Inferring Models from Cloud APIs and Reasoning over Them: A Tooled and Formal Approach

P H D T H E S I S

to obtain the title of

PhD of Science

Specialty : Computer Science

Defended on Friday, December 21, 2018 by

Stéphanie Challita

prepared at Inria Lille-Nord Europe, Spirals Team

Thesis committee:

Supervisor:	Philippe Merle	-	Inria (Lille)
Reviewers:	Benoit Combemale	-	University of Toulouse & Inria (Rennes)
	Christian Perez	-	Inria (Lyon)
Examiner:	Hélène Coullon	-	IMT Atlantique (Nantes)
Chair:	Laetitia Jourdan	-	University of Lille
Invited:	Faiez Zalila	-	Inria (Lille)

"Everything you can imagine is real."
–Pablo Picasso

To my parents for their constant support and endless sacrifices.
To Benjamin for his unlimited patience and love.

Acknowledgments

PhD is the biggest achievement but also the most challenging experience in my life, so far. Therefore, I would like to express my utmost gratitude to the people who helped me during this journey.

Foremost, I am truly grateful to my supervisor, **Philippe Merle** for many reasons. Thank you for taking my application for this thesis into consideration three years ago and for believing that I am a perfect fit for the job since our very first interview. Thank you for your guidance, which taught me the ropes of research, and for plenty of brilliant ideas, which were an inspiration for me. Thank you for helping me hone my skills and pushing me forward to be the best version of myself. I was determined to succeed to be worthy of the trust you placed in me. Our relationship made of taste for research, professionalism and kindness meant a lot to me. I highly admire your passion for your work, and I sincerely believe that you are an excellent researcher and a genuine person. And as I told you once before, I could not have imagined having a better mentor during my PhD. Thank you from the depths of my heart!

Next, I would like to thank the members of my thesis committee for devoting their time to read the manuscript and for their constructive feedback. **Benoit Combemale** and **Christian Perez**, thank you for accepting to review my manuscript. I would also like to thank **Hélène Coullon** for accepting to be part of my committee and **Laetitia Jourdan** for accepting to chair it.

Further gratitude is due to the members of the Spirals team at Inria research center, who I met since October 2015. Actually, during the last three years, I had the chance to be part of Spirals and it was a pleasure meeting many wonderful people there. Everyone was friendly and open for discussions, which made my stay extremely pleasant. I sincerely thank the team leader, **Lionel Seinturier**, for his effective direction, for providing a very motivating environment for preparing PhDs and for empowering my ambitions and helping me achieving them. I acknowledge your support and the support of **Laurence Duchien** when I came to you with my proposition to apply for the *L'Oréal-UNESCO For Women In Science* award. Also, thanks to Laurence for being a role model for many young female researchers like me and for your sincere advises for my career. I will always remember them. I would like to thank **Walter Rudametkin**, not only for your insightful comments, but also for your everyday friendship, for giving me access to your precious media server and for going out and drinking beers together. Besides, thank you and **Marcia** for receiving me in *"Chez Rudametkin"* and for those unforgettable tacos. I confirm that the 5 star on Google is well-deserved! ;) I salute **Clément Quinton** for his ambition and love of life. I enjoyed swimming with you on Fridays. Thank you as well for sharing with me your experience of becoming an associate professor. My warm wishes to you and your beautiful family. Thanks to **Simon Bliudze** with whom I shared the office for the last year. Thanks for working late so often, it helped me keeping focused and feeling well-surrounded. :) Also, thanks for giving me valuable feedback and propositions regarding my work, at each time I asked you. I hope we will work together sometime soon since many ideas emerged from these inspiring discussions. I would also like to thank my two former office mates, **Christophe Gourdin** and **Gustavo Sousa**. Thanks to Christophe for teaching me some "Chti"

language and for technical support when I started the implementation work in the OCCIware project. Here's to the prosperity of your startup! Thanks to Gustavo for the tips and recommendations when I first arrived to Lille and for the enriching conversations in the initial stage of my PhD. You were the first friend I made in the team and in Lille in general. Big thanks to **Faiez Zalila** for attending and efficiently participating to the weekly meeting with Philippe and me and for being the technical leader in the OCCIware project. I acknowledge your assistance with the modeling techniques, which allowed me to go further with my contributions. I would like to mention **Yahya Al-Dhuraibi** who started his PhD at the same time as me and under the supervision of Philippe too. I shared good moments with you when we attended the conference in Madeira and I admire your kindness, modesty and generosity. I wish you all the best for your future, you deserve it! Thanks to **Maxime Colmant** for helping me preparing my courses when I started teaching at the University of Lille.

I thank the OCCIware French project and the "Hauts-de-France" regional council for providing scholarships and appropriate facilities to pursue my doctoral studies. I also thank L'Oréal foundation for awarding me and providing research grants.

Thanks to my friends in Lille, **Tonie** and **Jad**. Tonie, I am so happy that I met you. I've always had fun with you and I really enjoyed our little tradition of Saturday lunch, although I missed some Saturdays because I needed to work. Jad, it was great news for me when I knew that you will be preparing a PhD also at the University of Lille, after we graduated together from the Antonine University. Thanks for the "Reeflex" evenings, for bringing me souvenirs when you visited a new city and for the catch up over coffee when you were at Inria. Who knows, maybe we will be colleagues again one more time!

Thanks to my cousin **Yara** and to my friends in Lebanon, **Alain**, **Alfred**, **Chantal** and **Rami** for always being there through WhatsApp, for your sense of humor and for the amazing outings at each time I visited Lebanon. You boosted my energy to reach this end.

I spare a moved thought for my guardian angel, my grandmother **Farida** who raised me during my early childhood and who left years ago. I wish that you were here with me and I hope that you are proud of me.

Big thanks to my dear parents, **Joseph** and **Rita**, who gave me the best of education and trusted my plan when I decided to move to France. Even from far away, I always felt your support. Dad, you taught me to aim high and I would not be who I am today without you. Mum, you are a perfect example of devotion and strength, I learn from you a lot and on daily basis.

