Model-Driven Configuration Management of Cloud Applications with OCCI

Fabian Korte[1], Stéphanie Challita[2], Faiez Zalila[2], Philippe Merle[2], Jens Grabowski[1]
[1] *University of Goettingen, Institute of Computer Science, Goldschmidstraße 7, Goettingen, Germany*
{fkorte, grabowski}@cs.uni-goettingen.de

[2] *Inria Lille - Nord Europe & University of Lille, CRIStAL UMR CNRS 9189, France*
{firstname.lastname}@inria.fr

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

Abstract: To tackle the cloud-provider lock-in, the Open Grid Forum (OGF) 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, the OGF also defines extensions that reflect the requirements of different cloud service levels, such as IaaS and PaaS. However, so far the OCCI PaaS extension is very coarse grained and lacks of supporting use cases and implementations. Especially, it does not define how the components of the application itself can be managed. In this paper, we present a model-driven framework that extends the OCCI PaaS extension and is able to use different configuration management tools to manage the whole lifecycle of cloud applications. We demonstrate the feasibility of the approach by presenting four different use cases and prototypical implementations for three different configuration management tools.

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, three different strategies can be identified in the literature: using code libraries that provide a common *Application Programming Interface* (API) for the different cloud provider APIs, e.g., Apache jclouds[1] [2], or fog[3], using techniques from *model driven engineering* (MDE) to decouple the cloud applications from the technical peculiarities of the different target platforms, e.g., OCCIware (Parpaillon et al., 2015), and SALOON (Quinton et al., 2016), and the development of common standards, e.g., the *Topology and Orchestration Specification for Cloud Applications* (TOSCA)[4], and the *Open Cloud Computing Interface* (OCCI) (Nyrén et al., 2016). Of the two mentioned standardization approaches, TOSCA currently receives more attention by both the industry and research community, but their focus is different and they can be used complementarily (Glaser et al., 2017). In this paper, we focus on OCCI, which is developed by the *Open Grid Forum* (OGF) and aims to standardize an API 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-as-a-Service* (IaaS) and *Platform-as-a-Service* (PaaS) layers. Several implementations and use cases for the IaaS extension already exist, that demonstrate its feasibility. However, implementations and use cases for the PaaS extension are rare. This might be due to the fact, that it only provides a very rough definition of cloud applications and its components, that does not include how these cloud applications can be configured and managed. Furthermore, it does not explain how application components are connected to the hosting infrastructure, such as, which component gets deployed on which virtual machine. To close these gaps, we demonstrate a model-driven approach for the configuration management of cloud applications using OCCI. Thereby,

[1] http://www.jclouds.org
[2] All URLs have been last retrieved on 01/03/2018.
[3] http://fog.io
[4] https://www.oasis-open.org/committees/TOSCA/

we provide the following contributions to tackle the identified shortcomings:

1. We propose an enhancement of the OCCI Platform lifecyle definition by adding new *deployed/undeployed* states to the application components and new actions to transit between these states,
2. we demonstrate how the OCCI model can be extended to enable the modeling of application component placements on an hosting cloud infrastructure,
3. we provide a common framework for utilizing OCCI for configuration management with different configuration management tools, and
4. show its feasibility by the means of different case studies.

The remainder of this paper is structured as follows. We introduce the OCCI model and the OCCIware tool chain in Section 2. Afterwards, in Section 3, we discuss the open issues that exist with the current version of OCCI related to the configuration management of cloud applications. In Section 4, we introduce the *Model-Driven Configuration Management of Cloud Applications with OCCI* (MoDMaCAO) framework, that aims to tackle these issues. To demonstrate the feasibility of the provided framework, we discuss how it can be applied to model different cloud applications and implement their configuration management with different configuration management tools in Section 5. We discuss related work in Section 6 and finally conclude this paper and provide an overview on future work in Section 7.

2 BACKGROUND

In the following, we provide a brief overview of OCCI standard and OCCIware model-driven tool chain.

