componentlink))->
    collect(l:occi::Link|l.parts)->select(mb:occi::
    MixinBase|mb.oclIsTypeOf(lamp::Tomcatdependency))->
    size()>=1

context TomcatDependency
inv TargetMustBeTomcat :
self.entity.oclAsType(occi::Link).target.oclAsType(occi::
    Resource).parts->exists(mlm.oclIsTypeOf(lamp::Tomcat)
    )

context MySQLDependency
inv TargetMustBeMySQL :
self.entity.oclAsType(occi::Link).target.oclAsType(occi::
    Resource).parts->exists(mlm.oclIsTypeOf(lamp::Mysql))
```

**Listing 6** OCL constraints of LAMP use case

### *5.2.3 Configuration Management Artifact Generation*

The capabilities of MoDMaCAO to generate Ansible playbook skeletons are used for the defined LAMP extension. Listing 7 shows the generated Ansible playbook (the blue parts) extended with a block from the `deploy` operation of the `Tomcat` mixin. The unarchive and copy mechanisms within the script (black parts) are manually filled by the developer.

```
- name: Deploy Tomcat
  block:
    - yum: name=java-1.8.0-openjdk-devel state=present
    - unarchive:
        src: https://www-eu.apache.org/dist/tomcat/tomcat-9/
             v9.0.14/bin/apache-tomcat-9.0.14.tar.gz
        dest: /tmp
        remote_src: yes
    - file:
        path: /opt/tomcat/
        state: directory
    - shell: cp -r /tmp/apache-tomcat-9.0.14/* /opt/tomcat/
  when: task == "DEPLOY"
  become: yes
```

**Listing 7** Ansible task for LAMP Tomcat Deploy operation.

Once the Ansible artifacts have been generated, they are later executed on our Openstack cloud. A demonstration,

available here[16], shows the deployment process of the LAMP stack.

## 6 Discussion

In this section, we discuss the applicability of the proposed approach and how it contributes to the OCCI standard, as well as the model driven cloud domain. Similarly to the problem statements, the contributions can be separated to the creation and extension of a precise modeling framework for OCCI (**P1**), the benefits of verifying cloud application models at design time (**P2**), and the combination of infrastructure as code tools and model-driven engineering (**P3**).

To address the problem of a missing precise modeling framework (**P1**), we extended the OCCI platform extension and built a framework around it. We introduced capabilities to the standard that allows to deploy components on IaaS resources that can be managed at runtime. In MoDMaCAO, we enhanced the OCCI `Application` and `Component` definition by adding three additional lifecycle operations. Our use cases confirmed that these extensions are able to reflect the requirements for the deployment of the selected applications. Furthermore, by providing the notion of a `PlacementLink`, we are able to establish a connection between the OCCI Platform extension and the OCCI Infrastructure extension. The `PlacementLink` is used in the implementations to derive the IP address of the hosting virtual machines to be able to connect to them for the configuration management. To extend the modeling capabilities of MoDMaCAO, we demonstrated how customized designers can be created for defined cloud resource types. We have shown that the utilization of simple annotation mechanisms allows to greatly improve the automated generation of graphical editors, especially when viewpoints for different hierarchies are required.

To verify designed cloud applications prior to a live deployment (**P2**), we annotated OCCI cloud resource types with OCL constraints. For the basic platform elements in MoDMaCAO, we introduced generic constraints to check whether basic structures of the standard are adhered to. This comprises, e.g., constraints to check whether an `Application` has `OneOrMoreComponents`. In our use cases we demonstrated how the MoDMaCAO framework can be easily extended for customized resource and component types allowing designers to incorporate their own OCL well-formedness rules. Throughout the creation of the case studies the validation reduced the overall development time as errors within the structure of the modeled application are directly detected. Still, only constraints regarding the structure or attribute configuration within the OCCI configuration can be checked.

Finally, to address the lack of IDEs for Infrastructure as Code (**P3**), we coupled model-driven engineering techniques with configuration management. We separated the configuration management tool-specific logic from the generic provisioning order. In this way, only a minimal set of tool-specific code needs to be provided for each configuration management tool. While we tested the integration of SaltStack and simple Bash scripts with MoDMaCAO, our case studies focused on the integration of the Ansible configuration management tool into the OCCI ecosystem. The combination of configuration management tools with a model-driven approach supported not only the design time, but also the runtime management of the modeled components. At design time, the generation of the artifact skeletons allowed to immediately start with directives to describe the deployment of newly modeled component types. At runtime, variable files accompanying the script can be generated which provides these directives access to the current state of the cloud application. Moreover, the direct connection of component lifecycle actions to individual blocks in the script allowed for smaller iterations when refining the configuration management scripts which greatly supported the development process.

