GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

ISSN 1612-6793
Nr. ZAI-MSC-2017-23

Master's Thesis

submitted in partial fulfilment of the
requirements for the course "Applied Computer Science"

Comparison And Adaptation Of Cloud Application Topologies Using Models At Runtime

Johannes Martin Erbel

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

26. September 2017

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 `www.informatik.uni-goettingen.de`

First Supervisor:	Prof. Dr. Jens Grabowski
Second Supervisor:	Prof. Dr. Dieter Hogrefe

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 26. September 2017

Abstract

Cloud computing, a service to dynamically rent computing resources on demand, establishes the common standard for computing being a utility. Due to high demands for such a service, several provider created and offer their own cloud infrastructure. This, however, has lead to a problem affecting the portability of cloud applications, as each provider requires the usage of different tools and frameworks. Consequently, a provider lock-in is created making it costly to change the service provider once an application is build. To bypass this problem, standards are developed which design interfaces and frameworks generalizing the access to cloud services. One of these standards is the Open Cloud Computing Interface (OCCI), which defines a Representational State Transfer (REST) Application Programming Interface (API), enabling the possibility to create, change, and release cloud resources over simple REST calls. To do so, OCCI defines a metamodel allowing to describe complete cloud application, whereby the single model elements represent cloud resources containing the information required for the REST calls. Nevertheless, OCCI does not define how to provision complete cloud application topologies or how to adapt applications already running. Therefore, we propose a models at runtime approach providing this capability. We examine multiple distinct steps required to transform the running cloud application into the desired one. Hereby, we propose a generic comparison process that utilizes different strategies in order to calculate a possible match of cloud resources from the models. Based on this match, we mark how the different resources have to be treated serving as input for the different steps of the adaptation process. These steps comprise different management tasks handling the deprovisioning, updating, and provisioning of single resources, which we compound in order to create an adaptation process. Overall, our evaluation of the this model-based approach shows its general feasibility to adapt running cloud application topologies using the OCCI standard.

Contents

Chapter 1

Introduction

High demand for computing as utility brought up a flourishing economy for cloud computing services, due to the elastic nature of virtualized resources. Hereby, the demand led to the creation of multiple companies providing such a service, whereby the way of accessing it differs from provider to provider. This diversity, however, resulted in the *provider lock-in* problem, i.e. the binding of a customer to a provider once a cloud application has been build. Meanwhile, many different approaches and standards evolved in order to solve this problem. One of them is the *Open Cloud Computing Interface (OCCI)* [1], a standard developed by the *Open Grid Forum (OGF)*. OCCI defines a universal interface to access and manage cloud resources over *Representational State Transfer (REST)* calls, a programming paradigm for web services [2]. For the derivation of the information required for the REST calls, OCCI defines a model which is capable of not only describing single cloud resources, but also depict complete cloud topologies.

In this thesis, we propose a *models at runtime* approach utilizing these models in order to automatically adapt a running cloud application topology. This allows not only to react to changing requirements and environments as soon as possible, but also to bypass a manual adaptation of the topology, which typically is a time-consuming and error prone process. To implement such an approach, the runtime model needs to be compared against the model depicting the desired state of the cloud application topology. This allows to evaluate which resources have to be changed, created or deleted.

The results of this comparison are utilized in several adaptation steps handling the deletion, creation, and update of single elements by performing the required REST calls. Summed up, the complete adaptation process is separated in the extraction of the runtime model, the comparison and required steps to be performed, which are described and evaluated in this thesis. In the following, the goals and scope of this thesis are enumerated.

1.1 Goals and Scope

The goals and scope of this paper can be summarized as follows:

- investigate comparison strategies to compare two OCCI models,
- examine required treatment of the cloud resources, based on this comparison,
- define adaptive steps to transform the cloud application into the desired state,
- evaluate the complete adaptation process and the limits of the comparison strategies.

1.2 Outline

Chapter 2 covers fundamental knowledge about cloud computing, the OCCI standard, and the principles of *Model Driven Engineering (MDE)*. Chapter 3 elicits requirements for the complete adaptation process highlighting different comparison strategies and required adaptation steps. Chapter 4 provides an overview of the complete adaptation process and a detailed design of the single adaptation steps and comparison strategies. Chapter 5 covers details about the implementation of the proposed approach, including the used tools and selected implementation challenges. Chapter 6 evaluates the comparison strategies and the complete adaptation process, using the implemented prototype. Chapter 7 discusses related work delimiting similar approaches to the proposed one. Finally, in Chapter 8, an overall conclusion is given, summing up the results of this thesis, followed by an outlook on future work.

Chapter 2

Basics

To understand the necessity and the process behind the adaptation and comparison techniques, this chapter provides fundamental knowledge about cloud computing and the MDE development paradigm. At first, in Section 2.1, a common definition of cloud computing is given in addition to an explanation of the service itself. Section 2.2 introduces the OCCI standard we utilize for the cloud management calls. Finally, Section 2.3 examines the MDE development paradigm highlighting its assumptions and techniques.

2.1 Cloud Computing

Cloud computing is a service which allows to dynamically rent computing resources on demand, establishing a way for computing to be a utility [3]. Due to this utilization of computing resources, cloud computing wipes out the necessity of companies to posses own physical hardware and human effort to maintain it. To offer such a service, a cloud provider pools a large amount of physical hardware together [4], which can be split via virtualization to make it available to the consumer [5]. This large amount of on-demand available resources eliminates the need of consumers to plan the required resources, as the amount of virtualized resources can be scaled at all time to quickly react to changing requirements [6]. Summed up, the elasticity of cloud resources allows a perfect utilization of computing resources, because the amount of resources can be adapted to the actual need, as depicted in Figure 2.1. This figure shows the problems of static resource provisioning, called Over- and Underprovisioning and how it is solved using dynamic provisioning of cloud resources. The Overprovisioning (Figure 2.1 a) comes with the drawback of paying unused resources on low demand periods, as the capacity is fit to the peak of the expected demand. If the capacity is set to a lower amount of peak demand times, as it is in the Underprovisioning case (Figure 2.1 b), not enough resources are available to handle all incoming requests, leading to a dissatisfaction of users and therefore to a decreasing demand overall (Figure 2.1 c). However, when using Cloud Provisioning (Figure 2.1 d), the capacity can be dynamically adjusted to the demand. This especially addresses an economical benefit, as only the required resources have to be rented. Consequently, cloud computing employs a *pay-as-you-go* pricing model, lowering the service operation cost for consumers [5].

To provide a common definition of cloud computing, Section 2.1.1, describes the definition developed by the *National Institute of Standards and Technology (NIST) [4]*. Furthermore, an abstract view of the architecture of cloud infrastructures is given in Section 2.1.2. Finally, Section 2.1.3 addresses the provider lock-in problem, one of the major drawbacks of cloud computing which we address in this thesis.

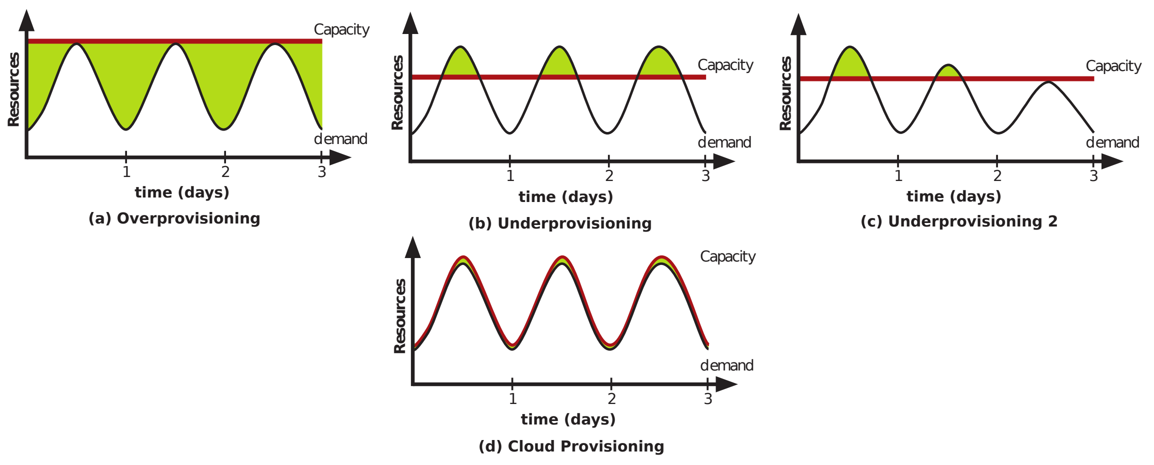


Figure 2.1: Dynamic versus static resource provisioning (adapted from Armbrust et al. [3]).

2.1.1 Cloud Computing Definition

Compared to the description of cloud computing provided above, the NIST standard provides a more detailed and formalized version: "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [4]. To deepen this definition, NIST describes five *essential characteristics* required by a cloud system, three *service models*, which describe the services a cloud provider can offer, and four *deployment models*, which define the scope of consumers having access to the cloud. In the following, these essential characteristics, service models, and deployment models are described.

Essential Characteristics

The NIST standard for cloud computing defines five essential characteristics of a cloud system:

On-demand self-service allows consumers to conveniently provision cloud resources, like computation time or storage, without the need for any human interaction. Furthermore, due to permanent availability, it grants the capability to rent the resources on demand.

Broad network access requires a cloud service to offer standardized mechanisms which allow heterogeneous client systems to access the cloud resources over a network. It should be noted that these standardized mechanisms differ from provider to provider.

Resource pooling composes the cloud system of a large pool of physical resources, which get shared by multiple consumers. These resources get virtualized and dynamically assigned and reassigned to the different consumers depending on their demand. Hereby, no information of the exact physical location of the resources are given. A consumer can only influence the location of the resources on a high abstraction level to configure for example, the country, state or data center in which the resources should be provisioned.

Rapid elasticity allows consumers to provision and release resources at any point in time, which creates the illusion of infinite available resources. Therefore, a rapid in- and out-scaling of rented resources is required.

Measured service requires a cloud to provide a monitoring service. This service makes the cloud's behavior transparent to both the consumer and provider. Moreover, by leveraging metering capabilities, the cloud system is able to optimize itself by adjusting its resource utilization. Hereby, the degree of optimization depends on the kind of resource measured (e.g., storage, processing, active user accounts).

Service Models

Overall, the NIST standard [4] defines three different service models a cloud provider can offer:

Software as a Service (SaaS) offers a pre-build software application, running on top of a cloud infrastructure. In case of SaaS, the consumer represents a user, as he has no control over the underlying infrastructure or the software configuration. Only a limited amount of user specific settings can be configured. One example for a SaaS is Google's *GSuite* [7], which comprises multiple kinds of office and email software.

Platform as a Service (PaaS) allows a consumer to deploy a self chosen software application onto a cloud infrastructure, whereby the application needs to meet the requirements of the cloud regarding supported tools. Compared to SaaS, the consumer needs to handle the hosting environment and software configurations of the chosen application. Nevertheless, the consumer has still no control over the underlying infrastructure. Two examples for PaaS are the *Google App Engine (GAE)* [8] and *Amazon Web Services (AWS) Elastic Beanstalk* [9], which allow consumers to deploy modern web and mobile applications.

In addition to deploying self chosen software, *Infrastructure as a Service (IaaS)* allows the management of complete infrastructure, as it grants direct access to any kind of provisioned resource, covering processing, storage or network resources. Therefore, the consumer is able to deploy and run any kind of software on the provisioned resources, including for example *Operating Systems (OSs)*. Nevertheless, the underlying infrastructure of the pooled physical resources is still inaccessible to the consumer. One example for IaaS is Amazon's *Elastic Compute Cloud (EC2)* [10].

Summed up, IaaS builds the most flexible service but also requires the largest amount of management effort. Compared to IaaS, PaaS comes with the benefit that only the deployed application needs to be maintained and configured, without paying attention to the underlying infrastructure. Finally, SaaS provides an easy and convenient way to make use of cloud resources, as complete software systems are rented including their maintenance. Overall, consumers can individually choose the different service models to fit their needs.

Deployment Models

NIST [4] names four kinds of deployment models, defining the scope of consumers having access to the cloud and the kind of organizations providing it:

Private clouds are cloud infrastructures exclusively managed to be used by single organizations having multiple consumers. Hereby, the infrastructure can either be managed by the organization itself or by a third party, and therefore can be on or off premises. For example, a cloud infrastructure provided by a research group, with only its employees having access to it.

Community clouds are provided exclusively for organizations sharing a specific concern. This infrastructure is managed by a subset of the community members or a third party. Thus, it can be on or off premises of one of the involved organizations.

Public clouds are cloud infrastructures available to the general public, which is managed by any kind of organization existing on their premises. Examples for public cloud providers are Amazon and Google.

Hybrid clouds combine at least two of the above listed deployment models, whereby the combined cloud infrastructures are unique entities. Hereby, these are connected over technologies that enable data and application portability. This combination of multiple cloud infrastructures allows for a larger pool of cloud resources and for a possible load balancing between the connected clouds.

2.1.2 Cloud Computing Architecture

In addition to the NIST standard [4], this section provides an overview of the architectural structure of a cloud infrastructure. The architecture is composed of four modular layers [5], as depicted in Figure 2.2: a hardware-, infrastructure-, platform- and an application layer. Furthermore, this figure shows the layers a consumer has access to depending on the cloud service models. The hardware layer represents the physical resources of the cloud system. These are typically managed in large data centers configuring the servers, routers and switches that together build the cloud infrastructure. On top of the hardware layer is the infrastructure layer, which is also known as *virtualization layer*. This layer partitions the physical resources of the cloud making them available to the consumer. For example, this layer instantiates *Virtual Machines (VMs)* requested by the consumer. Hereby, different virtualization technologies are used, such as *Xen* [11], *KVM* [12], and *VMWare* [13]. The platform layer defines frameworks and *Application Programming Interfaces (APIs)*, allowing for cloud applications to be deployed on. In the application layer the running cloud applications are contained utilizing the automatic-scaling capabilities of the underlying infrastructure. According to Zhang et al. [5], these layers are loosely coupled, which allows them to evolve separately with a minimum amount of management and maintenance overhead.

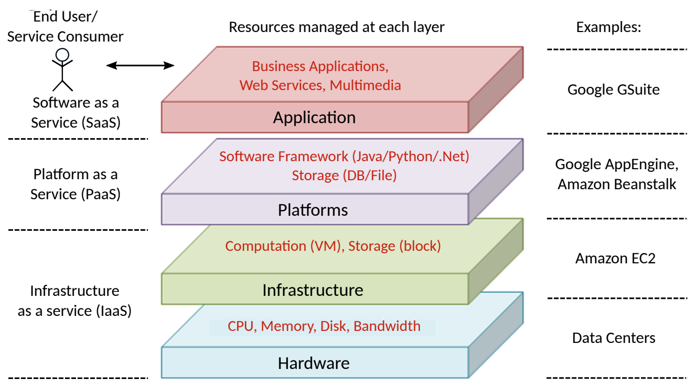


Figure 2.2: Cloud computing architecture (adapted from Zhang et al. [5]).

2.1.3 Provider Lock-In

Even though cloud computing establishes a convenient way to use computing resources as utility, several obstacles exist without a proper solution yet. One of these problems is the *provider lock-in*, which describes a vendor lock-in that binds a cloud consumer to a provider once a cloud application is build or established [14]. This lock-in results from the different APIs and frameworks used by cloud providers, as it is complex and costly to adapt an already build cloud application to meet the requirement of another cloud system. For example, one crucial point is the kind of virtualization technology used in the infrastructure layer, as it is not granted that a cloud application behaves in the same manner under the use of different virtualization techniques [14]. Meanwhile, many different cloud computing standards have been developed to tackle this issue, which are mainly accepted among open source cloud middlewares such as *OpenNebula* [15] and *OpenStack* [16]. One of these standards is OCCI, which is described in the following section.

2.2 Open Cloud Computing Interface

In the scope of this thesis we utilize the Open Cloud Computing Interface (OCCI) [17], a standard by the OGF [18], to build abstract representations of cloud applications, which we compare, deploy and adapt. In general, the standard defines an API and boundary protocol, which allows for a simple management of cloud resources over REST calls. Hereby, the OCCI interface can be implemented by the provider in addition to any proprietary API, as depicted in Figure 2.3. This figure depicts OCCI's place in the cloud infrastructure from a provider's point of view. Here, it is highlighted that a consumer can access the cloud resources over simple REST calls, which use the *Hypertext Transfer Protocol (HTTP)*. Moreover, it shows that the OCCI interface is connected to the provider's internal resource management framework, allowing for a uniform access of cloud resources [1]. Originally, OCCI was developed for the purpose of managing IaaS resources. Meanwhile, it is further enhanced to support the management of PaaS resources and *Service Level Agreements (SLAs)* [19]. OCCI's specification consists of multiple documents describing the core of the standard and multiple extensions to enhance the core's functionality. In the following these documents are described in order to examine the capabilities of the OCCI interface. Section 2.2.1 covers the definition of OCCI's *Core Model*. Section 2.2.2 deals with the IaaS extension of the Core Model, whereas Section 2.2.3 investigates the PaaS extension. Finally, Section 2.2.4 describes how to interact with OCCI using the REST API.

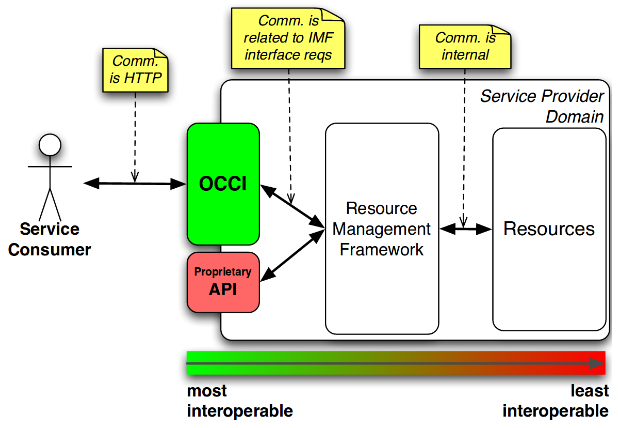


Figure 2.3: The OCCI API in a provider's architecture [1].

2.2.1 Core Model

The Core Model of OCCI defines a model used to abstract real-world cloud resources and, therefore, builds the main component of the standard [1]. Hereby, the model's elements represent single resources that are managed over REST calls. Figure 2.4 provides an overview of the OCCI Core Model. Overall, the OCCI Core Model is composed of two parts, the core base types and the classification and identification mechanisms [1]. In the following, these two parts are investigated in detail.

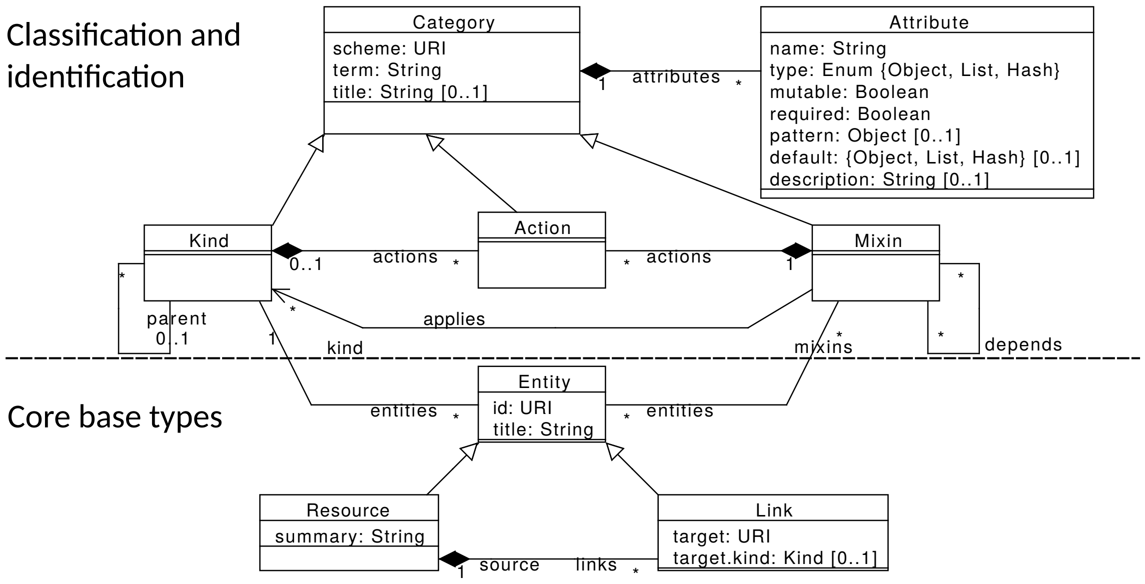


Figure 2.4: OCCI's Core Model (adapted from OCCI's Core specification [1]).

The core base types comprise the elements Entity, Resource, and Link. The abstracted real world resources, for example VMs storages or networks, are represented by instantiations of the Resource element. To describe a connection between two Resources, the source Resource contains a Link element, providing information about the target. Both, Resource and Link, inherit from the abstract Entity element, which defines title and id to identify those. Each Entity, i.e., each Resource and each Link, is bound to one Kind and can be related to any amount of Mixins classifying them.

The classification and identification mechanisms are build upon the elements Category, Attribute, Kind, Action and Mixin. The abstract Category element represents the basis of the identification mechanisms. It defines a scheme and term attribute, which uniquely identify Category instantiations and their location, e.g., http://example.com/category/scheme#term. Category instances may define any amount of Attributes representing client readable attributes. It should be noted, that each Attribute is uniquely bound to one specific Category instance, as depicted by the composition relationship.