2.1 Open Cloud Computing Interface

The OCCI Core model (Nyrén et al., 2016) is composed of eight elements (grey boxes in Figure 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.

The OCCI Core model is accompanied with several extensions. The OCCI Platform extension (Metsch and Mohamed, 2016) defines the two specialized `Resources`: `Application` and `Component` and a new `Link` type `ComponentLink` (see Figure 3). 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 API 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 Figure 1), has been proposed in (Merle et al., 2015) and enhanced in (Zalila et al., 2017). 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 (Medhioub et al., 2013), infrastructure extension (Metsch et al., 2016), platform extension (Yangui and Tata, 2013; Yangui and Tata, 2014; Metsch and Mohamed, 2016), application extension (Yangui and Tata, 2014), 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 domain. The constraints can be expressed on OCCI kinds and mixins. Finally, the OCCIWARE

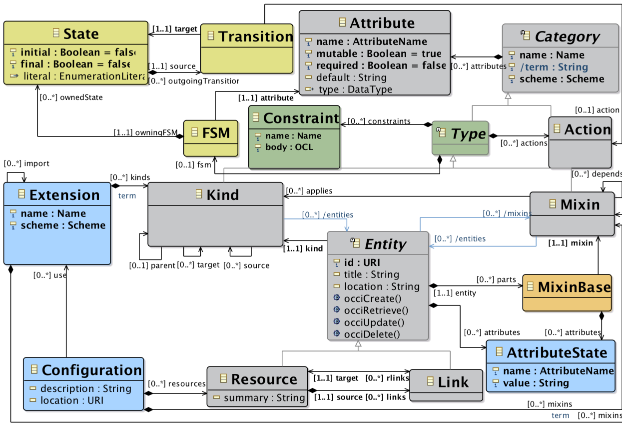

Figure 1: A subset of OCCIWARE METAMODEL.

METAMODEL provides the FSM mechanism. This later allows to describe the behavior of each OCCI kind/mixin as a *Finite State Machine* (FSM).

To resolve the second issue, a model-driven tool chain for OCCI, named OCCIWARE STUDIO, has been proposed (Zalila et al., 2017). 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, this generated connector is deployed on the OCCIware Runtime.

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:

- **Incomplete lifecycle model (P1)**: 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,
- **No connection between infrastructure and platform models (P2)**: The OGF provides two separate OCCI extensions for the IaaS and PaaS 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,
- **No support for configuration management (P3)**: In the current version of the specification, it is not defined how `Components` and `Applications` can be managed throughout their lifecycle and if and how additional tooling, e.g., configuration management tools can be integrated for this purpose,
- **Lack of use cases and implementations (P4)**: The current version of the specification lacks of any real-world use case for the application of the Platform extension. Furthermore, no canonical implementation is available.

To overcome these issues, we provide a framework for modeling and managing cloud applications with OCCI, which we will introduce in the next section.

4 MODMACAO

In the following, we will introduce the building blocks of the MoDMaCAO framework and how they tackle the problems identified above.

4.1 Overall Architecture

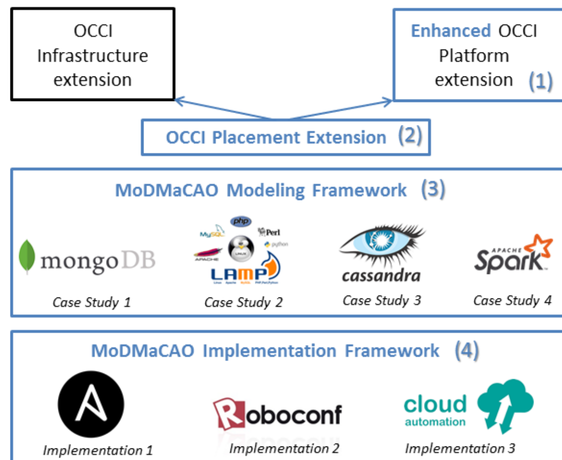

Figure 2: Overall Architecture.

