Model-Driven Cloud Resource Management with OCCIware

Faiez Zalila, Stéphanie Challita, Philippe Merle

Available online at www.sciencedirect.com

ScienceDirect

Future Generation Computer Systems 00 (2019) 1–21

FGCS

Model-Driven Cloud Resource Management with OCCIware

Faiez Zalila, Stéphanie Challita, Philippe Merle

Inria Lille - Nord Europe & University of Lille

Abstract

Cloud computing has emerged as the main paradigm for hosting and delivering computing resources as services over Internet. It provides a delivery model for computing resources at infrastructure, platform, and software levels. However, there is a plethora of cloud providers offering different resource management interfaces. Due to this diversity, the consumption, provisioning, management, and supervision of cloud resources are subjected to four key issues, *i.e.*, heterogeneity, interoperability, integration, and portability. To cope with these issues, Open Cloud Computing Interface (OCCI) is proposed as a community-based and open recommendation standard for managing any kind of cloud resources. Currently, only runtime implementations exist for OCCI, and each one targets a specific cloud service model such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS). Thus, OCCI lacks an approach to model and execute different OCCI artifacts. Our approach provides a generic modeling framework coupled with a generic runtime implementation. In this article, we propose the OCCIware approach, which represents the first approach to design, validate, generate, implement, deploy, execute, and supervise everything as a service with OCCI. This approach provides OCCIware Studio, the first model-driven tool chain for OCCI. It is built around OCCIware Metamodel, which defines the static semantics for the OCCI standard in Ecore and OCL. In addition, it proposes OCCIware Runtime, the first generic OCCI runtime implementation targeting all the cloud service models (IaaS, PaaS, and SaaS). OCCIware provides a unique and unified framework to manage OCCI artifacts and, at the same time, it represents a factory to build cloud domain-specific modeling frameworks where each framework targets a specific cloud domain. OCCIware has been applied in various cloud domains and use-cases, which validate its applicability.

Keywords: Cloud computing, Service computing, Model-Driven Engineering (MDE), Meta modeling, Models@runtime, Software standards, Computer-aided software engineering, Distributed information systems, Modeling environments

1. Introduction

During the last decade, cloud computing becomes the preferable delivery model for computing resources [1]. This model provides three popular layers of services known as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [2]. A cloud resource management application programming interface (CRM-API) [3] allows cloud developers to provision and manage their outsourced, on-demand, pay as you go and elastic resources. However, there is a plethora of CRM-APIs proposed by Amazon, Microsoft, Google, IBM, Oracle, Eucalyptus, OpenNebula, CloudStack, OpenStack, CloudBees, OpenShift, Cloud Foundry, Docker, to cite a few. Each CRM-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-APIs, *integration* of CRM-API for building multi-cloud systems, and *portability* of cloud management applications.

Several cloud computing standards have been proposed to resolve these issues such as the DTMF's Cloud Infrastructure Management Interface (CIMI) [4] standard, which defines a RESTful [1] API for managing IaaS resources only, and the OASIS's Cloud Application Management for Platforms (CAMP[2]) [5] standard targets the deployment of cloud applications on top of PaaS resources. However, the main drawback of these standards is their specificity for a particular cloud service model, *i.e.*, IaaS or PaaS.

Open Cloud Computing Interface (OCCI) has been proposed as the first and only open standard for managing any cloud resources [6]. OCCI provides a general purpose model for cloud computing resources and a RESTful API for efficiently accessing and managing any kind of these cloud resources. This will ease interoperability between clouds, as providers will be specified by the same resource-oriented model called the OCCI Core Model [7], that can be expanded through extensions and accessed by a common REST [8] API.

[1]REpresentational State Transfer

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

Currently, only runtime frameworks such as rOCCI[3], erocci[4], pySSF[5], pyOCNI[6], and OCCI4Java[7] are available, while OCCI designers/developers/users need software engineering tools to design, edit, validate, generate, implement, deploy, execute, manage, configure, and supervise new kinds of OCCI resources, and the configurations of these resources. In addition, the existing runtime implementations are targeting only a specific cloud service model (mainly IaaS). Thus, OCCI lacks a unified modeling framework to design its different artifacts, and verify them during the initial steps of the design process before their effective deployment. Added to that, OCCI stakeholders need a generic runtime implementation coupled with the expected modeling framework in order to seamlessly execute the different developed and/or generated artifacts. Finally, as OCCI is proposed as an open generic standard to manage Everything as a Service (XaaS), OCCI stakeholders need to obtain a specific modeling framework for each domain.

To overcome the issues presented above, we propose in this article the OCCIware approach, which can be summarized as:

- **Model-Driven Managing Everything as a Service with OCCIware.** OCCIware is a model-driven vision to manage XaaS. It allows one to model any type of resources. It provides OCCI users with facilities for designing, editing, validating, generating, implementing, deploying, executing, managing, and supervising XaaS with OCCI.
- **Generating Cloud Domain-Specific Modeling Frameworks with OCCIware.** OCCIware is a factory of cloud domain-specific modeling frameworks. Each generated Cloud Domain-Specific Modeling Studio (CDSMS) is dedicated for a particular cloud domain. Each CDSMS can be used to design configurations which conform to its related domain and hides the generic concepts of OCCI.

The OCCIware approach is composed of two main components: OCCIware Studio and OCCIware Runtime. OCCIware Studio is the first model-driven tool chain for OCCI [9]. OCCIware Studio has been built around OCCIware Metamodel. This metamodel represents a precise definition for OCCI [10]. It defines rigourously the static semantics of the OCCI Core Model [7], the core specification of OCCI, by resolving several identified lacks and drawbacks. This metamodel is encoded using the Eclipse Modeling Framework (EMF) [11], and its static semantics is defined using the Object Constraint Language (OCL) [12]. The second component of OCCIware approach is OCCIware Runtime, a generic OCCI runtime implementation. OCCIware Runtime provides a unified RESTful API that conforms to OCCI Core Model and puts forward the Models@run.time approach [13].

This article extends our previous work [9]. In this extended version, we exhaustively detail the different concepts of OCCIware Metamodel by presenting the whole meta-classes and also the recently added OCL invariants. In addition, we present an additional use-case for the OCCIware approach which consists in using OCCIware Studio as a factory to generate dedicated studios for each cloud domain. In this direction, we enhance OCCIware Studio to support the generation of graphical designers, each one is dedicated to a specific cloud domain. Furthermore, we introduce the different generators and features recently integrated in OCCIware Studio such as the Alloy Generator to generate formal specification from OCCI extensions, the LaTeX Generator to generate later a portable documentation from OCCI extensions, the Designer Generator to generate a specific model-driven graphical designer for each OCCI extension, and the Runtime Connector, a synchronization tool between OCCI configurations and any OCCI-compliant runtime. In addition, we introduce OCCIware Runtime, a generic OCCI runtime implementation. Finally, we discuss and evaluates OCCIware approach with six use-cases that exploit it.

This article is organized as follows. Section 2 explains the motivations behind our contribution. Section 3 gives a background on the OCCI standard. Section 4 presents an overview of the OCCIware approach. It details the different processes to use OCCIware approach. In Section 5, we present the OCCIware Metamodel by detailing its different concepts implemented with Ecore. Section 6 provides an overview of OCCIware Studio and its different implemented features. Section 7 presents OCCIware Runtime and details its architecture. Section 8 validates our approach by discussing five major use cases implemented with OCCIware approach. Section 9 presents the learned lessons from designing and implementing a model-driven tool chain for OCCI. We position our work with related approaches in Section 10. Finally, Section 11 concludes with future work and perspectives.

2. Motivations

Currently, cloud architects and developers have a lot of hope for the multi-cloud computing paradigm as an alternative to avoid the vendor lock-in syndrome, to improve resiliency during outages, to provide geo-presence, to boost performance and to lower costs. However, semantic differences between cloud provider offerings, as well as their heterogeneous CRM-APIs make migrating from a particular provider to another a very complex and costly process. We assume for example that a cloud 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-based 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 cloud developer would like a single API for both clouds to seamlessly access their resources [14].

Therefore, OCCI is an open standard that defines a generic extensible model for any cloud resources and a RESTful API for efficiently accessing and managing cloud resources. This will facilitate interoperability between clouds, as cloud provider's

[3]`http://gwdg.github.io/rOCCI`
[4]`http://erocci.ow2.org`
[5]`https://github.com/tmetsch/pyssf`
[6]`https://github.com/jordan-developer/pyOCNI`
[7]`https://github.com/occi4java/occi4java`

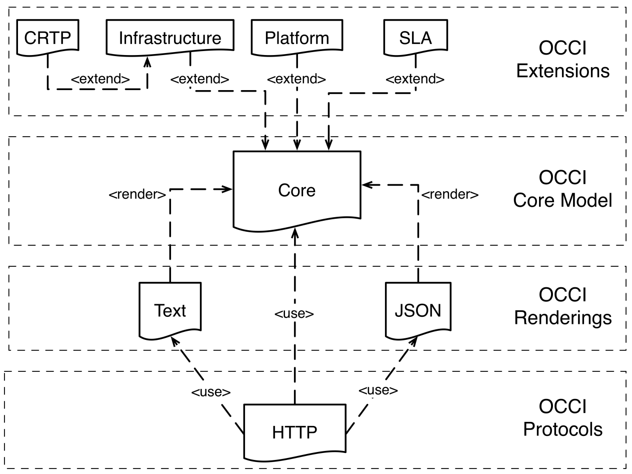


Figure 1: OCCI Specifications

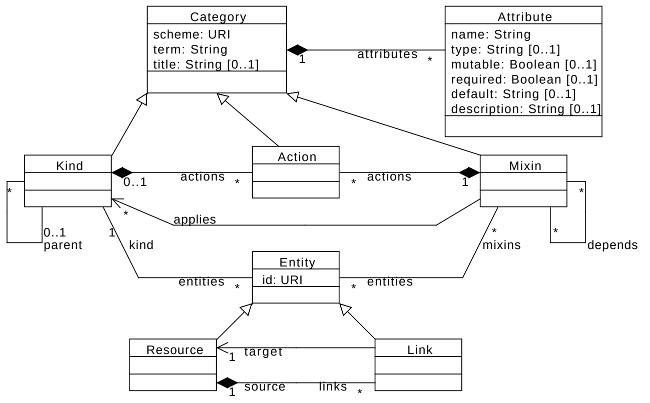


Figure 2: UML Class Diagram of the OCCI Core Model (from [7])

offerings 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 them to design and verify their configurations, neither to generate and deploy corresponding artifacts. This leads to several challenges:

1. Cloud architects, who are supposed to design the expected multi-cloud platform, are facing on one side to heterogeneity issues at different levels such as CRM-APIs heterogeneity (REST APIs versus SOAP APIs), delivery models heterogeneity (IaaS, PaaS, SaaS, etc.), deployment model heterogeneity (public, private and hybrid), and service providers heterogeneity. On the other side, cloud developers, who create and deploy running cloud systems, 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. They need a customized cloud framework dedicated to each cloud domain.

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 developers 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.

4. The CRM-APIs heterogeneity represents a major obstacle to seamlessly execute the deployment artifacts.

5. At the design level, the configuration represents a predefined architecture. However, the execution environment hosts a deployed system. A main challenge for the cloud developers is to provide a synchronization between the design level and the execution environment. When modifications occur in the predefined architecture, the update should be done in the executing environment. Conversely, when the deployed system changes, the modifications should be reflected in the predefined architecture.

Recently, we are witnessing several works that take advantage of MDE for the cloud [15, 16]. Therefore, to address the identified challenges, we believe that **there is a need for a tooled model-driven approach for OCCI** in order to:

1. Enable both cloud architects and developers to efficiently **design** their needs at a high-level of abstraction. This will be done by defining a metamodel, as a domain-specific modeling language (DSML), accompanied with graphical and textual concrete syntaxes. The expected DSML should be extensible in order to target different cloud domains.

2. Allow cloud architects to define structural and behavioral properties and **verify** them before any concrete deployments so they can a priori check the correctness of their cloud systems.

3. Automatically **generate** and **export** *(i)* textual documentations to assist cloud architects and developers to understand the concepts and the behavior of cloud-oriented APIs, *(ii)* specific designers dedicated to each cloud domain to assist cloud developers in the design of their configurations, *(iii)* formal specifications in order to formally **analyze** the different artifacts, and *(iv)* HTTP scripts that **deploy**, **provision**, **modify** or **de-provision** cloud resources.

4. **Execute** the generated scripts into a generic OCCI runtime implementation that must be able to host the developed connectors to concrete cloud resources.

5. **Discover** a configuration model by mapping a running cloud system into the expected modeling framework, **manage** this running cloud system via the configuration model

(for example, execute an action on the configuration model implies its execution on the running cloud system), and **bring back** the updates of the running system into the corresponding configuration model. These processes can be ensured via a connector between the cloud system and the modeling framework.

