A Model-Driven Tool Chain for OCCI

Faiez Zalila, Stéphanie Challita, Philippe Merle

A Model-Driven Tool Chain for OCCI

Faiez Zalila, Stéphanie Challita, and Philippe Merle

Inria Lille - Nord Europe & University of Lille
Email: firstname.lastname@inria.fr

Abstract. Open Cloud Computing Interface (OCCI) is the only open standard for managing any kinds of cloud resources, e.g., Infrastructure as a Service, Platform as a Service, and Software as a Service. However, no model-driven tooling exists to assist OCCI users in designing, editing, validating, generating, and managing OCCI artifacts (i.e., extensions that represent specific application domains and configurations that define running systems). In this paper, we propose the first model-driven tool chain for OCCI called OCCIware Studio. This tool chain is based on a metamodel defining the static semantics for the OCCI standard in Ecore and OCL. OCCIware Studio provides OCCI users facilities for designing, editing, validating, generating, and managing OCCI artifacts. We detail the tooled process to define an OCCI extension. In addition, we show how the cloud user can leverage the generated tooling for this extension to create his own OCCI configurations and manage them in the cloud. We illustrate our paper with the OCCI `Infrastructure` extension defining OCCI-compliant compute, network, and storage resources.

Keywords: Cloud computing, Service computing, Metamodeling, Software standards, Computer aided software engineering, Distributed information systems, Modeling environments

1 Introduction

Cloud computing has been adopted as the dominant delivery model for computing resources [3]. This model defines three well-discussed layers of services known as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [22]. Provisioning and managing these outsourced, on-demand, pay as you go, elastic resources require cloud resource management interfaces (CRM-API) [20]. However, there is a plethora of CRM-APIs, proposed by Amazon, Eucalyptus, Microsoft, Google, OpenNebula, CloudStack, OpenStack, CloudBees, OpenShift, Cloud Foundry, to name a few. Each API is based on different concepts and/or architectures. Therefore, provisioning and managing cloud resources are faced with four main issues: *heterogeneity* of cloud offers, *interoperability* between CRM-API, *integration* of CRM-API for building multi-cloud systems, and *portability* of cloud management applications.

To tackle these issues, Open Cloud Computing Interface (OCCI) defines the first and only open standard for managing any cloud resources [16]. OCCI provides a general purpose model for cloud computing resources and a RESTful

API for efficiently accessing and managing any kind of these resources. This will facilitate interoperability between clouds, as providers will be specified by the same resource-oriented model called the OCCI Core Model [33], that can be expanded through extensions and accessed by a common REST [18] API.

Today, only runtime frameworks such as erocci, rOCCI, pySSF, pyOCNI, and OCCI4Java are available, while OCCI designers/developers/users need software engineering tools to design, edit, validate, generate, and manage new kinds of OCCI resources, and the configurations of these resources. Nevertheless, OCCI lacks tools for modeling its extensions and configurations, despite the presence of a precise metamodel for OCCI [23] to be the reference of the implementation of such modeling tools.

To address this problem, the main contributions of this paper can be summarized as:

- The first model-driven tool chain for OCCI called OCCIware Studio which provides OCCI users with facilities for designing, editing, validating, generating, and managing OCCI artifacts.
- An enhanced metamodel for OCCI on which we build the OCCIware Studio. This new metamodel resolves several observed lacks in the previously proposed metamodel for OCCI [23] by introducing additional concepts such as *(i)* a mechanism to express business constraints, *(ii)* the Finite State Machine (FSM) concepts to define the behavior of OCCI resource types, *(iii)* a support for mixins, *(iv)* an own data type system for OCCI, and *(v)* a set of Ecore data types to assess the well-formedness of the OCCI artifacts.

Then, we detail the tooled process to define an OCCI extension. In addition, we show how the cloud user can leverage the generated tooling for this extension to create its own OCCI configurations and manage them in the cloud. To evaluate the efficiency of our OCCIware Studio, we elaborate a validation approach on the OCCI `Infrastructure` extension [26] defining OCCI-compliant compute, network and storage resources.

This paper is organized as follows. Section 2 explains the motivations behind our contribution. Section 3 presents our improvements made to the previous metamodel for OCCI. Then, we introduce the architecture of the OCCIware Studio and we detail the proposed approach to manage Everything as a Service with OCCI. Section 4 validates our proposed tool chain on the `Infrastructure` extension of OCCI. We position our work with related approaches in Section 5. Finally, Section 6 concludes on future work and perspectives.

2 Motivations

Today, multi-cloud computing is quite encouraged for cloud developers as a way to reduce vendor lock-in, to improve resiliency during outages and geo-presence, to boost performance and to lower costs. However, semantic differences between cloud providers, as well as their heterogeneous management interfaces, make changing from one provider to another very complex and costly. We assume for example that a developer would like to build a multi-cloud system spread over two clouds, Amazon Web Services (AWS) and Google Cloud Platform (GCP). AWS are accessible via a SOAP API, whereas GCP is based on a REST API, which leads to an incompatibility between these two different APIs. To use them, cloud consumers should be inline with the concepts and operations of each API, which is quite frustrating. The developer would like a single API for both clouds to seamlessly access their resources [9].

For this, OCCI is an open-source standard that defines a generic model for cloud resources and a RESTful API for efficiently accessing and managing resources. This will facilitate interoperability between clouds, as providers will be specified by the same resource model, and accessed by a common REST API. However, cloud developers cannot currently take advantage of this standard. Although there are several implementations of OCCI, there is no tool that allows the cloud developers to design and verify their configurations, neither to generate corresponding artifacts. This leads to several challenges:

1. Cloud users are focused on implementation details rather than cloud concerns, with the risk of misunderstandings for the concepts and the behavior that rely under cloud APIs.
2. The only way to be sure that the designed configurations will run correctly is to deploy them in the clouds. In this context, when errors occur, a correction is made and the deployment task can be repeated several times before it becomes operational. This is quite painful and expensive.
3. Cloud users need to provide various forms of documentation of their cloud configurations, as well as deployment artifacts. However, these tasks are complex and usually made in an ad-hoc manner with the effort of a human developer, which is error-prone and amplifies both development and time costs.

Recently, we are witnessing several works that take advantage of MDE for the cloud [8]. Therefore, to address the identified challenges, we believe that **there is a need for a model-driven tool chain for OCCI** in order to:

1. Enable both cloud architects and users to efficiently describe their needs at a high level of abstraction. This will be done by defining a metamodel accompanied with graphical and textual domain-specific modeling languages.
2. Allow cloud architects to define structural and behavioral constraints and validate them before any concrete deployments so they can a priori check the correctness of their systems.
3. Automatically generate *(i) textual documentation* to assist cloud architects, developers and users to understand the concepts and the behavior of the cloud API, and *(ii) HTTP scripts* that provision, modify or de-provision cloud resources.