The overall architecture of the proposed MoDMaCAO framework and its contributions

are depicted in Figure 2. Our first contribution (1) is to address **P1** by enhancing of the OCCI Platform extension via additional lifecycle `States` and `Actions`. Furthermore, we introduce a new `Link Kind` (2) to be able to connect `Components` of the OCCI Platform extension to `Compute` resources of the OCCI Infrastructure extension (addressing **P2**). As a third contribution (3), we define a new OCCI extension to be able to model application components that are managed with help of a configuration management tool (addressing **P3**). We demonstrate the feasibility of the defined extension by modeling four different distributed cloud applications and finally provide a framework for implementing model-driven configuration management with different configuration management tools (4), thereby addressing **P4**.

4.2 Enhanced OCCI Platform Lifecycle

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 it, we propose an enhancement of the OCCI Platform extension as shown in Figure 3. The different improvements are colored

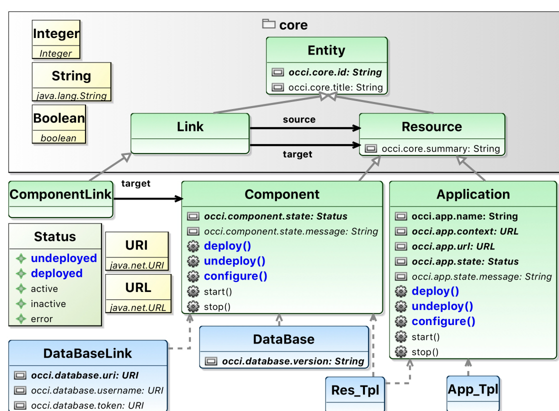

Figure 3: Enhanced OCCI Platform kinds.

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. Figure 4 shows the enhanced FSM for `Component` and `Application` kinds. Therefore, a `Component/Application` resource is initially `undeployed`. Once the `deploy` action is triggered, the resource is deployed. By triggering the `configure` action, the resource is configured and reaches the `inactive` state. Finally, a `Component/Application` can reach the `active` state by triggering the `start` action.

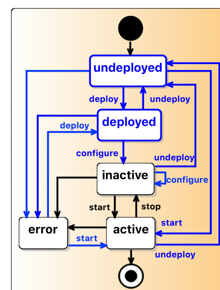

Figure 4: Enhanced OCCI Platform FSMs.

4.3 OCCI Placement Extension

Figure 5 depicts the definition of a new link kind `PlacementLink` that addresses the missing connection between the OCCI Platform extension and the OCCI Infrastructure extension. A `PlacementLink` has a `Component` resource as its `source` and a `Compute` resource as its `target`, and hence allows to model the placement of an application component on a virtual machine.

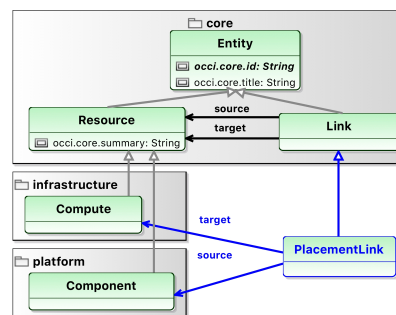

Figure 5: New OCCI Placement Extension.

4.4 MoDMaCAO Modeling Framework

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 that use the defined abstract types , and (3) check the validity of instances of cloud applications. Firstly, as

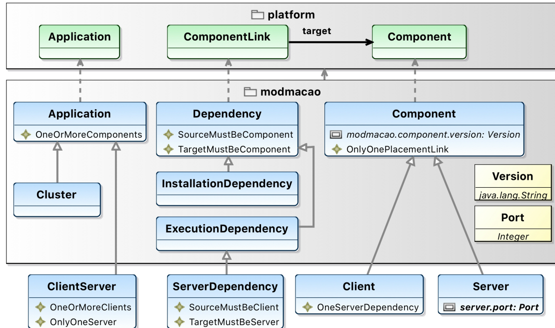

Figure 6: The MoDMaCAO Modeling Framework.