Last but not least, I want to thank with great affection, my handsome fiancé, **Benjamin Danglot**, for being my backbone and my everyday bundle of happiness for the last two years. You suffered with me the side effects of preparing a thesis. Thank you for everything you do to help my dreams come true and for loving me unconditionally. I feel so lucky to have you by my side "habibi".

Stéphanie Challita
Villeneuve d'Ascq, France
October 12, 2018

Abstract

In recent years, multi-cloud computing which aims to combine different offerings or migrate applications between different cloud providers, has become a major trend. Multi-clouds improve the performance and costs of cloud applications, and ensure their resiliency in case of outages. But with the advent of cloud computing, different cloud providers with heterogeneous cloud services *(compute, storage, network, applications, etc.)* and Application Programming Interfaces (APIs) have emerged. This heterogeneity complicates the implementation of an interoperable multi-cloud system. Several multi-cloud interoperability solutions have been developed to address this challenge. Among these solutions, Model-Driven Engineering (MDE) has proven to be quite advantageous and is the mostly adopted methodology to rise in abstraction and mask the heterogeneity of the cloud. However, most of the existing MDE solutions for the cloud remain focused on only designing the cloud without automating the deployment and management aspects, and do not cover all cloud services. Moreover, MDE solutions are not always representative of the cloud APIs and lack of formalization.

To address these shortcomings, I present in this thesis an approach based on Open Cloud Computing Interface (OCCI) standard, MDE and formal methods. OCCI is the only community-based and open recommendation standard that describes every kind of cloud resources. MDE is used to design, validate, generate and supervise cloud resources. Formal methods are used to effectively reason on the structure and behaviour of the encoded cloud resources, by using a model checker verifying their properties. This research takes place in the context of the OCCIware project, which provides OCCIware Studio, the first model-driven tool chain for OCCI. It is coupled with OCCIware Runtime, the first generic runtime for OCCI artifacts targeting all the cloud service models (IaaS, PaaS, and SaaS).

In this dissertation, I provide two major contributions implemented on top of the OCCIware approach. First, I propose an approach based on reverse-engineering to extract knowledge from the ambiguous textual documentation of cloud APIs and to enhance its representation using MDE techniques. This approach is applied to Google Cloud Platform (GCP), where I provide GCP Model, a precise model-driven specification for GCP. GCP Model is automatically inferred from GCP textual documentation, conforms to the OCCIware Metamodel and is implemented within OCCIware Studio. It allows one to perform qualitative and quantitative analysis of the GCP documentation. Second, I propose in particular the fclouds framework to achieve semantic interoperability in multi-clouds, *i.e.,* to identify the common concepts between cloud APIs and to reason over them. The fclouds language is a formalization of OCCI concepts and operational semantics in Alloy formal specification language. To demonstrate the effectiveness of the fclouds language, I formally specify thirteen case studies and verify their properties. Then, thanks to formal transformation rules and equivalence properties, I draw a precise alignment between my case studies, which promotes semantic interoperability in multi-clouds.

Keywords: Cloud Computing, Multi-Clouds, Open Cloud Computing Interface (OCCI), Model-Driven Engineering (MDE), Reverse-Engineering, Google Cloud Platform (GCP), Formal Methods, Formal Verification, Alloy, Interoperability

Résumé

Ces dernières années, l'informatique multi-nuages, qui vise à combiner différentes offres ou à migrer des applications entre différents fournisseurs de services en nuage, est devenue une tendance majeure. Les multi-nuages améliorent les performances et les coûts des applications hébergées dans les nuages et garantissent leur résilience en cas de panne. Mais avec l'avènement de l'informatique en nuage, différents fournisseurs offrant des services en nuage *(calcul, stockage, réseau, applications, etc.)* et des interfaces de programmation d'applications (APIs) hétérogènes sont apparus. Cette hétérogénéité complique la mise en oeuvre d'un système de multi-nuages interopérable. Plusieurs solutions pour l'interopérabilité de multi-nuages ont été développées pour relever ce défi. Parmi ces solutions, l'Ingénierie Dirigée par les Modèles (IDM) s'est révélée très avantageuse et constitue la méthodologie la plus largement adoptée pour monter en abstraction et masquer l'hétérogénéité du nuage. Cependant, la plupart des solutions IDM existantes pour le l'informatique en nuage restent concentrées sur la conception des nuages sans automatiser les aspects de déploiement et de gestion, et ne couvrent pas tous les services en nuage. De plus, les solutions IDM ne sont pas toujours représentatives des APIs de nuages et manquent de formalisation.

Pour remédier à ces limitations, je présente dans cette thèse une approche basée sur le standard Open Cloud Computing Interface (OCCI), les approches IDM et les méthodes formelles. OCCI est le seul standard ouvert qui décrit tout type de ressources de nuages. L'IDM est utilisée pour concevoir, valider, générer et superviser des ressources de nuage. Les méthodes formelles sont utilisées pour raisonner efficacement sur la structure et le comportement des ressources de nuage encodées, à l'aide d'un vérificateur de modèle analysant leurs propriétés. Cette recherche a lieu dans le contexte du projet OCCIware, qui fournit OCCIware Studio, la première chaîne d'outils pilotée par les modèles pour OCCI. OCCIware Studio est associé à OCCIware Runtime, le premier environnement d'exécution générique pour les artefacts OCCI ciblant tous les modèles de service de nuages (IaaS, PaaS et SaaS).

Dans cette thèse, je fournis en particulier deux contributions majeures qui sont mises en oeuvre en se basant sur l'approche OCCIware. Premièrement, je propose une approche basée sur la rétro-ingénierie pour extraire des connaissances des documentations textuelles ambiguës des APIs de nuages et améliorer leur représentation à l'aide des techniques IDM. Cette approche est appliquée à Google Cloud Platform (GCP), où je propose GCP Model, une spécification précise et basée sur les modèles pour GCP. GCP Model est automatiquement déduit de la documentation textuelle de GCP, est conforme à OCCIware Metamodel et est implémenté dans OCCIware Studio. Il permet d'effectuer des analyses qualitatives et quantitatives de la documentation de GCP. Deuxièmement, je propose le cadre fclouds pour assurer une interopérabilité sémantique entre plusieurs nuages, *i.e.,* pour identifier les concepts communs entre les APIs de nuages et raisonner dessus. Le langage fclouds est une formalisation des concepts et de la sémantique opérationnelle d'OCCI en employant le langage de spécification formel Alloy. Pour démontrer l'efficacité du langage fclouds, je spécifie formellement treize APIs et en vérifie les propriétés. Ensuite, grâce aux règles de transformation formelles et aux propriétés

d'équivalence, je peux tracer un alignement précis entre mes études de cas, ce qui favorise l'interopérabilité sémantique dans un système de multi-nuages.