The key feature of the classification system is the Kind element, which specifies related Entity sub-types. These define Attribute structures an Entity fills with values allowing for a detailed abstraction of the real world resource. Moreover, Kinds may define Actions that can be performed on related Entities. Each Kind is able to inherit from another Kind, whereby each Kind has to be related to one of the three core Kinds, which are equally named to the core base types Entity, Resource and Link. The inheritance hierarchy to the core Kinds ensures general functionality and information among each Entity and allows the Core Model to be extensible towards more domain-specific usage.

Another element used in OCCI's classification system is the Mixin element, allowing to add additional capabilities to related Entities at runtime. Mixins complement Kinds, regarding the classification mechanisms, as they define further specifications for Entities. In some cases Mixins can only be applied to Entities being of a specific Kind. Therefore, each Mixin element defines a list of Kinds on which it can be applied. Moreover, each Mixin stores a list of Mixins required in order to work properly. Mixins add the capability for service provider to define *Templates*, which can be assigned to an Entity at creation-time. Templates are Mixins containing default values for their Attributes. These are used for example to define the available VM sizes or *images*, specifying its OS, data files, and applications.

The Action element identifies operations that can be invoked on Entities. Hereby, an Action element only serves as an identifier and does not describe the implementation of the operation itself. Actions are only able to define Attributes, serving as input parameters required by the operation. Each Action is either bound to a Kind or Mixin, as it is considered as a capability of those. Typical examples of Actions are operations to start, stop or terminate VMs or storage resize operations. In the following, the IaaS extension of the OCCI Core Model is discussed.

2.2.2 Infrastructure Extension

The infrastructure extension [20] enhances the OCCI Core Model by adding specialized Entity types required to abstract and manage typical IaaS resources, such as VMs, networks, and storages. Figure 2.5 provides an overview of the infrastructure extension, depicting three new Resource types and two new Link types. Each type inherits from either Resource or Link and is assigned to an equivalent named Kind. Here, the Attributes defined by the Kind are depicted within the Entity type. Moreover, with the infrastructure extension, OCCI introduces two new Templates, which we, together with the new types, investigate in the following.

The IaaS Resource types are Compute, Storage and Network. Compute represents an abstraction of a generic processing resource, for example a VM or container [20]. Hereby, multiple Attributes of a compute resource must be filled, giving information about the machine's number of cores, memory and processor architecture. The Network type is used to create networking entities supporting the *Data link layer (L2)*, e.g., virtual switches. Networks are typically extended by a Mixin, called

IPNetwork, enabling *Network layer (L3)* and *Transport layer (L4)* capabilities such as *Transmission Control Protocol (TCP)* and *Internet Protocol (IP)*. IPNetwork defines Attributes to specify the address range, gateway and allocation type of a Network. The Storage type is used to describe data storage devices, whereby its size is defined over the size Attribute. It should be noted, that all of these Resource types posses a state Attribute and basic functionalities, in terms of Actions, such as starting and stopping an instance.

The Link types specified in the infrastructure extension, NetworkInterface and StorageLink, allow to connect Compute instances to Storages or Networks. A connection between a Compute and a Network instance is described by the NetworkInterface type, representing a L2 client device. Just as the Network type, NetworkInterface can be enhanced to support the L3 and L4 layer, by attaching a Mixin called *IPNetworkInterface*. The NetworkInterface instance allows to add information about the device interface, such as an ethernet interface, and *Media Access Control (MAC)*. To allow a communication on the L3 and L4 layer, the Attributes of the IPNetworkInterface specify the device's network address and gateway. The connection between a Compute and Storage instance is described over a StorageLink. Here, the device id and optionally the mount point of the Storage are defined. Finally, just like the Resource types, each Link type possesses an Attribute describing the state of the instance.

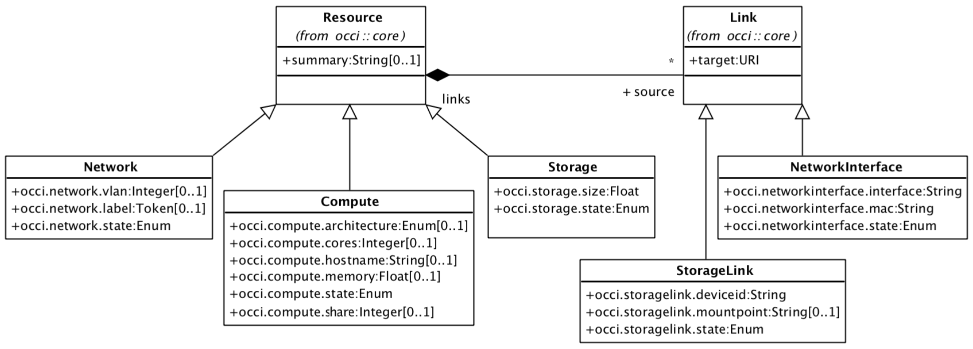

Resource
(from occi :: core)
+summary:String[0..1]
links
*
+ source
Link
(from occi :: core)
+target:URI
Network
+occi.network.vlan:Integer[0..1]
+occi.network.label:Token[0..1]
+occi.network.state:Enum
Compute
+occi.compute.architecture:Enum[0..1]
+occi.compute.cores:Integer[0..1]
+occi.compute.hostname:String[0..1]
+occi.compute.memory:Float[0..1]
+occi.compute.state:Enum
+occi.compute.share:Integer[0..1]
Storage
+occi.storage.size:Float
+occi.storage.state:Enum
NetworkInterface
+occi.networkinterface.interface:String
+occi.networkinterface.mac:String
+occi.networkinterface.state:Enum
StorageLink
+occi.storagelink.deviceid:String
+occi.storagelink.mountpoint:String[0..1]
+occi.storagelink.state:Enum

Figure 2.5: OCCI's infrastructure extension [20].

In addition to the Entity types, the infrastructure extension defines two major Templates, the *OS Template* and *Resource Template*, that can be utilized by the provider. The OS Template allows to specify pre-configured images that can be attached to Compute instance. Similar to the OS Template, the Resource Template defines a pre-set Resource configuration, which for example let the provider define sizes for VMs. If a new OS- or Resource Template is assigned to a Resource at runtime, the OCCI implementation is supposed to immediately remove the old Template and associate the new one. This kind of adaptation affects the Resource in a provider-specific way and has to return an error if the functionality is not supported. Furthermore, the extension defines Mixins, used to add *Secure Shell (SSH)* keys to a Compute instance or execute programs on it, once it is started. In the following, the platform extension of the OCCI Core Model is discussed.

2.2.3 Platform Extension

The platform extension [21] defines types to describe and support the management of PaaS resources. Overall, two Resource types and one Link type are specified, which are depicted in Figure 2.6. Furthermore, the platform extension describes two Templates used to create ready to use configurations for Applications. In the following, the types and Templates specified by the OCCI platform extension are described.

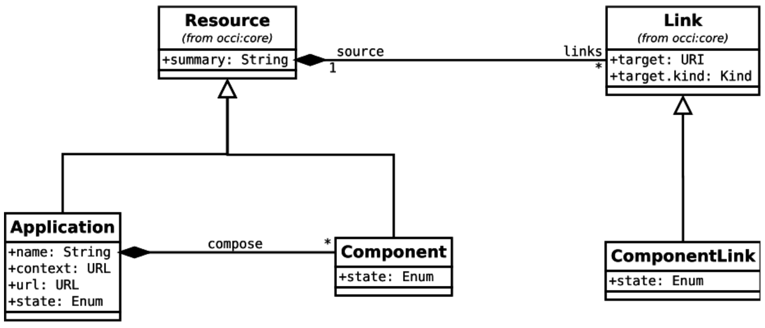


Figure 2.6: OCCI's platform extension [21].

The Application type represents the consumer-defined part of the service. Each Application specifies Attributes giving information about the application's context and the *Domain Name System (DNS)* entry, both in form of an *Uniform Resource Locator (URL)*. An Application instance is composed of Components providing it with business functionality. This composition of an Application from several Components forms an acyclic graph. Components define micro services that are responsible for the application's execution or hosting. To connect a hosting Component to an Application, the Link sub-type ComponentLink is used. Moreover, ComponentLink is used to connect two Components.

The platform extension defines two kinds of Templates. The *Application Template* and the platform *Resource Template*. Application Templates allow provider to pre-define underlying frameworks for applications, whereas platform Resource Templates specify preset resource configurations for Applications. Furthermore, the OCCI platform extension defines two Mixins that should be supported by an OCCI platform extension, *Database* and *DatabaseLink*. These handle the creation and connection of databases, whereby *Database* is applied to a *Component* instance and *DatabaseLink* to a *ComponentLink*, having a *Database* as target. The following section describes how the cloud resources and their connection described within the Resources and Links can be managed and accessed over the OCCI implementation.

2.2.4 RESTful API

As OCCI represents an API to manage cloud resources based on REST, this section does not only investigate how to interact with an implementation of the OCCI interface, but also gives a short introduction to REST itself. REST is a common web architecture style originally designed by Roy T. Fielding [2]. It defines multiple architectural constraints to optimize network communication, such as caching [22]. In general, a REST interface provides a uniform access to manage hypermedia resources over *Uniform Resource Identifiers (URIs)*. In case of OCCI, these hypermedia resources are Entities. To access and manage these resources, REST uses HTTP, a generic and stateless application-level protocol for distributed hypermedia systems [23]. REST is a stateless architecture style, as each request from the client must carry all necessary information to understand the request. Overall, four major methods are defined to perform management tasks: *GET*, *PUT*, *DELETE*, and *POST*. The GET request defines an operation to retrieve all information of the element hidden behind the URI [23]. POST is used to create a resource in the target system as specified in the request. PUT is used to update or create a resource at the specified in the URI, whereas DELETE is used to delete it. These methods are also known as *Create, Read, Update, Delete (CRUD)* operations. In the following, the behavior of these CRUD operations on the different OCCI elements is discussed. An overview is given in Table 2.1.

Path	**GET**	**POST**	**POST**(Action)	**PUT**	**DELETE**
Entity sub-type (`/compute/1`).	Retrieve the Entity.	Partial update of the Entity.	Perform an action on the Entity.	Create / Update the Entity.	Delete the Entity.
Entity collection (`/compute/`).	Retrieve the Entity collection.	Create a new Entity in this collection.	Perform actions on the Entity collection.	Not defined.	Remove all Entities from the collection.
Mixin-defined Entity collection (`/my_stuff/`).	Retrieve a collection of Entity of the given sub-type.	Add an Entity to this collection.	Perform actions on the Entity collection.	Update the Entity collection.	Remove all Entities from the collection.
Query interface (`/-/`)	Retrieve Category instances.	Add a user-defined Mixin.	Not defined.	Not defined.	Remove a user-defined Mixin.

Table 2.1: CRUD operation behavior (adapted from OCCI's HTTP specification [24]).

This table summarizes how the CRUD operations behave for specific URI paths. When a concrete Entity sub-type instance is stated in the URI, in this case `/1`, the specific instance is affected. When the path of the URI ends with a Kind, the operation chosen affects each Entity sub-type instance implementing this Kind. In this case `/compute/` or `/my_stuff/`. If the URI path is set to `/-/`, the *query interface* is accessed, which is used to retrieve and manage Category instances. This interface helps the client to determine the capabilities of the API implementing OCCI, for example to retrieve the information about possible VM images stored in OS Templates. The syntax of the management requests are defined within the rendering documents of OCCI's specification suite. Here, OCCI defines a *text rendering* [25] and a rendering based on the *JavaScript Object Notation (JSON)* [26]. As OCCI already defines a model to abstracts cloud applications, the following section investigates the MDE development paradigm, suitable of making use of such constructs.

2.3 Model Driven Engineering

Model Driven Engineering (MDE) is a development paradigm utilizing the multiple benefits of abstracting systems via models. Hereby, MDE does not only use these *models* as a descriptive sketch of the system, but further utilizes them as primary artifacts for the development process [27]. Compared to the object-oriented programming paradigm, where "everything is an object" [28], MDE assumes that "everything is a model" [28]. Hereby, MDE focuses on a precise modeling and *model transformations*, together building its core concepts [29]. A commonly known MDE approach is the *Model-driven architecture (MDA)* [30], published in November 2000 by the *Object Management Group (OMG)*, which is build around other OMG standards like the *Meta Object Facility (MOF)*, *XML Metadata Interchange (XMI)* and *Unified Modeling Language (UML)*. As the OMG provides a large portion of standards used for MDE, some concepts are already biased towards the ones OMG specifies [29]. To provide a sufficient introduction into the MDE development paradigm, the following section investigates its core concepts. Section 2.3.1 covers the definition of the term model. Section 2.3.2 describes the *metamodel* term and explains its relation to models. Section 2.3.3, covers the core operation used in MDE approaches, the model transformation. Finally, the concept of *models at runtime* is presented in Section 2.3.4.

2.3.1 Models

Due to the high abstraction and applicability of *models* in multiple scientific areas, multiple definitions of this term have emerged. In case of MDE, these definitions are partially shaped by the definition specified by the OMG. A generic definition by Kühne defines models as follows: "*A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made*" [31]. Here, the focus is the abstractive nature of models allowing for a simple and easy analysis of a system. To reach this goal, Stachowiak defines three general features a model requires [32]. The *mapping-*, *reduction-*, and *pragmatic feature*. The mapping feature describes that a model has to be based on a system, which Stachowiak refers to as *original*. The reduction feature requires a model to only describe a well-chosen selection of the system's properties, reducing the amount of information necessary to interpret the model to a minimum. These two features are responsible for models being a "*projection*" of a system, as some information is lost during the abstraction process. The pragmatic feature refers to the usability of the model, considering the individual or algorithm interpreting it. This feature is especially important as "an interpretation of a model gives a model meaning" [27]. Finally, it must be mentioned that a model either has a *descriptive* purpose, describing an already existing system, or a *prescriptive* purpose, to describe a system to be build [27, 31]. Summed up, a model is an abstract representation of relevant parts of a complex system tailored towards a specific group of persons in order to allow for an easy interpretation. To completely understand how models are created the term *metamodel* has to be investigated as "a model is an instance of a metamodel" [33].

2.3.2 Metamodels

In general, the term *meta* is a prefix used when an operation is applied twice [31]. In case of the term metamodel, it describes "a model of models" [30], which can also be interpreted as "a language of models" [29]. Overall, the concept of metamodels also appears under different names. [29]. For example, in the Extensible Markup Language (XML) [34], a text format to encode data structures, a *XML Schema (XSD)* is used to define its structure. Here, the XSD, even though it is referenced as schema, can be considered a metamodel. In case of OCCI, the OCCI Core Model plus its extensions serve as metamodel, as they describe how to model cloud applications. For example, the Compute type defines the structure and attributes of possible VMs. Using this metamodel element, a Compute instance can be created, representing a real world computing resource.

To allow the creation of such metamodels, as the OCCI Core Model, another language or model is required. These are known as *meta-metamodels*. Two of the more commonly known meta-meta modeling languages are the MOF [35], by the OMG, and *Ecore* of the *Eclipse Modeling Framework (EMF)*. In Figure 2.7, an overview of the different *meta-layers* is given, in which we contextualize the OCCI Core Model.

This figure shows an example four-layer model stack, which is a typical size for a meta-model hierarchy. Hereby, we must emphasize that this size can vary, especially due to the usage of extensions enlarging the metamodel [35]. At the top level of this hierarchy (M3) resides the meta-metamodel. Here, only a subset of its elements is shown, defining the generic element Class. This element can be used to create a variety of metamodels (M2.5), such as the OCCI Core Model. Each element of the OCCI Core Model is an instance of the Class element from the meta-meta model. As the infrastructure extension (M2) extends the OCCI Core Model, it also represents a metamodel. Here, the Compute type is instantiated, whereby compute, defining the Attributes occi.compute.cores and occi.compute.memory. Therefore, the OCCI Core Model itself can be seen as a metamodel, from which not only user models, but also extensions can be created. To completely specify the IaaS extension as part of the metamodel, an element Compute must be be provided, having compute as Kind. This allows to create a Compute instance in the OCCI User

Model (M1). In this case a Compute instance VM_0, having two cores and 4096 MB memory, is modeled. At the lowest level (M0), the actual cloud resource VM_0 from the real world is located, which is represented by the abstract Compute instance VM_0 (M1). It should be noted, that the OCCI Core Model allows to define additional Kinds within the user model, which serve as a meta-model within the model. For example, when the Kinds of the IaaS extension are defined within the user model, a VM can be represented over a Resource being of Kind compute.

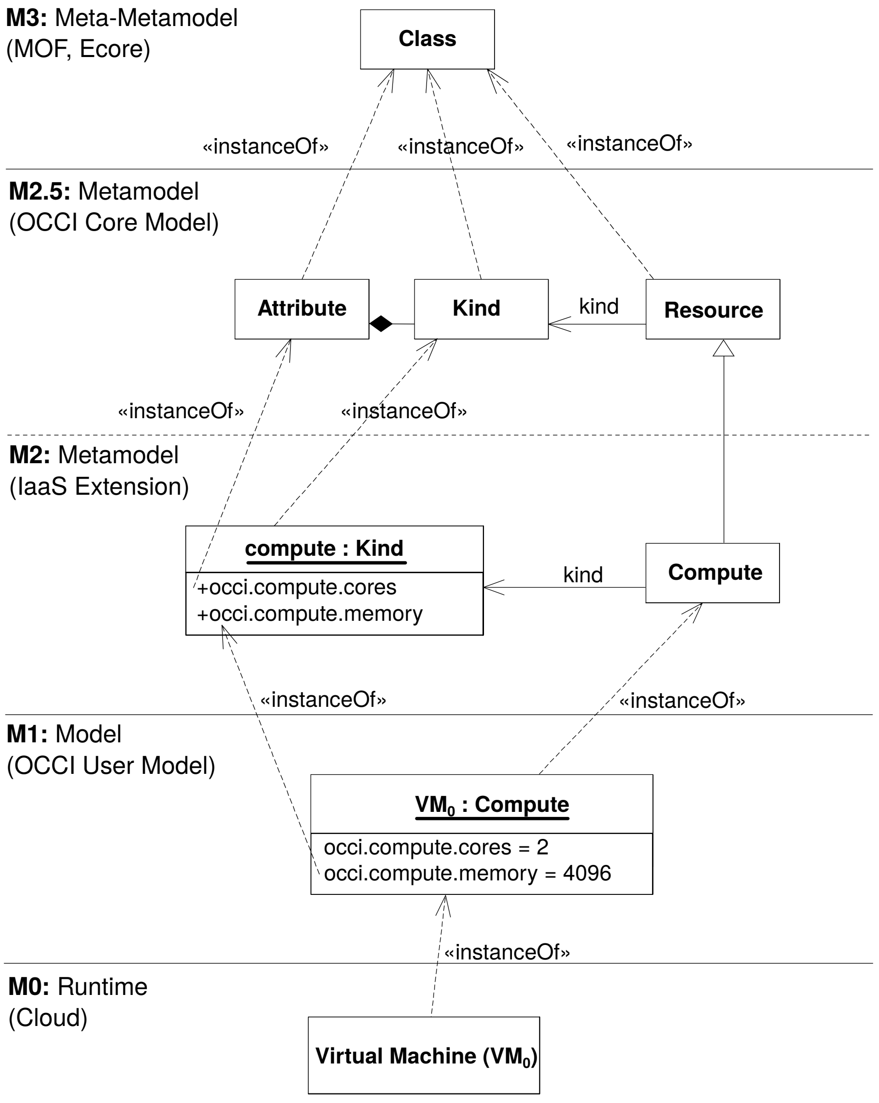


Figure 2.7: The OCCI Core Model in a four-layer metamodel hierarchy.

2.3.3 Model Transformations

Another fundamental concept of MDE is the model transformation, a methodology used to "develop, maintain, and evolve software" [36] based on models. Kleppe et al. define model transformations as follows: "*A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language*" [37]. Figure 2.8 illustrates the typical structure of a model transformation, which we adapted to highlight the different parts of a model transformation as defined above.

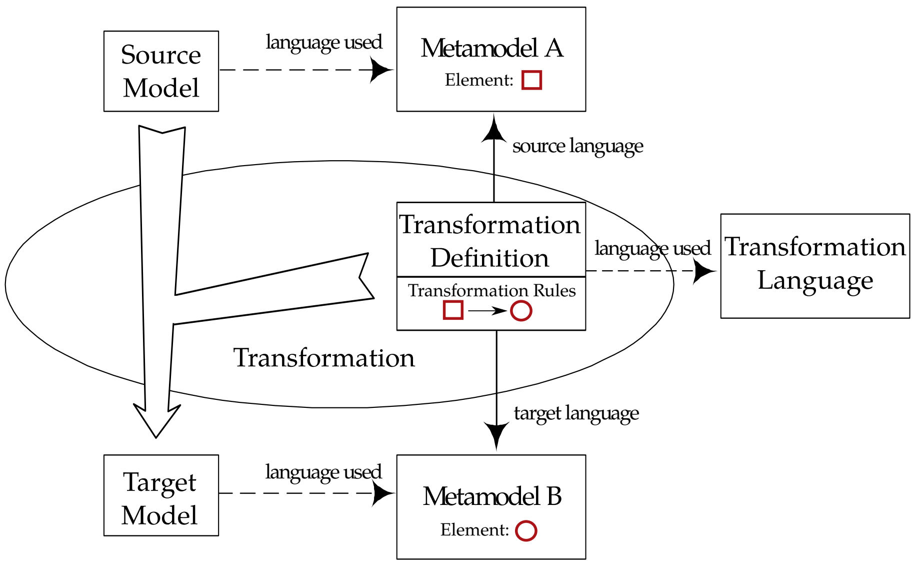


Figure 2.8: Typical model transformation procedure (adapted from the MDA Guide [30]).