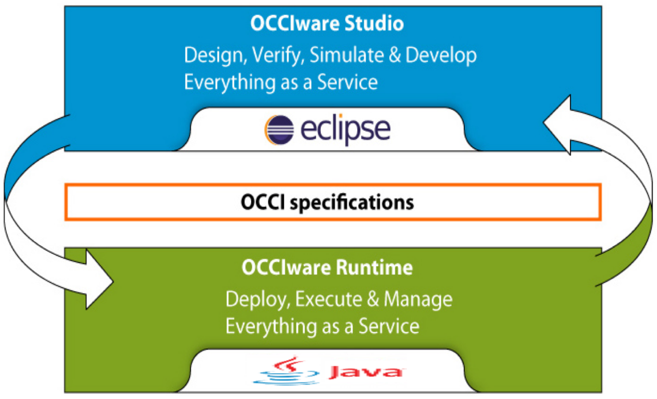


Figure 3: OCCIware Studio and OCCIware Runtime

3. Open Cloud Computing Interface (OCCI)

OCCI is an open cloud standard [6] 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 Figure 1:

OCCI Core Model. It defines the OCCI Core specification [7] proposed as a general purpose RESTful-oriented model. It is shown in Figure 2 and represented as a simple resource-oriented model composed of eight concepts: the heart of OCCI Core Model is the `Resource` type. It represents any cloud computing resource, such as a virtual machine, a network, and an application. A `Resource` owns a set of `links`. `Link` represents a relation between two resources, such as a virtual machine connected to a network and an application hosted by a virtual machine. A `Link` instance refers to both a `source` and `target` resource. `Entity` is an abstract concept. 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, using the `parent` relation between `Kinds`, 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 on zero or more other `Mixin` types. `Action` represents business specific behaviors, such as start/stop a virtual machine, and up/down a network, etc. `Category` is an abstract base class inherited by `Kind`, `Mixin` and `Action`. Each instance of kind, mixin or action is uniquely identified by both a `scheme` and a `term`, has also a human-readable `title`, a `description`, and owns a set of `attributes`. `Attribute` represents the definition of a customer-visible property, *e.g.*, the hostname of a machine, the IP address of a network, or a parameter of an action. An attribute has a `name`, can have a data `type`, can be (or not) `mutable` (*i.e.*, modifiable by customers), can be (or not) `required` (*i.e.*, value is provided at creation time), can have a `default` value and a human-readable `description`.

OCCI Protocols. Each OCCI Protocol specification describes how a particular network protocol can be used to interact with the OCCI Core Model. Multiple protocols can interact with the same instance of the OCCI Core Model. Currently, only the OCCI HTTP Protocol [17] has been defined. Other OCCI protocols would be proposed in the future such as AMQP[8].

OCCI Renderings. Each OCCI Rendering specification describes a particular rendering of the OCCI Core Model. Multiple renderings can interact with the same instance of the OCCI Core Model and will automatically support any OCCI extension. Currently, both OCCI Text [18] and JSON[9] [19] renderings have been defined. Other OCCI renderings would be specified in the future, such as an XML rendering for instance.

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 [20] is dedicated to IaaS. This extension provides compute, storage and network resources as services. Additional OCCI extensions are defined such as OCCI Compute Resource Templates Profile (CRTP) [21], OCCI Platform [22] and OCCI Service Level Agreements (SLA) [23]. Other OCCI extensions would be specified in the future, such as an OCCI monitoring extension [24].

4. OCCIware Approach

4.1. Managing Everything as a Service with OCCIware

The OCCIware funded project[10] [25] aims to provide a formal comprehensive, coherent, modular, model-driven tool chain for managing any kind of cloud computing resources. The OCCIware approach relies on model-driven engineering (MDE), a software engineering paradigm that proposes to reason on high-level artifacts, called *models*, rather than the code implementation. As MDE allows us to raise the level of abstraction, a *model* is an abstract representation of a system. It allows us to understand the designed system and answer the related queries. A model conforms to a *metamodel*, which defines the modeling language. OCCIware approach is composed of two main components: *(i)* OCCIware Studio, and *(ii)* OCCIware Runtime as depicted in Figure 3.

[8] Advanced Message Queuing Protocol
[9] JavaScript Object Notation
[10] `http://www.occiware.org`

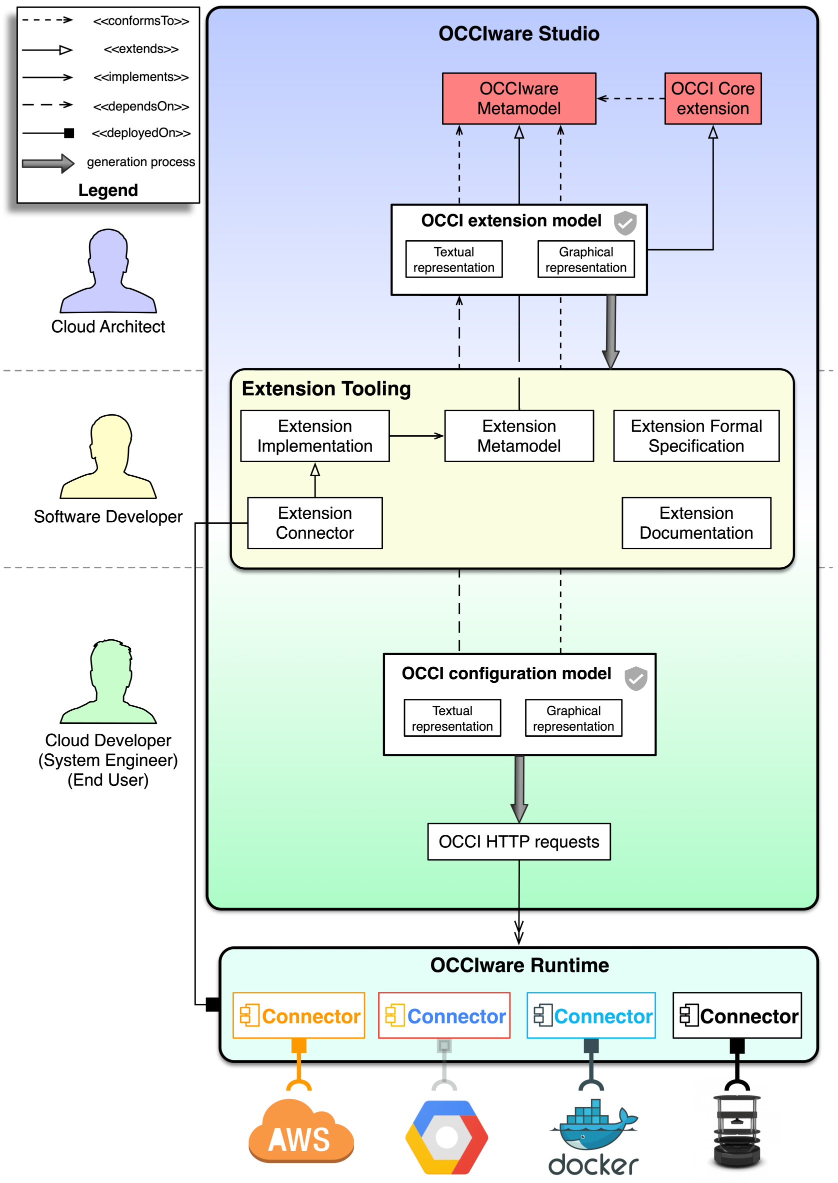


Figure 4: Model-Driven Managing Everything as a Service with OCCIware

OCCIware Studio, detailed in Section 6, is an OCCI model-driven tool chain that enables to design, verify, simulate, and develop every kind of resources as a service. Usually, a model-driven approach is based on, at least, a metamodel. The OCCIware approach is designed and developed based on a metamodel called OCCIware Metamodel and detailed in Section 5. This metamodel implements and extends the OCCI Core Model.

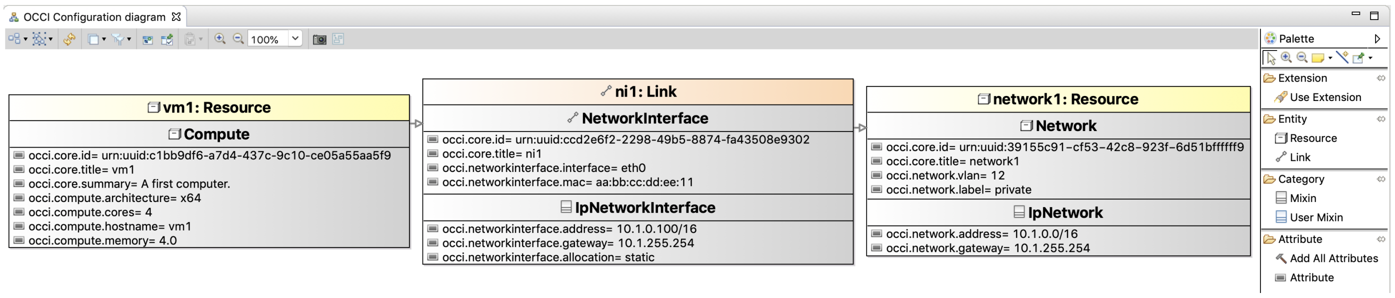
OCCIware Runtime, detailed in Section 7, is a generic OCCI-compliant models@run.time support and includes a cloud resource container, and tools for deployment, execution, and supervision of XaaS.

To benefit from OCCIware approach, a proposed process must be followed (cf. Figure 4). This process has two steps: the *design step* and the *use step*.

4.1.1. *Design step*

The *design step* (the top of Figure 4) consists in defining a new OCCI extension that extends the OCCI `Core` extension (an extension-like representation of the OCCI Core Model), and/or other OCCI extensions already defined. This step is ensured by the *Cloud Architect* who aims to have a tooled model-driven framework for his/her cloud domain such as infrastructure and platform. An OCCI extension model conforms to OCCIware Metamodel. It can be designed textually and/or graphically. Once the extension is designed and validated, a generation process of the `Extension Tooling` may be triggered. It consists to generate, from an OCCI extension model, a set of artifacts, which meet the needs of the cloud developers. They can be summarized as:

1. **Extension Documentation** represents a comprehensive



vm1: Resource

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= 4.0
```

ni1: Link

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
```

IpNetworkInterface

```
occi.networkinterface.address= 10.1.0.100/16
occi.networkinterface.gateway= 10.1.255.254
occi.networkinterface.allocation= static
```

network1: Resource

Network

```
occi.core.id= urn:uuid:39155c91-cf53-42c8-923f-6d51bffffff9
occi.core.title= network1
occi.network.vlan= 12
occi.network.label= private
```

IpNetwork

```
occi.network.address= 10.1.0.0/16
occi.network.gateway= 10.1.255.254
```

Figure 5: An OCCI Configuration Model

documentation of the designed extension. It serves as the reference document to describe for the cloud developer the different notions designed in the extension.

2. **Extension Formal Specification** defines formally the specification of the designed extension. This artifact can be later analyzed using a dedicated tool to check rigorously its correctness and its conformance to the OCCI specifications.

3. **Extension Metamodel** is a modeling language dedicated to the domain of the designed extension. It allows us to design conforming models representing running systems of this domain. This metamodel extends OCCIware Metamodel.

4. **Extension Implementation** represents a concrete implementation of the generated `Extension Metamodel`. It should provide an implementation for each concept of the designed extension.

5. **Extension Connector** extends the `Extension Implementation` and represents a skeleton of the causal link between designed models and running cloud resources. For a designed extension, this module represents the bridge between both OCCIware Studio/Runtime and the running cloud systems.

Once the different artifacts are generated, the **Software Developer** can complete the generated `Extension Connector`. It consists of implementing the business code required to handle each concept of the extension. Later, the completed connector must be deployed on OCCIware Runtime. The completed `Extension Connector` will be responsible in maintaining the synchronization between the designed configurations and running ones on the OCCIware Runtime. From now on, we can consider that the `Extension Tooling` is able to be used to manage conforming configurations.

4.1.2. *Use step*

Thanks to OCCIware Studio enriched with the `Extension Tooling` generated during the previous step, the cloud developer, who is the **System Engineer** (end user from our perspective) can design an OCCI configuration model conforms to OCCIware Metamodel. Figure 5 shows a small OCCI infrastructure configuration composed of a compute resource (`vm1`) connected to a network (`network1`), via an OCCI link, the network interface (`ni1`). 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 this generated configuration, the cloud developer can interact with the cloud by sending OCCI HTTP requests to OCCIware Runtime extended with the deployed `Extension Connector`. These scripts can be generated from the OCCI configuration model. To execute them, OCCIware Runtime invokes the appropriate `Extension Connector` to create the instance. Finally, the created resource is deployed in the cloud.