Mots-clés: Nuage informatique, Multi-nuages, Open Cloud Computing Interface (OCCI), Ingénierie dirigée par les modèles (IDM), Rétro-ingénierie, Google Cloud Platform (GCP), Méthodes formelles, Vérification formelle, Alloy, Interopérabilité

Contents

List of Figures

List of Tables

Part I

Preface

The first part of this manuscript introduces the scope, motivation and goals of this thesis.

Chapter 1

Introduction

Contents

Cloud computing, which is gaining the attention of both academia and industry for the last decade, was not born from scratch but is a normal evolution of many domains such as distributed computing, grid computing and service-oriented computing. Many computing researchers and practitioners have attempted to define cloud computing in various ways. I give below the most commonly used definitions:

- *"A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers."* [Buyya 2009].

- *"Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services."* [Armbrust 2010].

- *"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."* [Mell 2011].

To summarize, cloud computing enables computing resources, software, or data to be delivered as a *service* and *on-demand* through the Internet, so these resources have become *cheaper*, more *powerful* and more *available* than ever before.

More precisely, cloud computing is a model composed of three *deployment models*, three *service models* and three *delivery models*.

Deployment models. Cloud environments can have different access types, that are called **deployment models**. The latter can be *private*, *public* or *hybrid*. Private cloud environments are owned by a single organization and they can be built by relying on technologies like OpenStack [opea], whereas public cloud environments are owned by a third-party cloud provider such as *Amazon Web Services* (AWS) and *Google Cloud Platform* (GCP). Usually, a cloud developer requires using private clouds for testing a cloud application, then migrating to public clouds so the application can be publicly accessed by cloud users. And sometimes, the cloud developer requires using a *hybrid* cloud, *i.e.,* that comprises public and private cloud environments. It allows the cloud developer to make his/her application publicly accessed by hosting its Web server on a public cloud, and to privately store sensitive data by keeping his/her application database in a private cloud.

Service models. Cloud providers offer functionalities as services at different layers of the cloud stack, *i.e.,* **service models**: *Infrastructure-as-a-Service* (IaaS), *Platform-as-a-Service* (PaaS) and *Software-as-a-Service* (SaaS).

- *IaaS*: where the capability provided to the IaaS user is to provision virtual machines and to configure the infrastructure concerns: processing, storage, networking, and other computing resources.
- *PaaS*: where the capability provided to the PaaS user is to deploy an application and it is limited to the database(s), application server(s), compilation tools, libraries, etc.
- *SaaS*: where the capability provided to the SaaS user is to manage applications running on a cloud Infrastructure and/or Platform, and accessible through a web browser. A SaaS should rely on the principle of multi-tenancy, where

multiple independent instances of one or multiple applications operate in a shared environment.

Delivery models. Cloud resources can be provisioned from either a *single cloud* or *multiple clouds*. This is known as the **delivery model**. In this section, I describe each of the existing delivery models and I highlight the advantages of a multi-cloud delivery model, **which interests us in this dissertation**.

- *Single cloud*: where cloud applications are limited to be deployed on a single cloud among others, *i.e.,* to benefit from services of only one cloud provider at a time.

However, several cloud outages have taken place in the past [Ko 2013], which prove that the sentence "*do not place all your eggs in one basket*" is equally applicable to the cloud ecosystem. Therefore, some cloud application may require to exploit services from *multiple cloud* environments, at the infrastructure, platform, and software layers. In this case, the cloud developer should perfectly manage to deal with dependencies and to ensure separation of concerns. As Petcu explained in [Petcu 2013], there are two basic delivery models in multiple cloud systems: *Federated Clouds* and *Multi-Clouds*. Petcu has drawn a clear positioning of multi-clouds versus other cloud models. I summarize it as follows:

- *Federated clouds*: where the cloud providers are in agreement with each others to enhance the service offered to their consumers, *e.g., European Grid Infrastructure Federated Clouds* (EGI FC) which is a federation of private clouds.

- *Multi-clouds*: where the application provisions multiple cloud varying services, without a prior agreement with and between the cloud providers, but with a third party building a unique entry point for multiple clouds. This strategy has been adopted in the cloud computing industry since a while in order to improve disaster recovery and geo-presence, to use unique cloud services from different providers as they are needed, and to ensure unlimited scalability of cloud applications, as explained in [Petcu 2013].

The remainder of this introductory chapter is organized as follows. Section 1.1 presents the context of this thesis. In Section 1.2, I identify the problems that motivate this research. Section 1.3 introduces the research questions that this dissertation aims to answer. Next, in Section 1.4, I present the main goals of this thesis. Section 1.5 presents the vision of my research. Section 1.6 introduces my proposed solution. In Section 1.7, I summarize the structure of this dissertation. Finally, in Section 1.8, I detail the publications derived from my research.

1.1 Thesis Context

This thesis is supported by both the OCCIware [occc] research and development project funded by the French *Programme d'Investissements d'Avenir* (PIA), and the Hauts-de-France Regional Council. The OCCIware project promotes the OCCI standard to address the lack of unified cloud computing standard and facilitate the development of services. Therefore, the works carried on in this thesis are built on the OCCI standard by using the OCCIware approach.

This thesis is produced in the Spirals team. Spirals is a joint project-team between Inria Lille-Nord Europe research center and the University of Lille. Spirals currently consists of eight permanent members and about twenty-five non-permanent members. The research areas of Spirals are distributed systems and software engineering. The research areas of this thesis are particularly multi-cloud computing, *Model Driven Engineering* (MDE) and formal methods.