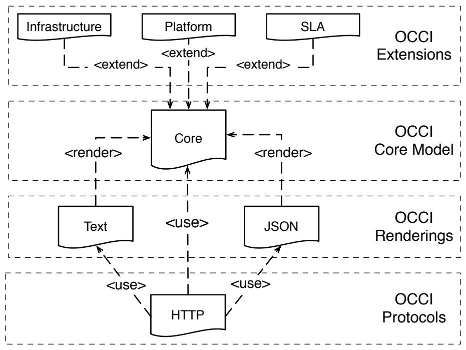


Fig. 1: OCCI Specifications

3 OCCIware Tool Chain

Designing is the key activity that must be addressed to resolve other encountered challenges such as verifying, generating, etc. Therefore, in order to assist OCCI users in modeling different OCCI artifacts, we propose a metamodel for OCCI named OCCIware Metamodel (cf. Section 3.3) extending the previous one described in Section 3.2. It defines the different concepts required to model OCCI extensions and configurations. In Section 3.4, we detail our approach to provide a tooled framework based on OCCI. It consists in mapping OCCIware Metamodel concepts into the chosen modeling framework, the Eclipse Modeling Framework (EMF). An overview of all the main OCCIware Studio features are described in Section 3.5. Section 3.6 presents our approach to manage Everything as a Service with OCCIware. It details the different interactions between the OCCIware Studio and stakeholders. Finally, Section 3.7 lists all the OCCI extensions currently provided with OCCIware Studio. But before that, let us start by introducing the OCCI standard.

3.1 OCCI

OCCI is an open cloud standard [16] specified by the Open Grid Forum (OGF). OCCI defines a RESTful Protocol and API for all kinds of management tasks on any kind of cloud resources, including Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). In order to be modular and extensible, OCCI is delivered as a set of specification documents divided into the four following categories as illustrated in Fig. 1:

– **OCCI Core Model**: It defines the OCCI Core specification [33] proposed as a RESTful-oriented model.
– **OCCI Protocols**: Each OCCI Protocol specification describes how a particular network protocol can be used to interact with the OCCI Core Model. Currently, only the OCCI HTTP Protocol [32] has been defined.

- **OCCI Renderings**: Each OCCI Rendering specification describes a particular rendering of the OCCI Core Model. Currently, both OCCI Text [15] and JSON [34] renderings have been defined.
- **OCCI Extensions**: Each OCCI Extension specification describes a particular extension of the OCCI Core Model for a specific application domain, and thus defines a set of domain-specific kinds and mixins. OCCI Infrastructure [26] is dedicated to IaaS. Additional OCCI extensions are defined such as OCCI Compute Resource Templates Profile (CRTP) [13], OCCI Platform [27] and OCCI Service Level Agreements [19].

3.2 The previous OCCIware Metamodel

Fig. 2 shows the OCCIware Metamodel. The entry point to define it was the OCCI Core Model whose main concepts are `Resource`, `Link`, `Kind`, `Mixin`, `Attribute`, and `Action`. `Resource` is the root abstraction of any cloud resource, such as a virtual machine, a network, and an application. `Link` represents a relation between two resources, such as a virtual machine connected to a network and an application hosted by a virtual machine. Each OCCI entity (resource or link) owns zero or more `Attribute`s, such as its unique identifier, the host name of a virtual machine, the Internet Protocol address of a network. As OCCI is a REST API, it gives access to cloud resources via classical CRUD operations (i.e., *Create*, *Retrieve*, *Update*, and *Delete*). In addition, each OCCI entity has zero or more `Action`s representing business specific behaviors, such as start/stop a virtual machine, and up/down a network. Each OCCI entity is strongly typed by a `Kind` and a set of `Mixin` instances. `Kind` represents the immutable type of OCCI entities and defines allowed attributes and actions. Single inheritance between `Kind`s allows us to factorize attributes and actions common to several kinds. `Mixin` represents cross-cutting attributes and actions that can be dynamically added to an OCCI entity. `Mixin` can be applied to zero or more kinds and can depend from zero or more other `Mixin` instances. The gray-colored classes in Fig. 2 show the OCCI Core Model. The latter can be interacted with over protocols/renderings and is expandable through extensions.

The OCCI Core Model does not explicitly define the notions of extension that represents a specific application domain and configuration that models a running system. For that, during a previous work [23], additional concepts, such as `Extension` and `Configuration`, are defined (the blue-colored classes in Fig. 2). This previous work has proposed the first metamodel for OCCI named, OCCIware Metamodel.

Definition 1. *`Extension` represents an OCCI extension, e.g., inter-cloud networking extension [21], infrastructure extension [26], platform extension [39,40,27], application extension [40], SLA negotiation and enforcement [14], cloud monitoring extension [10], and autonomic computing extension [30,31,28,29]. `Extension` has a `name`, has a `scheme`, owns zero or more `kinds`, owns zero or more `mixins`, owns zero or more `types`, and can `import` zero or more extensions.*

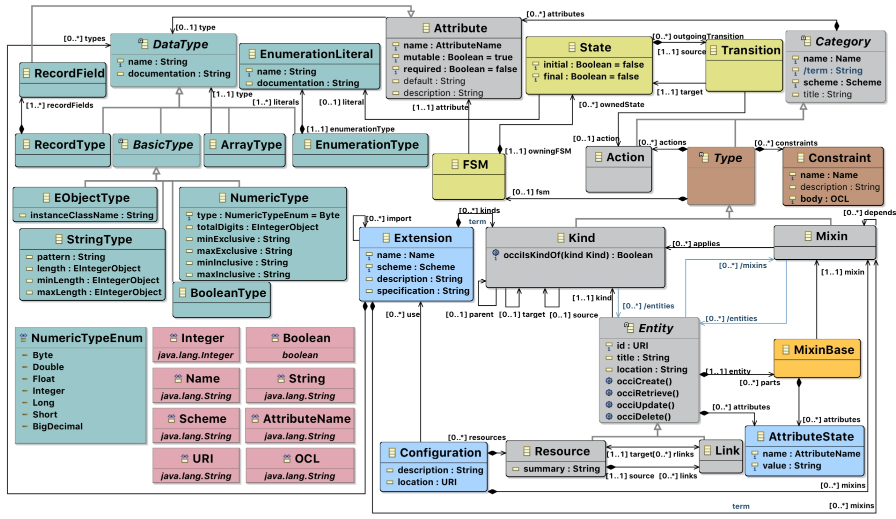


Fig. 2: Ecore diagram of OCCIware Metamodel

Definition 2. *`Configuration` represents a running OCCI system. `Configuration` owns zero or more `resources` (and transitively `links`), and `use` zero or more extensions. For a given configuration, the kind and mixins of all its entities (resources and links) must be defined by `used` extensions only. This avoids a configuration to transitively reference a type defined we do not know where.*

During the OCCIware research and development project[1], the proposed OCCIware Metamodel [23] was used to model various OCCI extensions: OCCI Infrastructure [26], OCCI Platform [27], OCCI SLA [19], OCCI Monitoring [11] and some additional proprietary extensions such as Docker [36] and simulation [1].