shown in Figure 6, the MoDMaCAO modeling framework defines the following set of abstract types:

- The **Application** mixin type abstracts the notion of 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).
- The **Cluster** mixin type abstracts the notion of clustered cloud application.
- The **Component** mixin type abstracts the notion of 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 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`, and `ServerDependency`. These new types could define 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.
- The **InstallationDependency** mixin type abstracts an installation dependency, i.e., the source component could be deployed only when the target component is already deployed.
- The **ExecutionDependency** mixin type abstracts an execution dependency, i.e., the source component could be started only when the target component is already started. 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 started.

Secondly, the MoDMaCAO modeling framework allows architects to model configured instances of cloud applications and their components. As illustration, Figure 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 `componentLinks`. Each `client` component is connected to the `server` component via a `ServerDependency` link. 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, the speed, and the memory of each virtual machine are configured.

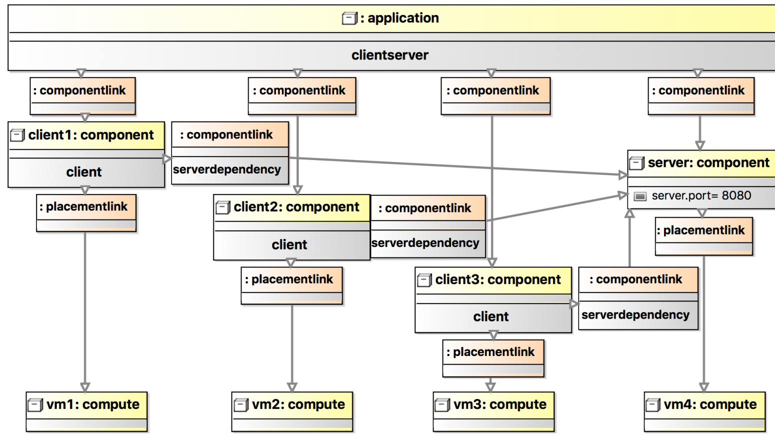

Figure 7: Modeling a Client/Server Application with MoDMaCAO.

Thirdly, MoDMaCAO checks the validity of cloud application configurations by evaluating all 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 `Dependency` link connects two `Component` resources (`SourceMustBeComponent` and `TargetMustBeComponent`), and each `ServerDependency` link connects 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 true, the cloud application can be deployed by the MoDMaCAO implementation framework.

4.5 MoDMaCAO Implementation

The MoDMaCAO implementation framework realizes 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, Roboconf and Cloud Automation by using model interpretation. As illustrated in Figure 8, 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, as discussed in Section 5.2.

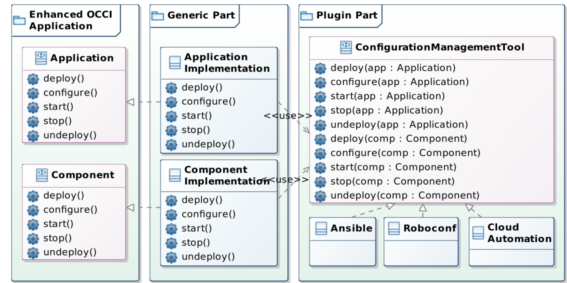

Figure 8: MoDMaCAO Implementation Class Diagram.

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 `InstallationDependency` links. Components not connected by `InstallationDependency` links are deployed in parallel when components connected by `InstallationDependency` links are deployed sequentially. For instance, the four components of the client-server application shown in Figure 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 are 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 in parallel 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 `ConfigurationManagementTool` interface shown in Figure 8. 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.