This figure shows two models, whereby both are based on a specific metamodel. In this case the source model is an instance of metamodel A, consisting of squares, whereas the target model is an instance of metamodel B, consisting of circles. Based on these metamodels a transformation definition is specified using a transformation language, such as the *Epsilon Transformation Language (ETL)* [38]. This transformation definition holds transformation rules, which in this case describe the transformation of squares into circles. It must be noted, that the transformation definition itself can be considered as models [30]. A complete taxonomy of model transformation is given by Mens and Van Gorp [36], which extend the definition of model transformation given above to cover the use of multiple source and target models.

2.3.4 Models at Runtime

Models can not only be used to depict, but also to manage running systems. In general, this methodology utilizes a system's runtime information to create a feedback loop with a causally connected model representing it [39]. Based on this up-to-date information, multiple objectives, such as adaptation and self-healing, can be reached. Furthermore, it allows to manipulate the system on the model layer, and therefore on a higher abstraction level. An overall understanding of models at runtime is given by participants of a workshop study by Bencomo et al., who define it as an "abstraction of a running system that is being manipulated at runtime for a specific purpose" [40]. In the scope of this thesis, the running system is represented by a cloud system, implementing an OCCI conform API, whereby the model representation is a model instantiating the OCCI Core Model. To grasp the large amount of benefits coping with this methodology, this section investigates common objectives and techniques of models at runtime.

According to Szvetits and Zdun [39], models at runtime is used to reach multiple kinds of objectives. Hereby, they examined that it is mainly used to create adaptive system allowing systems to change according to shifting environment. Moreover, as for any MDE approach, one objective is to create a system that operates on a high abstraction level, which is closer to the problem space. Furthermore, platform independence is a common goal, as this high abstraction level grants a generalized view on the system, and therefore does not consider specific platforms. Additional objectives comprise monitoring, error handling, checking the consistency and conformance and ensuring policies.

In addition to model transformation, Szvetits and Zdun [39] examined multiple techniques that are typically used in models at runtime approaches. First of all, *Autonomic control loops*, originally designed by Kephart and Chess [41], is a technique for self managing systems. The autonomic control loop itself is a technique to measure system parameters, which get analyzed in order to plan corrective actions and executes them. This control loop is commonly used to handle adaptation scenarios and is formerly known as *Monitor-Analyze-Plan-Execute (MAPE)*-loop [39]. This loop is depicted in Figure 2.9.

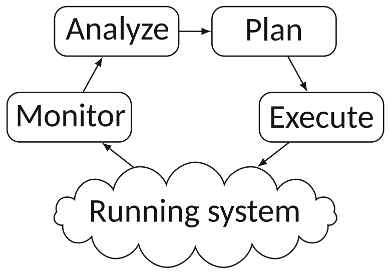


Figure 2.9: Visualization of the MAPE control loop [39].

The loop starts with the monitoring and measuring of chosen parameters. For example, parameters reflecting the workload of the system. To extract these parameters, different *introspection techniques* are used, such as checking *event-logs* or using a predefined API. Thereafter, these parameters are analyzed in order to identify corrective actions. Based on these, a plan is generated, describing imperative steps to bring the running system into the desired state. Then, this plan is executed, adapting the running system to fit into the new circumstances. Another technique commonly used in models at runtime approaches is *model conformance*, a technique to check whether data and processes are conform and consistent to each other. For example, the extracted runtime information is checked against the causally connected model representation of the system. Hereby, potential inconsistencies can be investigated and addressed, enabling self-healing capabilities. Furthermore, *model comparison* is a common technique used to identify the differences between two models, which can be used to examine the operations needed to transform one model into another one. Typically, one of these models represent a system state to transition to. This transition is performed by a model transformation. In the following section, we use the described basics in order to analyze the requirements for a comparison and adaptation approach which is able to adapt cloud applications over an OCCI API.

Chapter 3

Analysis

Even though cloud computing represents a milestone for dynamically providing computing resources as a utility, the provider lock-in still represents a major problem. To overcome this problem a comparison and adaptation process is required using a standardized API, such as OCCI, as it provides universal access to cloud resources. The different kind of resources OCCI can manage are described over the OCCI Core Model, a metamodel which allows to build complete cloud application topologies within a model.

Based on the model's elements, REST calls can be derived which create, update, and delete cloud entities such as networks, storages or compute instances. Furthermore, operations can be executed over these REST calls, such as starting or stopping a VM. In addition to these basic functionalities, the OCCI specification also defines optional features. Therefore, only a small subset of the possible functionality described in the OCCI standard is required for the API to be standard conform. Hence, a generic adaptation process is required utilizing basic functionalities of OCCI. Especially, because each provider's resource management framework has different constraints, OCCI implementations may miss optional functionalities.

In addition to the problem that each OCCI implementation has a different level of detail, OCCI only describes how to adapt single Entites. Hereby, no functionality is described within the OCCI standard that allows for complete cloud applications to be provisioned. Therefore, in order to adapt a cloud application, a process is required that not only provisions, but also transfers a cloud application from one state into another one. To reach this goal, we propose a complete model-driven cloud comparison and adaptation process filling this gap. In the following, we elicit the requirements for these processes and ask questions to be evaluated. Section 3.1 analyzes possible comparison strategies to match two OCCI models with each other in order to examine differences between the desired and the actual state of a cloud application. Section 3.2 investigates required adaptive steps to transform the cloud application into the desired state.

3.1 Comparison

To investigate the REST calls required to adapt running cloud applications, a comparison process is needed to map the different Entities of the cloud application onto the ones in the model to be deployed. Thus, we need to check whether an Entity is only present in the runtime and needs to be deleted, whether it is only present in the new model and needs to be created or whether it is present in both models and might require an update. Summed up, the major difficulty is to map the Entities in the runtime to the Entities in the target model resulting in the following question:

- **Q0:** How to recognize whether two elements from different models match?

Overall, model elements can be compared on an attribute level, a structure level, or a combination of both. When comparing models based on attributes and for an adaptation purpose, we are only allowed to consider unchangeable attributes, as changed attributes may indicate an updated element. In our case, the only immutable attributes are the Kind and the id of each Entity instance. Here problems arise when the id of a Resource can not manually be assigned during the provisioning process. Thus, the model id may differ from the id assigned by the cloud. Structural comparisons, however, have problems when models possess multiple identical substructures, as its elements can not exactly be matched. Moreover, it can not be recognized whether a matched element is adapted in terms of its attributes. Even though several model comparison frameworks already exist, such as *EMF Compare* [42], non of them is tailored towards cloud application topologies or allows to customize its strategy to utilize specific aspects from the attribute- or structural level. In the following the requirements for the adaptation process are investigated.

3.2 Adaptation Steps

The OCCI API offers four requests for its elements: GET, POST, PUT and DELETE. These operations can be mapped onto the different adaptation steps required to transfer a cloud application from one state to another. An extraction process (GET), which generates an OCCI model out of the running cloud application. A deprovisioning process (DELETE), to delete single Entities in the cloud, as the target model may be composed of fewer elements. An update process (PUT), that adjusts single Entities to fit its representation in the target model, for example a storage in need of more capacity. Finally, a process is required that allows for multiple Entity elements to be provisioned (POST), for example when another VM is required to handle additional workload. In addition to the analysis of the single adaptation step, the following question has to be answered:

- **Q1:** Is there a suitable order for the several adaptation steps?

In the following, these different processes are analyzed in more detail. Section 3.2.1 covers the extraction process, Section 3.2.2 the deprovisioning process, Section 3.2.4 the provisioning process, and Section 3.2.3 the update process.

3.2.1 Extraction

In order to derive adaptive steps, the adaptation process requires the system's runtime information of the system. Therefore, an extraction process is needed to gather the knowledge about the current cloud application's state. Hereby, this information must be in form of an OCCI model which can be compared to the OCCI model depicting the target state of the cloud application. This results in the following question:

- **Q2:** How to extract information from the running system in form of an OCCI model?

3.2.2 Deprovisioning

To handle elements in the runtime model that are absent in the target model, a deprovisioning process is necessary. Hereby, the process must handle all existing dependencies of the Entity to be deprovisined, before it gets deleted itself. This problem can be phrased as follows:

- **Q3:** Are there requirements for the deletion of OCCI elements?

3.2.3 Update

To change the characteristics of single Entities, an update process is required to identify whether a complete PUT request is needed in order to bring the Entity in the desired state or if a set of executed Actions are sufficient. Therefore, the following question arises:

- **Q4:** How to identify required adjustment calls?

3.2.4 Provisioning

To provision complete cloud applications, we need a process that automatically creates cloud resources and connects them as specified. Hereby, it is not sufficient to provision the Entities in any order, as each possess individual dependencies that need to be resolved. For example, when a VM is connected to a network, both the network and the VM have to be created before they can be linked. This leads to the following questions:

- **Q5:** How to resolve the dependencies between the different cloud resources?
- **Q6:** How to address already running resources within these dependencies?

Chapter 4

Design

In this chapter, the designs for the comparison and adaptation process are described. Overall, our system is composed of two modular processes, as depicted in Figure 4.1. The first process is the comparison which creates a match between the elements of the runtime- and target model. Here, the runtime model or source model is the extracted cloud application, whereas the target model represents the state of the cloud application to be reached. As multiple metamodels for OCCI exist, both the target and the source OCCI model have to be an instantiation of the same metamodel. Based on the matchings gathered throughout the comparison, the required adaptation steps for each single Entity are derived. Hereby, we indicate whether a resource has to be provisioned, deprovisioned, or updated, followed by an execution of the corresponding REST requests to the cloud. Through the extraction and utilization of runtime information, we derive the required adaptation steps. In future work, this information can also be utilized to establish a feed-back control loop between the cloud and our system, which also allows for self-healing capabilities. Hereby, the runtime model can be compared to the one expected running, followed by an execution of the adaptation process, when differences occur. To fulfill the analyzed requirements, a more detailed view into the modular parts of the adaptation process is required. Therefore, Section 4.1 covers the comparison- and Section 4.2 the adaptation process with special regards to the single adaptation steps.

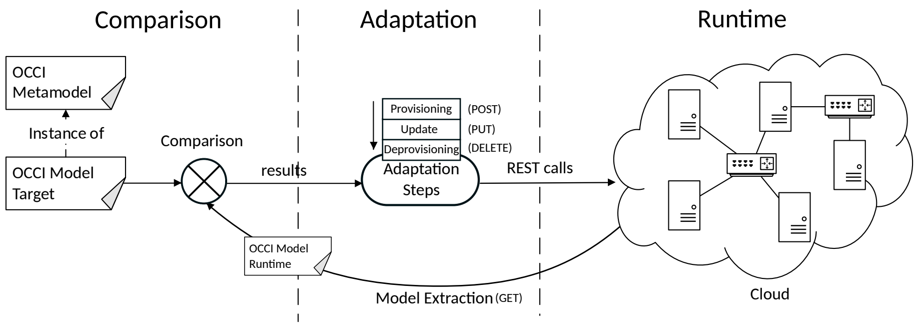


Figure 4.1: Overview of the complete adaptation process.

4.1 Comparator

During the comparison of two OCCI models, we check whether two Entities from the different models match. As Resources aggregate Links, we define a common structure that is based on a Resource matching, as depicted in Figure 4.2.

At first, the Resources of the models are matched. To create this match, multiple strategies can be used, which are discussed in the following sections. For the scope of this thesis, we define three Resource matching strategies: the *Simple comparator*, working on the attribute level, the *Complex comparator*, considering only the models structure, and the *Mixed comparator*, a combination of both. In case of the depicted models, a possible Resource match maps the Compute node of the source model to the one of the target model. The Storage and Network are not matched, as they are only contained within one of the two models. Based on this match, the Links of the matched Resources are compared to investigate whether the Resource possesses new or missing connections. Hereby, we check whether Links of the same Kind, source, and target exists. In the example case, the different Links of the models are not matched to each other, as they connect different resource types, and therefore possess different Kinds.

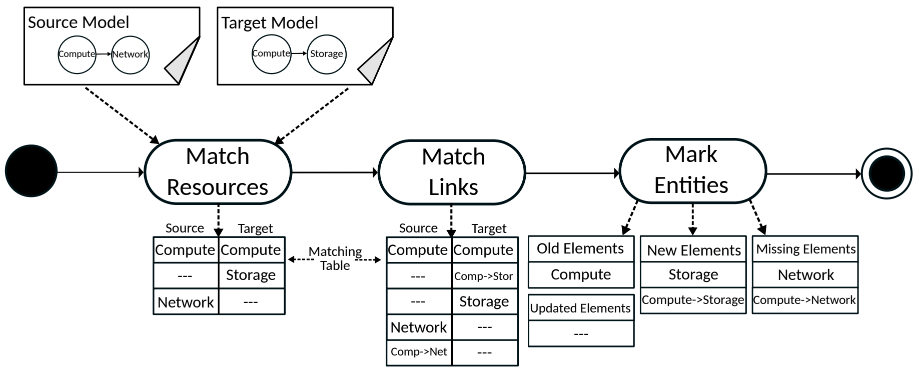


Figure 4.2: Overall structure of the comparison process.

Thereafter, the resulting matching table is used to investigate which adaptation step has to be performed on which Entity. For this purpose, we mark each of them as Old-, New-, Missing-, or Updated Element. This allows to characterize what kind of REST request has to be performed on each Entity. Furthermore, these marked elements serve as input for the different adaptation steps. Each Entity not having a corresponding match is either marked as New or Missing, depending on whether they are from the source or target model. In this case, Storage is marked as New, whereas Network is marked as Missing. To check whether an Entity is an Old- or Updated Element, the matched Entities are compared based on their Attributes in order to find differences. In the example, the Compute node is completely equivalent, and therefore marked as Old. Summed up, the comparison indicates the creation of a Storage, linked to Compute, as well as a deletion of the Network and its connections.

As the Link match and the extraction of steps from the matches are mainly based on the Resource match, the following sections describe three different model comparison strategies to calculate a Resource matching. Section 4.1.1 describes a process that compares two OCCI models based on the attribute level. Section 4.1.2 investigates a more complex approach, which checks whether two elements are the same based on the structure of the models. Finally, Section 4.1.3 presents a combination of both approaches.

4.1.1 Simple Comparator

The Simple comparator matches Resources based on their attributes. As theoretically each Attribute of a Resource can be used for a comparison, the problem exists that updated Resources cannot be identified. Therefore, we compare Resources based on their Kinds and ids, which are the only immutable Attributes. Besides some drawbacks, this allows for a discrete identification of equivalent Resources. Summed up, each Resource in the source model is matched to the Resource in the target model having the same id. The following section investigates a comparison strategy based on the structure level.

4.1.2 Complex Comparator

To compare two models based on their structure, we adapted the *Similarity Flooding algorithm*, a graph matching algorithm by Melnik et al. [43]. This algorithm is designed to iteratively calculate how well two nodes out of two different graphs match using the model's structure. Overall, the algorithm is separated into three major parts, as depicted in Figure 4.3. The creation of a *Pairwise connectivity graph (PCG)*, its transformation into an *Induced propagation graph (IPG)*, followed by a calculation of *fixpoint values*, indicating how well two elements match. Hereby, the PCG proposes possible map pairs, whereas the IPG adds additional edges and weights for the fixpoint calculation. These weights serve as *propagation coefficients*, indicating how well a map pair propagates to its neighbors and vice versa.

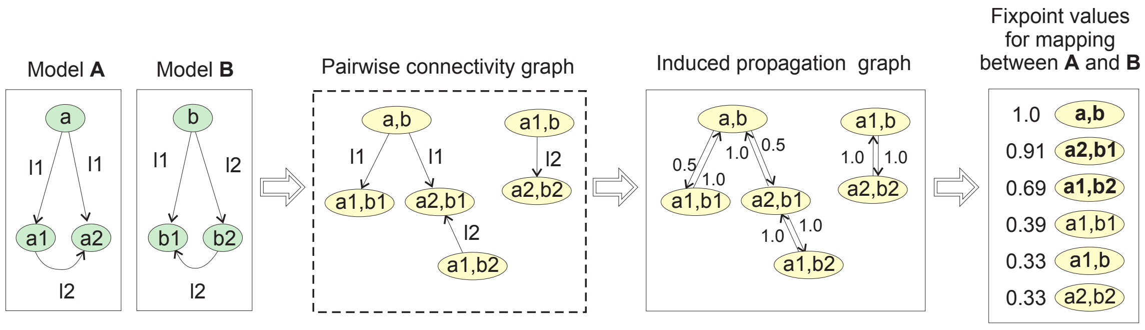


Figure 4.3: Example describing the Similarity Flooding algorithm [43].

To utilize this algorithm, we consider the OCCI models to be graphs, whereby each Resource is a node and each Link is an edge connecting two nodes. The algorithm assumes each edge in the graph to be a triple (s, p, o), whereby s stands for the source of the edge, o for the target and p for its label. As the label indicates the type of connection that connects two nodes, we set p as the Kind of the Link. From this perspective, a PCG can be derived, whereby we only create Resource map pairs with equal Kinds. A map pair is created when two nodes in the different graphs possess an edge with an equivalent label. Also, a map pair for the target nodes of the edge is created. Formally, a PCG for a model A and B is generated in the following manner:

$$((x, y), p, (x', y')) \in PCG(A, B) \iff (x, p, x') \in A \wedge (y, p, y') \in B \tag{4.1}$$

To give an example, in Figure 4.3 model A possesses an edge (a, l_1, a_1) and model B an edge (b, l_1, b_1), which creates the map pairs $((a, b), l_1, (a_1, b_1))$. Therefore, every pair in the PCG is an element from $A \times B$. These map pairs are created out of the intuition that if (a, b) are similar, (a_1, b_1) may represent the same element, due to the evidence of their shared connection l_1.

This PCG is transformed into an IPG, whereby for each edge another one is added in the opposite direction. Moreover, a weight w is assigned to each edge, which ranges from 0 to 1, indicating the propagation coefficient of the edge. Assuming that each edge label possesses the same contribution to the level of similarity, the weight can be computed by dividing 1.0 by the amount of outgoing edges of the same label from a map pair. For example the map pair (a, b) possesses two outgoing edges of label l_1 resulting in an equal distribution of the weight between $w((a, b), (a_1, b_1)) = 0.5$ and $w((a, b), (a_2, b_1)) = 0.5$. To fit the propagation coefficient onto OCCI models, we calculate the weight of an edge by dividing 1.0 by the amount of outgoing edges of the same Kind.

Based on the IPG, fixpoint values are iteratively calculated for each mapping $\sigma^i(x, y)$, with $x \in A$ and $y \in B$ for the i^{th} iteration. The values for σ^0 can be set based on previous comparison algorithms, for example a string matching of node labels. In case of the example provided above, no initial mapping is assumed, i.e., each σ^0 is set to 1.0. During each iteration, the σ value for each map pair is incremented by the σ values of the edges incoming from neighbor pairs multiplied by the edge's weight w in the following manner:

$$\begin{aligned} \sigma^{i+1} = \ & \sigma^i(x, y) \\ & + \sum_{(a_u, p, x) \in A, (b_u, p, y) \in B} \sigma^i(a_u, b_u) * w((a_u, b_u), (x, y)) \\ & + \sum_{(x, p, a_v) \in A, (y, p, b_v) \in B} \sigma^i(a_v, b_v) * w((a_v, b_v), (x, y)) \end{aligned} \tag{4.2}$$

Based on this equation, the value for the first iteration of map pair (a_1, b_1) is $\sigma^1(a_1, b_1) = \sigma^0(a_1, b_1) + \sigma^0(a, b) * 0, 5 = 1, 5$, whereas the value of (a, b) is $\sigma^1(a, b) = \sigma^0(a, b) + \sigma(a_1, b_1) * 1.0 + \sigma^0(a_2, b_1) * 1.0 = 3.0$. After each iteration σ^i is normalized by $\frac{\sigma^i}{\max(\sigma^i)}$, resulting in the fixpoint values $\sigma^1(a_1, b_1) = \frac{1.5}{3.0} = 0.5$ and $\sigma^1(a, b) = \frac{3.0}{3.0} = 1.0$. The algorithm stops, when either a maximum amount of iterations have been performed or the difference between the last iteration $\Delta(\sigma^n, \sigma^{n-1})$ is less than ϵ for $i > 0$. Based on the calculated fixpoint values, many possible mappings can be derived, depending on the filter technique used. When a filter only considers the highest values, the algorithm matches a to b, a_2 to b_1 and a_1 to b_2. In the work of Melnik et al. [43], further fixpoint value calculation variants and filter are presented.

4.1.3 Mixed Comparator

To utilize multiple comparison strategies, we propose the principle of a mixed comparator. Here, we use the results of the Simple comparator and perform the Similarity Flooding algorithm on Resources, which could not be matched over their id. To extract the most suitable matching from the possible map pairs, we add a tailored fixpoint value filter that compares Entities of similar values based on their attributes. Furthermore, we add a post-processing to bypass the problem of Resources without any Links, further discussed in Section 6.2. Figure 4.4 depicts the process of the Mixed Comparator, which is explained in the following.

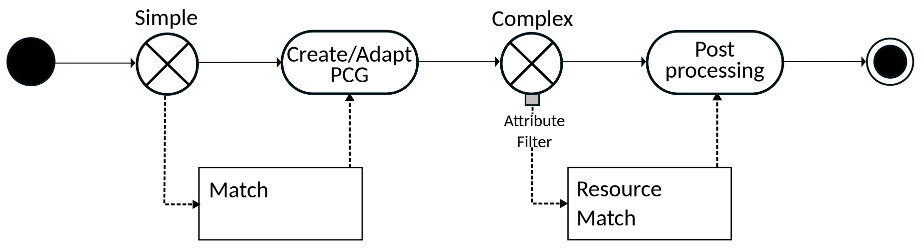


Figure 4.4: Mixed comparison process.