1.2 Problem Statement

Due to the emergence of numerous cloud providers and their heterogeneity, provisioning cloud services is not a straightforward task. I state the main problem addressed by this dissertation as follows:

> The cloud shows several favorable features like elasticity and pay-as-you-go. In order to take advantage of these features, the cloud computing market counts today variety of cloud providers like Amazon, Google, Microsoft, etc. Cloud providers offer varying infrastructure, platform or software services. Even at the same service layer, cloud providers use heterogeneous terms, concepts, and features, which usually are not aligned with those of competing providers. These semantic differences are critical in cloud computing as they make migrating an application across providers a very complicated and costly task. In addition, cloud providers give access to their resources through heterogeneous *Cloud Resource Management* (CRM)-*Application Programming Interface* (API)s. The management of a potentially large number of cloud services with heterogeneous CRM-APIs is a challenge, because of incompatibility between the different APIs. Worse still, the semantics of these CRM-APIs is informally described in English prose in their documentation available at the provider's website. Therefore, it is usually impossible to understand the behaviour of a cloud when the developer requests a virtual machine for example. For the above concerns, the dependency

to a single cloud provider is promoted, the multi-cloud environment is prevented and the migration from one cloud to another becomes a very complicated task.

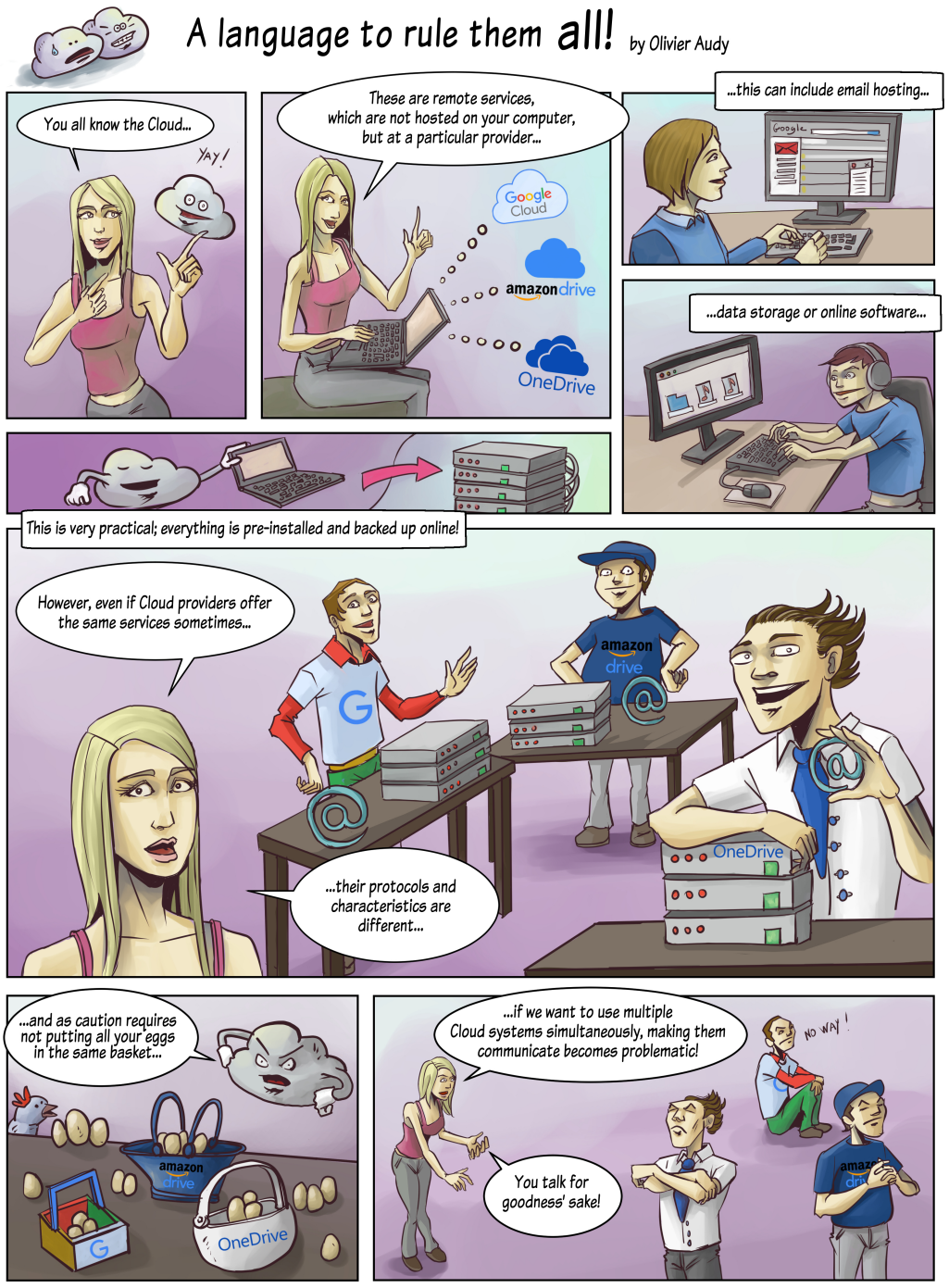


Figure 1.1: My Thesis in Comics - Part 1.

The comic strip in Figure 1.1 illustrates the problem above, *i.e.,* the heterogeneity in a multi-cloud context that leads to a lack of interoperability across providers. I credit the work for designing the amazing comics of my thesis to Olivier Audy.

More specifically, cloud stakeholders face the following challenges.

Heterogeneous service models. Cloud providers offer different services that belong to the Infrastructure (IaaS), Platform (PaaS) or Software (SaaS) layers. We use the abbreviation *Everything as a Service* (XaaS) to refer to all categories. These categories consist in the "Service Model". Service models contain highly heterogeneous cloud resources, which make difficult the overall management of a computer system from infrastructure to application resources.