4.2. *Generating Cloud Domain-Specific Modeling Studios with OCCIware*

As previously explained, OCCIware approach provides a set of tools to design, edit, validate, generate, and manage OCCI artifacts. Concretely, the main goal of OCCIware Studio consists in designing, at the end, a correct OCCI configuration model conforms to OCCIware Metamodel. Moreover, the ultimate goal for the cloud developer, the end user of OCCIware approach, consists in executing this model that represents an eventual running system in the cloud. In OCCIware approach, executing a configuration model invokes the OCCIware Runtime to create the different designed entities. In the model-driven development, two strategies to execute models are possible [26]: *Code Generation* and *Model Interpretation*.

Code Generation targets to produce running artifacts (a script, a piece of code, etc.) from a high-level model. It is similar to the compilation that produces executable binary files from a source code. Usually, the generated artifact is produced in a standard language that any developer can understand. In addition, the code generation strategy allows us to easily link a model-driven framework to existing tools and methods such as model-checkers, simulators and runtime environments.

Model Interpretation approach consists of parsing and executing the model on the fly, with an interpretation approach and using a generic engine. A major advantage of this approach is the capability to change the model at runtime without stopping the running application because the interpreter would continue the execution by parsing the new version of the model.

In the OCCIware approach, as shown on the left part of Figure 6, both strategies have been implemented. Code generation process allows us to integrate a generator of HTTP requests

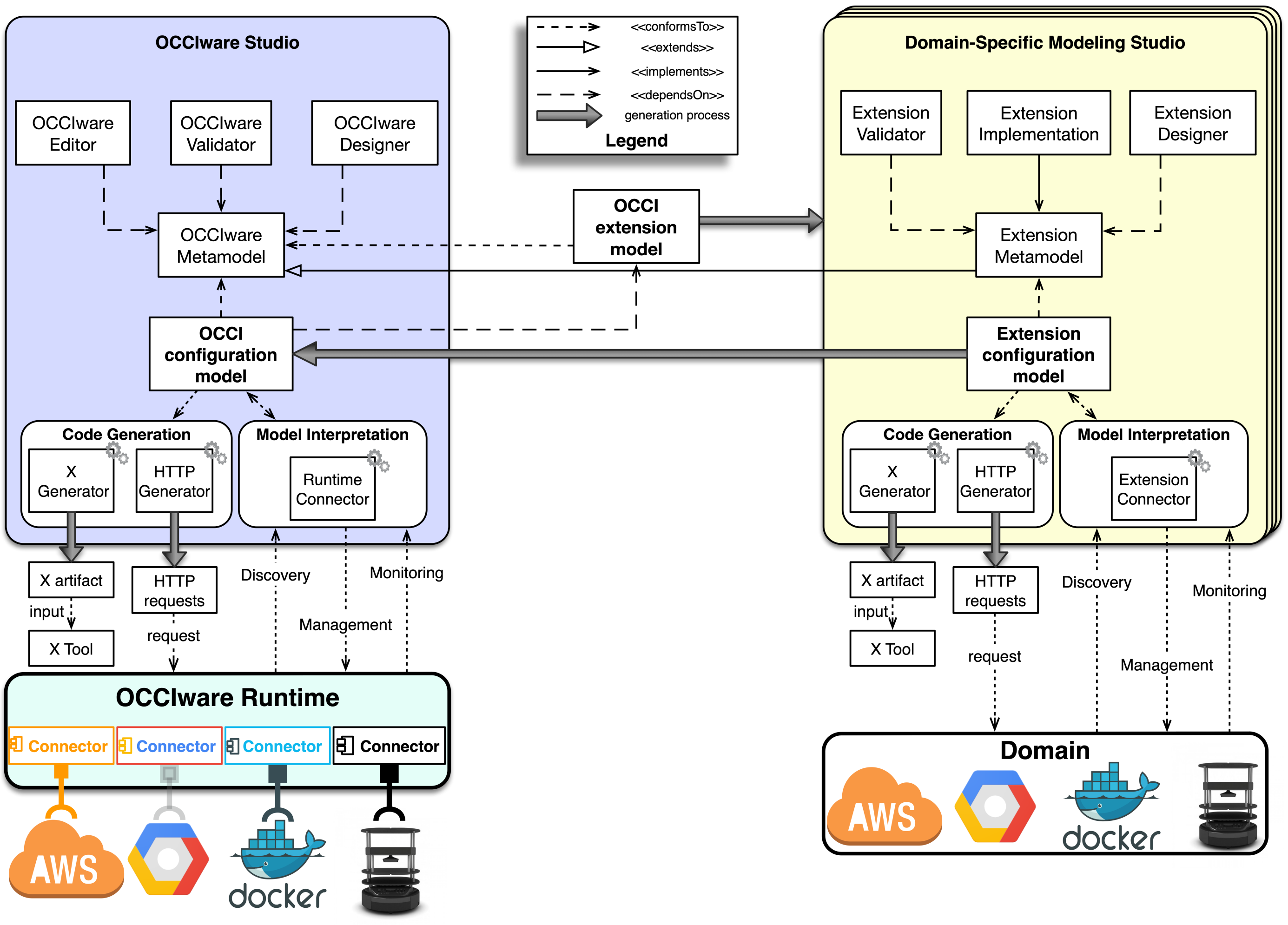


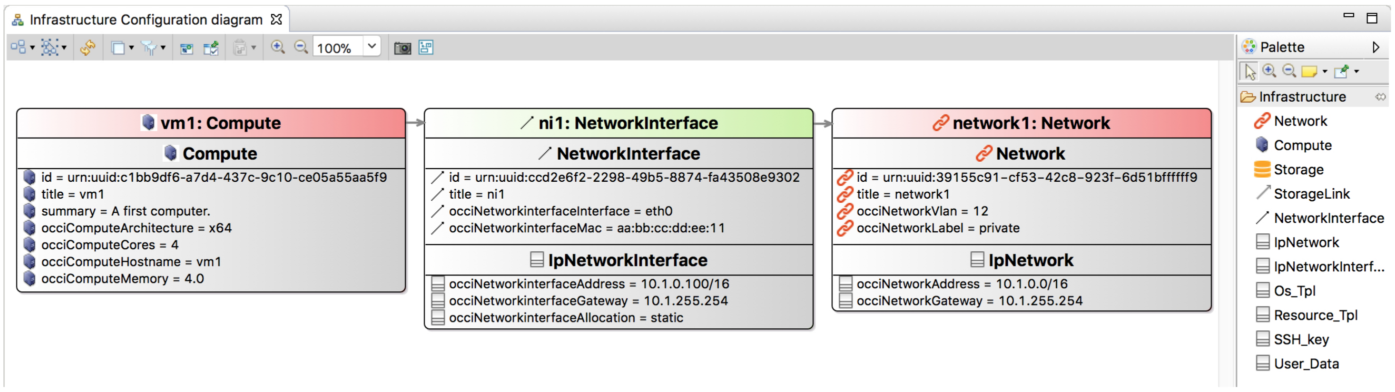
Figure 6: Generating Cloud Domain-Specific Modeling Studios with OCCIware

from OCCI configuration models. These requests can be later sent to OCCIware Runtime to create and deploy OCCI entities. The extensibility of OCCIware Studio permits software engineers to implement additional generators targeting other existing tools for other purposes such as the generation of deployment plans. Model interpretation is implemented by defining a `Runtime Connector`. Using this connector, we can *(i)* discover a configuration model by mapping a running system from OCCIware Runtime to OCCIware Studio, *(ii)* edit the obtained configuration model, *(iii)* send these modifications to the running system, and finally *(iv)* bring back the changes triggered by the runtime to the model.

OCCIware Studio represents the first model-driven framework to design OCCI artifacts. In addition, thanks to the different proposed generators, OCCIware Studio can be considered as a factory to build cloud domain-specific modeling studios, each one is specific to a particular cloud domain. As shown on the right part of Figure 6, once an OCCI extension model is defined, we can proceed to the generation of a Cloud Domain-Specific Modeling Studio (CDSMS) dedicated to a particular cloud domain. This specific studio provides *(i)* an `Extension Metamodel`, *(ii)* an `Extension Validator`, *(iii)* an `Extension Implementation`, and *(iv)* an `Extension Designer`. The latter is a dedicated model-driven graphical designer to the cloud domain of the extension. It gives a suitable framework for the cloud developer to graphically design the different resources of a running system, by instantiating the extension concepts and not OCCI concepts. For instance, Figure 7 shows an `Infrastructure` configuration model. This configuration corresponds to the OCCI configuration model shown in Figure 5. However, the palette of the `Infrastructure` Designer (right part of Figure 7) allows cloud developers to create an instance of `Infrastructure Metamodel` such as `Compute`, `Network`, and `Storage`. While in the palette of the OCCIware Designer (right part of Figure 5), the cloud developer can only create instances of OCCIware Metamodel, i.e., `Resource` and `Link` classes.

This approach allows us to obtain multiple CDSMSs (as shown on the right part of Figure 6) and each one targets a particular cloud domain/provider, i.e., a CDSMS for Amazon Web Services (AWS), a CDSMS for Docker containers, etc.

These generated tools allow the cloud developer to design configurations conform to a specific cloud domain. As OCCIware Studio, the specific studio can support both strategies to execute the designed `Extension` configuration model. The



vm1: Compute

Compute

```
id = urn:uuid:c1bb9df6-a7d4-437c-9c10-ce05a55aa5f9
title = vm1
summary = A first computer.
occiComputeArchitecture = x64
occiComputeCores = 4
occiComputeHostname = vm1
occiComputeMemory = 4.0
```

ni1: NetworkInterface

NetworkInterface

```
id = urn:uuid:ccd2e6f2-2298-49b5-8874-fa43508e9302
title = ni1
occiNetworkinterfaceInterface = eth0
occiNetworkinterfaceMac = aa:bb:cc:dd:ee:11
```

IpNetworkInterface

```
occiNetworkinterfaceAddress = 10.1.0.100/16
occiNetworkinterfaceGateway = 10.1.255.254
occiNetworkinterfaceAllocation = static
```

network1: Network

Network

```
id = urn:uuid:39155c91-cf53-42c8-923f-6d51bffffff9
title = network1
occiNetworkVlan = 12
occiNetworkLabel = private
```

IpNetwork

```
occiNetworkAddress = 10.1.0.0/16
occiNetworkGateway = 10.1.255.254
```

Figure 7: An Infrastructure Configuration Model

software developer completes the `Extension Connector` with the implementation related to a particular cloud API. In addition, he/she can implement several generators to generate specific artifacts for his/her cloud domain. If the cloud developer needs to come back and benefit from both OCCIware Studio tooling and OCCIware Runtime, it is always possible. A bridge from the `Extension` configuration models to OCCI configuration models has been provided.

Next sections provide more details on OCCIware Metamodel, OCCIware Studio, and OCCIware Runtime before discussing the several use cases validating the OCCIware approach.

5. OCCIware Metamodel

Designing is the key activity that must, at first, be addressed to later resolve other encountered challenges such as verifying, generating, and deploying artifacts. Therefore, in order to assist OCCI users in modeling different OCCI artifacts, we started in our previous works [10, 9] proposing a metamodel for OCCI named OCCIware Metamodel[11] shown in Figure 8. OCCIware Metamodel is composed of four main subsets of concepts: OCCI Core Model [7] concepts, OCCIware well-formedness concepts, concern-oriented concepts, and typing concepts. In the following, we detail these different subsets. After that, we will illustrate the OCCIware Metamodel with OCCI Infrastructure extension dedicated to the IaaS.

5.1. *OCCI Core Model [7] concepts*

This subset corresponds to the eight concepts of the OCCI standard, the gray-colored classes, i.e., `Resource`, `Link`, `Entity`, `Kind`, `Mixin`, `Action`, `Category`, and `Attribute`. These concepts are already presented in Section 3. We have improved these concepts to deal with the drawbacks previously identified in [10]. For example, each OCCI entity (resource or link) owns zero or more `attributes`, 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`). Finally, an entity owns zero or more instantiated mixins (`parts`).

5.2. *OCCIware well-formedness concepts*

Among the different drawbacks of the OCCI Core Model identified in [10], two ones point out the lack of concepts to design well-formed OCCIware artifacts. To resolve this issue, we introduced two new concepts (the blue-colored classes), i.e., `Extension` and `Configuration`. `Extension` represents an OCCI extension, *e.g.*, inter-cloud networking extension [27], infrastructure extension [20], platform extension [28, 29, 22], application extension [29], SLA negotiation and enforcement [30], cloud monitoring extension [31], and autonomic computing extension [32, 33, 34, 35]. `Extension` has a `name`, a `scheme`, a `description`, a `specification`, owns zero or more `kinds`, zero or more `mixins`, zero or more data `types`, and can `import` zero or more extensions. Each designed extension must, at least, extend the OCCI `Core` extension, the extension-like representation of the OCCI Core Model. The OCCI `Core` extension is composed of three kinds: a root `Entity` kind, and two children kinds, `Resource` and `Link`. `Configuration` is an added concept to represent a running OCCI system. A `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 only be defined by the `used` extensions. As a consequence, a configuration will not transitively reference an unknown type.