The process starts with a comparison of the models using the Simple comparator, i.e., elements that possess the same id in both models. This allows to utilize the benefits of the Simple Comparator, as we assume Resources of the same id are equivalent. Using this information, we create and adapt the PCG. Here, we remove every map pair consisting of one of the already mapped Resources, except the correct match. As we remove wrong map pair candidates, we reach not only a better performance of the Similarity Flooding algorithm but also a higher accuracy. Based on the adapted PCG, we perform the rest of the Similarity Flooding algorithm, i.e., we create an IPG and calculate the fixpoint values. To check for the most suited map pair, we filter for the highest fixpoint values among one Kind. Hereby, we inspect whether map pairs of similar high values exist that address the same source or target Resource. In this case, multiple suitable matches for a specific Resource are recognized, on which we perform an attribute comparison. This allows to investigate which map pair suites most not only on the structure-, but also on the attribute level. Finally, the resulting

Resource match is post-processed. Here, each non matched Resource is compared on the attribute level again, for example to check whether two Resources possess the same name. This allows to overcome the obstacle of the Complex comparator not being able to detect Resources without Links. In the following, the adaptation steps utilizing the results of the comparison process are discussed.

4.2 Adaptation Steps

To provide a clean adaptation process, this section discusses the order of the analyzed adaptation steps, as depicted in Figure 4.5. Here, we omit the extraction process for clarity, as it is only required for the comparison process (see Figure 4.1). First, every Entity marked as Missing is deleted to reduce the amount of rented resources to a minimum. We do not parallelize this process, as a user may be limited to the amount of resource he can provision, which may cause problems. For example, if the user is limited to two VMs, one must first be deleted before another one can be created. Then, every Entity marked as Updated gets adjusted, which brings the Entities in the required state for the rest of the cloud application to be provisioned. For example, when a former inactive VM is active in the target model, it needs to be started in order to allow for additional operations to be performed on it. Finally, the Entities marked as New are provisioned finalizing the adaptation process. In the following sections, we examine the design of each single adaptation step. Section 4.2.1 explains the extraction process (GET), Section 4.2.2 the deprovisioning process (DELETE), Section 4.2.3 the adjustment process (PUT), and Section 4.2.4 the provisioning process (POST).

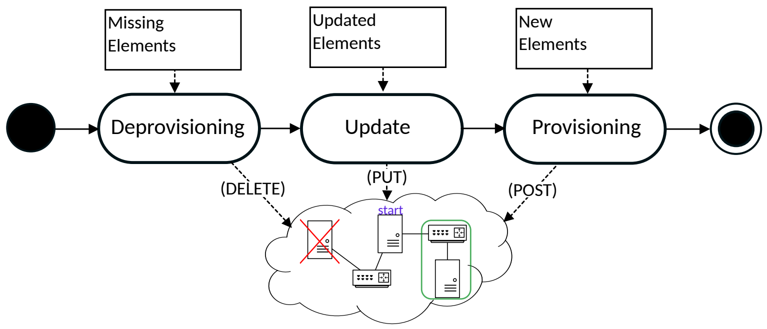


Figure 4.5: Adaptation step order and their comparison result utilization.

4.2.1 Extraction

The idea in the extraction process is that GET requests are send to the OCCI API of the cloud, as depicted in Figure 4.6. This allows to gather all information about the provisioned Category and Entity instances running at the moment. Based on that, the elements can be divided into the different Entity- and Category sub-types: Resource, Link, Mixin, Kind, and Action. This allows us to rebuild and serialize the cloud application into a model of any OCCI metamodel.

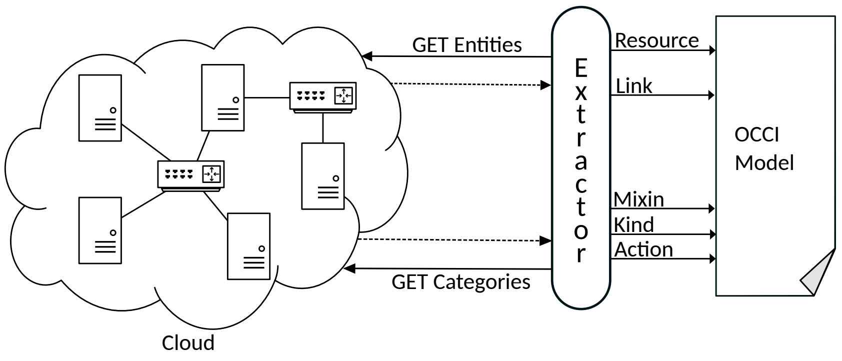


Figure 4.6: Extraction process creating an OCCI runtime model.

4.2.2 Deprovisioning

The deprovisioning process, depicted in Figure 4.7, is used to systematically delete resources already running on the cloud. In our case the Entities marked as Missing. Hereby, we separate the Missing Elements into Links and Resources. If necessary, these can be further adjusted by dividing them into different categories over their Kinds. After the separation, the Links are deleted, followed by the deletion of the Resources. This order allows to completely decouple a Resource before it is deleted, as each Link having the Resource either as target or as source is already marked as Missing. Even though a DELETE request should handle the complete decomposition of an element, this decoupling is required, because some OCCI implementations miss this capability. For example, in case of the popular cloud middleware OpenStack a network can only be deleted if it is not linked to any other resource. Here, an OCCI implementation should handle the DELETE request in that way that every connection to the network plus the network is deleted, which is not always granted. To provide a more robust process, we stick to the proposed deletion process.

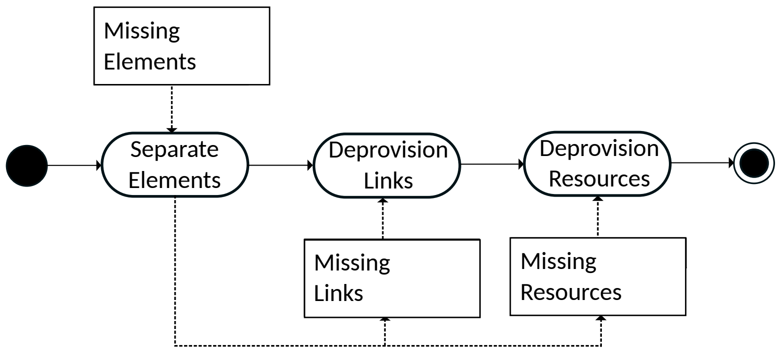


Figure 4.7: Deprovisioning process delete sequence.

4.2.3 Updating

To update an existing Entity, OCCI anticipates to use either a POST request for a partial, or a PUT request for an entire update (see Table 2.1). As POST(Action) requests can also be performed to change the state of specific Attributes, they can also be used to adapt single Entities. For example, Actions that resize the amount of storage of a Storage element, or Actions that start a VM, changing its state Attribute from inactive to active [1]. Therefore, we first separate the Updated Elements into steps that can be handled by the different kind of REST requests, as depicted in Figure 4.8. To allow for such a separation, a process is required that investigates how the change of specific Attributes have to be treated. In case of an update over Actions, it is not sufficient to only retrieve the possible Actions that can be performed by an Entity. It must also be known which Attribute the Action affects and how. To handle this issue, we define a list of OCCI specified Actions and their behavior. As OCCI does not exactly define the differences between the partial updates of POST requests and the full updates over PUT requests, the request to use depends on the OCCI implementation. It should be noted, that due to the automation of these requests and the complete information availability of each Entity, a full update over a POST request is always possible.

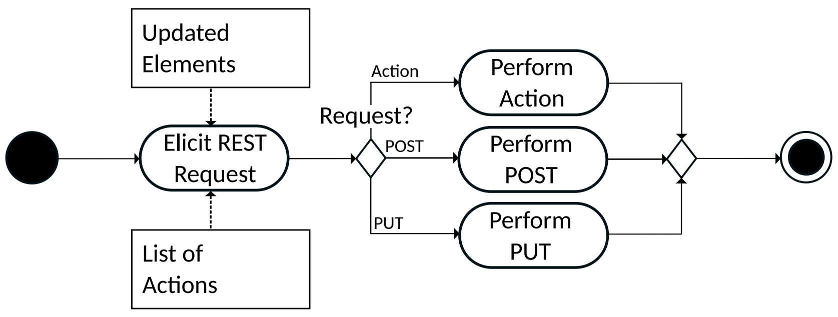


Figure 4.8: Update process adapting single Entities.

4.2.4 Provisioning

The provisioning of new Entities is the most crucial part of the adaptation process, as the dependencies between these Entities must be resolved. Furthermore, the already deployed Entities must be considered. To resolve the dependencies between the Entities within an OCCI model, we adapted an approach by Breitenbücher et al. [44], who created *provisioning plans* out of *Topology and Orchestration Specification for Cloud Applications (TOSCA)* models. We adapted this process to not only perform on OCCI models, but also provide a suitable provisioning plan interpreter which executes the required REST request in the described order. In a former paper [45] we already discussed this process only considering initial deployments without the utilization of runtime information. An overview of the original provisioning plan generation is depicted in Figure 4.9. At first, the OCCI model is transformed into a *Provisioning Order Graph (POG)*, which describes the dependencies of the OCCI model's elements as a directed graph. For this transformation, Breitenbücher et al. [44] define two types of dependency patterns that can occur, the depends-on- and use pattern. In our approach, we map these dependency types onto the different Kinds of Links, as they characterize the connection type of two Resources, and therefore their type of dependency. The depends-on pattern is a dependency in which the source Resource of the Link has to be created before the target Resource. This dependency occurs, for example, when a Component runs on a Compute node. The use pattern describes a dependency in which the source Resource as well as the target Resource need to be instantiated before the Link can be created. For example, when a Compute node is connected to a Network both Resources have to be provisioned before they can be linked.

Based on the analyzed dependencies within the POG, a second transformation is performed generating a provisioning plan. A provisioning plan is a workflow model that indicates the order in which the different Entities are created. To allow for the creation of a sufficient workflow model, its metamodel must at least consist of the following five elements: the initial element to start a control flow, the final element to stop it, a fork node that splits up tasks in order to allow for a parallel execution, a join node to join these tasks, and actions to perform basic functionalities, which are in this case POST requests. The details of the *model-to-model transformation (M2M)* are explained in Section 5.3.

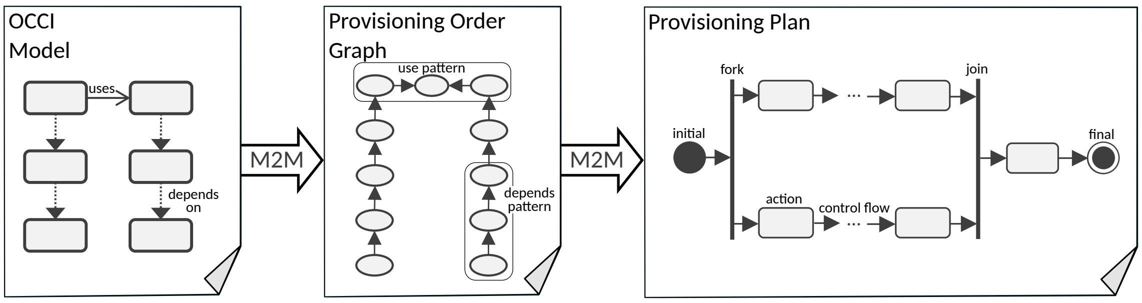


Figure 4.9: Overview of the provisioning plan generation (adapted from Breitenbücher et al. [44]).

To utilize the runtime information gathered through the extraction and comparison process, the generation of the provisioning plan is adjusted as depicted in Figure 4.10. Here, we utilize the results of the comparison process, as the Updated- and Old Elements give information about already deployed Entities. Therefore, the vertexes representing these Entities can be removed from the POG with all edges connected to it, because they do not need to be provisioned. Moreover, it indicates that the dependencies of the originally dependent vertexes are resolved. This process can be seen as a XOR operation, removing each element from the Updated- and Old Elements from the POG. Based on this adjusted PCG, the provisioning plan is generated as usual. Due to the deletion of vertexes of already running Entities from the POG, no actions are created within the provisioning plan.

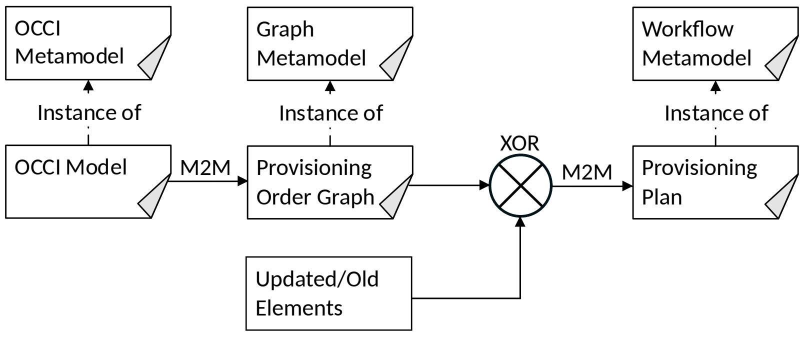


Figure 4.10: Utilization of runtime information to adapt the POG.

In addition to the provisioning plan, an interpreter is required to execute it. Therefore, we propose the mechanism depicted in Figure 4.11, which processes the plan node by node. The input defines the node of the provisioning plan it starts with. Then, it is checked what kind of element the node is. If it is the initial node, pre-processing steps are performed, whereas a final node results in post-processing steps and a closing of the workflow. If the node is a join, action or initial, the depicted operations are performed, followed by an incrementation of the current node to the following one starting the process from the beginning. Hereby, the operation of an action element performs the POST request to provision the Entity. At a join node the process waits until each incoming edge reached the node, before the procedure is continued with the next node. Otherwise, if the node is a fork, one thread is created for each outgoing edge starting the same procedure. This time however, the starting node is set to the node following the fork.

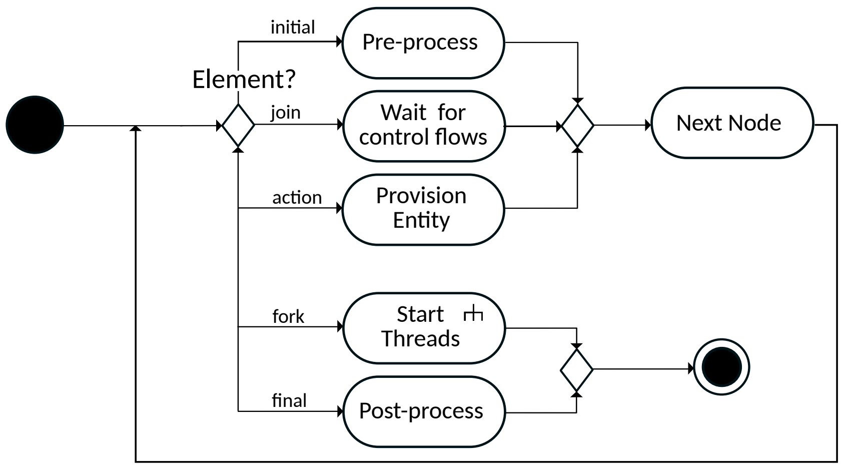


Figure 4.11: Process interpreting the provisioning plan.

In the following chapter, the challenges of the implementation of the proposed design for the comparison and adaptation process are discussed.

Chapter 5

Implementation

In this chapter, the realization of the proposed design is described with regards to implementation challenges. Furthermore, we utilize this implementation as a prototype to evaluate the feasibility of the comparison- and adaptation process. Section 5.1 covers an enumeration and description of the used tools, metamodels, and the cloud infrastructure. Section 5.2 discusses the implementation of the comparison processes. Finally, Section 5.3 investigates the coupling between the comparison results and the adaptation steps and furthermore goes into detail about the adaptation steps itself.

5.1 Tooling

To understand the different implementation decisions chosen during the development of a prototype, this section provides an overview of the different tools used. Especially, the used cloud system is discussed, delimiting the level of detail of the OCCI API implementation. Furthermore, as there are a lot of Model Driven Engineering (MDE) tools, this section covers the utilized languages and metamodels to implement the prototype. A quick summary of the utilized standards, metamodels, and tools is provided in Table 5.1. The reference to the UML standard is given as we instantiate our generated provisioning plan as *activity diagram*.

The adaptation prototype is developed using an OpenStack [16] cloud which additionally implements the *OpenStack OCCI Interface (OOI)* [46], an OCCI API for OpenStack. OOI is capable to interpret the basic REST requests, defined within the OCCI specification including the infrastructure extension. Thus, it supports the creation of VMs, storages, networks and their connections. However, among other drawbacks OOI lacks the capability to interpret PUT requests in its current version. Other implementation challenges that come with OOI are discussed in the representative sections. Furthermore, it is not capable of interpreting REST requests addressing elements of the OCCI platform extension. Unfortunately, no OCCI API exists that provide this capability at the moment, wherefore we can not provide an evaluation utilizing PaaS applications. Nevertheless, this evaluation is part of future work and an excursion about expected results is given in Section 6.4.2.

To efficiently work with models, we use *Epsilon* [38], a suite of languages and tools for MDE techniques. These languages are tailored to operate on EMF [47] models, such as Merle et al.'s [48] Ecore OCCI metamodel we make use of. Overall, for the model transformations, we use ETL, allowing for a convenient specification of transformation rules. Furthermore, we use the *Epsilon Flock* language for model migrations. In the following, implementation details about the comparison process utilizing these languages are given.

Specifications	**Version**	**Description**
OCCI	1.2	Standard for cloud resource management [1].
UML	2.5	Software modeling standard [49].
Metamodels		
OCCI Ecore metamodel	Version from 2015	OCCI 1.2 Ecore metamodel [48].
UML2 Ecore metamodel	5.2.0	UML 2.5 Ecore metamodel [50].
Tools		
OpenStack	Newton	Open source cloud middleware [16].
OOI	1.2.0	OpenStack OCCI API for Version 1.2 [46].
Epsilon	1.4.0	Family of languages for MDE techniques [38].

Table 5.1: Version table of utilized standards, metamodels and tools.

5.2 Comparison

Since we proposed multiple comparison strategies to generate Resource matchings, these are investigated in the following sections. In this section we focus on the implementation of the generalized parts: the data structure of matches, the Link matching- and marking process. To store matches and thus matching tables, we implement a class `Match` storing two objects. One object represents the elements from the source and the other from the target model. This allows to clearly identify the origin of the different Entities and whether a match for them exists. The matching table from the Resource matching serves as input for the Link matching procedure, which is depicted in Listing 5.1. Here, we iterate over each matched Resource (Line 2) and check whether the match is a direct match (Line 3), meaning that neither the source nor the target of the match equals `null`. If that is the case, we compare the Links of the matched Resources in order to create corresponding Link matches (Line 4). Hereby, all Links of equivalent Kind, source, and target are returned as a direct match. It should be noted that during this process the target of the Link is compared in such a manner that the already created Resource match is considered. Furthermore, new or missing Links of the target Resource are added as match to the matching table. If the `match` has either an empty source or target Resource (Line 7, 12), each Link of the non-matched Resource is added as Entity that could not be matched either. Hereby, either the first or second row of the match equals `null` (Line 8-10, 13-15).

```
protected void createLinkMatch() {
   List<Match> linkMatches = new ArrayList<Match>();
   for(Match match: this.matches){
      if(match.getSrc() != null && match.getTar() != null){
         linkMatches.addAll(matchLinksOfObject(match.getSrc(), match.getTar()));
      }
      else if(match.getSrc() == null){
         for(EObject link: extractLinks(match.getTar())){
            linkMatches.add(new Match(null, link));
         }
      }
      else if(match.getTar() == null){
         for(EObject link: extractLinks(match.getSrc())){
            linkMatches.add(new Match(link, null));
         }
      }
   }
   this.matches.addAll(linkMatches);
}
```

Listing 5.1: Link matching procedure (Java).

In the marking process we iterate over the complete matching table which now is filled with all Resources and Links. Hereby, each match with an empty target is marked as Missing, as it is not present in the target model, and therefore needs to be deleted. Each match with an empty source is marked as New, as it is not present in the source model. Thus, no corresponding resource exists in the running cloud application. To identify whether a direct match needs to be marked as an Old- or Updated Element, we perform a comparison of the matched Entities based on their Attributes, as depicted in Listing 5.2. Here, we iterate over the Attributes of the Entities to be compared (Line 2-3) and check whether equivalent Attributes (Line 4) possess equivalent values (Line 6). If a difference is found, the Entity is marked as Updated (Line 7) initiating the adjustment of it. Otherwise, the target Entity is marked as Old (Line 12). For this process, we exclude a set of Attributes over a *blacklist* (Line 5). This is required, because some Attributes can not be set by the OCCI API as defined within the model. For example, the id of Entities in OpenStack which results in the marking of each Entity as Updated Element.

As the Simple comparator presents a trivial design, it does not consist of any implementation challenges. Here, we simply perform an exact string matching based on the occi.core.id Attribute of each Entity in the target and source model, whereby we consider the entries of the *idSwapTable* introduced later in Section 5.3. The Complex comparator, however, is composed of two different model transformations, which we discuss in Section 5.2.1. Furthermore, we investigate the implementation details of the Mixed comparator, which manipulates the PCG based on the knowledge gathered through previously performed comparison strategies.

```java
protected boolean checkIfAdapted(EObject src, EObject tar) {
   for(AttributeState srcAttr: extractAttr(src)){
      for(AttributeState tarAttr: extractAttr(tar)){
         if(srcAttr.getName().equals(tarAttr.getName())
         && inBlacklist(srcAttr.getName()) == false){
            if(srcAttr.getValue().equals(tarAttr.getValue()) == false){
               return true;
            }
         }
      }
   }
   return false;
}
```

Listing 5.2: Marking process determining Old- and Updated Elements (Java).