3.3 The enhanced OCCIware Metamodel

We propose to extend the first version of OCCIware Metamodel in order to resolve the lacks and the limitations encountered during the definition of the different OCCI artifacts.

At first, we propose to add a mechanism enabling to express constraints related to each cloud computing domain. In fact, each extension targets a concrete cloud computing domain, e.g., IaaS, PaaS, SaaS, pricing, etc. Therefore, there are certainly business constraints related to each domain which must be respected by configurations that use this extension. For example, all IP addresses of all network resources must be distinct. For that, we have extended this metamodel by adding the `Constraint` concept (the brown-colored classes of Fig. 2). A `Constraint` has a `name`, a `description` and a `body` that can be defined with

[1] `http://www.occiware.org`

Object Constraint Language (OCL) [35]. Each kind/mixin has zero or more `constraints` inherited from the new introduced abstract `Type` class.

Then, we have extended the OCCIware Metamodel by introducing a Finite State Machine (FSM) modeling language. It allows us to model the behavior of OCCI concepts such as state diagrams of OCCI `Kind` instances used in both the Infrastructure [26] and Platform [27] extensions. For that, as shown in the yellow-colored classes of Fig. 2, the FSM metamodel has three classes: `FSM`, `State`, and `Transition`. A `FSM` owns a set of states (`State`); a transition (`Transition`) is necessarily associated to a state. Additional concepts and rules related to the OCCI domain are added in the FSM metamodel such as: "a `FSM` is associated to a particular `attribute` of the concerned `Kind/Mixin` instance", "the identifier of a `State` is a literal (`EnumerationLiteral`)", "an `Action` is associated to a `Transition` instance", etc. In the following, we detail a subset of the static semantics of the FSM modeling language integrated in the OCCIware Metamodel.

Definition 3. *The `type` of a `FSM attribute` must be `EnumerationType`.*

```
context FSM
inv AttributeType: attribute.type.oclIsTypeOf(EnumerationType);
```

Definition 4. *The `enumerationType` of a `State literal` is equals to the `type` of the `attribute` of the owner `FSM` instance.*

```
context State
inv LiteralType: owningFSM.attribute.type=self.literal.enumerationType;
```

Definition 5. *The `action` of a `Transition` instance must belong to the `actions` of the owner `Type` instance.*

```
context Transition
inv ActionMustBeDefined : self.oclContainer().oclAsType(State).oclContainer().
    oclAsType(FSM).oclContainer().oclAsType(Type).actions->includes(self.action)
```

Thereafter, we have defined an additional class named `MixinBase` (the orange-colored class) which refers to a `mixin`. This concept allows us to instantiate the attributes of the referenced `mixin` outside the owner `entity` in order to separate the `entity` attributes from the `mixin` ones.

Next, we have defined an own data type system for OCCI as shown at the left part of Fig. 2 (the green-colored classes). This data type system allows us to define primitive types, as provided in the first version of the OCCIware Metamodel, such as `StringType` to model string types, `NumericType` to model numeric types and `BooleanType` to model boolean types. In addition, it allows to model a Java-based type using `EObjectType` and enumerations using `EnumerationType`. It provides also the capability to model complex types like `ArrayType` to design array types and `RecordType` to design structured types.

Finally, we have defined a set of Ecore data types (the red-colored classes) such as `URI`, `Scheme`, `Name`, and `AttributeName`, etc. These string-based types

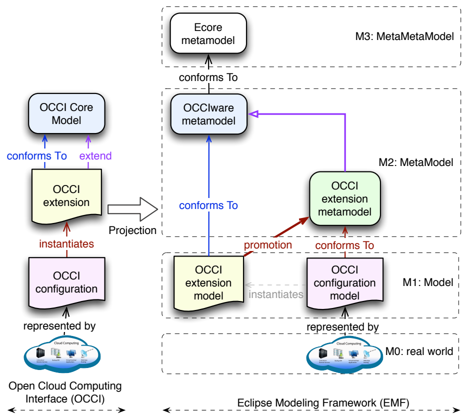


Fig. 3: Projection of OCCI to EMF

are enriched with the associated regular expression in order to ensure the correct values of different attributes. For example, the following rule "*Attribute names consist of alphanumeric characters separated by dots*" has been defined in the JSON rendering specification [34]. Accordingly, the user may define non-valid attribute names. Therefore, the bugs will be detected during the last steps of the modeling process and fixing them becomes a tricky task. For that, we have defined the following regular expression pattern for the `AttributeName` type: `value="[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)+"`.

Once the new OCCIware Metamodel is defined, we have tooled it with the EMF genmodel, the EMF model containing additional information related to the code generation, in order to generate the Java-based implementation.

3.4 Projection of OCCI to EMF

The OCCI Core Model is a simple resource-oriented model. It can be extended with several OCCI extensions.

As shown in the left part of Fig. 3, designing a new OCCI extension model consists in extending the OCCI `core` extension, the extension-like representation of the OCCI Core Model (as shown on the top part of Fig. 6). The OCCI `core` extension is composed of three kinds: a root `Entity` kind, and two children kinds: `Resource` and `Link`. Designing an OCCI configuration consists in defining an instance of an OCCI extension and represents a cloud architecture already deployed or to deploy.

The main goal of our work consists in introducing a tooled framework, based on OCCI, that manages any kind of resources as a service.

To do that, it was necessary to project different OCCI concepts into a modeling framework to benefit from the available facilities for building tools based on a metamodel (the right part of Fig. 3). To do that, EMF was chosen to embed OCCI and, thus, the OCCIware Metamodel was proposed as a precise metamodel for OCCI. Therefore, we can define either an OCCI extension model or an OCCI configuration model conform to the OCCIware Metamodel. However, the current tooling in EMF does not favor to encode that: an OCCI configuration is an *"instantiation"* of an OCCI extension. For that, we have proposed to promote the OCCI extension model by translating it into an Ecore metamodel, extending the OCCIware Metamodel. Consequently, the OCCI configuration model becomes an instance of this generated metamodel and, thus, an instance of the OCCIware Metamodel.

In the following, we detail the promotion process of OCCIware Metamodel concepts into the EMF concepts:

- Each OCCI kind instance is translated into an Ecore class. If its `parent` is the `Resource` kind, the generated class extends the `Resource` Ecore class of the OCCIware Metamodel. Otherwise, if its `parent` is the `Link` kind, the generated class extends the `Link` Ecore class of the OCCIware Metamodel.
- Each OCCI mixin instance is translated into an Ecore class extending the `MixinBase` class of the OCCIware Metamodel. Due to this added `MixinBase` class and this promotion rule, an instance of this generated class can now refer to the initial OCCI mixin instance.
- Each OCCI attribute instance, owned by an OCCI kind/mixin, is translated into an Ecore attribute owned by the corresponding generated Ecore class.
- Each OCCI action instance, owned by an OCCI kind/mixin, is translated into an Ecore operation owned by the corresponding generated Ecore class.
- Each OCCI constraint instance is translated into an OCL invariant.
- All Ecore data types defined in the OCCI extension are translated into the corresponding EMF concepts and/or types in the generated OCCI extension metamodel.