At this stage, no concept allows to instantiate mixins and attributes inside a configuration model. To resolve this issue, we introduce two new concepts, the orange-colored classes, i.e., `AttributeState` and `MixinBase`. `AttributeState` allows to represent an instantiated OCCI attribute. An `AttributeState` instance has a `name` and a `value`. `MixinBase` is an added concept strongly typed by a `mixin`. It represents an instantiated mixin and can own zero or more `attributes`.

5.3. *Concern-oriented concepts*

During the use of OCCIware Metamodel, several needs were encountered to deal with different modeling facets such as expressing structural business constraints, designing entity

[11] Available here `https://github.com/occiware/OCCI-Studio/blob/master/plugins/org.eclipse.cmf.occi.core/model/OCCI.ecore`

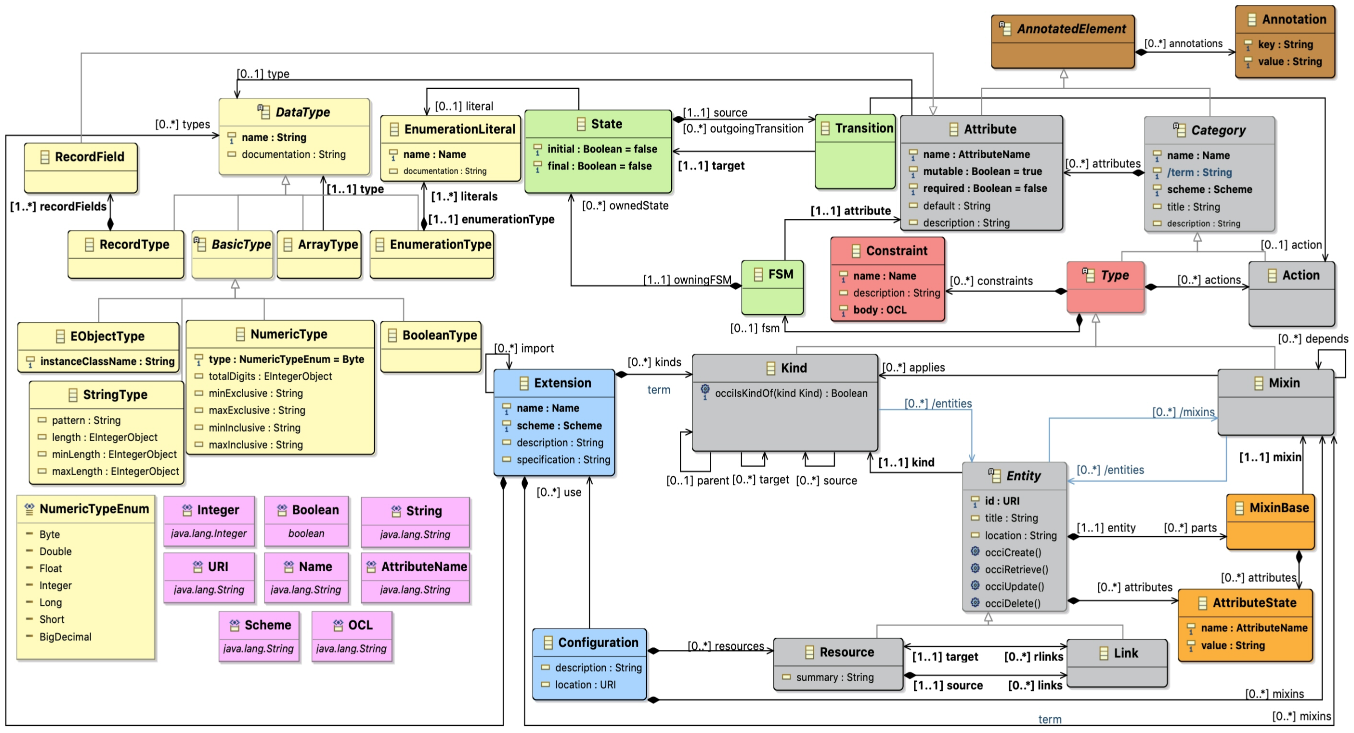


Figure 8: Ecore diagram of OCCIware Metamodel

behavior, etc. We addressed these different concerns in OCCIware Metamodel.

The first concern is the structural business constraints. In fact, each designed 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 the extension. For example, all IP addresses of all network resources must be distinct. To achieve this, we introduce two new concepts, the red-colored classes, i.e., Type and Constraint. Type is an added concept to represent an abstract type inherited by Kind and Mixin classes. Each type can own zero or more constraints. Constraint is a new concept to represent a detailed aspect related to a particular cloud computing domain. A Constraint has a name, a description and a body that can be defined with Object Constraint Language (OCL) [12].

The second concern is the behavior of entities. OCCI specifications provide a set of state diagrams. Each one defines the behavior of a particular resource kind or a mixin. Currently, no mechanism exists to design that. To resolve this issue, we introduce three new concepts, the green-colored classes, i.e., FSM (Finite State Machine), Transition and State. FSM allows us to design the behavior of an OCCI kind/mixin. It owns a set of states. State defines the state of a kind/mixin instance. It can be an initial and/or a final one. It has a literal and can own a set of transitions (outgoingTransitions). Transition represents a FSM transition from a source state to a target state. When a transition is triggered, an associated action is executed.

Brown-colored classes, i.e., AnnotatedElement and Annotation, provide the required concepts to express non-OCCI core information needed to perform several activities such as code generation and model visualization. AnnotatedElement represents the abstract base class inherited by Attribute and Category. Each attribute/kind/mixin/action can own zero or more annotations. Annotation represents an additional information that can be attached to an AnnotatedElement instance. An annotation is composed of a key and a value. This mechanism is usually used to limit changing the metamodel.

5.4. Typing concepts

Typing data is a necessary step to create precise artifacts. For that, in the OCCIware Metamodel, we provided two typing mechanisms. The first one is the purple-colored classes, i.e., Integer, Boolean, String, URI, Name, AttributeName, OCL and Scheme. Most of these defined data types are enriched with regular expressions in order to assess the well-formedness of OCCI artifacts in regards to OCCI specifications. For example, according to the OCCI Text Rendering specification [18], the AttributeName type is extended with the following pattern "[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)+" while the Name type is extended with "[a-zA-Z][a-zA-Z0-9_-]*" pattern. Scheme type is extended with a pattern constraint conforms to the Uniform Resource Identifier (URI) syntax [36].

The second set is the yellow-colored classes, i.e., DataType, its sub-classes, and their related ones, which define the OCCIware data type system. This set allows to model data types associated to the Attributes during the design of a cloud domain using the Extension mechanism. We can design prim-

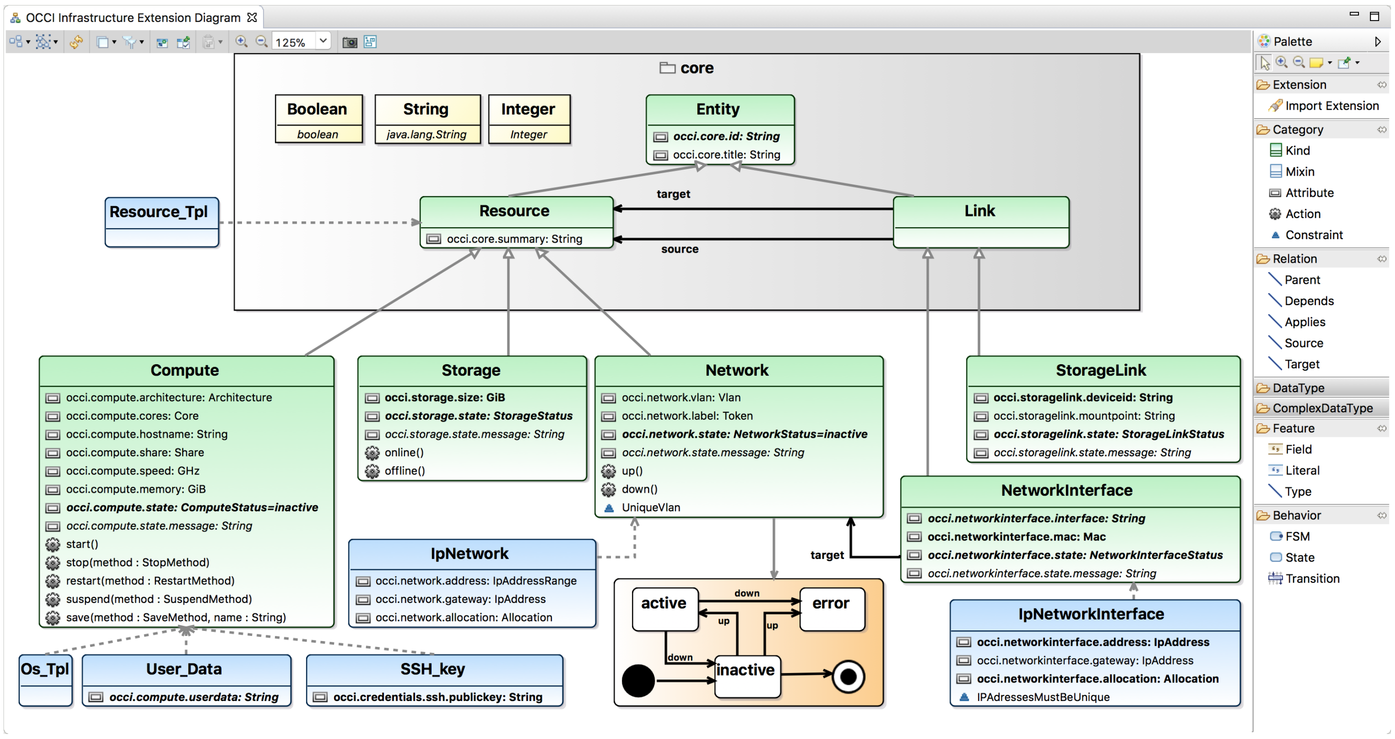

OCCI Infrastructure Extension Diagram
125%
core
Boolean
boolean
String
java.lang.String
Integer
Integer
Entity
occi.core.id: String
occi.core.title: String
Resource_Tpl
Resource
occi.core.summary: String
target
source
Link
Compute
occi.compute.architecture: Architecture
occi.compute.cores: Core
occi.compute.hostname: 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
NetworkInterface
occi.networkinterface.interface: String
occi.networkinterface.mac: Mac
occi.networkinterface.state: NetworkInterfaceStatus
occi.networkinterface.state.message: String
IpNetwork
occi.network.address: IpAddressRange
occi.network.gateway: IpAddress
occi.network.allocation: Allocation
target
active
down
error
up
up
down
inactive
IpNetworkInterface
occi.networkinterface.address: IpAddress
occi.networkinterface.gateway: IpAddress
occi.networkinterface.allocation: Allocation
IPAdressesMustBeUnique
Os_Tpl
User_Data
occi.compute.userdata: String
SSH_key
occi.credentials.ssh.publickey: String
Palette
Extension
Import Extension
Category
Kind
Mixin
Attribute
Action
Constraint
Relation
Parent
Depends
Applies
Source
Target
DataType
ComplexDataType
Feature
Field
Literal
Type
Behavior
FSM
State
Transition

Figure 9: OCCI Infrastructure Extension Model

itive types (i.e., the sub-classes of `BasicType`), enumeration types (`EnumerationType`), array types (`ArrayType`) and record types (`RecordType`).

The OCCIware Metamodel shown on Figure 8 only defines the structure of an OCCIware artifact. To design valid extensions and configurations with the respect of OCCI specifications, we have extended the OCCIware Metamodel with a set of OCL constraints.

To illustrate OCCIware Metamodel, let us consider OCCI Infrastructure [20] which defines compute, storage and network resource types and associated links. To design the `Infrastructure` extension, the OCCIware architect can use OCCIware Designer and/or OCCIware Editor.

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.). Figure 9 shows a subset of this extension. The complete one is available here[12].

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 Figure 9 illustrates the state diagram of a `Network` instance and describes its behavior. As shown previously, OCCIware Metamodel provides the required concepts to describe the behavior of each kind/mixin. In addition, it allows defining 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.