5.2.1 Complex Comparator

To transform the OCCI model into a graph required for the Similarity Flooding algorithm, we create a directed graph metamodel, which is capable to instantiate PCGs, as well as IPGs. This metamodel, depicted in Figure 5.1, consists of a Graph, comprising Vertexes and Edges. Hereby, a Vertex represents a possible Resource map pair by storing their corresponding id in a PCG-Resource. Additionally, each Vertex stores the Kind of the map pair and its fixpoint value, which is required for IPGs. As we require the current fixpoint value σ^i of a Vertex throughout a complete iteration, we added a second attribute, nextFixpointValue, to store the fixpoint value σ^{i+1} calculated during the iteration. As an Edge is responsible to describe the connection between two Vertexes, each Edge has attributes to store information about a target-, as well as a source Vertex. Additionally, these Edges can store the Kind and of the Link and a value for its weight, information required for the instantiation of IPGs.

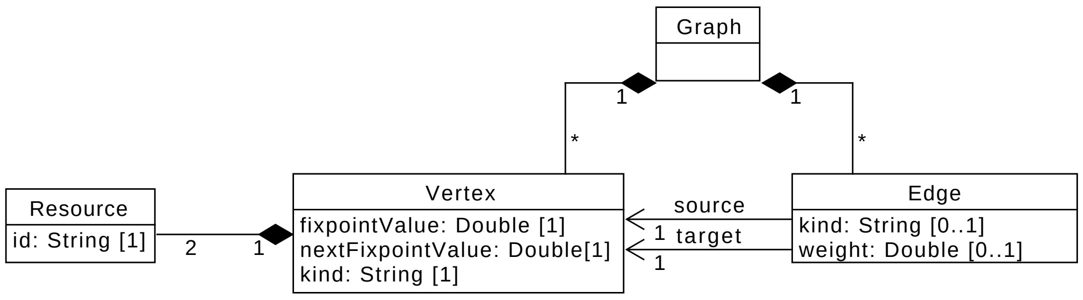


Figure 5.1: PCG and IPG metamodel.

We implemented the transformation of the OCCI to a PCG model as an ETL transformation, depicted in Listing 5.3. At first, we create a Graph element (Line 2) and iterate over the Resources and Links of the source and target OCCI models (Line 3-6). When a Link with the same Kind is found (Line 7-8), a Vertex is created for the corresponding Resources (Line 9). Then, it is checked whether a Vertex for the target of the Link already exists (Line 10). If not, it is added to the graph together with a corresponding Edge (Line 11, 13-14). This Edge links the new created source Vertex to the Vertex representing the target map pair.

```
1  pre {
2     var graph = new PCG!Graph();
3     for(srcRes in srcOCCI!Resource){
4        for(srcLink in srcRes.links){
5           for(tarRes in tarOCCI!Resource){
6              for(tarLink in tarRes.links){
7                 if(tarLink.kind.term == srcLink.kind.term
8                 and tarLink.kind.scheme == srcLink.kind.scheme){
9                    createVertex(srcRes, tarRes);
10                   if(not(vertexExists(srcLink.target.id, tarLink.target.id))){
11                      createVertex(srcLink.target, tarLink.target);
12                   }
13                   createEdge(srcLink.kind, getVertex(srcRes.id, tarRes.id),
14                      getVertex(srcLink.target.id, tarLink.target.id));
15                }
16       ...
17 }
```

Listing 5.3: OCCI to PCG transformation (ETL).

Due to the IPG being of the same metamodel as the PCG, the PCG to IPG transformation is a transformation in which only a minimal amount of information is added. Therefore, we utilized the Epsilon Flock language to perform a model migration, adding new Edges, fixpoint values, and weights. At first, for every Edge in the PCG another Edge is created pointing in the other direction. In a post block we calculate the weight for each Edge (Line 3) by dividing 1 through the amount of outgoing Edges of a specific Kind (Line 4), as shown in Listing 5.4.

```
1  post {
2     for(vert in IPG!Vertex){
3        for(edge in vert.getOutgoingEdges()){
4           edge.weight = 1.0d/vert.getOutgoingEdges().getEdgesOfKind(edge.kind).size();
5        }
6     }
7  }
```

Listing 5.4: PCG to IPG migration (Flock).

The IPG serves as input for the Similarity Flooding fixpoint calculation. In Listing 5.5, the algorithm for the fixpoint calculation and normalization is shown. The algorithm is performed until a maximum number of iterations, `maxIterations`, is reached or until no difference larger than `eps` is computed (Line 4). During each iteration of the algorithm, we calculate the fixpoint values of each Vertex, according to Equation 4.2. Hereby, we manipulate the nextFixpointValue of each Vertex (Line 6-7), as its current fixpoint value is required for the computation of other Vertexes' fixpoint values. Thereafter, the list `normValues` (Line 2), storing the highest pre-normalization value for each Kind, is updated (Line 8). Thus, `updateNormValues` checks whether a higher value for the corresponding Kind is calculated. Moreover, the update procedure creates an entry when a new Kind is detected. After the pre-normalization values for each Vertex are calculated, the fixpoint values are normalized (Line 10). During this process, the fixpointValue of the Vertex is set equivalent to the nextFixpointValue. Furthermore, this normalization process determines whether another iteration of the algorithm is necessary. Thus, the `stop` boolean is either set to true, for another iteration, or false to stop the algorithm.

```java
1  private void performSimilarityFlooding(Graph graph, int maxIterations, double eps) {
2     List<String[]> normValues = new ArrayList<String[]>();
3     boolean stop = false;
4     for(int i = 0; (i< maxIterations) && (stop == false); i++){
5        for(Vertex vertex: graph.getVertices()){
6           double nextFixVal = calculateFixpointValue(vertex, graph);
7           vertex.setNextFixpointValue(nextFixVal);
8           updateNormValues(vertex, normValues);
9        }
10       stop = normalizeValues(graph, normValues, eps);
11       normValues.clear();
12    }
13 }
```

Listing 5.5: Fixpoint value calculation (Java).

Finally, we extract the calculated fixpoint values into a map in order to investigate the most suitable matches. The keys of the map are the ids of the Resources from the source model, whereby its values are the map pairs for the corresponding source Resources. Based on this map, we iteratively investigate the most suitable map pair, which represents the filtering process. After one map pair is chosen, the map is updated by removing the source Resource key from the map. Additionally, all values taking the target Resource of the map pair into consideration are deleted. This prevents the possibility of matching already assigned Resources. To provide a detailed evaluation of the capabilities of the attribute- and structure level comparison, the Complex comparator only implements a filter, successively matching the Vertex with the highest fixpoint value. An enhanced filter is implemented in the Mixed comparator, which utilizes the attribute level.

5.2.2 Mixed Comparator

The Mixed comparator starts with the calculation of a Resource match based on the id comparison of the Simple comparator. Then, the OCCI models are transformed into a PCG. Utilizing the match results from the previous comparison, the PCG is adapted as shown in Listing 5.6. At first, we iterate over the direct Resource matches of the passed matching table (Line 4). Then we iterate over each map pair of the PCG (Line 8) in order to eliminate wrong candidates, considering one id of the already matched Resources (Line 13-16). As the code for the PCG is automatically generated, we always store the Resource from the source model in position (`0`), whereas the Resource from the target model resides on position (`1`). The boolean `missing` is used to identify whether a Vertex exists that describes the `match` (Line 8-12). If such a Vertex is missing, the corresponding Vertex is created at the end of the adjustment process. This is required for cases in which a Resource without Links is matched in the pre-comparison, as no map pair for them is created in the initial OCCI to PCG transformation. Based on this adapted PCG, the IPG is generated as usual, followed by a computation of the fixpoint values.

```
1  private void adjustElementsInGraph(Graph pcgGraph,  EList<Match> matches) {
2     List<Vertex> toRemove = new BasicEList<Vertex>();
3     List<Vertex> toAdd = new BasicEList<Vertex>();
4     for(Match match: extractDirectResourceMatch(matches)){
5        Resource srcRes = (Resource) match.getSrc();
6        Resource tarRes = (Resource) match.getTar();
7        boolean missing = true;
8        for(Vertex vertex: pcgGraph.getVertices()){
9           if(vertex.getResources().get(0).getId().equals(srcRes.getId())
10          && vertex.getResources().get(1).getId().equals(tarRes.getId())){
11             missing = false;
12          }
13          else if(vertex.getResources().get(0).getId().equals(srcRes.getId())
14          || vertex.getResources().get(1).getId().equals(tarRes.getId())){
15             toRemove.add(vertex);
16          }
17       }
18       if(missing == true){
19          toAdd.add(createMissingVertex(srcRes, matches));
20       }
21    ...
22 }
```

Listing 5.6: PCG adjustment (Java).

After the computation of the fixpoint values, we search the resulting map for the map pair with the highest fixpoint value to create a mapping. Once a map pair is chosen, it is checked whether map pairs with similar high fixpoint values exist that consider either the source or target Resource. At this point we perform an evaluation based on the attribute level to choose the most suitable match. Hereby, we perform the algorithm shown in Listing 5.7 two times. Once with a list of Vertexes considering the same source Resource and once with each Vertex Having the target Resource of the calculated map pair from the first step. Thus, we consider not only the best match for the source being the target, but also how well the target fits to other sources. It should be noted, that this algorithm is only executed if multiple possible map pairs are detected, resulting in a list size larger than two. To evaluate the most fitting vertex from a list of Vertexes we initialize two variables, `bestFit` and `maxVertex`, indicating the best fitting Vertex and its position (Line 2). Then, we iterate over each Vertex (Line 3) and compare it to the best fitting Vertex (Line 4). The comparison is performed based on the Attributes, whereby we simply count the number of equivalences in order to evaluate the most suitable Resource match. This calculation can be further improved by weighting the different Attributes or the amount of Links. If the Vertex on position `i` is the `maxVertex` (Line 5), we update the position of best fitting map pair by updating `bestFit` to `i` (Line 6). Finally, the most suiting Vertex is returned indicating the most suitable Resource match (Line 9). In the following, the implementation details of the adaptation process are discussed.

```java
1  private static Vertex mostFittingVertex(List<Vertex> vertexes, EList<EObject> srcM,
       EList<EObject> tarM) {
2     int bestFit = 0; Vertex maxVer;
3     for(int i=1; i < vertexes.size(); i++){
4        maxVer = compVertexes(vertexes.get(bestFit), vertexes.get(i), srcM, tarM);
5        if(vertexes.get(i).equals(maxVer)){
6           bestFit = i;
7        }
8     }
9     return vertexes.get(bestFit);
10 }
```

Listing 5.7: Attribute filter for Entities having similar fixpoint values (Java).

5.3 Adaptation Process

The adaptation process is composed of the multiple modular adaptation steps described in the design phase. As the adaptation steps are directly related to the cloud, several implementation challenges have to be overcome, addressing constraints of the used OCCI implementation. This covers especially the behavior of establishing a connection to the OCCI API and the interpretation of responses. Hereby, some decisions regarding the OCCI implementation are examined, providing specific constraints to the implementation of the adaptation process. Furthermore, this section provides implementation details about the single adaptation steps.

To communicate with the cloud, a *session token* is required to identify the user and the *tenant* he or she wants to access. Therefore, we request a session token at the beginning of the adaptation process, which is used by the single adaptation step. As OpenStack does not provide the capability to manually choose an Entity's id, we implement an idSwapTable to map the ids of the cloud onto the ones defined in the model, as shown in Figure 5.2. Thus, we can identify which id in the model is which in the cloud and runtime model. To allow the usage of the adaptation process on multiple cloud applications, we store the idSwapTable for each specific project and user. Hereby, we also store the last model that got provisioned using our approach, which allows to check whether the cloud application is in the expected state.

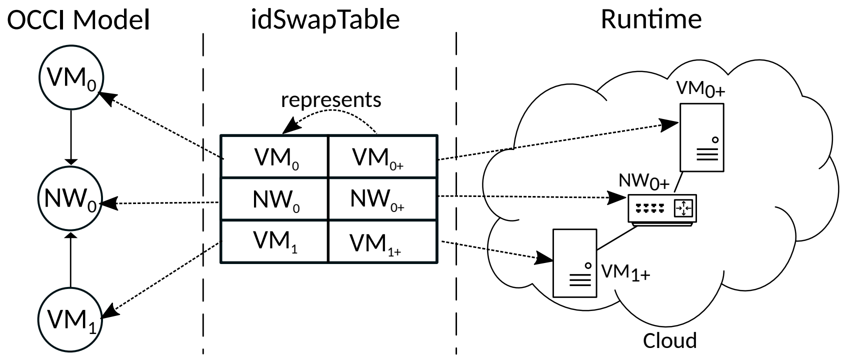


Figure 5.2: Principle of the idSwapTable associating ids from the OCCI model to the runtime.

Another generic principle is the generation of REST calls based on the information contained within the Entity to be managed. As OOI implements the text rendering specification [25] of OCCI, we serialize the information to fit onto this structure. This general process is shown in Listing 5.8 using the POST request as example. The text rendering passes the information about the Entity over request headers. One header addresses related Categories, whereas the other describes the Attributes of the element. At first, we extract the request's URI from the Kind of the Entity (Line 3). Thereafter, a connection to the cloud is established indicating a specific REST request to be performed(Line 4). Then we gather information about Category instances linked to the Entity and put them into the *Category header* (Line 5). This, for example, addresses the Kind of the Entity and its Mixins. Finally, we add the information about the Entity itself in the *Attribute header*, whereby we extract the Attribute of the Entity to be managed (Line 6). It should be noted, that these extractions represent trivial processes, as the Kind, Mixins, and Attributes are extracted in a similar manner. When the connection is successful, multiple actions are executed, depending on the kind of REST operation performed (Line 8-11). For example, after a POST request, the Entities' new id is extracted from the output and stored together with its original id in the idSwapTable. In case of an unsuccessful connection attempt we re-send the request up to five times to mitigate connection problems. In the following, the implementation of the different adaptation steps are investigated in more detail. Section 5.3.1 covers the extraction-, Section 5.3.2 the deprovisioning-, Section 5.3.3 the update-, and Section 5.3.4 the provisioning process.

```java
private String executePostOperation(EObject element) {
      Entity entity = (Entity) element;
      String adaptedAddress = getEntityKindURI(entity);
      HttpURLConnection conn = establishConnection(adaptedAddress, "POST", true,
        "text/occi", this.connection.getToken());
      conn.setRequestProperty("Category", generateCategoryHeader(entity));
      conn.setRequestProperty("X-OCCI-Attribute", generateAttributeHeader(entity));
      ...
      if(connectionSuccessful(conn)){
         ...
          return output;
      }
      ...
   }
```

Listing 5.8: Generation of a REST request from OCCI elements (Java).

5.3.1 Extraction

The extraction process represents a pre adaptation step process, as it is only required for the comparison between the desired and actual state of the cloud application. Therefore, we also investigate its connection to the comparison process and the idSwapTable. We use an extraction module generating an OCCI model that was implemented in another students research project. However, we shortly explain this process and how we utilized it. For the extraction of the runtime OCCI model, the *jOCCI-API* [51], a java library for OCCI queries, is used. Based on the received model information, it is serialized to fit to the OCCI Ecore metamodel of Merle et al. [48]. After the extraction, we synchronize the idSwapTable with the extracted runtime model, to only consider ids actually assigned by the cloud. This is required, as changes to the cloud can be made outside of the proposed adaptation process. Moreover, the OCCI model side of the idSwapTable is updated using the comparison results, whereby the ids of matched Entities are exchanged in the idSwapTable. This allows the other processes to work with ids contained in the model, which are simply swapped out with the corresponding id in the idSwapTable once the request is performed. In the following, the deprovisioning process is examined that handles the deletion of elements marked as Missing Elements.

5.3.2 Deprovisioning

As the deprovisioning process simply separates the Missing Elements into Resources and Links and performs DELETE requests on them, it represents a trivial implementation. Therefore, we focus on the details of REST requests deleting an Entity. Compared to the other requests, the DELETE request, does not require any information about the Category instances and Attribute instances of the Entity. It only requires the correct URI of the element. As each Entity to be deleted originates from the runtime model, we extract the address and id of it directly from the source model. On a successful deletion request, the corresponding idSwapTable entry is deleted. In the following, the update process is investigated, which together with the provisioning process makes use of the updated idSwapTable.

5.3.3 Update

Every Entity marked as Updated is handled by the updating process, as shown in Listing 5.9. Therefore, we iterate over `updatedElements` (Line 2) and extract for each Resource its `counterpart` (Line 3). Based on the counterpart, we gather information about the exact difference between the Attributes of the Entities (Line 5). Here, we create a list storing not only the differing Attributes, but also their values. In addition to this list, an Action list is required to check whether the update of the Entity can be performed by Actions only. The Action list specifies which Action affects which Attributes. Thus, the Action list must not only store the name and Attribute of each Action, but also the source and target state of the Attribute's value. I.e., to start an inactive VM, a list must contain occi.compute.state, whereby the value inactive to active can be changed over the Action start. If the Entity can be adjusted over Actions (Line 6), the corresponding Action requests are send (Line 7). Otherwise, a PUT request is performed, as the request contains all information about the element to be changed, and therefore all parts to be updated (Line 10). As OOI does not implement the possibility to perform PUT request, this operation presents a `stub`. For the execution of an Action, the term of the Action to be performed is added to the address of a POST request. Furthermore, instead of the Entity's attributes filling the Category- and Attribute header, the information is extracted out of the Action. Compared to the deprovisioning process, the id of the Entity must be extracted out of the idSwapTable, as we perform the update on the element of the target model. Hereby, we address the correct Entity, as we updated the idSwapTable using the comparison results. In the following, the provisioning process is discussed in detail.

```java
1  public void update(EList<EObject> updatedElements, EList<Match> match) {
2     for(EObject element: updatedElements){
3        EObject counterpart = getCounterpart(element, match);
4        EList<AttributeState[]> differences = new BasicEList<AttributeState[]>();
5        differences = investigateDifferences(element, counterpart);
6        if(canBeHandledByActions(differences)){
7           performActions(element, differences);
8        }
9        else{
10          performPut(element); //stub
11       }
12    }
13 }
```

Listing 5.9: Generation of a suite of REST requests updating an Entity (Java).

5.3.4 Provisioning

The provisioning process was already developed and implemented during a former students research project [45]. It is composed of two ETL model transformations, the OCCI to POG and POG to provisioning plan transformation. To instantiate POGs we created a directed graph metamodel. This graph is composed of vertexes only storing ids of corresponding Entities and edges connecting them. The provisioning plan is an *activity diagram*, which instantiates the UML Ecore metamodel [50]. To perform the OCCI to POG transformation a configuration of the dependency types for each Link type is required. Every infrastructure extension is mapped to the use pattern, as in each case both source- and target Resource need to be provisioned beforehand. The transformation itself generates a vertex for each Resource and each Link, as shown in Table 5.2. Thus, vertexes represent not only Resources, but also Links to be provisioned. After the vertexes are created, a post processing connects them as shown in Listing 5.10. Here, two edges are generated for each Link (Line 5-6, 9-10). Depending on the Kind of the Link, they either connect the vertexes using the depends-on- or use pattern, as shown in Figure 5.3. The *equivalent* method addresses the origin of a transformed element, i.e., in this case it addresses the vertex generated from a Link. The depends-on pattern creates an edge that connects the target Resource of the Link to the Link itself and an edge from the Link to its source Resource (Line 5-6). The use pattern creates an edge that connects the Links target to the Link and an edge connecting the source of the Link to the Link (Line 9-10). Due to a missing OCCI platform implementation, currently no mapping of the dependency pattern to the platform Link types is provided.

OCCI	**POG**
Resource	Vertex
Link	Vertex
Link Kind	Use/Depends-on pattern

Table 5.2: OCCI to POG mapping.

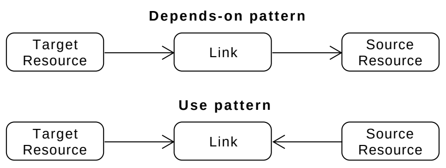


Figure 5.3: Dependency pattern.

```
1  operation createEdges() {
2     for(link in OCCI!Link){
3        var linkBaseKind = new String;
4        if(dependsOnDependencies.exists(kind | kind.compareTo(link.kind.baseKind()))){
5           Edge(link.target.equivalent(), link.equivalent(), graph);
6           Edge(link.equivalent(), link.source.equivalent(), graph);
7        }
8        if(useDependencies.exists(kind | kind.compareTo(link.kind.baseKind()))){
9           Edge(link.target.equivalent(), link.equivalent(), graph);
10          Edge(link.source.equivalent(), link.equivalent(), graph);
11       }
12    }
13 }
```

Listing 5.10: Linking process of the OCCI to POG transformation (ETL).