5 EVALUATION

To evaluate the proposed approach, we selected four different case studies that represent distributed cloud applications. Furthermore, we demonstrate how our approach can be easily adapted to different configuration management tools by providing experimental implementations for three different configuration management and cloud orchestration tools.

5.1 Case Studies

As case studies, we selected a distributed MongoDB database[5], the popular LAMP web-application stack[6], a distributed Cassandra database[7] and an Apache Spark cluster[8]. Due to space constraints, we focus on the description of the defined MongoDB extension of MoDMaCAO, and we provide a configuration model that is conform to this extension. For the other use cases, we give only a brief overview of the defined extensions. Full-sized configuration models using each of these three remaining extensions can be found in the supplemental material[9].

5.1.1 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 of 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.

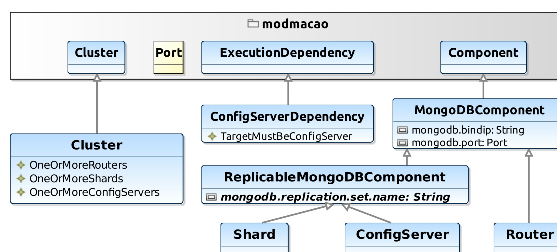

Figure 9: Modeling MongoDB with MoDMaCAO.

Figure 9 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 attributes `mongodb.bindip`, and `mongodb.port`, that specifies the IP address and port on which the MongoDB service should be listening.
- The **ReplicableMongoDBComponent** mixin type defines the base type for components that can be replicated. It defines the attribute `mongodb.replication.set.name` 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

[5] https://www.mongodb.com/
[6] https://help.ubuntu.com/community/ApacheMySQLPHP
[7] http://cassandra.apache.org/doc/latest/
[8] https://spark.apache.org/docs/latest/
[9] https://github.com/occiware/MoDMaCAO

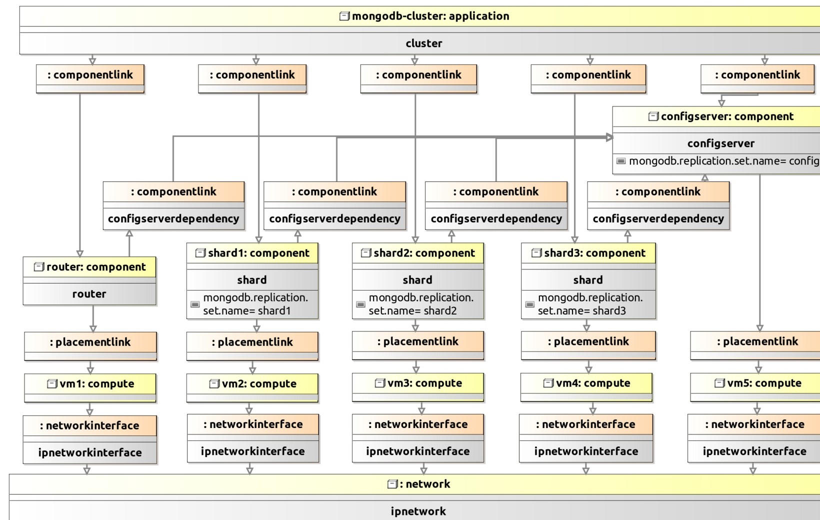

Figure 10: Modeling a MongoDB Cluster with MoDMaCAO.

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 be able 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 Figure 10. For the sake of brevity, we omit the depiction of `Attributes`. The MongoDB-cluster `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`. The components are placed on five different virtual machines, `vm1` to `vm5`, using `PlacementLinks`, which are connected to a network using `NetworkInterfaces`.

5.1.2 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.

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

- 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`).