In order to ease the definition, edition, validation, and generation of different OCCI artifacts, we have benefited from the associated technologies to EMF in order to design and implement the OCCIware Studio.

3.5 OCCIware Studio Features

OCCIware Studio is a set of plugins for the Eclipse integrated development environment. Fig. 4 shows all the main features of OCCIware Studio:

- **OCCI Designer** is a graphical modeler to create, modify, and visualize both OCCI extensions and configurations. The OCCI standard does not define any standard notation for the graphical or textual concrete syntax. This tool is implemented on top of the Eclipse Sirius framework.
- **OCCI Editor** is a textual editor for both OCCI extensions and configurations. Our OCCI textual syntax is described in [25]. This tool is implemented on top of the Eclipse Xtext framework.
- **OCCI Validator** is a tool to validate both OCCI extensions and configurations. This tool checks all the constraints defined in the OCCIware Metamodel, *i.e.*, both Ecore and OCL ones. In addition, it checks the OCL invariants generated by the promotion process.

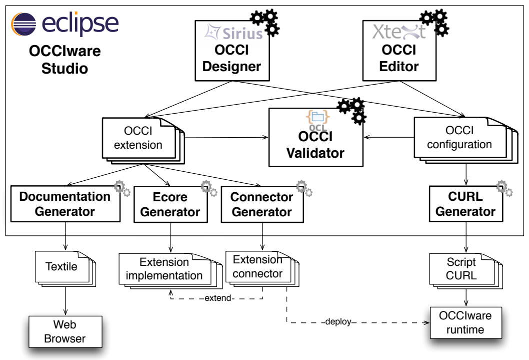


Fig. 4: OCCIware Studio features

- **Documentation Generator** is a tool to generate a Textile documentation from an OCCI extension model. Textile is a Wiki-like format used for instance by Github projects. This tool is implemented on top of the Eclipse Acceleo framework.
- **Ecore Generator** is a tool to generate the promoted Ecore metamodel and its associated Java-based implementation code from an OCCI extension. This tool is directly implemented in Java.
- **Connector Generator** is a tool to generate the OCCI connector implementation associated to an OCCI extension. This generated connector code extends the generated Ecore implementation code. 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 will be deployed on OCCIware Runtime [2]. This tool is implemented on top of the Eclipse Acceleo framework.
- **CURL Generator** is a tool to generate a curl-based script from an OCCI configuration model. These generated scripts contain HTTP requests to instantiate OCCI entities into any OCCI-compliant runtime. These scripts are used for offline deployment. This tool is implemented on top of the Eclipse Acceleo framework.

3.6 Managing Everything as a Service with OCCIware

OCCIware Studio proposes a tooled framework to manage any resource in the cloud. To benefit from it, a proposed process must be followed (cf. Fig. 5). This process contains two phases: the **Design phase** and the **Use phase**.

The **Design phase** consists in defining a new OCCI extension that extends the OCCI `core` extension and/or others OCCI extensions already defined. This phase contains three steps are shown in Fig. 5.

At first, an OCCIware architect designs his/her extension using both **OCCI Designer** and **OCCI Editor** tools (Step 1 in Fig. 5). Once the OCCI extension is defined, the generation process of the extension tooling may be triggered (Step 2 in Fig. 5). It proceeds in three steps:

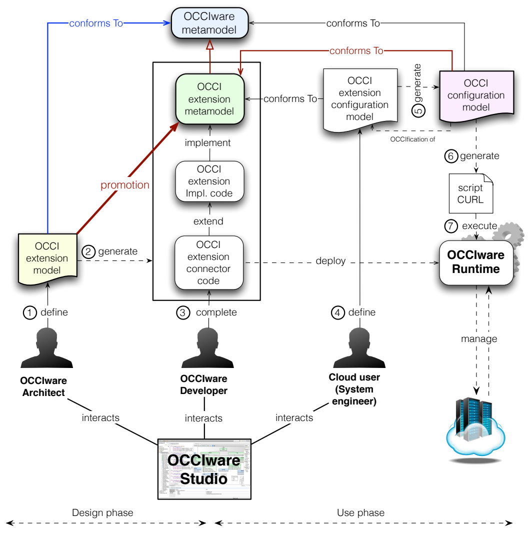


Fig. 5: OCCIWARE Approach to manage Everything as a Service

1. The **Ecore Generator** tool is used to generate the extension metamodel which extends the OCCIWARE METAMODEL because:
   – the generated classes from kinds, inheriting the `Resource` kind, will inherit the `Resource` Ecore class,
   – the generated classes from kinds, inheriting the `Link` kind, will inherit the `Link` Ecore class, and
   – the generated classes from mixins will inherit the `MixinBase` Ecore class.
2. Thereafter, the EMF tooling generates the Java-based implementation of the extension metamodel.
3. Finally, the **Connector Generator** tool is used to generate the implementation of a connector for each subclass of `Resource`, `Link` and `MixinBase`.

The connector code defines an implementation of the OCCI specific callback methods (the CRUD operations) and a pseudo-code for all kind/mixin-specific actions. The code of these actions is deducted from the defined FSM.

Once the generation step (Step 2 in Fig. 5) is achieved, the OCCIWARE developer can complete the generated connector classes (Step 3 in Fig. 5). It consists in implementing the business code for each connector class. The completed connector code must be deployed on the OCCIWARE RUNTIME [2], a full OCCI-compliant model-driven server.

From now on, we can consider that the **Design phase** is achieved and the OCCI extension is completely tooled and able to be used to manage conforming configurations.

The **Use phase** can now start. In fact, thanks to OCCIWARE STUDIO enriched with the extension tooling provided during the previous design step, users can design an extension configuration model conforms to the extension metamodel (Step 4 in Fig. 5). To benefit from the OCCI-compliant tools of OCCIware Studio, this extension-specific model must be translated into an OCCI configuration model (Step 5 in Fig. 5) conforms to the OCCIware Metamodel.

In order to deploy and manage the designed OCCI configuration, users interact with the cloud by sending OCCI HTTP requests to OCCI Runtime [2]. One possibility consists in generating CURL scripts (Step 6 in Fig. 5) using the **CURL Generator** tool. Then, the generated script is executed (Step 7 in Fig. 5) via the OCCIware Runtime [2] that invokes the `occiCreate()` method of each `Resource` class. This method implements how to create the considered `Resource` instance in the cloud. Finally, the created resource is deployed in the cloud.