Heterogeneous CRM-APIs. Cloud services are often exposed as Web services, which follow the industry standards such as *Web Services Description Language* (WSDL)[1], *Simple Object Access Protocol* (SOAP)[2] and *Universal Description, Discovery and Integration* (UDDI)[3] [Paraiso 2012]. They frequently rely on *REpresentational State Transfer* (REST)ful [Fielding 2000] APIs that provide programmatic access to the resources offered by a cloud provider through *Create, Retrieve, Update and Delete* (CRUD) operations. For example, the Amazon cloud services are accessible via a SOAP API, whereas other clouds are based on a REST API, which leads to an incompatibility between these two different APIs.

Semantic differences. The semantics refers to the description of a cloud service by its provider. These descriptions are heterogeneous because a cloud provider employs concepts, which usually do not directly map to those of a competing provider. In fact, even if cloud providers offer the same service, the latter may have different names, characteristics and functionalities. For instance, GCP refers to its compute service as *"instance"*, whereas DigitalOcean calls it *"droplet"*. These semantic differences are critical in cloud computing as they make migrating an application across providers a very complicated and costly task.

Lack of verification. Cloud solutions provide services, libraries or model-driven tools to provision cloud resources. However, once provisioned, the deploy-

[1]WSDL is an *Extensible Markup Language* (XML)-based language that is used for describing the functionality offered by a Web service.

[2]SOAP is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks.

[3]UDDI is a platform-independent, XML-based registry by which businesses worldwide can list themselves on the Internet, and a mechanism to register and locate Web service applications.

ment of the applications can face several problems such as misconfiguration of links between resources, lack of resources on the hosts in which the applications are deployed, human errors, etc. The only way to be sure that the cloud configurations will run or fail is to deploy them on the target executing environment. Moreover, there is no way to verify that deployed configurations are conform with those desired. The lack of verification tool becomes quickly painful and expensive when the deployment task is repeated several times.

Lack of formalization. The semantics of cloud APIs is informally described in their documentation available at provider websites within English prose. It is then difficult to understand the behaviour of a cloud when the developer requests a virtual machine for example. Moreover, the cloud solutions are numerous and also lack of precise documentation, which complicate their understanding and comparison. This lack of formalization hinders the understanding of the cloud APIs and solutions, thus complicates the provisioning process and also misleads the alignment and comparison between cloud offerings.

Vendor lock-in. It is recognized as one of the greatest challenges to cloud adoption where cloud clients are locked-in to a specific cloud provider due to the heterogeneity. Therefore, vendor lock-in is a serious result of all the problems that I discussed above. This problem hinders the complete exploitation of the full capabilities of cloud computing since it prevents two main intended aspects: portability and interoperability, which are closely related terms and may often be confused. Cloud interoperability is the integration between several cloud offerings, whereas portability is the ability to move applications between different cloud providers. Cohen clarifies in [Cohen 2009] the similarities and the differences among these terms in an attempt to exemplify and differentiate them.

The work presented in this thesis aims to alleviate the challenges presented above.

1.3 Research Questions

More specifically, this thesis aims to answer the following three research questions (RQs):

- **RQ#1:** *Is it possible to have a solution that allows to represent all kinds of cloud resources despite their heterogeneity, and a complete framework for managing them?*

  - How to **design** the cloud developer needs at a high-level of abstraction?
  - How to **verify** the cloud structural and behavioral properties before any concrete deployments?
  - How to **deploy** and **manage** cloud configurations?

- **RQ#2:** *Is it possible to automatically extract precise models from cloud APIs and to synchronize them with the cloud evolution?*
  - How to **provide an accurate description** for a cloud API?
  - How to **correct** the existing drawbacks in a cloud API documentation?
  - How to **analyze** a cloud API documentation?

- **RQ#3:** *Is it possible to reason on cloud APIs and identify their similarities and differences?*
  - How to **better understand** cloud solutions?
  - How to **make sure** that a cloud solution reflects the desired behaviour?
  - How to **ensure an accurate migration** from a cloud solution to another?

These research questions are explored in next sections.

1.4 Thesis Goals

The objective of this thesis topic was to propose the first formal framework to rigorously handle cloud resources. This framework allows to model, analyze, design, deploy, manage every kind of cloud resources, and to reason over them. This framework is based on the *Open Cloud Computing Interface* (OCCI) [Edmonds 2012, occa] of the *Open Grid Forum* (OGF) recommendation. The tooling of this framework relies on MDE techniques, particularly the *Eclipse Modeling Framework* (EMF) and the Models@run.time approach. The formalization of this framework relies on formal specification languages such as the Alloy [Jackson 2012] language developed by Professor Daniel Jackson from the *Massachusetts Institute of Technology* (MIT). To achieve this objective, I decompose it into the following goals.

Regarding *RQ#1*, this thesis aims to provide mechanisms to interact with heterogeneous cloud environments. These mechanisms allow one to model, analyze, design, deploy and manage every kind of cloud resources.

Regarding *RQ#2*, this thesis aims to propose mechanisms to automatically build a cloud model from the corresponding cloud API. These mechanisms rely on reverse-engineering techniques. They consist in extracting knowledge from a cloud API

documentation in order to infer the concepts to be defined in the cloud model so it correctly reflects the real cloud API. Also, these mechanisms allow to automatically update the cloud model in case changes occurred to the cloud API.

Regarding *RQ#3*, this thesis intends to provide mechanisms to draw a precise alignment between cloud APIs. For such purpose, I exploit formal languages to rigorously and precisely encode cloud concepts and operations and to reason over them.

1.5 Thesis Vision

We discussed earlier in this chapter that cloud computing encompasses heterogeneous cloud providers. As illustrated in Figure 1.2, this thesis takes advantage of model-based and formal approaches in order to rise in abstraction from the heterogeneous real-world and promote multi-cloud computing. The approaches presented in this thesis are represented in blue. The OCCIware model-driven approach is discussed in the background part of this thesis, and the FCLOUDS formal approach is discussed in the contributions part of this thesis. More precisely, this thesis aims at inferring models from multi-clouds using the OCCIware platform, and then formally reasoning on these models using the FCLOUDS framework.