5.1.3 Apache Cassandra

Apache Cassandra is an open-source distributed NoSQL database management system. It is designed to handle large amounts of data by offering support for *clusters* across multiple datacenters. To achieve scalability in Cassandra clusters, the architect adds new *nodes* to an existing cluster without having to disconnect it first. When a new node is added, the cloud architect has to enter the new node name in the *seed* component list and then the new node will be part of the Cassandra database architecture, called "ring". Also, unlike other sharded systems like MongoDB and other Master/Worker systems like Apache Spark (cf. Section 5.1.4), a single point of failure does not affect the whole cluster in Cassandra. Therefore, it is capable of offering continuous availability. To do so, it automatically distributes, replicates and maintains data across the nodes of a cluster. We detail in the following each element of the Cassandra Model:

- The **Cluster** mixin type abstracts the notion of a clustered Cassandra database. This mixin depends on MoDMaCAO modeling framework by inheriting from its `Cluster` mixin. It defines two constraints, i.e., `OneOrMoreNodes` and `OneOrMoreSeeds` that enforce that a `Cluster` contains at least one `Node` and one `Seed`, respectively.
- The **Node** mixin type abstracts the notion of a Cassandra component. All nodes play an identical role, they communicate with each other via a distributed, scalable protocol called "gossip".
- The **Seed** mixin type inherits from the `Node` mixin. The seed node maintains all the nodes list in a cluster.

5.1.4 Apache Spark

Apache Spark is an open-source cluster computing platform for big data processing. We define the following extension to be able to model Apache Spark clusters:

- The **ApacheSparkCluster** mixin type abstracts the notion of a Spark application that runs as independent sets of processes on a cluster. The `ApacheSparkCluster` connects to several types of `ApacheSparkComponent` instances, i.e., only one master (`OnlyOneMaster`) and at least one worker (`OneOrMoreWorkers`).
- The **ApacheSparkComponent** mixin type abstracts the notion of a Spark component. Each Spark component has `apache.spark.port` and `apache.spark.webui.port` attributes representing the port to access the Spark console and the Spark Web interface, respectively. An `ApacheSparkComponent` instance may be either a `Master` or a `Worker`.
- The **Master** mixin type abstracts the notion of a managing component that has a pool of jobs and assigns them to workers.
- The **Worker** mixin type represents any node that can execute a job, i.e., run application code in the cluster. Each worker has an `apache.spark.worker.cores` attribute representing the number of cores in each worker. It also has an `apache.spark.worker.memory` attribute representing the worker memory in gibibyte (GiB). A `Worker` instance depends on on the execution of only one `Master` instance (`OnlyOneMasterDependency`). We also define `WorkerCoresSmallerThanComputeCores` and `WorkerMemorySmallerThanComputeMemory` constraints that enforce that the cores and the memory of each worker are smaller than those of its hosting virtual machine.
- The **MasterDependency** mixin type abstracts a Spark execution dependency by always connecting a `Worker` instance to a `Master` instance (`SourceMustBeWorker` and `TargetMustBeMaster`).

5.2 Implementations

In the following we briefly discuss the implementation for the configuration management tools Ansible, Roboconf[10] and Cloud Automation[11] with help of the MoDMaCAO implementation framework.

5.2.1 Ansible

We implemented an Ansible-specific plugin that implements the `ConfigurationManagementTool` interface. For each of the defined `Mixins`, an Ansible role[12] is created that bundles the steps and files that

[10] https://roboconf.net/

[11] https://www.activeeon.com/cloud-automation/

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

are necessary to install the corresponding software component on a specific machine. For the prototypical implementation, we assume that these roles are already accessible from the OCCIware Runtime. When executing the `deploy` action, this role is triggered by our plugin. We further provide Ansible roles for the `undeploy`, `start`, `stop`, and `configure` actions for each `Mixin`. The `Attributes` defined by the `Mixins` are getting passed to Ansible in form of Ansible variables and are accessible in the configuration steps defined in the Ansible roles.