3.7 Supported OCCI extensions

Each OCCI extension is implemented as an Eclipse modeling project containing one extension model, which is an instance of OCCIware Metamodel. Currently, OCCIware Studio supports the five OCCI extensions defined by the OGF's OCCI working group:

– **OCCI Infrastructure** [26] defines compute, storage and network resource types and associated links.
– **OCCI Compute Resource Templates Profile** [13] defines a set of preconfigured instances of the OCCI compute resource type.
– **OCCI Platform** [27] defines application and component resource types and associated links.
– **OCCI SLA** [19] defines OCCI types for modeling service level agreements.
– **OCCI Monitoring** [11] defines sensor and collector types for monitoring cloud systems.

4 OCCIware Approach Validation

In this section, we validate our approach by illustrating the proposed process on an OCCI extension with OCCIware Studio. We choose the `Infrastructure` extension. Then, we show how the cloud user leverages the generated tooling around this extension to create/manage his/her configuration models with OCCIware Studio and deploy them in the cloud.

4.1 Design process of OCCI `Infrastructure` extension

To design the `Infrastructure` extension, the OCCIware architect can use **OCCI Designer** and/or **OCCI Editor** tools (Step 1 in Fig. 5).

This extension defines five kinds (`Network`, `Compute`, `Storage`, `StorageLink` and `NetworkInterface`), six mixins (`Resource_Tpl`, `IpNetwork`, `Os_Tpl`, `SSH_key`, `User_Data`, and `IpNetworkInterface`), and around twenty data types (`Vlan` range, `Architecture` enumeration, various status enumerations, etc.), as shown in Fig. 6.

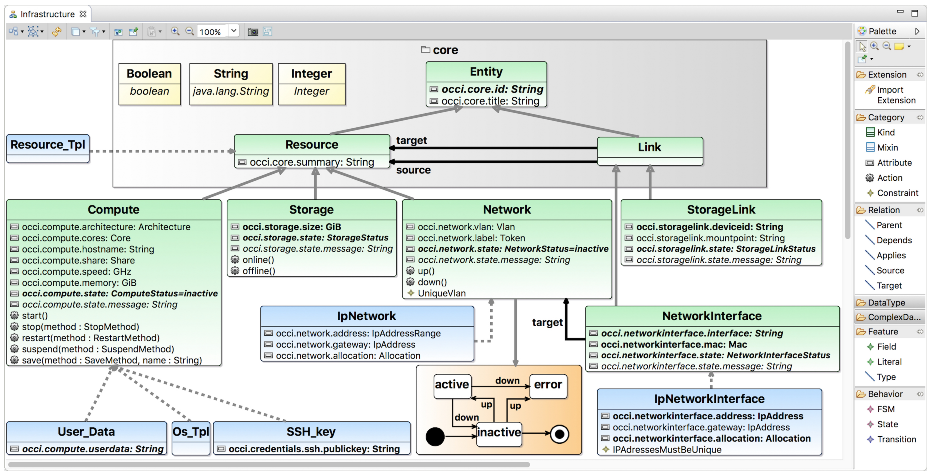

Infrastructure
100%
core
Boolean
boolean
String
java.lang.String
Integer
Integer
Entity
occi.core.id: String
occi.core.title: String
Resource_Tpl
Resource
occi.core.summary: String
target
source
Link
Compute
occi.compute.architecture: Architecture
occi.compute.cores: Core
occi.compute.hostname: String
occi.compute.share: Share
occi.compute.speed: GHz
occi.compute.memory: GiB
occi.compute.state: ComputeStatus=inactive
occi.compute.state.message: String
start()
stop(method : StopMethod)
restart(method : RestartMethod)
suspend(method : SuspendMethod)
save(method : SaveMethod, name : String)
Storage
occi.storage.size: GiB
occi.storage.state: StorageStatus
occi.storage.state.message: String
online()
offline()
Network
occi.network.vlan: Vlan
occi.network.label: Token
occi.network.state: NetworkStatus=inactive
occi.network.state.message: String
up()
down()
UniqueVlan
StorageLink
occi.storagelink.deviceid: String
occi.storagelink.mountpoint: String
occi.storagelink.state: StorageLinkStatus
occi.storagelink.state.message: String
IpNetwork
occi.network.address: IpAddressRange
occi.network.gateway: IpAddress
occi.network.allocation: Allocation
target
NetworkInterface
occi.networkinterface.interface: String
occi.networkinterface.mac: Mac
occi.networkinterface.state: NetworkInterfaceStatus
occi.networkinterface.state.message: String
active
down
error
up
up
down
inactive
IpNetworkInterface
occi.networkinterface.address: IpAddress
occi.networkinterface.gateway: IpAddress
occi.networkinterface.allocation: Allocation
IPAdressesMustBeUnique
User_Data
occi.compute.userdata: String
Os_Tpl
SSH_key
occi.credentials.ssh.publickey: String
Palette
Extension
Import Extension
Category
Kind
Mixin
Attribute
Action
Constraint
Relation
Parent
Depends
Applies
Source
Target
DataType
ComplexDa...
Feature
Field
Literal
Type
Behavior
FSM
State
Transition

Fig. 6: OCCI Infrastructure extension model

The `Compute` kind represents a generic information processing resource, e.g., a virtual machine or container. It inherits the `Resource` defined in the OCCI `core` extension. It has a set of OCCI attributes such as `occi.compute.architecture` to specify the CPU architecture of the instance, `occi.compute.core` to define the number of virtual CPU cores assigned to the instance, `occi.compute.memory` to define the maximum RAM in gigabytes allocated to the instance, etc. The `Compute` kind exposes five actions: `start`, `stop`, `restart`, `save` and `suspend`.

The `Network` kind is an interconnection resource and represents a Layer 2 (L2) networking resource. This is complemented by the `IpNetwork` mixin. It exposes two actions: `up` and `down`.

The orange-colored box in Fig. 6 illustrates the state diagram of a `Network` instance and describes its behavior. As shown previously, the new OCCIware Metamodel provides the required concepts to describe the behavior of each kind/mixin. In addition, it allows to define extension-specific constraints. For example, the following OCL constraint specifies that each `Network` instance must have a unique `vlan`.

```
inv UniqueVlan: Network.allInstances()->isUnique(occi.network.
    vlan)
```

In addition, we define, in the following, an additional OCL constraint in the `IpNetworkInterface` mixin which checks that all IP addresses must be different.

```
inv IPAddressesMustBeUnique: IpNetworkInterface.allInstances()
    ->isUnique(occi.networkinterface.address)
```

The `NetworkInterface` kind inherits the `Link` kind. It connects a `Compute` instance to a `Network` instance. The `Storage` kind represents data storage devices. The StorageLink kind inherits the Link kind. It connects a Compute instance to a Storage instance.

Once the Infrastructure extension is defined, the generation process of tooling may be triggered (Step 2 in Fig. 5). It generates three main elements: *(i)* the Infrastructure extension metamodel, *(ii)* the Java-based implementation of the Infrastructure extension metamodel, and *(iii)* the implementation of the Infrastructure connector.