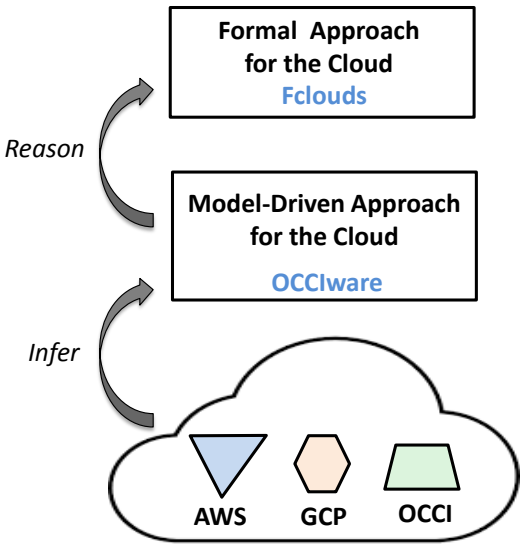


Figure 1.2: Thesis Vision.

1.6 Proposed Solution

In this section, I provide an overview of the contributions described in this dissertation. As stated before, the goal of my thesis is to provide approaches, languages, tools for inferring and enhancing the knowledge of cloud APIs, precisely representing this knowledge and efficiently reasoning over it. The main contributions of our work are summarized as follows:

Precise models for cloud APIs. My first contribution is to enhance the knowledge representation in cloud APIs by automatically inferring a precise model from the cloud textual documentation. My approach is applied on a major cloud provider, GCP. To address the drawbacks of GCP textual documentation, I propose a precise model that describes GCP API. It consists in a precise specification that describes without ambiguity the knowledge and activities in GCP to avoid confusion and misunderstandings. This model-driven specification, called GCP Model, is automatically inferred from the textual documentation of GCP. GCP Model conforms to the OCCIware Metamodel and is implemented within the open source model-driven Eclipse-based OCCIware tool chain. Thanks to our GCP Model, I offer corrections to the drawbacks I identified in GCP textual documentation. Also, I analyze GCP by drawing conclusions regarding their documentation and quantifying their services.

The fclouds framework. I provide as second contribution fclouds, the first formal framework for semantic interoperability between cloud APIs. By semantic interoperability I mean to identify the similarities and differences between cloud APIs concepts and to mathematically reason over them. fclouds contains a catalog of cloud APIs that are precisely described. It will help the cloud customer to understand the behaviour of the cloud API but also how to migrate from one API to another, thus to promote semantic interoperability. To implement the formal language that will encode all the APIs of our fclouds framework, I advocate the use of formal methods, *i.e.,* techniques based on mathematical notations. They will allow us to rigorously encode cloud concepts and behaviour, validate cloud properties and finally define formal transformation rules between cloud concepts. I adopt the concepts of the OCCI common standard to define the formal language of the fclouds framework. I choose to formalize OCCI with Alloy [Jackson 2012], a lightweight promising formal specification language designed by Daniel Jackson from the MIT.

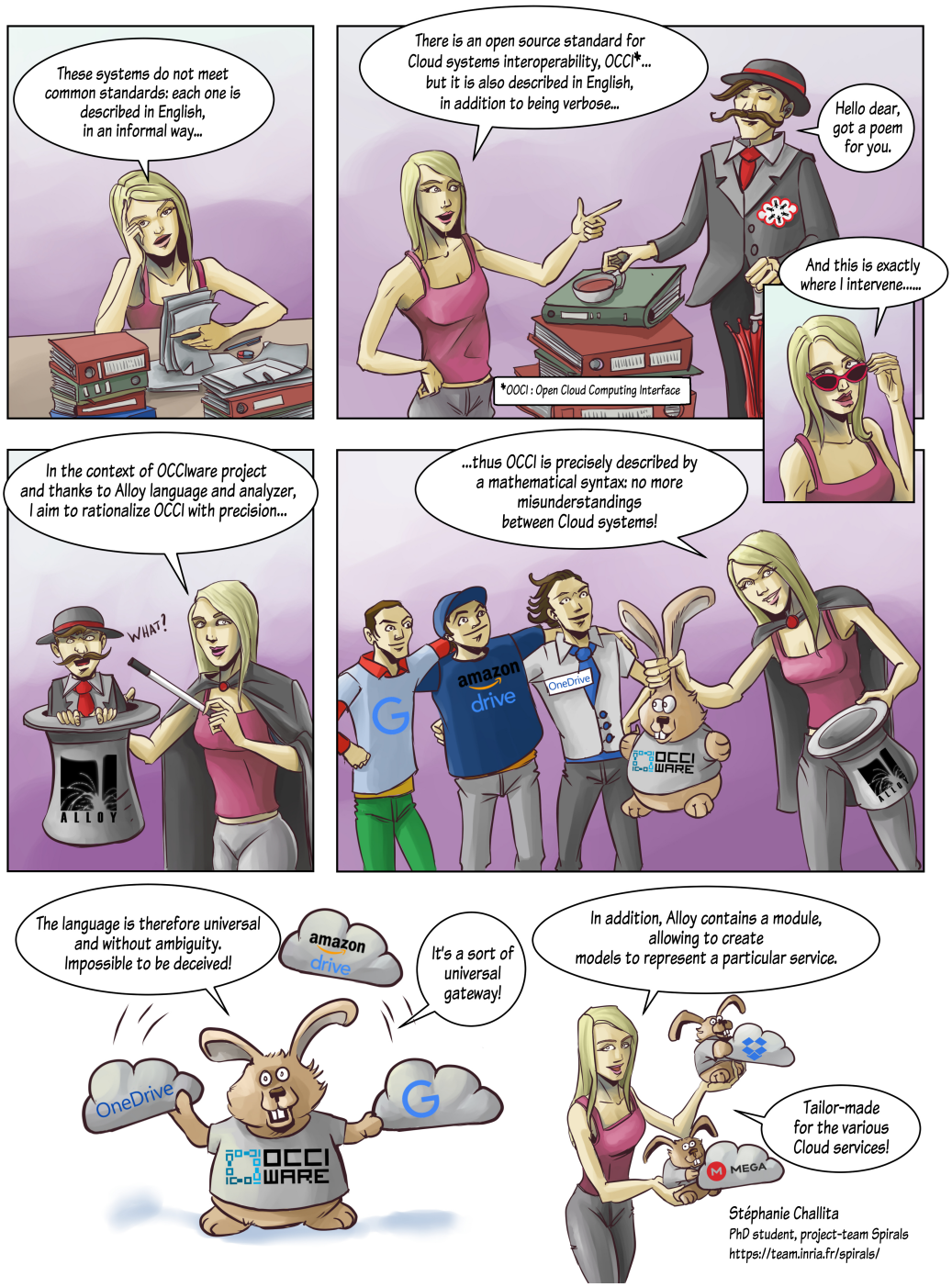


Figure 1.3: My Thesis in Comics - Part 2.