The second transformation generates a provisioning plan out of the POG. Hereby, we define a transformation rule that transforms each vertex into an action, as they represent the Entities to be provisioned. These actions possess input pins in which we store the id of the corresponding Entity. The order of the actions is determined over the incoming edges of each vertex, as they represent the Entity's requirements and therefore characterize when the action can take place. The generation of the corresponding workflow is done in three steps. At first, we determine the initial part of the workflow. Here, each vertex without incoming edges is directly linked to a fork connected to the initial node, as they posses no requirements, and therefore can be parallelized. Then, we link the remaining actions to the rest of the control flow, as depicted in Listing 5.11. At first, we check whether all actions are linked to the control flow (Line 3). If that is not the case, we iterate over each vertex without any unresolved requirements and attach them to the control flow (Line 3-19). These `provisionableVertexes` are detected by determining whether all required vertexes have their corresponding actions linked to the control flow. Based on the amount of required vertexes (Line 5), the action of the current vertex is attached to the control flow in a specific manner. If the vertex has multiple dependencies (Line 6-13), a join is created joining all required actions (Line 8). Moreover, we connect the current action to this join binding it as following action to the rest of the control flow (Line 9-13). Thus, all required actions are performed before the action of the current vertex takes place. If the vertex has only one requirement, the control flow of the required action is directly connected to the action represented by the current vertex (Line 16). The third step creates a join leading to the final node in which all actions without outgoing control flows are joined.

```
1  operation linkRemainingActions(){
2     while(notProvisioned.isEmpty() == false){
3        for(vertex in provisionableVertexes()){
4           var reqVertexes = vertex.requiredVertexes();
5           if(reqVertexes.size() > 1) {
6              var join = new UML!JoinNode;
7              activity.ownedNode.add(join);
8              reqVertexes.joinIn(join);
9              var cFlow = new UML!ControlFlow;
10             cFlow.source = join;
11             cFlow.target = vertex.equivalent();
12             cFlow.activity = activity;
13             join.outgoing.add(cFlow);
14          }
15          else{
16             cFlow(reqVertexes.first().equivalent(), vertex.equivalent(), activity);
17          }
18          notProvisioned.remove(vertex);
19       }
20    }
21 }
```

Listing 5.11: Provisioning plan control flow generation (ETL).

As the design of the interpretation process of the resulting provisioning plan is already discussed in the design chapter (see Figure 4.11), we focus on the implementation details of action nodes. When an action is reached, a POST request is performed. Hereby, we extract the id contained within the action to search for the corresponding Entity in the OCCI model. Based on the information contained within this Entity, we fill the Category- and Attribute header of the request, whereby we adjust contained Attribute values according to the information contained within the idSwapTable. This is required, as the ids within the values of the Attributes of an Entity contain only the model ids and do not consider the actual ids assigned by the cloud. On a successful request, the id assigned by the cloud is extracted from the server's response and stored together with its original id in the idSwapTable. It must be noticed, that the utilization of this process using OOI requires a specific pre- and post processing step, performed on the initial and final node. Here, we provision and release a *stub network* to which all Compute resources are initially connected. This stub network is required, as otherwise OOI connects each VM to the public network by default. In the following chapter, the implemented comparison- and adaptation process is evaluated using a suite of sample cases.

Chapter 6

Evaluation

To investigate the feasibility of our approach, this section evaluates and discusses the comparison- and adaptation process based on a set of sample cases. These cases are separated into *basic-* and *complex cases*. For the evaluation of the comparison process we provide a ground truth for both case suites describing the expected results. Based on these we compare the expected against the actual outcome in order to investigate the individual drawbacks of the comparison strategies. Section 6.1 introduces these cases in detail. Section 6.2evaluates the different comparison strategies and checks whether they are able to overcome the obstacles of the complex cases. Section 6.3 evaluates the complete adaptation process. Here, we examine the feasibility of the process by investigating an example adaptation, whereby we highlight the interaction between the comparison process and the single adaptation steps. Finally, Section 6.4 discusses the proposed approach based on the evaluation results and answers the questions asked in Chapter 3. Additionally, as no OCCI implementation for OpenStack exists that considers the platform extension, Section 6.4 gives a short excursus on the expected behavior of our approach when applied on PaaS cloud applications.

6.1 Case Descriptions

In our case suite, we differentiate between two kinds, the basic- and complex cases. The basic cases, described in Section 6.1.1, are used in order to evaluate basic functionalities, such as recognizing whether an Entity needs to be created, deleted, or updated. The complex cases are specifically designed to build a burden for the different comparison strategies, in order to evaluate their limitations. These cases are described in Section 6.1.2. As visualizations of complete OCCI application topologies are too large for a depiction of our example cases, we compress the OCCI models in a graph based manner. Hereby, circles represent Resources, whereas edges represent Links. The different labels of the elements represent the Kind of the Resource. Thus, a node labeled VM is a Compute node, NW a Network node, and Stor a Storage node. The subscripts of the node labels represent the id of the Resource. As the standard provides only one Kind of Link that connects two different Resources, the annotation of Kinds for Links are omitted. To also consider the attribute layer, we highlight specific Attributes on the bottom of the node.

6.1.1 Basic Cases

The basic cases, depicted in Table 6.1, describe a suite of simple cloud application topologies. Hereby, each case represents an addition of a specific set of Resources to the Basis case, e.g., the case +VM adds a further VM. For each comparison strategy, these cases are compared one with another. Thus, we created a test suite comprising not only new-, but also missing Entities. Moreover, with these cases we can check whether the comparison strategies are able to handle multiple new or missing Entities. A detailed evaluation of the marking process is not provided, as the process simply investigates the differences of already matched Resources. To provide an idea of how an actual OCCI model looks like refer to Figure 6.1 in which we depict the Basis case. This depiction shows the Basis case within an OCCI model editor. The information about the different Mixins and the Kind of each Entity are stored directly within each element and are therefore not visible in the given figure.

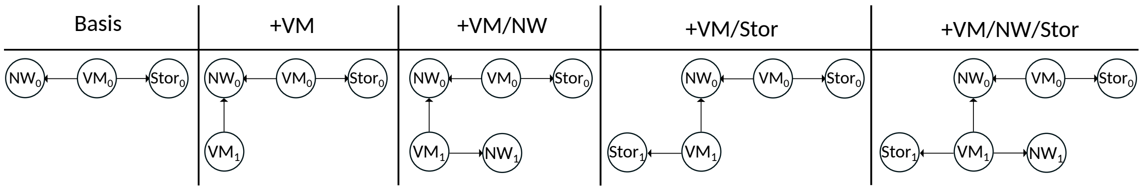
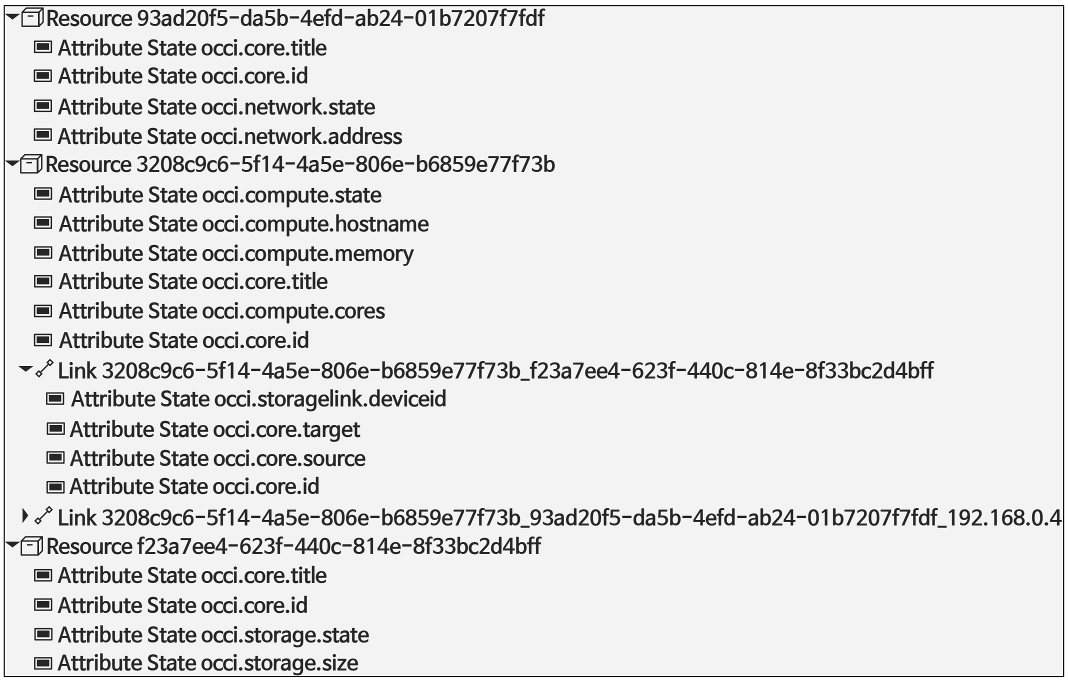


Table 6.1: Basic cases.



```
▾ Resource 93ad20f5-da5b-4efd-ab24-01b7207f7fdf
    Attribute State occi.core.title
    Attribute State occi.core.id
    Attribute State occi.network.state
    Attribute State occi.network.address
▾ Resource 3208c9c6-5f14-4a5e-806e-b6859e77f73b
    Attribute State occi.compute.state
    Attribute State occi.compute.hostname
    Attribute State occi.compute.memory
    Attribute State occi.core.title
    Attribute State occi.compute.cores
    Attribute State occi.core.id
  ▾ Link 3208c9c6-5f14-4a5e-806e-b6859e77f73b_f23a7ee4-623f-440c-814e-8f33bc2d4bff
      Attribute State occi.storagelink.deviceid
      Attribute State occi.core.target
      Attribute State occi.core.source
      Attribute State occi.core.id
  ▸ Link 3208c9c6-5f14-4a5e-806e-b6859e77f73b_93ad20f5-da5b-4efd-ab24-01b7207f7fdf_192.168.0.4
▾ Resource f23a7ee4-623f-440c-814e-8f33bc2d4bff
    Attribute State occi.core.title
    Attribute State occi.core.id
    Attribute State occi.storage.state
    Attribute State occi.storage.size
```

Figure 6.1: OCCI model showing the Basis case.

To check whether the comparison strategies are able to correctly compare two models, Table 6.2 depicts the expected outcome of the Resource matching phase in matching tables. This figure shows, that if a case is compared to itself, it should result in only direct mappings between all of its elements. But when a new or missing element within the models is detected, the left or right side of the specific row should be empty, visualizing whether a match is found for the element or not. To highlight the differences between the comparisons, the direct matches are only initially shown for the comparison between the same model.

Target / Source	Basis		+VM		+VM /NW		+VM /Stor		+VM/NW /Stor	
Basis	VM$_0$	VM$_0$	---	VM$_1$	---	VM$_1$	---	VM$_1$	---	VM$_1$
	NW$_0$	NW$_0$			---	NW$_1$	---	Stor$_1$	---	NW$_1$
	Stor$_0$	Stor$_0$							---	Stor$_1$
+VM	VM$_1$	---	VM$_1$	VM$_1$	---	NW$_1$	---	Stor$_1$	---	NW$_1$
									---	Stor$_1$
+VM /NW	VM$_1$	---	NW$_1$	---	VM$_1$	VM$_1$	NW$_1$	---	---	Stor$_1$
	NW$_1$	---			NW$_1$	NW$_1$	---	Stor$_1$		
+VM /Stor	VM$_1$	---	Stor$_1$	---	---	NW$_1$	VM$_1$	VM$_1$	---	NW$_1$
	Stor$_1$	---			Stor$_1$	---	NW$_1$	NW$_1$		
+VM/NW /Stor	VM$_1$	---	NW$_1$	---	Stor$_1$	---	NW$_1$	---	VM$_1$	VM$_1$
	NW$_1$	---	Stor$_1$	---					NW$_1$	NW$_1$
	Stor$_1$	---							Stor$_1$	Stor$_1$

Table 6.2: Expected Resource matching of the basic cases.

6.1.2 Complex Cases

Overall, we consider five complex cases, which are depicted with their expected match in Table 6.3. The first complex case, Different Ids, tests completely different ids for each Entity in both models, the major limitation of the Simple comparator. As the Similarity Flooding comparison strategy creates a matching of Resources based on their Links, we examine the Independent Resources case. Here, the source model contains a VM without any Link, which is connected to a NW in the target model. The third complex case is used to check how the different comparison strategies handle the occurrence of Symmetric Graphs, the major limitation for a structure based comparison. It should be noted, that the basic cases +VM/Stor depict an equivalent case. However, this time extra information on the attribute level is provided. Moreover, we investigate if the comparison strategies are able to recognize the reattachment of a Stor to another VM. The last complex case allows to evaluate the outcome between an interlinked source model, which is split into two subgraphs.

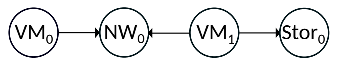
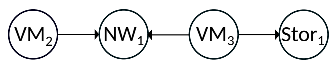
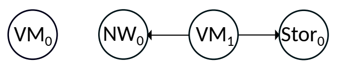
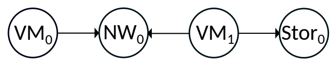
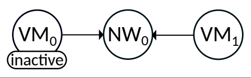
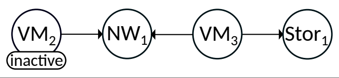
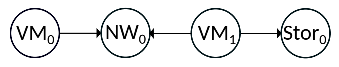
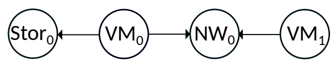
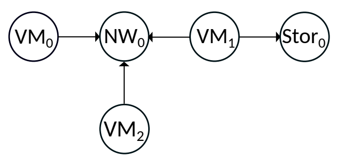
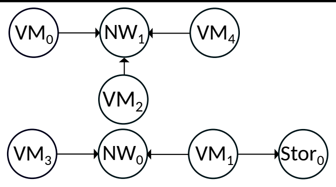
	Source	Target	Expected Match			
Different Ids			VM_0 VM_1	VM_2 VM_3	NW_0 $Stor_0$	NW_1 $Stor_1$
Independent Resources			VM_0 VM_1	VM_0 VM_1	NW_0 $Stor_0$	NW_0 $Stor_0$
Symmetric Graphs			VM_0 VM_1	VM_2 VM_3	NW_0 ---	NW_1 $Stor_1$
Storage Reattachment			VM_0 VM_1	VM_0 VM_1	NW_0 $Stor_0$	NW_0 $Stor_0$
Split into Subgraphs			VM_0 --- VM_2 ---	VM_0 NW_1 VM_2 VM_4	--- NW_0 VM_1 $Stor_0$	VM_3 NW_0 VM_1 $Stor_0$

Table 6.3: Suite of complex cases and expected matches.

6.2 Comparator

In this section the different comparison strategies are evaluated. Hereby, we check if they are able to correctly match the Resources of the basic and complex cases. The investigation of the Link match and the state of matched Entities is shortly discussed in the following. After the description of the comparison outcome, the results are concluded in a short discussion. Section 6.2.1 covers the results of the Simple comparator, Section 6.2.2 the results of the Complex comparator and Section 6.2.3 the results of the Mixed comparator. Based on the evaluated Resource matching, the Link match always results in a correct behavior, as each source Resource is treated in such a manner that it represents the matched target Resource. Also, the marking procedure is not evaluated, as it simply serializes the matching table. Here, each one sided matching is marked either as New or Missing, and every direct matching is compared based on their Attributes to mark them as Old or Updated. Therefore, the resulting match always leads to an adaptation of the source into the target model, independent of how well the comparison is.

6.2.1 Results of the Simple Comparator

The Simple comparator represents a deterministic comparison strategy that matches the Resources of the source and target model based on their unique id. The Simple comparator's Resource matching is identical to the expected results depicted in Figure 6.2. Therefore, it is capable to match every Resource correctly, as long as the ids of the source model Resources are equivalent to the ids of the corresponding Resources in the target model.

For the Simple comparator, the main obstacle is represented by the Different Ids case, as depicted in Table 6.5. Here, all ids from the source and target model are different and therefore no Resource can be matched, although the Resources are expected to be the same. It follows, that each Resource in the source model is marked as Missing Element and each Resource from the target model as New Element. Therefore, this Resource match indicates a complete deprovisioning and re-provisioning of the cloud application. For the rest of the complex cases, the Simple comparator matches every Resource with the same id, leading to the expected Resource match, as long as the ids are set correctly. Thus, the cases Independent Resources, Storage Reattachment and Split into Subgraphs are correctly matched, whereas the cases Different Ids and Symmetric Graphs result in a match indicating a recreation of the cloud application.

Different Ids		Independent Resources		Symmetric Graphs		Storage Reattachment		Split into Subgraphs	
VM_0	---	VM_0	VM_0	VM_0	---	VM_0	VM_0	VM_0	VM_0
NW_0	---	VM_1	VM_1	VM_1	---	VM_1	VM_1	---	NW_1
VM_1	---	NW_0	NW_0	NW_0	---	NW_0	NW_0	VM_2	VM_2
$Stor_0$	---	$Stor_0$	$Stor_0$	---	$Stor_1$	$Stor_0$	$Stor_0$	---	VM_4
---	VM_2			---	VM_2			---	VM_3
---	NW_1			---	VM_3			NW_0	NW_0
---	VM_3			---	NW_1			VM_1	VM_1
---	$Stor_1$							$Stor_0$	$Stor_0$

Table 6.4: Simple comparator results for the complex cases.

Summed up, the Simple comparator represents the perfect comparison strategy, when the ids of the Resources in the target and source model match as supposed and nothing is changed manually in the cloud application due to missing entries in the idSwapTable. As this is a common issue, it represents a viable comparison strategy under the right conditions. Moreover, it can be used as preprocessing for further comparison approaches, as it is the case in the Mixed comparator.

6.2.2 Results of the Complex Comparator

The Complex comparator considers the structure of Resources in the OCCI topology models to create a Resource match. Hereby, the basic cases can only be partially solved, as symmetric structures result in non-deterministic solutions. An overview of cases that can be solved are depicted in Table 6.6, also addressing requirements to solve the rest of the basic cases. As soon as $Stor_1$ is within the source or target model the comparator is not able to correctly classify, whether VM_0 in the target model is VM_0 or VM_1 in the source model. For the +VM/NW/Stor cases, the map pair (VM_0, VM_1) is favored, because it is not only connected to the (NW_0, NW_0) but also to (NW_0, NW_1), resulting in another term in the fixpoint value calculation. However, when this case is compared to itself, the comparator is able to calculate a correct match, as VM_1 is connected to an additional network. Overall, the basic cases uncover one flaw of the Similarity Flooding algorithm, which are symmetric structures, as they lead to similar fixpoint values. In such cases, either the user has to decide how the Resource has to be matched or an attribute filtering process is required.

Compared to the Simple comparator, the structure based approach is able to correctly match the Different Ids case. For the Similarity Flooding algorithm however, one of the limiting cases is represented by the Independent Resources case. Here, VM_0 does not possess any Links and therefore can not be matched to any of the target model's Resources. Therefore, VM_0 is detected as a new as well as a missing Resource. Moreover, the Symmteric Graphs case pushes the Complex comparator to its limits, as similar problems to the ones discovered within the basic cases +VM/Stor and +VM/NW/Stor occur. When comparing the Storage Reattachment case, the algorithm assumes that VM_1 is VM_0, as both are linked to a Storage and the rest of the models are identical.

Different Ids		Independent Resources		Symmetric Graphs		Storage Reattachment		Split into Subgraphs	
VM_0	VM_2	VM_0	---	VM_0	VM_3	VM_0	VM_1	VM_1	VM_1
NW_0	NW_1	---	VM_0	VM_1	VM_2	VM_1	VM_0	NW_0	NW_0
VM_1	VM_3	VM_1	VM_1	NW_0	NW_1	NW_0	NW_0	$Stor_0$	$Stor_0$
$Stor_0$	$Stor_1$	NW_0	NW_0	---	$Stor_1$	$Stor_0$	$Stor_0$	VM_2	VM_3
		$Stor_0$	$Stor_0$					VM_0	VM_2
								---	NW_1
								---	VM_0
								---	VM_4

Table 6.5: Complex comparator results for the complex cases.

For the Split into Subgraphs case the Similarity Flooding algorithm calculates the values as shown in Figure 6.2. The corresponding matching results of these fixpoint values is shown in Table 6.5. Here, the Complex comparator correctly chooses the match for NW_0, VM_1, and $Stor_0$, due to the high fixpoint values resulting from their unique part within the structure. The next highest fixpoint value is the one for $VM_{0,2}$ being VM_3, as their structure fits to the lower subgraph, even though they are connected to a new NW in the target model. Due to the occurrence of equivalent fixpoint values, either may be chosen for the match. In the case depicted in Figure 6.2, VM_2 is chosen. Finally, VM_0 is matched to one of the Compute nodes $VM_{0,2,4}$, as they posses equivalent fixpoint values. However, these fixpoint values are so low, that they even could be discarded as possible matchings. Finally, the left over target VMs are not matched.

Network Resources

NW_0, NW_1	1.0
NW_0, NW_0	<0.001

Compute Resources

VM_1, VM_1	1.0
VM_1, VM_3	0.14
VM_1, $VM_{\{0,2,4\}}$	<0.001
$VM_{\{0,2\}}$, $VM_{\{1,3\}}$	0.14
$VM_{\{0,2\}}$, $VM_{\{0,2,4\}}$	<0.001

Storage Resources

$Stor_0$, $Stor_0$	1.0

Figure 6.2: Fixpoint values and match for the Split Into Subgraphs case.

Assumed that the OCCI models exactly represent the state of the cloud application, the Complex comparator represents a strategy that is able to correctly match each Resource. Unfortunately, that is not the case as the software installed on the different VMs may differ. Still, the Complex comparator represents a viable comparison strategy to match models based on their structure. Nevertheless, it is individually insufficient to compare OCCI models only considering the OCCI IaaS extension, as the amount of different Kinds is low, which leads to non unique model structures. This drawback results in a large amount of equivalent fixpoint values for map pairs considering for example VMs. Therefore, a good filter mechanism is required to identify the correct one, which we evaluate as part of the Mixed comparator in the following section. It must be noted that the algorithm is able to correctly specify the structural requirements of resources. For example, it detects that a VM requires a storage, as in the Split into Subgraphs case. Summed up, the Complex comparator is a viable comparison strategy, especially when more extensions are utilized as they allow for more distinguishable structures.