[12]`https://github.com/occiware/OCCI-Studio/blob/master/plugins/org.eclipse.cmf.occi.infrastructure/model/Infrastructure.occie`

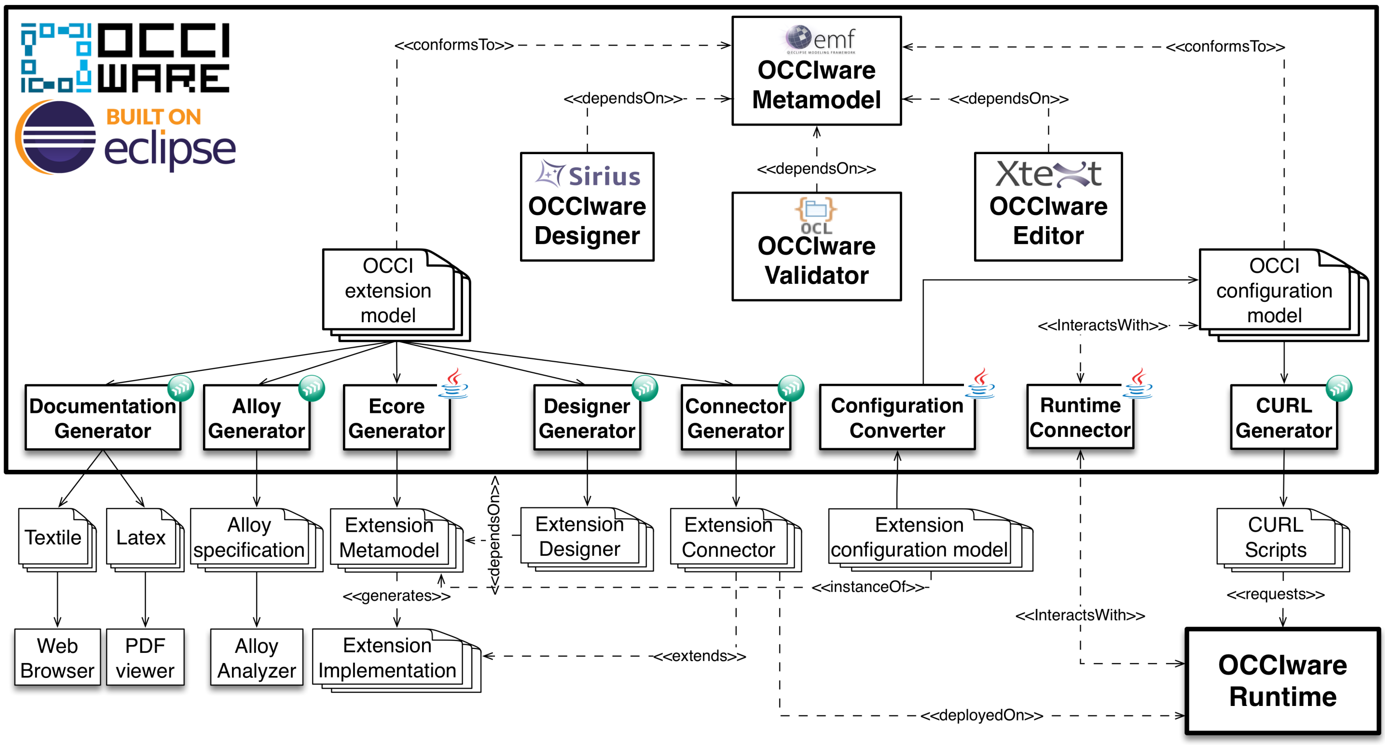


Figure 10: OCCIware Studio features

To summarize, during the OCCIware research and development project, the proposed OCCIware Metamodel was used to model various OCCI extensions: OCCI Infrastructure [20], OCCI Platform [22], OCCI SLA [23], OCCI Monitoring [24] and some additional proprietary extensions such as Docker [37], simulation [38, 39], OMCRI [40], ModMaCAO [41], GCP [42], and Multi-Cloud[13]. 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.

6. OCCIware Studio

Once OCCIware Metamodel was defined, we have tooled it to propose a model-driven framework called OCCIware Studio. This latter is a set of plugins for the Eclipse[14] Integrated Development Environment (IDE). Figure 10 shows all features of OCCIware Studio, which can be categorized into three main sets based on the accepted artifacts. In the following, we detail these categories.

6.1. OCCIware *generic tools*

In order to ease the use of a DSML, it is mandatory to provide such facilities to create and edit conforming models. The main tools that should be defined are those which implement the different concrete syntaxes, i.e., textual and graphical ones. These tools are developed on the top of OCCIware Metamodel. They provide the required tools to design, edit and validate OCCIware artifacts. For instance, **OCCIware 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[15]. **OCCIware Designer** is used to design the OCCI Infrastructure extension (Figure 9) and an OCCI configuration (Figure 5). To textually design and edit OCCIware artifacts, we implement a textual editor called **OCCIware Editor**. It is implemented on top of the Eclipse Xtext framework[16]. Finally, to validate OCCIware artifacts, we implement the **OCCIware Validator** tool. It checks all the constraints defined in OCCIware Metamodel, *i.e.*, both Ecore and OCL ones.

6.2. OCCIware *extension-specific tools*

OCCIware Metamodel allows us to define extensions. Each extension represents a particular cloud domain. An extension should be verified, documented and tooled in order to easily create correct conforming configurations. To do that, we have developed a set of tools that accept an extension and generate the appropriate artifacts.
First of all, once the extension is designed, it is necessary to verify it. To achieve that, we have chosen Alloy, which is a

[13]`https://github.com/occiware/Multi-Cloud-Studio`
[14]`https://www.eclipse.org`
[15]`https://eclipse.org/sirius/`
[16]`https://eclipse.org/Xtext/`

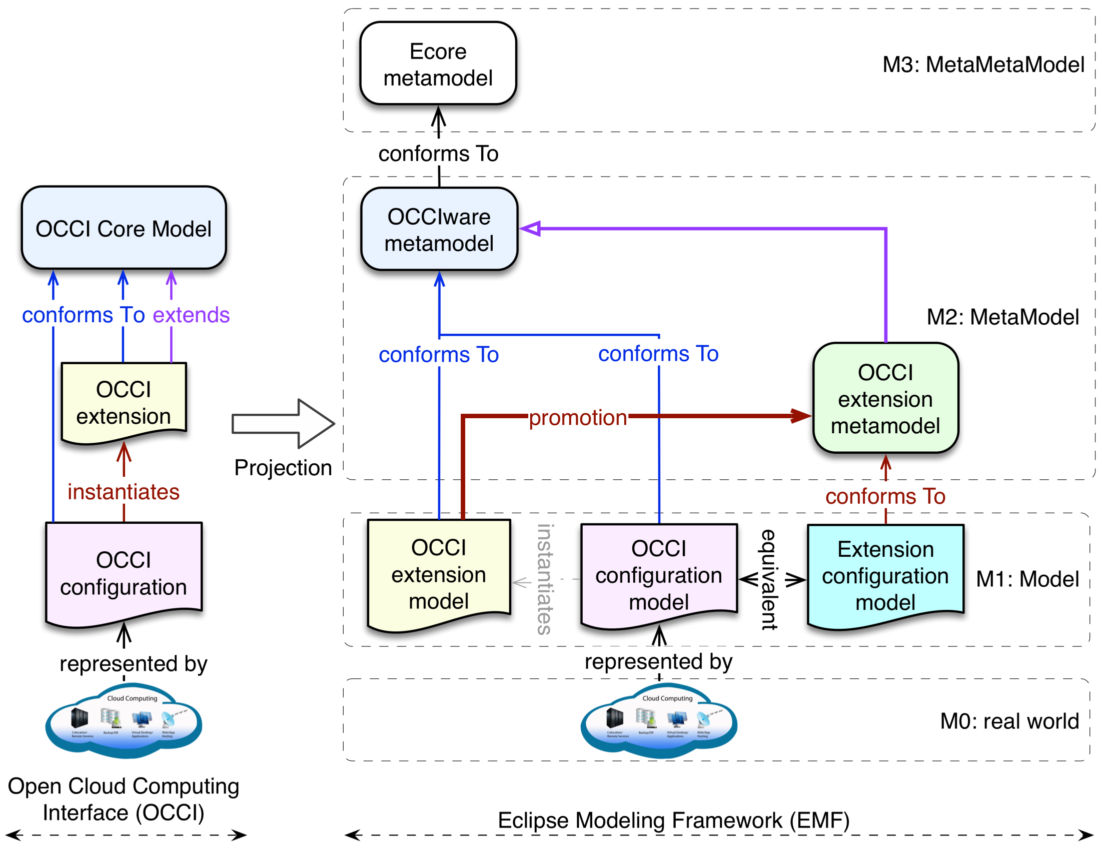


Figure 11: Projection of OCCI to EMF

lightweight formal specification language based on the first-order relational logic [43]. In a previous work, we have used Alloy to precisely define the OCCI concepts and their properties [44]. In addition, we were also able to formally analyze OCCI extensions with Alloy Analyzer, which is a solver that takes the constraints of a model and finds instances that satisfy them. To formally verify extensions, we have implemented **Alloy Generator**, which is a tool to generate an Alloy specification from an OCCI extension model. Latter, Alloy Analyzer takes an OCCI extension in our case and finds instances that satisfy it. Alloy Analyzer also verifies the properties of the extensions by searching for counterexamples. **Alloy Generator** is implemented on top of the Eclipse Acceleo framework.
To seamlessly create conforming configurations, it is necessary to document the designed and verified extension. To do that, we have defined two documentation generators. **Textile 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[17]. **Latex Documentation Generator** is a tool to generate a Latex documentation from an OCCI extension model. It allows us to later generate a portable document describing the OCCI extension. This tool is implemented on top of the Eclipse Acceleo framework.
The main goal of OCCIware consists in introducing a tooled framework, based on OCCI, that manages any kind of resources as a service. To achieve that, it was necessary to map different OCCI concepts into a modeling framework to benefit from the available facilities for building tools based on a metamodel (the right part of Figure 11). EMF was chosen to embed OCCI and, thus, OCCIware Metamodel was proposed as a precise metamodel for OCCI [10]. Therefore, we can define either an OCCI extension model or an OCCI configuration model which conforms to OCCIware Metamodel. However, the current tooling in EMF does not allow us to encode that: an OCCI configuration is an *"instantiation"* of an OCCI extension. For that, we have implemented the **Ecore Generator**, which is a tool to generate the Ecore metamodel and its associated Java-based implementation code from an OCCI extension. It allows us to promote the OCCI extension model by translating it into an Ecore metamodel, extending OCCIware Metamodel. Consequently, we can design an Extension configuration model, instance of this generated metamodel. Later, we can deduce a semantically equivalent OCCI configuration model, instance of OCCIware Metamodel. This tool is directly implemented in Java. In the following, we detail the generation 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 OCCIware Metamodel. Otherwise, if its `parent` is the `Link` kind, the generated class extends the `Link` Ecore class of OCCIware Metamodel.

[17]`https://www.eclipse.org/acceleo/`

- Each OCCI mixin instance is translated into an Ecore class extending the `MixinBase` class of OCCIware Metamodel.
- 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.

Table 1 outlines the mapping process of OCCIware Metamodel concepts into EMF concepts.

OCCI concept	**EMF concept**
Kind	EClass
Kind's source	OCL invariant
Kind's target	OCL invariant
Attribute	EAttribute
Action	EOperation
Mixin	EClass
Constraint	OCL invariant
BasicType	EDataType
EnumerationType	EEnum
RecordType	EClass
ArrayType	EClass

Table 1: The mapping process of OCCI concepts into EMF concepts

The OCCIware approach proposes to deploy a designed configuration on the cloud and maintain it synchronized with the running one. To ensure that, we have implemented a **Connector Generator**, which is a tool to generate the OCCI connector implementation associated to an OCCI extension. This generated connector code extends the generated Ecore implementation code. Later, it must be completed by software developers to implement concretely how OCCI CRUD operations and the specific actions must be executed on a real cloud infrastructure. Finally, this generated connector will be deployed on OCCIware Runtime. This tool is implemented on top of the Eclipse Acceleo framework.

Using **Ecore Generator**, OCCIware Studio allows to create a DSML for each OCCI extension. To easily create and edit conforming configurations, we have implemented **Designer Generator**, which is a tool to generate a graphical extension-specific designer from an OCCI extension. Unlike **OCCIware Designer** where we can only create OCCI-specific configurations, the generated designer can be later customized to be able to represent the concepts related to the extension domain. This tool is implemented on top of the Eclipse Acceleo framework and generates an Eclipse Sirius designer.

For our OCCI Infrastructure use-case, as shown previously, Figure 7 presents an `Infrastructure` configuration model that conforms to the `Infrastructure Metamodel`. This configuration is designed using the **Infrastructure Designer** generated from the `Infrastructure` extension using the **Designer Generator**. In addition, Listing 1 shows a subset of the generated `Network` connector class generated from the `Infrastructure` extension by the **Connector Generator**. It extends the `NetworkImpl` class generated by the **Ecore Generator** 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.