In comparison to the original OCCI platform extension, the cloud developer would have created manually coded deployment artifacts for each application to deploy. This activity is very hard, increases drastically defects in final code and decreases the productivity of developers. In addition, the developer needs to check all functional constraints such as the necessity to have a link between a compute resource and a component one. Furthermore, the cloud developer needs to perform all previously listed tasks without any visual support such as a graphical editor. This situation affects the understandability of the designed systems and complicates its reusability, maintenance and evolution. All these enhancements provided in MoDMACAO avoid users manually creating infrastructure codes and deploying applications without any guarantee of a successful execution.

The case studies presented in this paper only cover a small subset of application possibilities for MoDMaCAO. The original work additionally covers the modeling of a distributed Apache Cassandra database and an Apache Spark cluster [3]. In addition, MoDMaCAO has already been successfully applied in scope of resource deployment for scientific workflows in the cloud [14] and for adaptive sensor management [15].

## 7 RELATED WORK

As already mentioned in Section 1 and as explained in [16], there are two strategies to address the heterogeneity between cloud offerings. Since the first strategy, which is multi-cloud libraries, is only focused on the infrastructure interoperability, we detail in the following the state-of-the-art of the sec-

[16] `https://github.com/occiware/MoDMaCAO/tree/master/videos/LAMP-Stack`

ond remaining strategy, especially the solutions that tackle the management of applications.

1) MDE for the cloud: Nowadays, model-based solutions are becoming increasingly popular in cloud computing. Some of them are commercial application provisioning solutions enabling developers and administrators to specify deployment artifacts and dependencies. Notable examples include Ubuntu juju[17] that targets the modeling of applications and their hybrid deployment. In the same vein of this commercial graphical interface, several research projects are providing domain-specific modeling languages and frameworks that enable architects to describe and manage cloud platforms. Among these model-based solutions, we identify OCCIware [2] [6], which our work is an extension of. OCCIware has been successfully applied for the management of resources from different domains, including the management of Docker containers [12], and the management of mobile robots [17]. COAPS [18] is a PaaS interface for managing cloud applications. It extends the OCCI Core model, i.e., the Resource and Link concepts, without extending the OCCI platform extension. Moreover, COAPS complies to the previous, non-enhanced version of the OCCI standard, hence it lacks of the resource state management and the conformance verification provided by the OCCIware tool chain and MoDMaCAO. SALOON [19] is a model-driven multi-cloud configurator. It uses feature models to represent infrastructure and platform variability, as well as ontologies to describe the cloud applications requirements. SALOON targets four PaaS providers and the authors claim it can be extensible by adding new provider models that conform to the metamodel they define. However, this can be difficult and error-prone since this framework is not based on a standard, nor on some formal specification. TUNe [20] is a management system that is based on the Fractal component model for describing the software encapsulation and on two UML profiles, one for the deployment of legacy distributed applications and one for their reconfiguration using state diagrams. TUNe was applied for the administration of J2EE applications. Like most of the available model-driven configuration management approaches, TUNe allows changes only at design-time. This means that the deployment process may be repeated several times, which is costly and time-consuming.

Regarding runtime support, a strong analogy can be made between our approach and DeployWare [21], while the former is applied on cloud APIs and the latter on grid infrastructures. In fact, DeployWare provides a modeling language to deploy applications on Grid'5000[18] and a graphical interface to manage them at runtime. CloudML [22] is a cloud modeling language that helps to provision cloud infrastructure and platform resources by a semi-automatic matching between the defined application requirements and the cloud offerings. CloudML is exploited both at *design-time* to describe the application provisioning of cloud resources after performing the necessary orchestration, and at *runtime* to manage the deployed applications. Furthermore, its corresponding management framework CloudMF [23] presents follows a model driven approach to maintain multi-cloud applications. Another language to model cloud application is *Cloud Application Modelling and Execution Language* (CAMEL) [24] also utilizing the benefits of a models at runtime approach. In the CAMEL approach the Cloudiator toolkit [25] is used which provides a deployment engine building upon self-contained components described via executable artifacts and life-cycle actions.

Unlike our work, CloudML and CAMEL are not based on standards and requires the user to learn a new DSL. In addition, the different extensions of CloudML DSL are required to address different needs (monitoring, QoS, etc.) [26]. However, in OCCI, and thus OCCIware, this aspect is simplified by providing a single, simple and concise core DSL to capture the different concepts that could emerge to represent everything-as-a-service.