The comic strip in Figure 1.3 vulgarizes this contribution based on OCCI and Alloy formal language and its analyzer for precisely describing cloud APIs. It mainly

highlights how the formalization of OCCI in Alloy allows a standardization of the the various cloud services. Consequently, this formalization helps the developer to avoid the misunderstandings that result from the English documentations.

1.7 Dissertation Roadmap

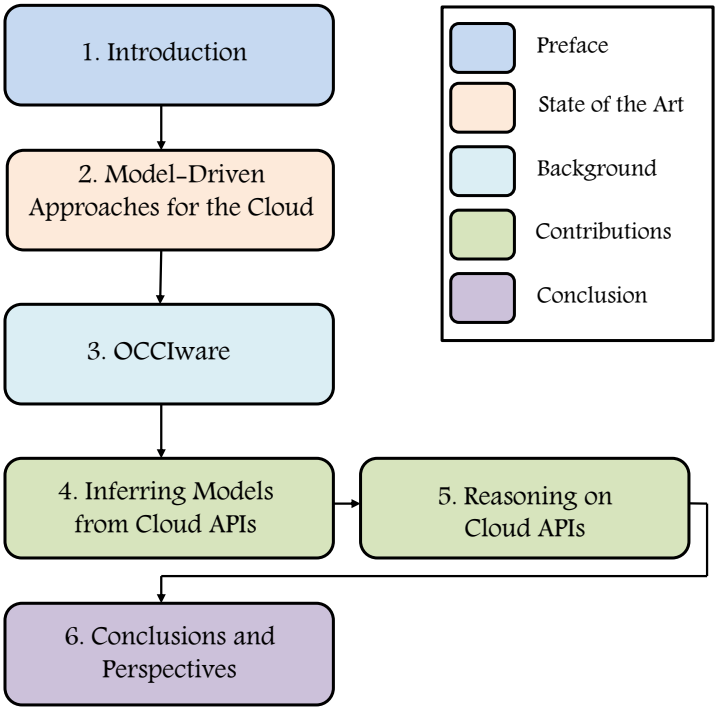


Figure 1.4: Thesis Outline.

This dissertation is divided in five parts and six chapters, as shown in Figure 1.4. While this introductory chapter is part of the first part, the second one encloses the State of the Art. In the third part, I present OCCIware, which is the model-driven environment on which I rely to implement my works. The fourth part presents the two contributions of this dissertation. Finally, the last part includes the conclusions and perspectives of this dissertation. Below, I present an overview of the chapters that compose the different parts.

Part II: State of the Art

Chapter 2: Model-Driven Approaches for the Cloud In this chapter, I present the approaches that are used in order to ensure multi-clouds, namely standards, services, libraries and models. I focus on model-based cloud solutions and I propose a taxonomy to provide a better understanding of the concerns in which our work takes place. I list and describe the most relevant related works in terms of our taxonomy criteria. Since our work presents a solution for multi-clouds, the idea of this chapter is to explore the existing solutions and their limitations.

Part III: Background

Chapter 3: Modeling, Verifying, Generating and Managing Cloud Resources with OCCIware. In this chapter, I present OCCIware, the project that supports this thesis and the paltform that I used to implement my contributions. OCCIware proposes to textually and graphically encode cloud APIs and cloud configurations via OCCIware Studio. The latter is a model-driven environment for OCCI standard, based on an Ecore metamodel. Then, OCCIware Studio is linked with OCCIware Runtime, an execution environment for OCCI artifacts. Therefore, from a designed and verified OCCI configuration, we can generate a deployment script via the CURL Generator tool. Later, these configurations can be managed at runtime via generated connectors deployed on OCCIware Runtime.

Part IV: Contributions

Chapter 4: Inferring Precise Models from Cloud APIs Textual Documentations. In this chapter, I present my approach for retrieving information from cloud APIs, improving their representation and discovering new knowledge from them. This approach is experimented by studying the textual documentation of GCP, one of the leaders in the cloud market. Then, I build a precise model for GCP, called the GCP Model. Thanks to this model, I study GCP API and provide corrections to the six drawbacks of its current informal documentation.

Chapter 5: Specifying Heterogeneous Cloud Resources and Reasoning over them with fclouds. In this chapter, I present my approach for formally specifying cloud APIs, called the fclouds framework. Based on Alloy formal language and the OCCI standard, I define a formal language for cloud computing that relies on first-order logic paradigm combined with relational algebra. Then, I show how having formal specifications of cloud solutions allow to check their behaviour,

detect their inconsistencies, and remove their ambiguity to understand their similarities and promote their interoperability.

Part V: Conclusion

Chapter 6: Conclusions and Perspectives. In this chapter, I conclude the work presented in this dissertation. I discuss some limitations that motivate new ideas and future directions as short-term and long-term perspectives.

1.8 Publications

The contributions derived from this thesis have been published in international peer-review conferences. In this section, I detail all the publications resulted from my research for the last three years. These publications are ordered by year of publication.

1.8.1 International Conferences

- **Stéphanie Challita**, Faiez Zalila, and Philippe Merle. *"Specifying Semantic Interoperability between Heterogeneous Cloud Resources with the FCLOUDS Formal Language."* 11th IEEE International Conference on Cloud Computing (CLOUD), San Francisco, California, USA, 2018, p. 367-374 [Challita 2018b] (CORE rank B, acceptance rate: 20%).