Once the generation step is achieved, the software developer can complete the generated connector classes by updating their method implementations (`TODO` sections in Listing 1) with business code related to targeted API. For the `NetworkConnector` class, the software 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`).

```java
public class NetworkConnector extends NetworkImpl {
  NetworkConnector() {}
  // OCCI CRUD callback operations.
  public void occiCreate()    { /* TODO */ }
  public void occiRetrieve()  { /* TODO */ }
  public void occiUpdate()    { /* TODO */ }
  public void occiDelete()    { /* TODO */ }

  // Network actions.
  public void up() {
    if(getState().equals(NetworkStatus.INACTIVE)) {
      if ( true ) {
        // TODO: Transition inactive -up-> active
        setState(NetworkStatus.ACTIVE);
      } else {
        // TODO: Transition inactive -up-> error
        setState(NetworkStatus.ERROR);
      }
    }
  }
  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: The Generated `Network` Connector Class

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 later deployed on OCCIware Runtime. Then, the software developer can proceed to the customization of the generated `Infrastructure Designer`, which will be used to create `Infrastructure` configuration models.
Hence, we can consider that the OCCI `Infrastructure` extension is completely tooled and able to be used to manage conforming configurations.

6.3. OCCIware *configuration-specific tools*

Once an OCCI configuration is designed and verified, it can be deployed on the cloud. Usually, this activity is performed by manually writing the artifact defining the configuration to be deployed. However, it is an error-prone task. To resolve this issue, OCCIware Studio proposes **CURL Generator**, which 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.
To synchronize OCCI configuration models with running OCCI configurations hosted by any OCCI-compliant runtime, we have implemented **Runtime Connector**, which allows cloud developers to introspect an OCCI runtime in order to build the corresponding OCCI configuration model, then update this model and send changes back to the OCCI runtime. This tool integrates the jOCCI API[18], a Java library implementing transport functions for rendered OCCI queries.
OCCIware Studio allows us to design Extension configurations, which instantiate the specific concepts of an extension. In order to reuse the OCCI tools such as the CURL Generator to deploy later the configuration into an OCCI-compliant runtime, we have developed a tool called **Configuration Converter**. It translates an Extension configuration model into an OCCI configuration model (the **equivalent** relation in Figure 11). This tool is directly implemented in Java.

For our OCCI Infrastructure use-case, cloud developers can design an OCCI `Infrastructure` configuration model conforms to the `Infrastructure Metamodel`. As shown previously, Figure 7 illustrates a small infrastructure configuration composed of a compute (`vm1`) connected to a network (`network1`), via a `NetworkInterface` link (`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 a `vm1` hostname, an `x64`-based architecture, 4 cores, and 4 GiB of memory.

To benefit from the OCCI-compliant tools defined in OCCIware Studio, an `Infrastructure` configuration model must be translated into an OCCI configuration model that conforms to OCCIware Metamodel. Figure 5 shows the generated OCCI configuration model from the `Infrastructure` configuration model.

In order to deploy and manage the generated OCCI configuration models, cloud developers interact with the cloud by sending OCCI HTTP requests to OCCIware Runtime. These requests can be automatically generated as CURL scripts using the CURL Generator tool. Listing 2 shows the CURL script that interacts with OCCIware Runtime via both OCCI HTTP Protocol [17] and OCCI Text Rendering [18] to create the `network1` instance.

```
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 'Category: ipnetwork;
      scheme="http://schemas.ogf.org/occi/infrastructure#";
      class="mixin";'
–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 2: The generated CURL script to create a `Network` instance

To summarize, OCCIware Studio is an open-source tool available here[19]. It can be extended to deal with other needs and challenges. Currently, based on the feedback of OCCIware Studio users, we plan to implement a generator of JSON artifacts from OCCI configurations. The generated artifacts must respect the OCCI JSON Rendering specification [19].

7. OCCIware Runtime

To enact OCCI configuration models, we adopt the Models@run.time approach [13] that extends the use of modeling techniques beyond the design and implementation phases. It seeks to extend the applicability of models and abstractions to capture the behavior of the executing environment. End users of OCCIware Studio require interaction with cloud APIs to create, retrieve, update and delete cloud resources. Therefore, we implemented OCCIware Runtime, a generic Java implementation of OCCI, available as an open-source project[20]. OCCIware Runtime can be deployed as a standalone server including an embedded Jetty server or as a Java library that is based on Java Servlet API specification. OCCIware Runtime is composed of four main parts, as illustrated in Figure 12:

- **OCCI Server** that implements the following OCCI specifications: *(i)* OCCI HTTP Protocol [17], *(ii)* OCCI Text Rendering [18], OCCI JSON Rendering [19] based on the Jackson[21] library, and *(iii)* OCCI Core Model [7] based on OCCIware Metamodel.

- **OCCI Extensions** that represent the OCCIware-based modeling of concrete cloud domains such as OCCI `Infrastructure` [20], OCCI `Platform` [22], OCCI `SLA` [23], Docker [37], cloud mobile robotics [40], GCP [42], etc.

- **OCCI Configurations** which are instances of OCCIware configuration, such as a configuration using OCCI `Infrastructure` and containing a running virtual machine with 4 cores, 3.2 GHz and 16 GiB.

[18]`https://github.com/EGI-FCTF/jOCCI-api`

[19]`https://github.com/occiware/OCCI-Studio`
[20]`https://github.com/occiware/MartServer`
[21]`https://github.com/FasterXML/jackson`

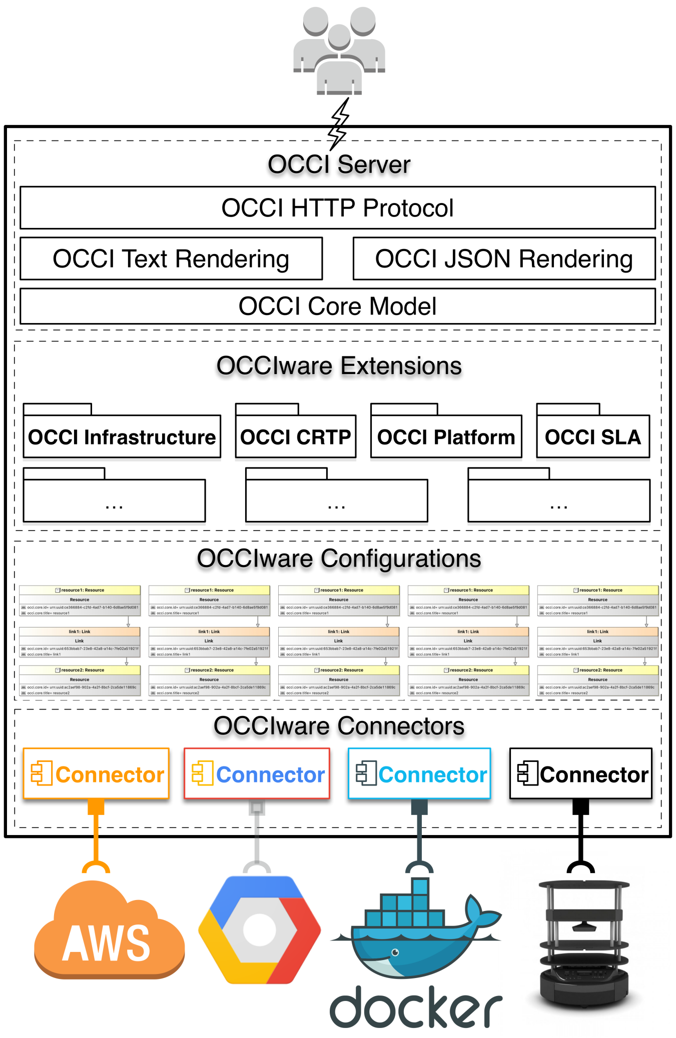


Figure 12: OCCIware Runtime Architecture

- **OCCI Connectors** that consist in the pivot between OCCIware extensions/configurations and CRM-APIs. Each connector is dedicated to a specific cloud domain, *e.g.*, AWS, GCP, Docker, or mobile robotics. It implements the corresponding OCCI extension. It executes CRUD (Create, Retrieve, Update and Delete) operations and extension-specific actions, such as *start compute* for the OCCI `Infrastructure` extension. OCCIware Runtime is extensible by design: Supporting a new kind of cloud resources consists of adding a new connector.

Users send their HTTP requests to OCCIware Runtime to manage an *OCCI configuration* and wait for a reply. In OCCIware Runtime, the processing of a user's request consists of managing the *OCCI HTTP protocol*, decoding the HTTP request body according to its *text or JSON* format, forwarding the request to the *OCCI Core Model*, verifying if the request is allowed by the OCCIware *extension*, calling the *connector* related to the targeted cloud API, preparing the request to send to the cloud provider, communicating with the cloud API via its associated network protocol, processing the request by the cloud provider, encoding the HTTP reply body, and return the reply to the user. Then the user processes the reply.

For our Infrastructure use-case, once the HTTP request (Listing 2) is sent, OCCIware Runtime invokes the `occiCreate()` method of the `NetworkConnector` class, which implements how to create the considered network instance in the cloud. Finally, the `Network` resource is created and deployed in the cloud.

8. Synthesis on the OCCIware Approach

OCCIware approach has been successfully applied in six use cases and domains. Each one implements some required features in a cloud modeling framework. Table 2 states an overview on the implementation of these usages in the different use cases.

1. For the `Infrastructure` use case, cloud architects have taken advantage of the OCCIware Studio tools to design the structural and behavioral aspects of the extension, verify it according to the requirements specific to the domain, and generate its documentation. To deploy designed configurations, `Infrastructure` use case implements the code generation strategy to produce CURL scripts from an OCCI configuration. In addition, model interpretation strategy is also implemented. Indeed, an OCCI extension for the `VMware` technology has been defined[22]. `VMware` extension extends the `Infrastructure` extension. A `VMware` connector has been developed. It supports the deployment of a designed `VMware` configuration in the cloud environment. In addition, it ensures the reconfiguration of a running system at runtime (management) and the synchronization between the design and the execution environment by affecting changes occurred in the executing environment to the existing architecture (monitoring).

2. For the `Cloud Simulation` use case [38, 39], it illustrates several usages of the OCCIware approach. At first, it represents the capability to create a specific designer for the `Simulation` domain. In addition, it also shows how we can map an existing OCCI configuration to a simulated one. Finally, this work implements the model interpretation strategy by simulating an OCCI configuration model using CloudSim [45]. This use case illustrates how we can plug an external API, here CloudSim, into the OCCIware framework. In fact, once a configuration has been designed, it can be reused for several activities such as simulation, deployment, and cost analysis.

3. OMCRI use case [40] endorses the fact that we can manage Everything as a Service (XaaS) with the OCCIware approach, even mobile robots. OMCRI use case validates

[22]Available here `https://github.com/occiware/Multi-Cloud-Studio/tree/master/plugins/org.eclipse.cmf.occi.multicloud.vmware`

the genericity of OCCIware Runtime. In addition, the evaluation shows that adopting the OCCIware approach into another domain, other than cloud computing, does not cause a latency that damages the responsiveness of the system. This work has been awarded the best paper award during the 2nd IEEE International Congress on Internet of Things (ICIOT 2017).

4. Docker use case [37] is considered as the first OCCI-based studio implemented with OCCIware technology. It illustrates our approach to consider OCCIware as a factory to build cloud modeling frameworks (right part of Figure 6). Docker Studio represents a concrete use case, which implements both strategies, code generation, and model interpretation, to execute designed models specific to a particular domain. The most important feature developed in the Docker use case is the mapping and the synchronization of a running architecture from the execution environment to the modeling framework. It allows end users to benefit from the Docker Studio while the configuration was not initially designed using it.

5. MoDMaCAO use case [41] validates the applicability of the OCCIware approach on different cloud layers. It can even be applied to connect two layers in the cloud such as IaaS and PaaS. Model interpretation strategy has been experimented by developing a connector, which allows to deploy and manage different designed cloud applications using Ansible[25], a flexible configuration management system. Code generation strategy may also be experimented by generating Ansible playbook artifacts from MoDMaCAO configurations.

6. GCP use case [42] represents the first use case that applies the OCCIware approach on a big cloud provider, aka GCP. The authors target to integrate both code generation and model interpretation strategies. With the code generation approach, they aim to use GCP configurations to generate GCP artifacts, such as JSON files containing the needed structured information for creating a VM for example. In addition, this use case aims to experiment the model interpretation approach, by defining the business logic of GCP connector, which defines the relationship between GCP configurations and their executing environment.

To summarize, OCCIware approach provides a software product line, named OCCIware Studio Product Line as shown in Figure 13. OCCIware Studio is considered as a factory to create other studios where each one targets a particular cloud domain. The different generated studios share a common base, which is OCCIware Metamodel. Therefore, composing the different generated studios can be an interesting perspective to create an Internet of Everything (IoE) Studio that allows us to compose heterogeneous concepts and domains in the same modeling framework.

[23]Robot as a Service

[24]Container as a Service

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

9. Learned Lessons

Based on our experience with OCCIware, we have identified four major feedbacks.