2) Cloud standards: Our work is also a standard-based approach since it adopts the OCCI standard metamodel. Besides OCCI, several cloud computing standards for managing cloud applications exist. The *Organization for the Advancement of Structured Information Standards* (OASIS)'s *Cloud Application Management for Platforms* (CAMP)[19] standard targets the deployment of cloud applications on top of PaaS resources. The OASIS's TOSCA standard defines a language to describe and package cloud application artifacts and deploy them on IaaS and PaaS resources. The Eclipse Winery[20] project provides an open source Eclipse-based graphical modeling tool for TOSCA when the OpenTOSCA project provides an open source container for deploying TOSCA-based applications [27]. Cloudify[21] 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.

. For the deployment aspects, OpenTOSCA chose to use management plans implemented as BPEL and/or BPMN workflows to deploy applications and adaptation plans to update a deployment based on the given situation at runtime. In the OCCI based adaptation process utilized in our use cases similar plans are generated with MoDMaCAO providing the capability to plug our models into the management configuration tools such as Ansible. The declarative aspect of the latter and its idempotency allows it to only affect the runtime part concerned by the update in the modeling level. In ad-

[17] http://juju.ubuntu.com/

[18] https://www.grid5000.fr/

[19] https://www.oasis-open.org/committees/camp/

[20] https://www.eclipse.org/proposals/soa.winery/

[21] http://cloudify.co/

dition, CAMP and TOSCA can use OCCI-based IaaS/PaaS resources, so these standards are complementary. This standards "marriage" will be a main pillar of our future work, as discussed in Section 8.

## 8 CONCLUSION

This article presents our approach, named MoDMaCAO, for model-driven configuration management of cloud applications at runtime by using an enhanced version of OCCI standard. MoDMaCAO has the following features: *i)* a modeling framework to design cloud applications based on OCCI standard, *ii)* the visualization facilities to seamlessly create cloud applications, *iii)* a verification mechanism to ensure the correctness of the designed applications, and finally *iv)* a generative approach to automatically produce configuration management artifacts. We used the OCCIware tool chain to model the proposed enhancements and used its capabilities to generate prototypical implementations for different configuration management tools. Furthermore, we showed how the proposed framework can be used to model, deploy and manage two different distributed cloud applications, a MongoDB cluster and a LAMP stack.

As future work, we will investigate how the proposed framework can be extended to support multiple configuration management tools to be used side-by-side for managing a single cloud application. We also want to incorporate concepts that support the reuse of defined `Component` mixins in other applications. Second, we plan to reduce the manual written parts of scripts. For that, we need to define the semantics of a configuration management tool in order to finely capture the behavior of each operation. Then, we can extend the Deployment artifacts generator to generate scripts based on the chosen configuration management tool. In addition, we intend to explore in depth the behavior definition of OCCIware resources. One possible improvement consists to incorporate an assertion based on the dependency mixin in the finite state machine of the related kind. Our long-term goal is to extend the provided approach and tooling with the support of additional cloud standards, including TOSCA and CAMP. We already defined a preliminary mapping between TOSCA and OCCI [28]. We will further refine this mapping as a basis for providing an integrated solution for model-driven cloud orchestration utilizing both standards.

## ACKNOWLEDGEMENTS

We thank the Simulationswissenschaftliches Zentrum Clausthal-Göttingen (SWZ), the French PIA OCCIware project (`www.occiware.org`), and the Hauts-de-France Regional Council for supporting this work.

## AVAILABILITY

Readers can find the open source code base of MoDMaCAO on `https://github.com/occiware/MoDMaCAO`.

## References

1. R. Nyrén, A. Edmonds, A. Papaspyrou, T. Metsch, and B. Parák, "Open Cloud Computing Interface - Core," September 2016, [Available online: http://ogf.org/documents/GFD.221.pdf].
2. J. Parpaillon, P. Merle, O. Barais, M. Dutoo, and F. Paraiso, "Occiware-a formal and tooled framework for managing everything as a service," in *Projects Showcase@ STAF'15*, vol. 1400, 2015, pp. 18–25.
3. F. Korte, S. Challita, F. Zalila, P. Merle, and J. Grabowski, "Model-driven configuration management of cloud applications with occi," in *8th International Conference on Cloud Computing and Services Science (CLOSER)*, 2018, pp. 100–111.
4. T. Metsch and M. Mohamed, "Open Cloud Computing Interface - Platform," September 2016, [Available online: https://www.ogf.org/documents/GFD.227.pdf].
5. P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, and S. Tata, "A Precise Metamodel for Open Cloud Computing Interface," in *8th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2015, pp. 852–859.