Listing 1.1 shows a subset of the generated Network connector class. It extends the NetworkImpl class generated by the EMF tooling and contains the OCCI specific callback methods for the CRUD operations and all Network kind-specific actions (i.e., up and down). The generated code of specific actions is deducted from the defined FSM on the Network kind.

```
public class NetworkConnector extends NetworkImpl {
  NetworkConnector() {}
  // OCCI CRUD callback operations.
  public void occiCreate()   { /* TODO */ }
  ...
  // Network actions.
  ...
  public void down() {
    if(getState().equals(NetworkStatus.ACTIVE)) {
      if ( true ) {
        // TODO: Transition active -down-> inactive
        setState(NetworkStatus.INACTIVE);
      } else {
        // TODO: Transition active -down-> error
        setState(NetworkStatus.ERROR);
      }
    }
  }
}
```

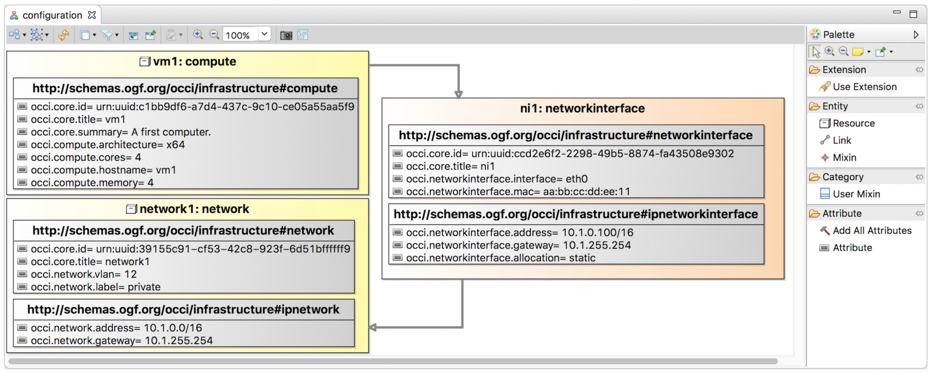
Listing 1.1: The generated Network connector class

Once the generation step is achieved, the OCCIware developer can complete the generated connector classes (Step 3 in Fig. 5) by updating their methods implementations (TODO sections in Listing 1.1) with business code related to targeted API. For the NetworkConnector class, the developer completes the code to trigger that the OCCI Network resource was created (occiCreate), will be retrieved (occiRetrieve), was updated (occiUpdate) and will be deleted (occiDelete). In addition, he/she completes the generated methods (up and down) related to specific actions defined in the Network kind. The completed connector code must be deployed on the OCCIware Runtime [2], a full OCCI-compliant model-driven server.

From now on, we can consider that the OCCI Infrastructure extension is completely tooled and able to be used to manage conforming configurations.

4.2 Use process of OCCI Infrastructure extension

Using OCCIware Studio enriched with the Infrastructure extension tooling, users can design an OCCI Infrastructure configuration model conforms to the Infrastructure extension metamodel (Step 4 in Fig. 5). To benefit from the OCCI-compliant tools defined in the OCCIware Studio, an Infrastructure



```
vm1: compute
http://schemas.ogf.org/occi/infrastructure#compute
occi.core.id= urn:uuid:c1bb9df6-a7d4-437c-9c10-ce05a55aa5f9
occi.core.title= vm1
occi.core.summary= A first computer.
occi.compute.architecture= x64
occi.compute.cores= 4
occi.compute.hostname= vm1
occi.compute.memory= 4

network1: network
http://schemas.ogf.org/occi/infrastructure#network
occi.core.id= urn:uuid:39155c91–cf53–42c8–923f–6d51bffffff9
occi.core.title= network1
occi.network.vlan= 12
occi.network.label= private
http://schemas.ogf.org/occi/infrastructure#ipnetwork
occi.network.address= 10.1.0.0/16
occi.network.gateway= 10.1.255.254

ni1: networkinterface
http://schemas.ogf.org/occi/infrastructure#networkinterface
occi.core.id= urn:uuid:ccd2e6f2-2298-49b5-8874-fa43508e9302
occi.core.title= ni1
occi.networkinterface.interface= eth0
occi.networkinterface.mac= aa:bb:cc:dd:ee:11
http://schemas.ogf.org/occi/infrastructure#ipnetworkinterface
occi.networkinterface.address= 10.1.0.100/16
occi.networkinterface.gateway= 10.1.255.254
occi.networkinterface.allocation= static
```

Fig. 7: An OCCI Infrastructure Configuration

configuration model must be translated into an OCCI configuration model (Step 5 in Fig. 5) conforms to the OCCIware Metamodel.

Fig. 7 illustrates a small IaaS configuration composed of a compute (`vm1`) connected to a network (`network1`), via an OCCI link (orange-colored box), the network interface (`ni1`). As this configuration uses an IP-based network, the `Network` resource and the `NetworkInterface` link have an `IpNetwork` and `IpNetworkInterface` mixin, respectively. Each OCCI entity is configured by its attributes, *e.g.*, `vm1` has the `vm1` hostname, an `x64`-based architecture, `4` cores, and `4` GiB of memory.

In order to deploy and manage the designed OCCI `Infrastructure` configuration, users interact with the cloud by sending OCCI HTTP requests to OCCIware Runtime [2]. These requests can be automatically generated as CURL scripts using the **CURL Generator** tool (Step 6 in Fig. 5).

```
OCCI_SERVER_URL=$1

curl $CURL_OPTS −X PUT
      $OCCI_SERVER_URL/network/39155c91−cf53−42c8−923f−6d51bffffff9
−H 'Content−Type: text/occi'
−H 'Category: network; scheme="http://schemas.ogf.org/occi/infrastructure#";
    class="kind";'
−H 'X−OCCI−Attribute: occi.core.id="39155c91−cf53−42c8−923f−6d51bffffff9"'
−H 'X−OCCI−Attribute: occi.core.title="network1"'
−H 'X−OCCI−Attribute: occi.network.vlan=12'
−H 'X−OCCI−Attribute: occi.network.label="private"'
−H 'X−OCCI−Attribute: occi.network.address="10.1.0.0/16"'
−H 'X−OCCI−Attribute: occi.network.gateway="10.1.255.254"'
```

Listing 1.2: The generated CURL script to create a `network` instance

Listing 1.2 shows the CURL script that requests OCCIware Runtime [2] via both OCCI HTTP Protocol [32] and OCCI Text Rendering [15] to create the `network1` instance. Then, OCCIware Runtime [2] invokes the `occiCreate()` method of the `NetworkConnector` class, which implements how to create the considered network instance in the cloud (Step 7 in Fig. 5). Finally, the created `Network` resource is deployed in the cloud.

In summary, thanks to OCCIware Studio, cloud architects and developers can design their OCCI extensions graphically via the **OCCI Designer** tool, validate their extensions via the **OCCI Validator** tool, generate a Web-based documentation automatically via the **Documentation Generator** tool, generate the promoted metamodel and its Java-based implementation via the **Ecore Generator** tool, generate the connector Java classes via the **Connector Generator** tool, and deploy their implemented extensions on OCCIware Runtime [2]. In addition, users can design their OCCI configurations graphically via the **OCCI Designer** tool, validate their configurations via the **OCCI Validator** tool, generate a deployment script via the **CURL Generator** tool, and manage their configurations at runtime via OCCIware Runtime [2].