Specifying OCCI precisely. At first, our starting point to implement the OCCIware approach was OCCI specification documents, and more precisely the OCCI Core Model. However, as mentioned in its specification [7], the OCCI Core Model *in itself is not a complete definition of the model*, *i.e.*, it is proposed only for documenting reasons. Based on this incomplete and informal specification, it would have been difficult and even impossible to propose a tooling chain to design, edit, verify and manage any kind of cloud resources. That is why our first step was to study the OCCI Core Model, identify the lacks and drawbacks, and finally resolve them by introducing the required concepts. For example, in the OCCI Core specifications [7], authors mention the notion of `Extension` but this concept is never defined at all throughout the specification documents. Now, based on the OCCIware approach, we can precisely define an OCCI extension and even we can verify it before creating conforming configurations.

Modeling any cloud resources with OCCI. OCCI claims to be a generic standard. From the cloud architect perspective, it is suitable to serve many other models in addition to IaaS, including PaaS and SaaS. In addition, it is extremely helpful to have a concise and small set of concepts to design different types of resources through the different cloud layers. In fact, the OCCIware approach provides this facility through the inheritance mechanism. Indeed, from OCCI perspective, a compute, an application, or a component is finally a resource. Even more, the OCCIware use-cases have extended to a border scope by targeting other domains rather than the cloud domain. OMCRI [40] use-case provides the first implementation of the use of OCCIware in the Cloud Robotics field.

Generating cloud domain-specific studios. The genericity of OCCI Core Model allows us to reason on a high-level of abstraction. However, the genericity aspect is not a fundamental point for a cloud developer who usually defines a cloud configuration inside a particular cloud provider like AWS. The most urgent need for a cloud developer is to obtain a tool, which should be as near as possible to the concepts of the cloud provider. So, a harmony between the generic and the specific tooling must be preserved for such an approach. The OCCIware approach has succeeded in this challenge. Beyond the generic OCCI configuration designer, OCCIware Studio allows us to generate a designer for each domain. This designer can later be customized with the notation of the targeted domain. Therefore, by using OCCIware Studio, an AWS cloud developer considers that he/she handles specifically AWS concepts such as EC2 and Lambda. However, in the reality, the instantiated concepts are OCCI resources and links.

	Infrastructure	Simulation	OMCRI	Docker	MoDMaCAO	GCP	Coverage
Cloud Domain	IaaS	IaaS	RaaS[23]	CaaS[24]	PaaS	[I||P||S]aaS	XaaS
Design	✓	✓	✓	✓	✓	✓	✓
Verification	✓	✓	✓	✓	✓	✓	✓
Documentation	✓	✓	✓	✓	✓	✓	✓
Code generation	✓			✓		✓	✓
Model interpretation	VMware API	CloudSim API	Robots API	Docker API	Ansible API	GCP API	✓
Deployment	✓		✓	✓			✓
Discovery				✓			✓
Management	✓		✓	✓	✓		✓
Monitoring	✓			✓			✓

Table 2: OCCIware Use Cases

OCCI is not enough. Creating a cloud configuration using a modeling tool and deploying it in the cloud is always a challenging task and requires a huge effort from the cloud stakeholders. To address this challenge, several mechanisms like reusing pre-defined templates and expressing resource behaviors are proposed. A template is a mechanism to provide default values for instances. For example, when creating an AWS EC2 instance, a set of pre-defined templates[26] (like t2.micro template) are proposed and the user needs to choose a particular one. In addition, AWS defines for each kind of resource a life cycle (like EC2 instance here [27]). OCCI proposes the notion of Mixin to deal with the template mechanism and the OCCIware metamodel defines the required concepts to explicitly define the behavior of an OCCI kind/mixin. However, both mechanisms are limited in the scope of a particular entity. OCCI and OCCIware do not allow us to reason on an inter-entity layer in order to assemble a subset of kinds and create a template that can be later instantiated. In addition, the different introduced concepts to define the behavior of a cloud resource does not allow us to create a workflow, which describes the interaction between different entities. To deal with these identified weaknesses, it would be interesting to refer to the OASIS's Topology and Orchestration Specification for Cloud Applications (TOSCA) [46] standard. It provides the first answers to our research questions. Finally, the first works on combining both TOSCA and OCCI standards [47] show interesting results and can be considered as a cornerstone to continue in this way.

10. Related work

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

Cloud Standards. Besides OCCI, many elaborated and mature standards emerged in the cloud community to provide some principles to be commonly used by the cloud stakeholders for different purposes. The OASIS's Topology and Orchestration Specification for Cloud Applications (TOSCA) [46] is a modeling language for describing the topology/structure/architecture of cloud applications, 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 modeling language is only defined as a textual XML or YAML document so it is complicated to have an overview of all its concepts. While both are standards, TOSCA provides a cloud application description language while OCCI provides a cloud resource management API. In the future, our MoDMaCAO use case presented in Section 8 will be extended to support TOSCA topologies as input. The DTMF's Cloud Infrastructure Management Interface (CIMI) standard [4] 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, and even mobile robots. The OASIS's Cloud Application Management for Platforms (CAMP) 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.

Cloud Modeling Languages. Several cloud modeling languages exist in the literature to describe aspects of the cloud domain, by rising in abstraction from technical details. CloudML [48][49] 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 an inspiration source for future efforts in modeling the cloud, however, it lacks implementations. Unlike OCCIware, which offers a set of tools to model, validate, document, generate, develop, deploy, manage, and supervise 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. CloudML is only a cloud DSML that we could very well encode with OCCIware Studio. ClOud sOlution design tooL (COOL) [50] is an approach for designing cloud architectures based on high-level requirements. The architectures produced can then be deployed via OCCIware or CloudML for example. SALOON [51, 52] is based on Ex-

[26]`https://aws.amazon.com/ec2/instance-types/`
[27]`https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-lifecycle.html`

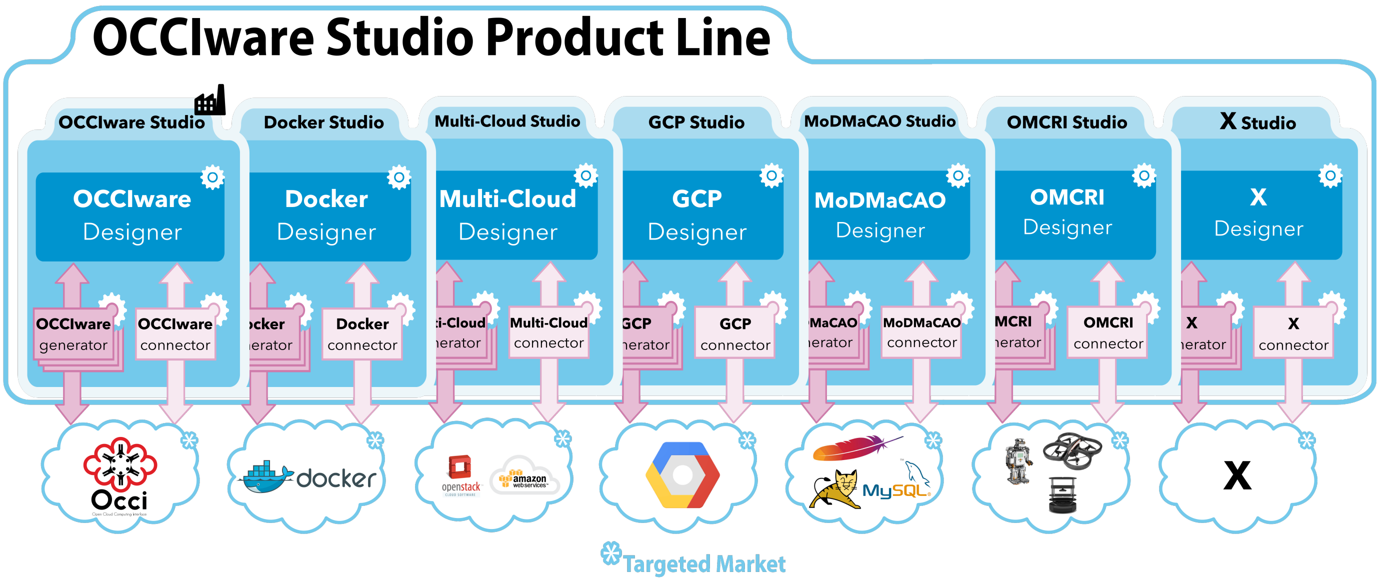


Figure 13: OCCIware Studio Product Line

tended 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. On 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 developers 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. The CompatibleOne Resource Description System (CORDS) [53] is the model of the open-source CompatibleOne broker that offers services from several cloud providers. 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 [53] defined this model to describe only IaaS and PaaS resources managed by their broker. The authors in [54] propose domain-specific models for designing cloud resources as entities and relationships (Resource Description Model) and for managing these resources (Action Model and Event Model). They also implement connectors to automatically generate cloud resource descriptions, deployment and management scripts. Once both the domain-specific models and the connector are registered, cloud resources can be created and managed.

Cloud Runtime Implementations. Besides OCCIware Runtime, we identified seven OCCI implementations of which five are inactive (ACCORDS, erocci, OCCI4Java, pyOCNI, pySSF) and two are active (OOI and rOCCI). The Advanced Capabilities for CORDS (ACCORDS) [55] is a C implementation developed by Prologue. It is the execution platform of the CORDS models. It allows to handle the user's requirements, validate, and execute the provisioning plan, and to deliver the cloud services. However, it does not support the graphical design of cloud configurations nor the generation of documentations, Ecore extensions, CURL scripts, etc. Erocci is an implementation in Erlang developed by Jean Parpaillon, member of the OCCI working group of the OGF, and partly as part of the OCCIware collaborative project. OCCI4Java is a Java implementation developed by Sebastian Laag. OpenStack OCCI Interface (OOI) [56] is a Python implementation developed by the Spanish National Research Council (NTS) as part of EGI-Engage and INDIGO-Datacloud collaborative projects. OOI is used in the EGI FC community. pyOCNI is a Python implementation developed by the Institut Mines Telecom-Telecom SudParis (IMT-TSP). pySSF is a Python implementation developed by Thijs Metsch, founder/chairman of the OGF's OCCI working group and engineer at Intel. rOCCI is a Ruby implementation developed by the GWDG, the computer center of the University of Göttingen in Germany. rOCCI is actively maintained and used in the EGI FC community. All these OCCI implementations, except for erocci, are not generic but mainly specific to the IaaS domain. Also, they are not all related to modeling practices through the use of Models@run.time. The running OCCI configurations might be error-prone due to the lack of a verification process, and correcting them requires huge time and development costs. Finally, these OCCI implementations remain code-level implementations, which makes them tightly coupled to their programming languages instead on focusing on cloud concerns.

The literature encompasses several model-driven efforts for building a runtime for interpreting and executing cloud models. Several implementations of TOSCA were developed. For example, Winery[28] provides an open source Eclipse-based graphical modeling tool for TOSCA. OpenTOSCA [57] is the TOSCA runtime environment to deploy and manage cloud applications.

[28]https://projects.eclipse.org/projects/soa.winery

It enables the automated provisioning of applications that are modeled using TOSCA standard. 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. Unlike our approach, OpenTOSCA is not connected to Winery to manage the running TOSCA topologies via the TOSCA model and bring back the updates from the running system to the model. Also, OpenTOSCA focuses on one delivery model, which is PaaS. The Cloud Modeling Framework (CloudMF) [58] is the execution engine (model interpreter) of CloudML. CloudMF aims to manage at runtime the deployed applications. It automatically handles the model at design-time and returns a runtime model of the provisioning resources, according to the Models@run.time approach. However, CloudMF is not based on a standard that defines common principles for the cloud computing.

11. Conclusion

OCCI proposes a generic extensible model and a REST 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 ease interoperability with other clouds. However, OCCI has proven its utility in several contexts. For example, the 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.

We argue in this article 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, the first model-driven approach for OCCI. Our approach provides two main components: OCCIware Studio and OCCIware Runtime. OCCIware Studio is a model-driven tool chain to design OCCI artifacts. It is built on the top of a metamodel, named OCCIware Metamodel, which defines the precise semantics of OCCI in Ecore and OCL. This metamodel can be seen as a domain-specific modeling language (DSML) to define and exchange OCCI extensions and configurations between end users and resource providers. OCCIware Runtime is a generic OCCI-compliant runtime environment.

The OCCIware approach has been proposed as a framework to manage everything as Service with OCCI. Indeed, thanks to OCCIware Studio, both cloud architects and developers can encode OCCI extensions and configurations, respectively, graphically via the OCCIware Designer tool, and textually via the OCCIware Editor tool. They can also automatically verify the consistency of these extensions and configurations via the OCCIware Validator tool. OCCIware approach supports two strategies to link the OCCIware Studio artifacts with the OCCIware Runtime: code generation and model interpretation, Therefore, from a designed and verified OCCI configuration, we can generate a deployment script via the CURL Generator tool. In addition, we can generate dedicated model-driven tooling via both Ecore Generator and Connector Generator tools. Later, we can manage these configurations at runtime via the generated connectors deployed on OCCIware Runtime.