6.2.3 Results of the Mixed Comparator

The Mixed comparator utilizes both benefits of the Simple- and Complex comparator in addition to an attribute level filter applied on the fixpoint values. Thus, Resources are first compared based on their id, followed by a comparison of left Resources based on the topologies structure. Hereby, Resources of equivalent structure are compared on their attribute level. Due to the usage of the Simple comparator as first comparison strategy, the Mixed comparator is able to handle every basic case. To provide a sufficient evaluation of the Similarity Flooding using the Mixed comparator's attribute filter, we tested each basic case whereby every Resource from the target and source model possesses different ids. The results, depicted in Table 6.6, show that the Mixed comparator is able to correctly match each +VM/Stor case, due to the implementation of the attribute filter.

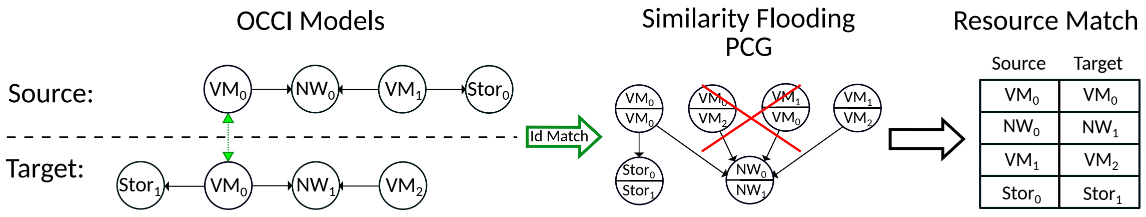
Target / Source	Basis	+VM	+VM /NW	+VM /Stor	+VM/NW /Stor
Basis	▨	▨	▨	●	
+VM	▨	▨	▨	●	
+VM /NW	▨	▨	▨	●	
+VM /Stor	●	●	●	●	●
+VM/NW /Stor				●	▨

▨	Similarity Flooding
●	Attribute Filter
□	One VM Matched

Table 6.6: Mixed comparator results for the basic cases with required features and information.

However, the filter is not enough to correctly match the +VM/NW/Stor cases. That is because the fixpoint values between the storage and VM nodes differ too much, due to the influence of the high fixpoint value from (NW_0, NW_1) on VM_1. This problem can be bypassed, when at least one VM is matched by the Simple comparator, as it recognizes and excludes wrong map pairs in the PCG, leading to a correct matching. For example, in the basis to +VM/NW/Stor case, if the network is matched, the case would be exactly as the +VM/Stor case and can therefore be matched over the attribute filter. If the storage or VM is matched, the map pair ($Stor_0$, $Stor_1$) is removed and therefore VM_0, as well as NW_0, can be correctly matched. Overall, the basic cases uncover that a complete autonomous comparison is not always possible and may require human intervention, especially on small non-unique models.

For the complex cases, the Mixed comparator is able to correctly match the Resources in each case, as shown in Table 6.3. The Different Ids case, due to the Similarity Flooding step and the Independent Resources case, due to the utilization of the id matching of the Simple comparator. The Symmetric Graphs case is handled because of the attribute filter. If the id of VM_0 would differ in the target model, the Resources can still be matched if they posses the same name, due to the post processing attribute comparison of non-matched Resources. The Storage Reattachment and Split into Subgraphs case results in a correct Resource matching, as the ids of the source model match to the ones of the target model. In case of different ids, depicted in Figure 6.3, the minimum information of one matched VM is required to calculate a correct Resource match, using the Similarity Flooding algorithm. Here, the information gained from the Simple comparison step, the match of VM_0 to the other VM_0, results in a deletion of every wrong case in the PCG and therefore in a correct matching, as only one match per node is still available. The same behavior occurs when only the id of VM_1 can be matched, as the amount of possible solutions for the Similarity Flooding algorithm shrink. For the Split into Subgraphs case, two flaws of the Similarity Flooding algorithm are uncovered. One flaw is the matching of Networks and the equivalent fixpoint values of the VM map pairs. Because the case can be solved with the help of the Simple comparator, we shortly discuss this case with the target model having different ids, as it is the drawback of the Simple comparator. The attribute filter helps to identify the most suitable VM matching. This comes with the drawback of correctly recognizing VMs being updated in the target model as they differ on the attribute layer. In this case, a more unique structure of the VMs or a manual adjustment of the matching is required.



Source	Target
VM_0	VM_0
NW_0	NW_1
VM_1	VM_2
$Stor_0$	$Stor_1$

Figure 6.3: Adjusted Storage Reattachment case in which only one id of the VM nodes match.

The Mixed comparator delivers the best results of the proposed comparison strategies, as the information about the ids of the Entities, which uniquely identifies them, grants extra information for the Complex comparison process analyzing its structure. This extra information about already matched Resources affects the amount of possible map pairs in the PCG used in the Similarity Flooding algorithm. Even the information about one correctly identified Resource greatly supports the Similarity Flooding process, as an anchor point is found. Thus, the fixpoint values of these nodes are increased rolling up the complete structure of the model. The attribute filter grants a huge advantage over a complete structure based filter, as it allows to filter out map pairs that are less suitable than others. Nevertheless, to make a perfect use out of this filter, its threshold must be evaluated for the different OCCI Resource types. Finally, the post-processing performed on non-matched Resources serves as a last gateway to perform a reasonable matching. Overall, the Mixed comparator represents a suitable approach to match OCCI topology models, as it utilizes all information available in the models. Nevertheless, due to the high abstraction level of the OCCI infrastructure extension, cases exist in which the source and target model can not be perfectly matched, as only an excerpt of the cloud application is considered. Therefore, a manual inspection of the Resource matching phase is recommended, as wrong matchings may result in fatal changes. The following section investigates the complete adaptation process, focusing on the adaptation steps making use of the comparison.

6.3 Adaptation Process

To discuss the complete adaptation process and show how actual REST requests are performed, we examine the procedure of the different adaptation steps, based on a minimal example. The target model used as input is depicted as an object diagram in Figure 6.4. This figure gives a detailed overview of the information contained within each Entity, which we extract for corresponding REST calls. The target model is composed of an Ubuntu 16.04 Server VM of size m1.medium, as stated in the Mixin Templates. This VM is connected to a network nw0 and a storage stor0, whereby the network and its interface are attached to the Ipnetwork mixins. A simplified version is depicted in Figure 6.5, which additionally shows the runtime model and the comparison results.

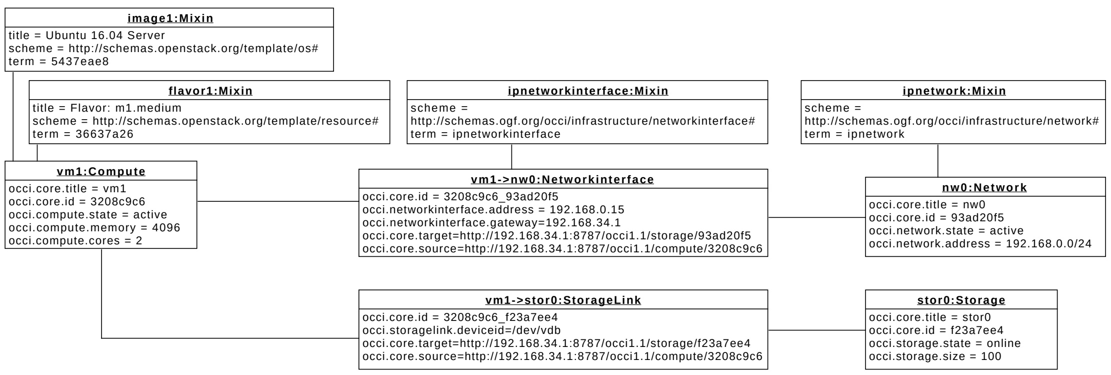


image1:Mixin
title = Ubuntu 16.04 Server
scheme = http://schemas.openstack.org/template/os#
term = 5437eae8
flavor1:Mixin
title = Flavor: m1.medium
scheme = http://schemas.openstack.org/template/resource#
term = 36637a26
ipnetworkinterface:Mixin
scheme =
http://schemas.ogf.org/occi/infrastructure/networkinterface#
term = ipnetworkinterface
ipnetwork:Mixin
scheme =
http://schemas.ogf.org/occi/infrastructure/network#
term = ipnetwork
vm1:Compute
occi.core.title = vm1
occi.core.id = 3208c9c6
occi.compute.state = active
occi.compute.memory = 4096
occi.compute.cores = 2
vm1->nw0:Networkinterface
occi.core.id = 3208c9c6_93ad20f5
occi.networkinterface.address = 192.168.0.15
occi.networkinterface.gateway=192.168.34.1
occi.core.target=http://192.168.34.1:8787/occi1.1/storage/93ad20f5
occi.core.source=http://192.168.34.1:8787/occi1.1/compute/3208c9c6
nw0:Network
occi.core.title = nw0
occi.core.id = 93ad20f5
occi.network.state = active
occi.network.address = 192.168.0.0/24
vm1->stor0:StorageLink
occi.core.id = 3208c9c6_f23a7ee4
occi.storagelink.deviceid=/dev/vdb
occi.core.target=http://192.168.34.1:8787/occi1.1/storage/f23a7ee4
occi.core.source=http://192.168.34.1:8787/occi1.1/compute/3208c9c6
stor0:Storage
occi.core.title = stor0
occi.core.id = f23a7ee4
occi.storage.state = online
occi.storage.size = 100

Figure 6.4: Object diagram describing the target model.

Here, the running cloud application is composed of two VMs, which are connected to a network. This application got extracted into a runtime model consisting of two Compute nodes, VM_0 and VM_1, which both are connected to a Network, NW_0. The extraction step itself is not discussed in detail, as it simply creates an OCCI model using an already existing interface. This runtime model is compared to the target model possessing an additional Storage node, which is connected to the now active VM_1. This example is chosen, because its comparison results comprise not only Missing- and New-, but also Old- and Updated Elements. After the extraction of the runtime model and its

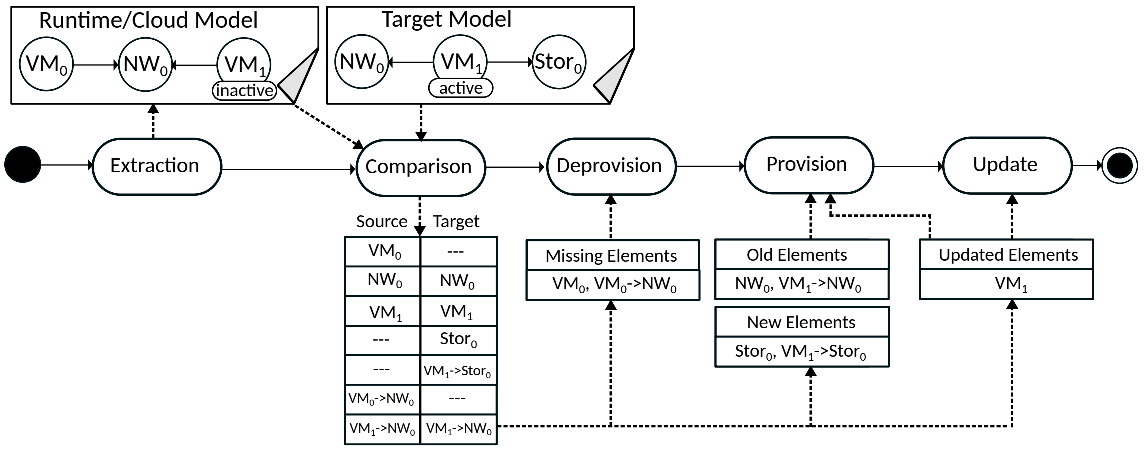


Figure 6.5: Minimal adaptation example.

comparison to the target model, the deprovisioning of the Missing Elements takes place, which sorts them for Links and Resources. Therefore, at first the Link, VM_0->NW_0 is deleted, detaching VM_0 from the network, followed by a DELETE request of VM_0 itself. It should be noted that, when a Resource is marked as missing, each Link of it is also marked as missing, e.g, each Resource is completely detached before it is deleted. This detachment process is especially important, because networks sometimes can only be deleted once nothing is connected to them. The information required to delete these elements is hereby read out of the runtime model. The DELETE request to deprovision VM_0 simply requires the information about its URI, and therefore about its Kind and id. This results in the request depicted in Listing 6.1. Hereby, TOKEN represents the session token requested, which is generated at the beginning of the adaptation process. After the deletion, the id of the VM_0 is removed from the idSwapTable.

```
DELETE http://192.168.34.1:8787/occi1.2/compute/VM0
-H 'X-Auth-token: ' TOKEN
```

Listing 6.1: Example DELETE request.

Thereafter, the Updated Elements are handled, changing the state of VM_0 to active. In the proposed example, we intentionally chose an element adaptation that can be performed by Actions, as OOI does not provide any implementation for PUT or POST update requests. To bring the state Attribute from inactive to active, the start Action is executed, which is shown in Listing 6.2. Here, the runtime id of the VM, `3208c9c6+`, is extracted using the matching table and idSwapTable.

```
POST http://192.168.34.1:8787/occi1.2/compute/3208c9c6+?action=start
-H 'Content-Type: text/occi'
-H 'X-Auth-Token: ' TOKEN
-H 'Category: start; scheme="http://schemas.ogf.org/occi/infrastructure/compute/
    action#"; class="action"'
```

Listing 6.2: Example POST(Action) request.

During the provisioning process, the POG is created out of the target model from which we remove the Old Elements and Updated Elements, as shown in Figure 6.6. Additionally, this figure depicts the provisioning plan for the original POG. Thus, we highlight the difference the actions taken when the target model would be deployed without considering the runtime model. As this process is already discussed in [45], a shortened version is given in the following. As each Resource provisioning is independent from each other, they represent vertexes without incoming edges. Due to the mapping of each Link type of the infrastructure to the use dependency, the Link vertexes depend on the creation of the Resources it connects. Therefore, a workflow is created which starts with the parallel provisioning of each Resource followed by the provisioning of Links connecting them. As we adapt the POG by deleting the Old- and Updated Elements, we remove VM_0, VM_0->NW_0, and NW_0. Thus, the POG is transformed into a workflow that only contains of actions that provision $Stor_0$ and VM_0->$Stor_0$. Concluding, the POG and workflow plan only consist of the New Elements to be created. Furthermore, the adaptation removes the dependencies for the Link VM_0->$Stor_0$, as it is already resolved in the cloud application.

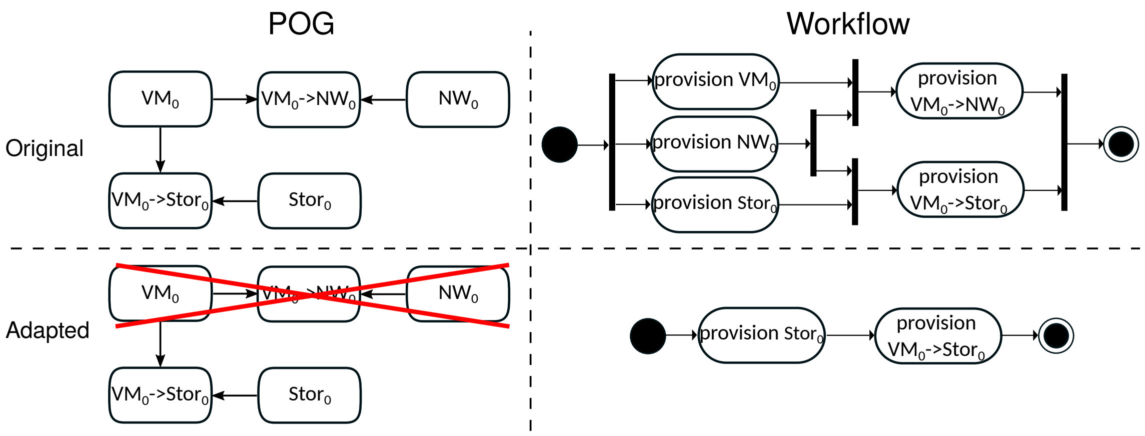


Figure 6.6: Adjusted POG and activity diagram for the provisioning process.

As the complete information of a Resource is required for a POST request, we extract the Category and Attribute header directly out of the corresponding Entities from the target model. This results in the request of Listing 6.3, depicting the creation of Stor_0. This request dynamically assigns an id to the provisioned Entity, which we extract from the server's response. In the following, we refer to this id as `f23a7ee4+`, which is stored together with its original id f23a7ee4 in the idSwapTable.

```
POST http://192.168.34.1:8787/occi1.2/storage/
-H 'Content-Type: text/occi'
-H 'X-Auth-Token: ' TOKEN
-H 'Category: storage; scheme="http://schemas.ogf.org/occi/infrastructure\#"; class="
    kind"'
-H 'X-OCCI-Attribute: occi.core.title="stor0", occi.core.id="f23a7ee4", occi.storage.
    state="online", occi.storage.size="100"'
```

Listing 6.3: Example Resource POST request.

Thereafter, the StorageLink connecting VM_0 to Stor_0 is provisioned, resulting in the request shown in Listing 6.4. Here, the id of the Link's target Resource, `stor0`, is already stored within the idSwapTable as `f23a7ee4+`. Furthermore, the id of the source Resource, `vm1`, is stored as `3208c9c6+`. Again, the cloud assigns a dynamic id for this Resource, which we store together with its counterpart in the idSwapTable. After this request, the provisioning workflow reaches the final node, closing the workflow.

```
POST http://192.168.34.1:8787/occi1.2/storagelink/
-H 'Content-Type: text/occi'
-H 'X-Auth-Token: ' TOKEN
-H 'Category: storagelink; scheme="http://schemas.ogf.org/occi/infrastructure\#";
    class="kind"'
-H 'X-OCCI-Attribute: occi.core.id="3208c9c6+_f23a7ee4+", occi.core.source="3208c9c6
    +", occi.core.target="f23a7ee4+", occi.storagelink.deviceid="/dev/vdb"'
```

Listing 6.4: Example Link POST request.

The proposed adaptation process is capable of changing the state of a cloud application using a runtime representation of it. Hereby, we adapted the proposed provisioning process, presented in [45], to utilize runtime information. The required adaptive steps itself are derived from the results of the comparison step. As these steps are loosely coupled, any arbitrary comparison strategy can be used, whereby the Mixed comparator is recommended as discussed in Section 6.2. Also, the single steps are loosely coupled, representing an approach easy to maintain. Nevertheless, when further OCCI extensions are utilized this design may result in problems. For example, the dependencies resulting from updating elements may affect the provisioning process and vice versa. In the following, an overall discussion of the proposed approach is given including a short excursion about expected influences regarding OCCI's platform extension [21].

6.4 Discussion

To show the feasibility of the proposed approach we separately evaluated the comparison- and adaptation process. For the comparison process we evaluated each strategy based on two example suites. One suite was focused on the evaluation of basic functionality, whereas the second suite tried to investigate the limitation of the different strategies. Hereby, we showed that the Simple comparator is able to provide deterministic matching results as long the ids of the different cloud resources are set correctly. The complete opposite is represented by the results of the Complex comparator, which is not able to deterministically match symmetric structures, but therefore is independent from the attribute level. Finally, the Mixed comparator, using the benefits of both approaches, yielded in the best results, as not only a deterministic match based on ids is performed, but also a structural comparison followed by an iterative identification of the most suitable matches using an attribute filter. However, cases exist that require specific Resource matches from the first comparison phase to be correctly solved. Thus, on large and complex adaptations a manual inspection of the calculated comparison may be necessary.

The adaptation process was evaluated based on an example cloud application topology, which adapts a previously defined runtime state. Hereby, we created a case in which the comparison indicates an execution of all adaptation steps. Thus, we were able to evaluate each step of the complete adaptation process covering the comparison-, the deprovisioning-, the update-, and the provisioning step. Hereby, we highlighted specific requirements such as the resolution of provisioning dependencies and the constraints regarding the deprovisioning of resources. Overall, we were able to show the feasibility of our proposed approach using one minimal example. To conclude the evaluation, the following answers the questions stated in Chapter 3 of this thesis:

- **Q0:** How to recognize whether two elements from different models match?

To answer this question we proposed three different comparison strategies. Hereby, we showed that a reasonable matching can be calculated based on the information contained within the elements attributes and structure. Nevertheless, we examined multiple limitations when only one of these levels is considered. However, when these levels are combined these limitations can be bypassed, as shown by the results of the Mixed comparator.

- **Q1:** Is there a suitable order for the several adaptation steps?

The derivation of required adaptive steps requires a model representation of the running cloud application. Thus, the extraction process represents the first step. As cloud tenants may be limited to an amount of cloud resources that can be assigned simultaneously, we next deprovision cloud resources not needed anymore. Thereafter, we adapt each element requiring for an update to bring it into the desired state. This is required for the provisioning process, as the target model depicts a working state of the cloud application and thus inconsistencies in the runtime have to be avoided.

- **Q2:** How to extract information from the running system in form of an OCCI model?

We extracted the running cloud application into an OCCI model by using the *jOCCI-API* [51], which is able to perform OCCI queries. Nevertheless, a serialization of the information gathered by this API is required to fit the extracted model onto the metamodel of Merle et al. [48] we use.

- **Q3:** Are there requirements for the deletion of OCCI elements?