5.2.2 Roboconf

Secondly, we implemented a Roboconf-specific plugin (Pham et al., 2015), which is responsible of managing the application lifecycle via its `SoftwareInstanceManager` concept. The latter extends the `ConfigurationManagementTool` interface of MoDMaCAO implementation framework. The `SoftwareInstanceManager` comprises three connectors for managing the `Application` instances, their `Component` instances and the `ComponentLink` instances, respectively. A Roboconf method `deployAndStartAll()`, which deploys an application and directly starts its components, is executed by only calling the `deploy()` method of MoDMaCAO. The `start()` method of MoDMaCAO is implicit and not implemented in Roboconf. A `Mixin` will start directly when it is deployed. The same behavior is applied to `undeployAll()` method of Roboconf which will implicitly `stop()` the defined `Mixin`.

5.2.3 Cloud Automation

Finally, the third plugin integrated to our MoDMaCAO implementation framework is the Cloud Automation orchestrator. Cloud Automation is based on workflows, which are series of automated actions that the cloud developer triggers to occur based on the `Application` or `Component` state. For example, Cloud Automation organizes workflows for stopping an application, as follows: `stop` all the application components, `undeploy` them, and then `stop` the application. These workflows are implemented thanks to OCCI finite state machine.

5.3 Discussion

We enhanced the OCCI `Application` and `Component` definition by adding three additional lifecycle operations. Our case studies 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. 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. By applying our implementation framework successfully to three different configuration management tools, we were able to show, that our approach is generic enough to cover the requirements of different tools. Nevertheless, we also discovered that there are some tool-specific configuration needed that should also become part of the modeling framework. For example, a SSH Key and a username is required by Ansible to connect to the virtual machines for the configuration management. Such information could be either provided by tool-specific `Mixins` that are added to the models or by enhancing the OCCI Infrastructure extension to be able to cover authentication information for provided virtual machines. MoDMaCAO proved to be powerful enough to model different distributed cloud applications and we were also able to use its implementation framework to successfully provide plugins for different configuration management tools.

6 RELATED WORK

As already mentioned in Section 1 and as explained in (Challita et al., 2017), there are three 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 two remaining strategies, especially the solutions that tackle the management of applications.

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[13] 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

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

frameworks that enable architects to describe and manage cloud platforms. Among these model-based solutions, we identify OCCIware (Parpaillon et al., 2015) (Zalila et al., 2017), 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 (Paraiso et al., 2016), and the management of mobile robots (Merle et al., 2017). COAPS (Sellami et al., 2013) is a PaaS API 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 (Quinton et al., 2016) 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 now 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 (Chebaro et al., 2009) 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 (Flissi et al., 2008), 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[14] and a graphical interface to manage them at runtime. CloudML (Ferry et al., 2013) 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. Unlike our work, CloudML is not based on standards and requires the user to learn a new DSL.

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)[15] 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[16] 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 (Binz et al., 2013). Cloudify[17] 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.

In contrast to CAMP and TOSCA, OCCIware models are executable inside a Models@run.time (Blair et al., 2009) interpreter framework. In addition, 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 7.

7 CONCLUSION

We presented an approach for model-driven configuration management of cloud applications at runtime by using an enhanced version of OCCI. 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 four different distributed cloud applications. 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

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

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

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

[17]http://cloudify.co/

to incorporate concepts that support the reuse of defined `Component` mixins in other applications.

Our long-term goal is to extend the provided concept and tooling with the support for additional cloud standards, including TOSCA and CAMP. We already defined and a preliminary mapping between TOSCA and OCCI (Glaser et al., 2017). 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

Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., and Wagner, S. (2013). OpenTOSCA–a runtime for TOSCA-based cloud applications. In *International Conference on Service-Oriented Computing*, pages 692–695. Springer.

Blair, G., Bencomo, N., and France, R. B. (2009). Models@run.time. *Computer*, 42(10).

Challita, S., Paraiso, F., and Merle, P. (2017). Towards Formal-based Semantic Interoperability in Multi-clouds: The fclouds Framework. In *10th IEEE International Conference on Cloud Computing (CLOUD)*, pages 710–713. IEEE.