5 Related work

In this section, we present some of the cloud metamodels, tools, and standards that were recently proposed and are relevant to our contribution.

CloudML [7][17] is a cloud modeling language that allows both cloud providers and developers to describe cloud services and application components, respectively. Then, it helps to provision cloud 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. In fact, the model at design-time is automatically handled by the Cloud Modeling Framework (CloudMF), which returns a runtime model of the provisioning resources, according to the Models@Run.Time approach [6]. CloudML is an inspiration source for future efforts in modeling the cloud, however, it lacks implementations. Unlike OCCIware which offers a set of tools to model, edit, validate, document, deploy, and manage any kind of cloud resources and their corresponding configurations, CloudML only provides a JSON and/or XML textual syntax to specify deployment and management concerns in IaaS and/or PaaS clouds.

SALOON [37] is an EMF graphical framework that is based on Extended Feature Models (EFMs) to represent clouds variability, as well as on ontology concepts to model the various semantics of cloud systems. It allows to translate these ontology concepts into a Constraint Satisfaction Problem (CSP) in order to select the adequate cloud environment. Although the authors state that SALOON supports the discovery and selection of multiple providers, in practice it does not. In the contrary, it selects one suitable provider at a time. SALOON targets ten IaaS/PaaS cloud environments. The objective of OCCI differs from SALOON; OCCI helps cloud users to overcome the differences between the resource kinds by defining a unified API, but it does not provide a way to select the suitable configuration.

CompatibleOne [38] is an open-source broker that offers services from several cloud providers and consists of a model called CompatibleOne Resource Description System (CORDS) and an execution platform called Advanced Capabilities for CORDS (ACCORDS). Similar to OCCIware Metamodel, CORDS is based on the OCCI standard and is an object based description of cloud applications, services, and resources. However, the authors in [38] defined this model to describe only IaaS and PaaS resources managed by their broker. ACCORDS allows to handle the user's requirements, validate, and execute the provisioning plan, and to deliver the cloud services. CompatibleOne does not support the graphical design of cloud configurations nor the generation of documentations, Ecore extensions, CURL scripts, etc.

Topology and Orchestration Specification for Cloud Applications (TOSCA) [5] is a language for describing the topology/structure/architecture of a cloud application, i.e., the software components that constitute the application, the physical or virtual nodes on which the components will be deployed, and the relationships between components and nodes. The TOSCA language is only defined as a textual XML or YAML document so it is complicated to have an overview of all its concepts. Several implementations of TOSCA were developed. For example, Winery[2] provides an open source Eclipse-based graphical modeling tool for TOSCA and the OpenTOSCA [4] project provides an open source container for deploying TOSCA-based applications, hence it is responsible for translating a TOSCA description into actions to be performed in clouds. These actions are sent to the clouds through their respective APIs. While both are standards, TOSCA provides a cloud application description language when OCCI provides a cloud resource management API.

Besides OCCI and TOSCA, several elaborated and mature cloud computing standards exist. For instance, the DTMF's Cloud Infrastructure Management Interface (CIMI) standard [12] defines a RESTful API for managing IaaS resources only. OCCI Infrastructure is concurrent to CIMI because both address IaaS resource management but OCCI has a more general purpose as it can be used also for any kind of PaaS and SaaS resources.

The OASIS's Cloud Application Management for Platforms (CAMP[3]) standard targets the deployment of cloud applications on top of PaaS resources. CAMP and TOSCA can use OCCI-based IaaS/PaaS resources, so these standards are complementary.

6 Conclusion

OCCI proposes a generic model and API for managing any kind of cloud computing resources. Unfortunately, it is obvious that leading cloud providers have no interest in adopting a standard API like the one offered by OCCI to facilitate interoperability with other clouds. However, OCCI has proven its utility in several contexts. For example, the European Grid Infrastructure Federated Cloud (EGI FC), which is a hybrid cloud, is based on OCCI Infrastructure to ensure interoperability among 20 cloud providers and over 300 data centers. Furthermore, OCCI attracts several cloud brokers such as CompatibleOne that aims at ensuring seamless access to the heterogeneous resources of cloud providers.

[2] `https://projects.eclipse.org/projects/soa.winery`

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

We argue in this paper that OCCI suffers from the lack of modeling, verification, validation, documentation, deployment, and management tools for both OCCI extensions and configurations. To address this issue, we propose OCCIware Studio, the first model-driven tool chain for OCCI. This tool chain is based on a metamodel defining the precise semantics of OCCI in Ecore and OCL. Our metamodel can be seen as a domain-specific modeling language to define and exchange OCCI extensions and configurations between end-users and resource providers. More precisely, thanks to OCCIware Studio, both cloud architects and users can 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, generate dedicated model-driven tooling via both **Ecore Generator** and **Connector Generator** tools, generate a deployment script via the **CURL Generator** tool, and manage their configurations at runtime via the generated connectors deployed in OCCIware Runtime [2]. OCCIware Studio is tightly integrated with the Java IDE, to facilitate the addition of functionality to the service skeletons and generates ready-to-deploy configurations for OCCI extensions. Our tool is validated by encoding and automatically verifying, provisioning, and managing the `Compute`, `Network` and `Storage` resources of the OCCI Infrastructure extension. OCCIware Studio also succeeded in managing mobile robots in [24], which proves the ability of our tool chain in managing any kind of resources. As for previous OCCI implementations, our OCCIware Studio can be easily integrated and used with them. For example, the CURL scripts generated by OCCIware Studio can be executed on OpenStack, OCCI4Java, etc.

In the future, we target industrial validation for OCCIware Studio. Therefore, an ongoing work aims to get this tool tested and adopted within Scalair[4]. We will also validate OCCIware Studio and our improved metamodel for OCCI on all the already published OCCI extensions. In order to cover the whole cloud market, we will also continuously enrich OCCIware Studio with new extensions such as AWS, GCP, OpenStack, etc. We would also like to provide dedicated studios for these OCCI extensions as components of OCCIware Studio, i.e., Docker Studio, GCP Studio, etc. This will ensure a specific environment for designing configurations that conform to each promoted extension metamodel.

Availability

Readers can find OCCIware Studio including OCCIware Metamodel and all the model-driven tools at `http://github.com/occiware/OCCI-Studio`. This work is supported by the OCCIware research and development project (`www.occiware.org`) funded by French Programme d'Investissements d'Avenir (PIA).