The utilization of the OpenStack cloud infrastructure showed that some OCCI elements have to be decoupled from the rest of the infrastructure before they can be deleted. Thus, we created a process which systematically decouples a Resource by removing all attached Links to it, followed by a deletion of the Resource itself. To address requirements not yet found or introduced with further extensions, we designed the process in such a manner that it can be individually customized for each Kind.

- **Q4:** How to identify required adjustment calls?

To identify which requests are required to update an Entity it must be evaluated whether an adjustment over Actions or a complete update over a PUT call is required. Therefore, information about which Actions change which Attributes into which states is required. At this point in time, a manual creation of an Action list storing this information is necessary. Because we evaluated the proposed approach using OOI as OCCI API, the behavior of PUT requests could not be evaluated due to a missing implementation.

- **Q5:** How to resolve the dependencies between the different cloud resources?

Finally, to solve the dependencies of the single elements within the cloud topology we adapted an approach by Breitenbücher et al. [44]. Using this approach, we created a provisioning plan describing the order of the REST calls to be performed. Even though this process was already discussed in a former research project [45], we presented it in this thesis to evaluate its adjustment to fit into an adaptation process.

- **Q6:** How to address already running resources within these dependencies?

To address already running resources within the generation of a provisioning plan, we adjusted the POG by removing Entities already running. Thus, the dependencies considering already running resources are resolved from which the provisioning plan can be generated as usual. Hereby, we identified the already running resources over the Old- and Updated Elements resulting from the comparison process. In the following, Section 6.4.1 discusses validity threats for the proposed process, whereas Section 6.4.2 provides a short excursion of expected influence from OCCI's platform extension [21] on our approach.

6.4.1 Threats to Validity

In this section circumstances that threaten the validity of the proposed approach are discussed. One of the biggest impacts is the utilization of additional OCCI extensions, which may result in different requirements for the resolution of Resource dependencies and thus for a more specialized configuration of our approach. However, these could not be considered due to the lack of compatible OCCI implementations. Nevertheless, a short excursus covering the utilization of the platform extension [21] is given in Section 6.4.2. Additionally, major changes in OCCI's basic structure may influence parts of the comparison- and adaptation processes.

Even though we presented a large variety of different cases for the evaluation of the proposed approach, we did not test how the comparison- as well as the adaptation process behave for larger infrastructure topologies. Nevertheless, these were sufficient to investigate the advantages and drawbacks of the different comparison strategies and show the feasibility of our approach. Additionally, an evaluation of the proposed adaptation process on multi-cloud environments is missing which may result in requirements not discovered yet.

6.4.2 Excursion: OCCI PaaS Extension

The platform extension comes with the capability to manage Applications running on the different Compute nodes. For this, the platform extension defines the Application type, which gets attached to Compute elements. This additional information for a Compute node especially supports the Similarity Flooding comparison process, as the cloud application's structure and therefore the Resources get more distinguishable. Hereby, the degree of uncertainty of matching two Compute nodes is reduced, due to an Application being composed of multiple Components. Summed up, the platform extension would provide a huge advantage for the comparison process, as OCCI models get more detailed.

To correctly represent the running cloud application topology as OCCI model, the extraction process needs to be adapted as information about the software state's running on the individual machines is required. The rest of the adaptation steps mainly require an OCCI API, providing the platform extension. As the interface is responsible for the management of single elements, the form of the REST requests itself does not differ, e.g, the creation of a Storage represents the same procedure as provisioning an Application. When adjusting the proposed approach to support the platform extension, the update and provisioning process require for a further evaluation, as the creation of the POG is in addition composed of depends-on patterns. Especially, the interconnection between several software parts deployed on different VMs represent a circumstance to be investigated. In the following, an introduction into related work is given in which we delimit our approach to similar ones.

Chapter 7

Related Work

In addition to OCCI, other cloud standards exists having a model based nature that tackle the provider lock-in. One of these standards is the Topology and Orchestration Specification for Cloud Applications (TOSCA) [52], which we shortly introduce in this chapter. Furthermore, we delimit similar approaches to the proposed one in order to further highlight its necessity. Also, we describe the different processes we adapted in this thesis to compare their and our objective.

TOSCA [52], developed by *Organization for the Advancement of Structured Information Standards (OASIS) [53]*, is another cloud standard aiming to solve the provider lock-in. Its main goal is to increase the portability of cloud applications to allow for convenient and provider independent switching of the cloud infrastructure. Just like OCCI, TOSCA defines a metamodel for the creation of cloud resources. Hereby, it provides two kinds of model structures, one based on XML [54], whereby the metamodel is in the form of an XSD, and one based on *YAML Ain't Markup Language (YAML)* [55], which lacks a formal metamodel [56]. The major difference between those two standards lies within their goals. TOSCA aims for reusability and persistence of cloud application models without defining its actual deployment, whereas OCCI provides the definition of a standardized API [56]. In Section 8.1, an outlook into future work is given where we combine our adaptation process with a former approach [56] that provides a mapping of TOSCA and OCCI elements.

In the approach of Breitenbücher et al. [44] we adapted originally describes the generation of a provisioning plan from TOSCA models. Nevertheless, this approach does not consider resources already running on the cloud, and therefore is not capable to perform an adaptation. Additionally, it does not define a standardized way of the actual provisioning, as TOSCA does not specify an API.

Kolovos et al. [57] present multiple types of model comparisons- and matching approaches. One of them is EMF Compare [42], which adds the capability to compare models to the EMF framework. Even though it is a powerful, customizable, and extensible tool that creates a matching between elements of the source and target model, its comparison mainly operates on the attribute level, as it evaluates the similarity of elements based on their features. Therefore, we stick to the Similarity Flooding algorithm to evaluate how well a pure structure based comparison strategy behaves. Moreover, as we consider each attribute to be object of change, our attribute comparison is restricted to one attribute, the id.

A similar adaptation approach is proposed by Holmes [58], who extracts a runtime model and compares it against a model to be deployed. To create and extract these models, he created a generalized metamodel tailored towards the OpenStack API and thus is not conform to any existing cloud standard. Furthermore, compared to our approach only a short concept is given, addressing the resolution of dependencies between the required adaptation calls. For the derivation of the adaptive steps, he compared the cloud application topologies using EMF Compare. Hereby, he configured the comparison in such a manner that only the elements contents are considered ignoring their identifiers. Unfortunately, no evaluation of the proposed adaptation- or comparison process is given making a comparison to our results impossible.

Another models at runtime process capable of adapting cloud applications is proposed by Ferry et al. [59]. To enable an adaptation, they define a *Domain-Specific Language (DSL)* for cloud application models, called *CloudML*, and a runtime environment to execute calculated adaptation plans. Here, they use a three layer architecture for self-adaptive systems by Kramer and Magee [60]. In terms of this architectural style, our approach can be classified as the *short-term layer*, being responsible for the management and deployment of the application. Compared to our approach, they use a self-created metamodel instead of accepted cloud standards and therefore do not tackle the provider lock-in.

One further big project using the OCCI standard is *OCCIware* [61], a framework capable of modeling, designing, and deploying every kind of computing resources as a service, utilizing the OCCI Core Model. Hereby, it allows to manage any kind of resource by combining multiple OCCI back-ends, which are hidden behind its generic front-end. Furthermore, OCCIware provides the capability to deploy, reconfigure, and monitor cloud applications. However, it is not able to adapt complete cloud applications. In the following, a final conclusion about the proposed approach is given.

Chapter 8

Conclusion

We presented a model based approach capable of adapting cloud applications, using the OCCI standard. To perform this adaptation, we designed two modular processes, the comparison between the runtime and target model and the execution of required adaptation steps. For the comparison process, we investigated three approaches to create possible Resource matchings. Based on these, we identified Link matches and marked Entities as Old-, Updated-, New-, or Missing Element. Hereby, we evaluated the capabilities of comparison strategies operating on the attribute-, structure-, and mixed-level. This evaluation showed that each of these strategies result in reasonable matchings if applied correctly. However, as the Mixed comparator presents the most flexible and to the problem space tailored strategy, it results in the most accurate matches. Furthermore, we showed that even when the matches are not well-chosen, the state described in the target model is always reached. However, the results should always be checked manually in order to prevent wrong adaptations.

Based on this comparison we created an adaptation process, which we decoupled into three adaptation steps responsible to manage DELETE, PUT, and POST requests. Depending on the mark of each Entity, we assigned them to one of these steps. For each of these management tasks, we investigated different requirements. To deprovision a Resource, we decoupled it from the rest of the application. To update an Entity, we checked whether it can be handled over Actions. For the provisioning process, we identified dependencies of the single Entities from which we generated a provisioning plan. Overall, we presented a modular approach that is capable of adapting a cloud application, only requiring the desired state of the cloud application as input. Thus, we neglect not only the need for any human intervention, but also bypass the provider lock-in problem by extracting standardized REST calls directly from the model. As the proposed prototype merely represents a portion of possible capabilities, we present future work in the following section, discussing new and enhanced functionalities.

8.1 Future Work

Even though we presented a short excursus into the PaaS extension of OCCI where we described the influence of additional element types on the comparison process, a more formal evaluation is necessary. Moreover, further comparison strategies can be evaluated against each other. Especially, to investigate further pre-comparison steps that can be utilized in the Mixed comparator. Hereby, it is worth evaluating how the Similarity Flooding algorithm performs when only the highest fixpoin values are extracted as a map pair, followed by another iteration of the algorithm with an adapted PCG. Moreover, the attribute filter used for the fixpoint values can be enhanced by adding weights to specific attributes or by recognizing how well an updated Attribute fits to the structure of the Resource. For example, a Resource having multiple Links can be favored for an upscale process.

To enhance the functionality of the adaptation process, multiple functionalities can be added. For the update process, state machines can be extracted from the runtime model that list Actions and how they affect specific Attributes. Thus, the need for a manual configuration is neglected. Using these state machines, the shortest path between two Attribute states can be calculated leading to the minimal amount of Actions needed to reach a certain state. Furthermore, it can be evaluated whether a set of Actions is more suitable than one PUT request to adapt single Entities. Another functionality that can be added is an engine enabling self managing capabilities for cloud applications, in which a complete MAPE loop is utilized in order to choose the correct target model for different environmental requirements. E.g, a monitoring step can be created that chooses a suitable OCCI model for the analyzed parameters, which then serves as input for the proposed adaptation process. Moreover, as already mentioned, the proposed process is able to enable self healing mechanisms for the deployed cloud application, as the desired state can be stored and compared to the current runtime state of the cloud application. If any differences occur, the proposed adaptation process is simply executed. Therefore, a process is required that periodically extracts the runtime model of the cloud application and checks for occurring differences using the comparison process. When differences are detected, i.e. not only Old Elements exist, the adaptation process is started, adapting the corrupted state of the cloud application. However, this process requires a further blacklist indicating attributes to be ignored. Thus, it can be configured that the process only cures resource failure states.

Finally, as we described the benefits of TOSCA and the similarities of it to OCCI, it would be interesting to investigate how the comparison strategies perform on TOSCA topology models, as these are designed for a permanent storage of cloud topologies. Furthermore, as we presented a transformation of TOSCA to OCCI models [56], we are going to evaluate how well these transformed models can be used for the proposed adaptation process.

Bibliography

[1] Open Grid Forum, "Open Cloud Computing Interface - Core," 2016, Available online: https://www.ogf.org/documents/GFD.221.pdf, last retrieved: 20.09.2017.

[2] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," *Electrical Engineering and Computer Sciences, University of California at Berkeley*, 2009.

[4] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *National Institute of Standards and Technology*, vol. 53, no. 6, p. 50, 2009.

[5] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[6] W. Vogels, "A Head in the Clouds—The Power of Infrastructure as a Service," in *First workshop on Cloud Computing and in Applications (CCA'08)(October 2008)*, vol. 5, 2008.

[7] Google, "GSuite," Available online: https://gsuite.google.com/index.html, last retrieved: 20.09.2017.

[8] Google, "Google App Engine," Available online: https://cloud.google.com/appengine/, last retrieved: 20.09.2017.

[9] Amazon, "Amazon Web Services Elastic Beanstalk," Available online: https://aws.amazon.com/elasticbeanstalk/, last retrieved: 20.09.2017.

[10] Amazon, "Amazon Web Services Elastic Compute Cloud," Available online: https://aws.amazon.com/ec2/, last retrieved: 20.09.2017.

[11] Linux Foundation, "Xen," Available online: https://www.xenproject.org/, last retrieved: 20.09.2017.

[12] Open Virtualization Alliance, "Kernel-based Virtual Machine (kvm)," Available online: https://www.linux-kvm.org/page/Main_Page, last retrieved: 20.09.2017.

[13] VMware Inc., "VMWare," Available online: http://www.vmware.com/, last retrieved: 20.09.2017.

[14] A. Salam, Z. Gilani, and S. U. Haq, *Deploying and Managing a Cloud Infrastructure*. John Wiley & Sons, 2015.

[15] R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "IaaS Cloud Architecture: From

Virtualized Datacenters to Federated Cloud Infrastructures," *Computer*, vol. 45, pp. 65–72, 2012.

[16] OpenStack, "Newton," 2016, Available online: https://releases.openstack.org/newton/, last retrieved: 20.09.2017.

[17] Open Grid Forum, "Open Cloud Computing Interface," 2010, Available online: http://occi-wg.org/, last retrieved: 20.09.2017.

[18] Open Grid Forum, "An Open Global Forum for Advanced Distributed Computing," 2006, Available online: https://www.ogf.org/ogf/doku.php, last retrieved: 20.09.2017.

[19] Open Grid Forum, "Open Cloud Computing Interface - Service Level Agreements," 2016, Available online: https://www.ogf.org/documents/GFD.228.pdf, last retrieved: 20.09.2017.

[20] Open Grid Forum, "Open Cloud Computing Interface - Infrastructure," 2016, Available online: https://www.ogf.org/documents/GFD.224.pdf, last retrieved: 20.09.2017.

[21] Open Grid Forum, "Open Cloud Computing Interface - Platform," 2016, Available online: https://www.ogf.org/documents/GFD.227.pdf, last retrieved: 20.09.2017.

[22] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[23] Internet Engineering Task Force, "Hypertext Transfer Protocol Versuib 2–HTTP/2," 2015, RFC 7540, Available online: https://www.rfc-editor.org/rfc/pdfrfc/rfc7540.txt.pdf, last retrieved: 20.09.2017.

[24] Open Grid Forum, "Open Cloud Computing Interface - Text Rendering," 2016, Available online: https://www.ogf.org/documents/GFD.224.pdf, last retrieved: 20.09.2017.

[25] Open Grid Forum, "Open Cloud Computing Interface - HTTP Protocol," 2016, Available online: https://www.ogf.org/documents/GFD.223.pdf, last retrieved: 20.09.2017.

[26] Open Grid Forum, "Open Cloud Computing Interface - JSON Rendering," 2016, Available online: https://www.ogf.org/documents/GFD.226.pdf, last retrieved: 20.09.2017.

[27] E. Seidewitz, "What Models Mean," *IEEE software*, vol. 20, no. 5, pp. 26–32, 2003.

[28] J. Bézivin, "In Search of a Basic Principle for Model Driven Engineering," *Novatica Journal, Special Issue*, vol. 5, no. 2, pp. 21–24, 2004.

[29] J.-M. Favre, "Towards a Basic Theory to Model Model Driven Engineering," in *3rd workshop in software model engineering, wisme*, 2004, pp. 262–271.

[30] Object Management Group, "MDA Guide Version 1.0.1," 2003, Available online: http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf, last retrieved: 20.09.2017.

[31] T. Kühne, "Matters of (meta-) modeling," *Software & Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[32] H. Stachowiak, *Allgemeine Modelltheorie.* Springer-Verlag, 1973.

[33] Object Management Group, "Unified Modeling Language Infrastructure Specification," 2011, Available online: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF, last retrieved: 20.09.2017.

[34] World Wide Web Consortium, "Extensible Markup Language," 2006, Available online: https://www.w3.org/TR/2008/REC-xml-20081126/, last retrieved: 20.09.2017.

[35] Object Management Group, "OMG Meta Object Facility (MOF) Core Specification," 2016, Available online: http://www.omg.org/spec/MOF/2.5.1/PDF/, last retrieved: 20.09.2017.

[36] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[37] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[38] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. Polack, "The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering," in *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*. IEEE, 2009, pp. 162–171.

[39] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Software & Systems Modeling*, vol. 15, no. 1, pp. 31–69, 2016.

[40] N. Bencomo, G. Blair, S. Götz, B. Morin, and B. Rumpe, "Report on the 7th International Workshop on Models@run.time," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 1, pp. 27–30, 2013.

[41] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[42] Eclipse, "EMF Compare," 2011, Available online: http://www.eclipse.org/emf/compare/, last retrieved: 20.09.2017.

[43] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching," in *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 2002, pp. 117–128.

[44] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 87–96.

[45] J. Erbel, "Declarative Cloud Resource Provisioning Using OCCI Models," in *Informatik 2017, 47. Jahrestagung der Gesellschaft für Informatik*, 2017.

[46] OpenStack OCCI Interface, "OpenStack OCCI Interface," 2015, Available online: http://ooi.readthedocs.io/en/stable/, last retrieved: 20.09.2017.

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

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

[49] Object Management Group, "Unified Modeling Language," 2015, Available online: http://www.omg.org/spec/UML/2.5/PDF, last retrieved: 20.09.2017.

[50] Eclipse, “UML2, an EMF-based implementation of the Unified Modeling Language (UML) 2.x,” Available online: https://wiki.eclipse.org/MDT/UML2, last retrieved: 20.09.2017.

[51] M. Kimle, B. Parák, and Z. Šustr, “jOCCI–General-Purpose OCCI Client Library in Java,” in *International Symposium on Grids and Clouds (ISGC)*, vol. 15, no. 20, 2015.

[52] Organization for the Advancement of Structured Information Standards, “Topology and Orchestration Specification for Cloud Applications,” 2013, Available online: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca, last retrieved: 20.09.2017.

[53] Organization for the Advancement of Structured Information Standards, “Advancing open standards for the information society,” 1993, Available online: https://www.oasis-open.org/, last retrieved: 20.09.2017.

[54] Organization for the Advancement of Structured Information Standards, “Topology and Orchestration Specification for Cloud Applications (TOSCA) 1.0,” 2013, Available online: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html, last retrieved: 20.09.2017.

[55] Organization for the Advancement of Structured Information Standards, “TOSCA Simple Profile in YAML Version 1.0,” 2013, Available online: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html, last retrieved: 20.09.2017.

[56] F. Glaser, J. Erbel, and J. Grabowski, “Model driven cloud orchestration by combining tosca and occi,” in *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: (CLOSER 2017)*, INSTICC. SciTePress, 2017, pp. 672–678.

[57] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, “Different Models for Model Matching: An analysis of approaches to support model differencing,” in *Comparison and Versioning of Software Models, 2009. CVSM’09. ICSE Workshop on*. IEEE, 2009, pp. 1–6.

[58] T. Holmes, “Facilitating Migration of Cloud Infrastructure Services: A Model-Based Approach,” in *CloudMDE@MoDELS*, 2015, pp. 7–12.

[59] N. Ferry, G. Brataas, A. Rossini, F. Chauvel, and A. Solberg, “Towards bridging the gap between scalability and elasticity,” in *Proceedings of the 4th International Conference on Cloud Computing and Services Science - Volume 1: (CLOSER 2014)*, INSTICC. SciTePress, 2014, pp. 746–751.

[60] J. Kramer and J. Magee, “Self-Managed Systems: an Architectural Challenge,” in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 259–268.

[61] OCCIware, “A formal framework for the management of any digital resource in the cloud,” 2015, Available online: http://www.occiware.org/bin/view/Main/, last retrieved: 20.09.2017.

List of Abbreviations

API	Application Programming Interface	9
AWS	Amazon Web Services	8
CRUD	Create, Read, Update, Delete	15
DNS	Domain Name System	14
DSL	Domain-Specific Language	72
EC2	Elastic Compute Cloud	8
EMF	Eclipse Modeling Framework	17
ETL	Epsilon Transformation Language	19
GAE	Google App Engine	8
HTTP	Hypertext Transfer Protocol	10
IPG	Induced propagation graph	29
IP	Internet Protocol	13
IaaS	Infrastructure as a Service	8
JSON	JavaScript Object Notation	16
L2	Data link layer	12
L3	Network layer	13
L4	Transport layer	13
M2M	model-to-model transformation	35
MAC	Media Access Control	13
MAPE	Monitor-Analyze-Plan-Execute	20
MDA	Model-driven architecture	16
MDE	Model Driven Engineering	2
MOF	Meta Object Facility	16
NIST	National Institute of Standards and Technology	6
OASIS	Organization for the Advancement of Structured Information Standards	71
OCCI	Open Cloud Computing Interface	1
OGF	Open Grid Forum	1
OMG	Object Management Group	16
OOI	OpenStack OCCI Interface	39
OS	Operating System	8

PCG	Pairwise connectivity graph	29
POG	Provisioning Order Graph	35
PaaS	Platform as a Service	7
REST	Representational State Transfer	1
SLA	Service Level Agreement	10
SSH	Secure Shell	13
SaaS	Software as a Service	7
TCP	Transmission Control Protocol	13
TOSCA	Topology and Orchestration Specification for Cloud Applications	34
UML	Unified Modeling Language	16
URI	Uniform Resource Identifier	15
URL	Uniform Resource Locator	14
VM	Virtual Machine	9
XMI	XML Metadata Interchange	16
XML	Extensible Markup Language	17
XSD	XML Schema	17
YAML	YAML Ain't Markup Language	71

List of Figures

List of Tables

List of Listings