Chebaro, O., Broto, L., Bahsoun, J.-P., and Hagimont, D. (2009). Self-TUNe-ing of a J2EE Clustered Application. In *6th IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems, 2009. EASe 2009*, pages 23–31. IEEE.

Ferry, N., Rossini, A., Chauvel, F., Morin, B., and Solberg, A. (2013). Towards Model Driven Provisioning, Deployment Monitoring, and Adaptation of Multi-Cloud Systems. In *6th IEEE International Conference on Cloud Computing (CLOUD)*, pages 887–894. IEEE.

Flissi, A., Dubus, J., Dolet, N., and Merle, P. (2008). Deploying on the Grid with DeployWare. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 177–184. IEEE.

Glaser, F., Erbel, J. M., and Grabowski, J. (2017). Model Driven Cloud Orchestration by Combining TOSCA and OCCI. In *7th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 644–650. SciTePress.

Medhioub, H., Msekni, B., and Zeghlache, D. (2013). OCNI – Open Cloud Networking Interface. In *22nd International Conference on Computer Communications and Networks (ICCCN)*, pages 1–8. IEEE.

Merle, P., Barais, O., Parpaillon, J., Plouzeau, N., and Tata, S. (2015). A Precise Metamodel for Open Cloud Computing Interface. In *8th IEEE International Conference on Cloud Computing (CLOUD)*, pages 852–859. IEEE.

Merle, P., Gourdin, C., and Mitton, N. (2017). Mobile Cloud Robotics as a Service with OCCIware. In *2nd IEEE International Congress on Internet of Things (ICIOT)*, pages 50–57. IEEE.

Metsch, T., Edmonds, A., and Parák, B. (2016). Open Cloud Computing Interface - Infrastructure. [Available online: http://ogf.org/documents/GFD.224.pdf].

Metsch, T. and Mohamed, M. (2016). Open Cloud Computing Interface - Platform. [Available online: https://www.ogf.org/documents/GFD.227.pdf].

Nyrén, R., Edmonds, A., Papaspyrou, A., Metsch, T., and Parák, B. (2016). Open Cloud Computing Interface - Core. [Available online: http://ogf.org/documents/GFD.221.pdf].

Paraiso, F., Challita, S., Al-Dhuraibi, Y., and Merle, P. (2016). Model-Driven Management of Docker Containers. In *9th IEEE International Conference on Cloud Computing (CLOUD)*, pages 718–725. IEEE.

Parpaillon, J., Merle, P., Barais, O., Dutoo, M., and Paraiso, F. (2015). OCCIware-A Formal and Tooled Framework for Managing Everything as a Service. In *Projects Showcase@ STAF'15*, volume 1400, pages 18–25.

Pham, L. M., Tchana, A., Donsez, D., De Palma, N., Zurczak, V., and Gibello, P.-Y. (2015). Roboconf: a Hybrid Cloud Orchestrator to Deploy Complex Applications. In *8th IEEE International Conference on Cloud Computing (CLOUD)*, pages 365–372. IEEE.

Quinton, C., Romero, D., and Duchien, L. (2016). SALOON: A Platform for Selecting and Configuring Cloud Environments. *Software: Practice and Experience*, 46(1):55–78.

Sellami, M., Yangui, S., Mohamed, M., and Tata, S. (2013). PaaS-independent Provisioning and management of applications in the cloud. In *6th IEEE International Conference on Cloud Computing (CLOUD)*, pages 693–700. IEEE.

Yangui, S. and Tata, S. (2013). CloudServ: PaaS resources provisioning for service-based applications. In *27th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 522–529. IEEE.

Yangui, S. and Tata, S. (2014). An OCCI Compliant Model for PaaS Resources Description and Provisioning. *The Computer Journal*, 59(3):308–324.

Zalila, F., Challita, S., and Merle, P. (2017). A Model-Driven Tool Chain for OCCI. In *25th International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS)*, pages 389–409. Springer, Cham.