[4] `https://www.scalair.fr`

References

1. Ahmed-Nacer, M., Tata, S.: Simulation Extension for Cloud Standard OCCIware. In: 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE. pp. 263–264 (2016)
2. Alshabani, I., Parpaillon, J., Plouzeau, N., Gibello, P.Y., Tata, S.: OCCI Core Architecture. Deliverable D4.1.1, OCCIware Project (May 2015)
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A View of Cloud Computing. Communications of the ACM 53(4), 50–58 (2010)
4. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In: International Conference on Service-Oriented Computing, pp. 692–695. Springer (2013)
5. Binz, T., Breiter, G., Leyman, F., Spatzier, T.: Portable Cloud Services Using TOSCA. IEEE Internet Computing (3), 80–85 (2012)
6. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer 42(10), 22–27 (2009)
7. Brandtzæg, E., Mosser, S., Mohagheghi, P.: Towards CloudML, a Model-Based Approach to Provision Resources in the Clouds. In: 8th ECMFA. pp. 18–27 (2012)
8. Bruneliere, H., Cabot, J., Jouault, F.: Combining Model-Driven Engineering and Cloud Computing. In: 4th edition of Modeling, Design, and Analysis for the Service Cloud Workshop (MDA4ServiceCloud'10) (2010)
9. Challita, S., Paraiso, F., Merle, P.: Towards Formal-based Semantic Interoperability in Multi-clouds: The fclouds Framework. In: 10th IEEE International Conference on Cloud Computing (CLOUD). IEEE (2017)
10. Ciuffoletti, A.: A Simple and Generic Interface for a Cloud Monitoring Service. In: 4th International Conference on Cloud Computing and Services Science (CLOSER 2014). pp. 143–150
11. Ciuffoletti, A.: Open Cloud Computing Interface - Monitoring Extension. Specification Document 1.2, Open Grid Forum, OCCI-WG (Jan 2016)
12. Davis, D., Pilz, G.: Cloud Infrastructure Management Interface (CIMI) Model and REST Interface over HTTP. vol. DSP-0263 (May 2012)
13. Drescher, M., Parák, B., Wallom, D.: OCCI Compute Resource Templates Profile. Recommendation GFD-R-P.222, Open Grid Forum (Oct 2016)
14. Edmonds, A., Metsch, T., Papaspyrou, A.: Open Cloud Computing Interface in Data Management-Related Setups. In: Fiore, S., Aloisio, G. (eds.) Grid and Cloud Database Management, pp. 23–48. Springer (2011)
15. Edmonds, A., Metsch, T.: Open Cloud Computing Interface – Text Rendering. Recommendation GFD-R-P.229, Open Grid Forum (Oct 2016)
16. Edmonds, A., Metsch, T., Papaspyrou, A., Richardson, A.: Toward an Open Cloud Standard. IEEE Internet Computing 16(4), 15–25 (2012)
17. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-Cloud Systems. In: IEEE Sixth International Conference on Cloud Computing (CLOUD 2013). pp. 887–894
18. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
19. Katsaros, G.: Open Cloud Computing Interface – Service Level Agreements. Recommendation GFD-R-P.228, Open Grid Forum (Oct 2016)

20. Martin-Flatin, J.: Challenges in Cloud Management. IEEE Cloud Computing 1(1), 66–70 (2014)
21. Medhioub, H., Msekni, B., Zeghlache, D.: OCNI – Open Cloud Networking Interface. In: 22nd International Conference on Computer Communications and Networks (ICCCN). pp. 1–8. IEEE (2013)
22. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. NIST Special Publication 800(145) (Sep 2011)
23. Merle, P., Barais, O., Parpaillon, J., Plouzeau, N., Tata, S.: A Precise Metamodel for Open Cloud Computing Interface. In: Proceedings of the 8th IEEE International Conference on Cloud Computing (IEEE CLOUD 2015). pp. 852–859
24. Merle, P., Gourdin, C., Mitton, N.: Mobile Cloud Robotics as a Service with OCCIware. In: 2nd IEEE International Congress on Internet of Things (2017)
25. Merle, P., Parpaillon, J., Barais, O.: OCCI Specific Language - Structural Part. Deliverable D2.3.1, OCCIware Project (May 2015)
26. Metsch, T., Edmonds, A., Parák, B.: Open Cloud Computing Interface – Infrastructure. Recommendation GFD-R-P.224, Open Grid Forum (Oct 2016)
27. Metsch, T., Mohamed, M.: Open Cloud Computing Interface – Platform. Recommendation GFD-R-P.227, Open Grid Forum (Oct 2016)
28. Mohamed, M.: Generic Monitoring and Reconfiguration for Service-based Applications in the Cloud. Ph.D. thesis, INT, Evry, France (2014)
29. Mohamed, M., Amziani, M., Belaid, D., Tata, S., Melliti, T.: An autonomic approach to manage elasticity of business processes in the cloud. Future Generation Computer Systems pp. 49–61 (2015)
30. Mohamed, M., Belaïd, D., Tata, S.: Monitoring and Reconfiguration for OCCI Resources. In: 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2013). vol. 1, pp. 539–546. IEEE, Bristol, United Kingdom
31. Mohamed, M., Belaïd, D., Tata, S.: Autonomic Computing for OCCI Resources. Tech. rep., Telecom Sud Paris (Jan 2014), `http://www-inf.it-sudparis.eu/SIMBAD/tools/OCCI/autonomic/AutonomicComputingForOCCIResources.html`
32. Nyrén, R., Edmonds, A., Metsch, T., Parák, B.: Open Cloud Computing Interface – HTTP Protocol. Recommendation GFD-R-P.223, Open Grid Forum (Oct 2016)
33. Nyrén, R., Edmonds, A., Papaspyrou, A., Metsch, T., Parák, B.: Open Cloud Computing Interface – Core. Recommendation GFD-R-P.221, Open Grid Forum
34. Nyrén, R., Feldhaus, F., Parák, B., Sustr, Z.: Open Cloud Computing Interface – JSON Rendering. Recommendation GFD-R-P.226, Open Grid Forum (Oct 2016)
35. OMG: Object Constraint Language, Version 2.4. OMG Specification OMG Document Number: formal/2014-02-03, Object Management Group (Feb 2014)
36. Paraiso, F., Challita, S., Al-Dhuraibi, Y., Merle, P.: Model-Driven Management of Docker Containers. In: 9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016. pp. 718–725 (2016)
37. Quinton, C., Haderer, N., Rouvoy, R., Duchien, L.: Towards Multi-Cloud Configurations Using Feature Models and Ontologies. In: Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds. pp. 21–26
38. Yangui, S., Marshall, I.J., Laisne, J.P., Tata, S.: CompatibleOne: The Open Source Cloud Broker. Journal of Grid Computing 12(1), 93–109 (2014)
39. Yangui, S., Tata, S.: CloudServ: PaaS resources provisioning for service-based applications. In: 27th IEEE International Conference on Advanced Information Networking and Applications (AINA 2013). pp. 522–529. IEEE (2013)
40. Yangui, S., Tata, S.: An OCCI Compliant Model for PaaS Resources Description and Provisioning. The Computer Journal pp. 308–324 (2016)