Moreover, OCCIware approach is considered as a factory of cloud domain-specific modeling languages and studios due to its capability to generate a complete framework to manage resources specific to a particular cloud domain. OCCIware Studio is tightly integrated with the Eclipse IDE, to ease the addition of functionality to the service skeletons and generates ready-to-deploy configurations for OCCI extensions.

OCCIware approach has been validated via several use cases, which target different domains (RaaS, IaaS, CaaS, PaaS, and SaaS). Each use case illustrates a specific usage of OCCIware approach and demonstrates its genericity and extensibility. For example, OCCIware approach has been applied to define a modeling framework for the VMware, Docker, GCP, CloudSim, and Ansible cloud technologies, and even cloud mobile robotics with OMCRI.

In the future, we target industrial validation for OCCIware approach. Therefore, an ongoing work aims to get this approach tested and adopted in Scalair[29], a hybrid cloud provider. In order to cover the whole cloud market, we are currently developing the Multi-Cloud Studio that supports two other cloud providers: AWS and OpenStack. Both OCCI extensions for AWS and OpenStack are under development in order to provide a modeling studio to design both AWS and OpenStack configurations.

For managing cloud applications, we aim to implement a model-driven cloud orchestrator based on two complementary standards, TOSCA and OCCI. Therefore, we are refining the mapping between the concepts of these two standards [47], and building TOSCA Studio, a dedicated model-driven environment for designing applications with TOSCA. This approach will allow TOSCA to have a complementary tool to take better advantage of deployed applications in production environments. At runtime, TOSCA Studio will be able to: *(i)* communicate with an OCCI Infrastructure such as the EGI FC to provision virtual machines for example, or *(ii)* be exposed via the OCCI Plaform API in order to create, retrieve, update and delete any kind of cloud application resources. The research questions about the coherence of the model at runtime can also be addressed.

Finally, we target to extend OCCIware Studio in order to support the automatic generation of deployment plans from OCCI configurations. Currently, the cloud developer does this task manually. This feature allows us to analyze the different resources and links between them available in an OCCI configuration and deduce a deployment plan, which will be automatically executed on OCCIware Runtime.

Acknowledgements

This work is supported by the OCCIware research and development project (`www.occiware.org`) funded by French Pro-

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

gramme d'Investissements d'Avenir (PIA), and the Hauts-de-France Regional Council. We would like to thank all the OCCIware partners for having tested and used our tool chain on their business use cases. Finally, we would like to address special thanks to William Piers from Obeo company, who helps us to implement OCCIware Studio, and Christophe Gourdin from Inria, who fully implemented OCCIware Runtime.

Availability

Readers can find OCCIware Studio including OCCIware Metamodel and all the model-driven tools at `https://github.com/occiware/OCCI-Studio`. OCCIware Runtime is available at `https://github.com/occiware/MartServer`.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al., A View of Cloud Computing, Communications of the ACM 53 (4) (2010) 50–58.

[2] P. Mell, T. Grance, The NIST Definition of Cloud Computing, Vol. 800-145, IGI Publication, 2011.

[3] J. Martin-Flatin, Challenges in Cloud Management, IEEE Cloud Computing 1 (1) (2014) 66–70.

[4] D. Davis, G. Pilz, Cloud Infrastructure Management Interface (CIMI) Model and REST Interface over HTTP, vol. DSP-0263.

[5] M. Carlson, M. Chapman, A. Heneveld, S. Hinkelman, D. Johnston-Watt, A. Karmarkar, T. Kunze, A. Malhotra, J. Mischkinsky, A. Otto, et al., Cloud Application Management for Platforms, OASIS, http://cloudspecs.org/camp/CAMP-v1. 0. pdf, Tech. Rep.

[6] A. Edmonds, T. Metsch, A. Papaspyrou, A. Richardson, Toward an Open Cloud Standard, IEEE Internet Computing 16 (4) (2012) 15–25.

[7] R. Nyrén, A. Edmonds, A. Papaspyrou, T. Metsch, B. Parák, Open Cloud Computing Interface – Core, Recommendation GFD-R-P.221, Open Grid Forum.

[8] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Ph.D. thesis, University of California, Irvine (2000).

[9] F. Zalila, S. Challita, P. Merle, A Model-Driven Tool Chain for OCCI, in: On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I, 2017, pp. 389–409.

[10] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, S. Tata, 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.

[11] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.

[12] OMG, Object Constraint Language, Version 2.4, OMG Specification OMG Document Number: formal/2014-02-03, Object Management Group (Feb. 2014).

[13] G. Blair, N. Bencomo, R. B. France, Models@run.time, Computer 42 (10) (2009) 22–27.

[14] S. Challita, F. Paraiso, P. Merle, Towards Formal-based Semantic Interoperability in Multi-Clouds: The fclouds Framework, in: 10th IEEE International Conference on Cloud Computing (CLOUD), IEEE, 2017, pp. 710–713.

[15] H. Bruneliere, J. Cabot, F. Jouault, Combining Model-Driven Engineering and Cloud Computing, in: 4th edition of Modeling, Design, and Analysis for the Service Cloud Workshop (MDA4ServiceCloud'10), 2010.

[16] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann, A Systematic Review of Cloud Modeling Languages, ACM Computing Surveys (CSUR) 51 (1) (2018) 22.

[17] R. Nyrén, A. Edmonds, T. Metsch, B. Parák, Open Cloud Computing Interface – HTTP Protocol, Recommendation GFD-R-P.223, Open Grid Forum (Oct. 2016).

[18] A. Edmonds, T. Metsch, Open Cloud Computing Interface – Text Rendering, Recommendation GFD-R-P.229, Open Grid Forum (Oct. 2016).

[19] R. Nyrén, F. Feldhaus, B. Parák, Z. Sustr, Open Cloud Computing Interface – JSON Rendering, Recommendation GFD-R-P.226, Open Grid Forum (Oct. 2016).

[20] T. Metsch, A. Edmonds, B. Parák, Open Cloud Computing Interface – Infrastructure, Recommendation GFD-R-P.224, Open Grid Forum (Oct. 2016).

[21] M. Drescher, B. Parák, D. Wallom, OCCI Compute Resource Templates Profile, Recommendation GFD-R-P.222, Open Grid Forum (Oct. 2016).

[22] T. Metsch, M. Mohamed, Open Cloud Computing Interface – Platform, Recommendation GFD-R-P.227, Open Grid Forum (Oct. 2016).

[23] G. Katsaros, Open Cloud Computing Interface – Service Level Agreements, Recommendation GFD-R-P.228, Open Grid Forum (Oct. 2016).

[24] A. Ciuffoletti, Open Cloud Computing Interface - Monitoring Extension, Specification Document 1.2, Open Grid Forum, OCCI-WG (Jan. 2016).

[25] J. Parpaillon, P. Merle, O. Barais, M. Dutoo, F. Paraiso, OCCIware - A Formal and Tooled Framework for Managing Everything as a Service, in: CEUR (Ed.), Projects Showcase @ STAF'15, Vol. 1400 of Proceedings of the Projects Showcase @ STAF'15, L'Aquila, Italy, 2015, pp. 18 – 25. URL `https://hal.archives-ouvertes.fr/hal-01188826`

[26] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2012.

[27] H. Medhioub, B. Msekni, D. Zeghlache, OCNI – Open Cloud Networking Interface, in: 22nd International Conference on Computer Communications and Networks (ICCCN), IEEE, 2013, pp. 1–8.

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

[29] S. Yangui, S. Tata, An OCCI Compliant Model for PaaS Resources Description and Provisioning, The Computer Journal 59 (3) (2016) 308–324.

[30] A. Edmonds, T. Metsch, A. Papaspyrou, Open Cloud Computing Interface in Data Management-Related Setups, in: S. Fiore, G. Aloisio (Eds.), Grid and Cloud Database Management, Springer, 2011, pp. 23–48.

[31] A. Ciuffoletti, 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.

[32] M. Mohamed, D. Belaïd, S. Tata, Monitoring and Reconfiguration for OCCI Resources, in: 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2013), Vol. 1, IEEE, Bristol, United Kingdom, pp. 539–546.

[33] M. Mohamed, D. Belaïd, S. Tata, Autonomic Computing for OCCI Resources, Tech. rep., Telecom Sud Paris (Jan. 2014).

[34] M. Mohamed, Generic Monitoring and Reconfiguration for Service-based Applications in the Cloud, Ph.D. thesis, INT, Evry, France (2014).

[35] M. Mohamed, M. Amziani, D. Belaid, S. Tata, T. Melliti, An autonomic approach to manage elasticity of business processes in the cloud, Future Generation Computer Systems 50 (2015) 49–61.

[36] T. Berners-Lee, R. Fielding, L. Masinter, Uniform Resource Identifiers (URI): Generic Syntax (1998).

[37] F. Paraiso, S. Challita, Y. Al-Dhuraibi, P. Merle, 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, 2016, pp. 718–725.

[38] M. Ahmed-Nacer, S. Tata, Simulation Extension for Cloud Standard OCCIware, in: 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE, 2016, pp. 263–264.

[39] M. Ahmed-Nacer, W. Gaaloul, S. Tata, OCCI-Compliant Cloud Configuration Simulation, in: IEEE International Conference on Edge Computing, EDGE 2017, Honolulu, HI, USA, June 25-30, 2017, 2017, pp. 73–81.

[40] P. Merle, C. Gourdin, N. Mitton, Mobile Cloud Robotics as a Service with OCCIware, in: IEEE International Congress on Internet of Things, ICIOT 2017, Honolulu, HI, USA, June 25-30, 2017, 2017, pp. 50–57.

[41] F. Korte, S. Challita, F. Zalila, P. Merle, J. Grabowski, Model-driven configuration management of cloud applications with OCCI, in: Proceed-

ings of the 8th International Conference on Cloud Computing and Services Science, CLOSER 2018, Funchal, Madeira, Portugal, March 19-21, 2018., 2018, pp. 100–111.

[42] S. Challita, F. Zalila, C. Gourdin, P. Merle, A precise model for google cloud platform, in: 2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018, 2018, pp. 177–183.

[43] D. Jackson, Software Abstractions: Logic, Language, and Analysis - Revised edition, MIT Press, 2012.

[44] S. Challita, F. Zalila, P. Merle, Specifying semantic interoperability between heterogeneous cloud resources with the FCLOUDS formal language, in: 11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018, 2018, pp. 367–374.

[45] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, R. Buyya, CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms, Software: Practice and Experience 41 (1) (2011) 23–50.

[46] T. Binz, G. Breiter, F. Leyman, T. Spatzier, Portable Cloud Services Using TOSCA, IEEE Internet Computing (3) (2012) 80–85.

[47] F. Glaser, J. Erbel, J. Grabowski, Model Driven Cloud Orchestration by Combining TOSCA and OCCI, in: 7th International Conference on Cloud Computing and Services Science (CLOSER), 2017, pp. 644–650.

[48] E. Brandtzæg, S. Mosser, P. Mohagheghi, Towards CloudML, a Model-Based Approach to Provision Resources in the Clouds, in: 8th ECMFA, 2012, pp. 18–27.

[49] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg, Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-Cloud Systems, in: IEEE Sixth International Conference on Cloud Computing (CLOUD 2013), IEEE, pp. 887–894.

[50] H. R. M. Nezhad, K. Yorov, P. Yin, T. Nakamura, S. Trent, G. Shurek, T. Kushida, U. Subramanian, COOL: A Model-Driven and Automated System for Guided and Verifiable Cloud Solution Design, in: International Conference on Service-Oriented Computing, Springer, 2016, pp. 194–198.

[51] C. Quinton, N. Haderer, R. Rouvoy, L. Duchien, 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.

[52] C. Quinton, D. Romero, L. Duchien, SALOON: a platform for selecting and configuring cloud environments, Software: Practice and Experience 46 (2016) 55–78.

[53] S. Yangui, I.-J. Marshall, J.-P. Laisne, S. Tata, CompatibleOne: The Open Source Cloud Broker, Journal of Grid Computing 12 (1) (2014) 93–109.

[54] D. Weerasiri, M. C. Barukh, B. Benatallah, J. Cao, A Model-Driven Framework for Interoperable Cloud Resources Management, in: International Conference on Service-Oriented Computing, Springer, 2016, pp. 186–201.

[55] S. Yangui, I.-J. Marshall, J.-P. Laisne, S. Tata, CompatibleOne: The Open Source Cloud Broker, Journal of Grid Computing 12 (1) (2013) 1–17.

[56] Á. L. García, E. F. del Castillo, P. O. Fernández, OOI: OpenStack OCCI Interface, SoftwareX 5 (2016) 6–11.

[57] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner, OpenTOSCA – A Runtime for TOSCA-based Cloud Applications, in: International Conference on Service-Oriented Computing, Springer, 2013, pp. 692–695.

[58] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, A. Solberg, CloudMF: Model-Driven Management of Multi-Cloud Applications, ACM Transactions on Internet Technology (TOIT) 18 (2) (2018) 16.