6. F. Zalila, S. Challita, and P. Merle, "A Model-Driven Tool Chain for OCCI," in *25th International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS)*. Springer, Cham, 2017, pp. 389–409.
7. H. Medhioub, B. Msekni, and D. Zeghlache, "OCNI – Open Cloud Networking Interface," in *22nd International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2013, pp. 1–8.
8. T. Metsch, A. Edmonds, and B. Parák, "Open Cloud Computing Interface - Infrastructure," September 2016, [Available online: http://ogf.org/documents/GFD.224.pdf].
9. S. Yangui and S. Tata, "CloudServ: PaaS resources provisioning for service-based applications," in *27th IEEE International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2013, pp. 522–529.
10. ——, "An OCCI Compliant Model for PaaS Resources Description and Provisioning," *The Computer Journal*, vol. 59, no. 3, pp. 308–324, 2014.
11. F. Zalila, S. Challita, and P. Merle, "Model-driven cloud resource management with occiware," *Future Generation Computer Systems*, vol. 99, pp. 260 – 277, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18306071
12. F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-Driven Management of Docker Containers," in *9th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 718–725.
13. J. Erbel, F. Korte, and J. 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 (CLOSER)*, 2018.
14. E. Johannes, K. Fabian, and G. Jens, "Scheduling architectures for scientific workflows in the cloud," in *System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering*, K. Ferhat and G. Reinhard, Eds. Cham: Springer International Publishing, 2018, pp. 20–28.
15. J. Erbel, T. Brand, H. Giese, and J. Grabowski, "OCCI-compliant, fully causal-connected architecture runtime models supporting sensor management," in *Proceedings of the 14th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2019)*, May 2019.

16. S. Challita, F. Paraiso, and P. Merle, "Towards Formal-based Semantic Interoperability in Multi-clouds: The fclouds Framework," in *10th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2017, pp. 710–713.
17. P. Merle, C. Gourdin, and N. Mitton, "Mobile Cloud Robotics as a Service with OCCIware," in *2nd IEEE International Congress on Internet of Things (ICIOT)*. IEEE, 2017, pp. 50–57.
18. M. Sellami, S. Yangui, M. Mohamed, and S. Tata, "PaaS-independent Provisioning and management of applications in the cloud," in *6th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2013, pp. 693–700.
19. C. Quinton, D. Romero, and L. Duchien, "SALOON: A Platform for Selecting and Configuring Cloud Environments," *Software: Practice and Experience*, vol. 46, no. 1, pp. 55–78, 2016.
20. O. Chebaro, L. Broto, J.-P. Bahsoun, and D. Hagimont, "Self-TUNe-ing of a J2EE Clustered Application," in *6th IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems, 2009. EASe 2009*. IEEE, 2009, pp. 23–31.
21. A. Flissi, J. Dubus, N. Dolet, and P. Merle, "Deploying on the Grid with DeployWare," in *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE, 2008, pp. 177–184.
22. N. Ferry, G. Brataas, A. Rossini, F. Chauvel, and A. Solberg, "Towards Bridging the Gap Between Scalability and Elasticity," in *4th International Conference on Cloud Computing and Services Science (CLOSER)*, 2014, pp. 746–751.
23. N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "Cloudmf: Model-driven management of multi-cloud applications," *ACM Trans. Internet Technol.*, vol. 18, no. 2, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3125621
24. A. P. Achilleos, K. Kritikos, A. Rossini, G. M. Kapitsaki, J. Domaschka, M. Orzechowski, D. Seybold, F. Griesinger, N. Nikolov, D. Romero *et al.*, "The cloud application modelling and execution language," *Journal of Cloud Computing*, vol. 8, no. 1, p. 20, 2019.
25. D. Baur, D. Seybold, F. Griesinger, H. Masata, and J. Domaschka, "A provider-agnostic approach to multi-cloud orchestration using a constraint language," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '18. IEEE Press, 2018, p. 173–182. [Online]. Available: https://doi.org/10.1109/CCGRID.2018.00032
26. A. Bergmayr, A. Rossini, N. Ferry, G. Horn, L. Orue-Echevarria, A. Solberg, and M. Wimmer, "The Evolution of CloudML and its Manifestations," in *3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, 2015, pp. 1–6.
27. T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA–a runtime for TOSCA-based cloud applications," in *Service-Oriented Computing*. Springer, 2013, pp. 692–695.
28. F. Glaser, J. Erbel, and J. Grabowski, "Model Driven Cloud Orchestration by Combining TOSCA and OCCI," in *7th International Conference on Cloud Computing and Services Science (CLOSER)*. SciTePress, 2017, pp. 644–650.