- **Stéphanie Challita**, Faiez Zalila, Christophe Gourdin, and Philippe Merle. *"A Precise Model for Google Cloud Platform."* 6th IEEE International Conference on Cloud Engineering (IC2E), Orlando, Florida, USA, 2018, p. 177-183 [Challita 2018a] (acceptance rate: 19%).

- Fabian Korte, **Stéphanie Challita**, Faiez Zalila, Philippe Merle, and Jens Grabowski. *"Model-Driven Configuration Management of Cloud Applications with OCCI."* 8th International Conference on Cloud Computing and Services Science (CLOSER), Funchal, Madeira, Portugal, 2018, p. 100-111 [Korte 2018] (acceptance rate: 22%).

- Faiez Zalila, **Stéphanie Challita**, and Philippe Merle. *"A Model-Driven Tool Chain for OCCI."* 25th International Conference on Cooperative Information Systems (CoopIS), Rhodes, Greece, 2017, p. 389-409 [Zalila 2017a] (CORE rank A, acceptance rate: 20%).

- **Stéphanie Challita**, Fawaz Paraiso, and Philippe Merle. *"Towards Formal-based Semantic Interoperability in Multi-Clouds: The FCLOUDS Framework."* 10th IEEE International Conference on Cloud Computing (CLOUD), Honolulu, Hawaii, USA, 2017, p. 710-713 [Challita 2017b] (CORE rang B, acceptance rate: 18%).

- **Stéphanie Challita**, Fawaz Paraiso, and Philippe Merle. *"A Study of Virtual Machine Placement Optimization in Data Centers."* 7th International Conference on Cloud Computing and Services Science (CLOSER), Porto, Portugal, 2017, p. 343-350 [Challita 2017a] (acceptance rate: 22.5%).

- Fawaz Paraiso, **Stéphanie Challita**, Yahya Al-Dhuraibi, and Philippe Merle. *"Model-driven Management of Docker Containers."* 9th IEEE International Conference on Cloud Computing (CLOUD), San Francisco, California, USA, 2016, p. 718-725 [Paraiso 2016] (CORE rank B, acceptance rate: 15%).

1.8.2 International Journal

In addition, one journal article is under submission:

- Faiez Zalila, **Stéphanie Challita**, and Philippe Merle. *"Model-Driven Cloud Resource Management with OCCIware."* Future Generation Computer Systems (FGCS), 2018 [Zalila 2018] (Impact factor 4.639).

1.9 Awards

During this thesis, I was selected as an ambassador of the French fellowship **L'ORÉAL-UNESCO FOR WOMEN IN SCIENCE** 2018. Among 900 applications, 20 female PhD candidates and 10 female postdocs were granted this award.

Also, I received two student travel grants from:

- IEEE CLOUD 2017 conference that took place in Honolulu, Hawaii, USA, and,

- FormaliSE 2018 conference (co-located with International Conferences on Software Engineering (ICSE)) that took place in Gothenburg, Sweden.

Part II

State of the Art

In this part, I review approaches related to cloud computing and I classify the existing models for the cloud.

Chapter 2

Model-Driven Approaches for the Cloud

Contents

Today, the plethora of cloud providers and their heterogeneity hinder their interoperability. Therefore, many solutions have emerged to add abstraction between the cloud providers and provide mechanisms to automate the provisioning of services from multi-clouds. Among these solutions, MDE has received a significant attention in the development of software for cloud computing. MDE is a software development methodology that allows software developers to design the software concerns at a high level of abstraction, hide different implementation details, reduce complexity, ease reuse, and thus improve software quality. Since 2010, several MDE approaches for the cloud have emerged. However, each one targeted a particular problem and resolved it within an ad-hoc manner. In fact, some years after the emergence of the cloud computing, several works [Bruneliere 2010, Baryannis 2013] were interested to the synergy between the cloud and MDE. However, no work gave a consensus on the set of models, languages, model transformations and software processes for the model-driven development of the cloud applications. In this chapter, I present the first detailed study about the use of MDE for the cloud.

This chapter is structured as follows. Section 2.1 recalls the concept of "Cloudware engineering" and classifies the existing Cloudware engineering solutions into three categories that I call *spaces*. Section 2.2 describes a taxonomy for explaining *Model-Driven Approaches for the Cloud* (MDAC). Therefore, I list the different identified usages for MDAC, during the different phases of building an application for/in the cloud. Afterwards, I discuss what an eventual MDAC can contain as concepts to reply to the different usages needs. Then, I discuss how these concepts can be encoded and what are the different possible approaches to do that. Section 2.3 details twenty-two existing MDAC. Then, I discuss these solutions according to my taxonomy criteria. Section 2.4 identifies some limitations in the existing approaches. Finally, Section 2.5 concludes the chapter.

2.1 Multi-Cloud Ecosystem

The emergence of the virtualization and the cloud computing has fostered the deployment of the software on the cloud. This specific kind of software is called Cloudware. The Cloudware engineering requires us to update the classical software engineering approaches to be adapted to the cloud computing specificities such as elasticity and portability. To promote interoperability between clouds, *i.e.,* to enable multi-clouds, the Cloudware engineering market counts numerous solutions at different levels of abstraction, traditionally called service layers. From my point of view, these solutions can be classified into three spaces, as shown in Figure 2.1. I identify the *Provider Space* that offers solutions for the cloud provider, the *Programming Space* that offers solutions for the cloud developer and the *Modeling Space* for the cloud architect. Multi-cloud solutions, whether they belong the *Provider Space*, *Programming Space* or *Modeling Space*, follow sometimes the current emerging cloud standards.

Standards. Cloud standards result from collective agreements and aim at providing some concepts, characteristics and implementations to be commonly used by cloud providers. Among cloud standards, I identify:

- *Cloud Application Management for Platforms* (CAMP) [Carlson 2012, cam]: the *Organization for the Advancement of Structured Information Standards* (OASIS)'s CAMP standard targets the deployment of cloud applications on top of PaaS resources.
- *Cloud Data Management Interface* (CDMI) [cdm]: defines a RESTful interface that allows cloud applications and users to retrieve and perform operations on the data from the cloud.

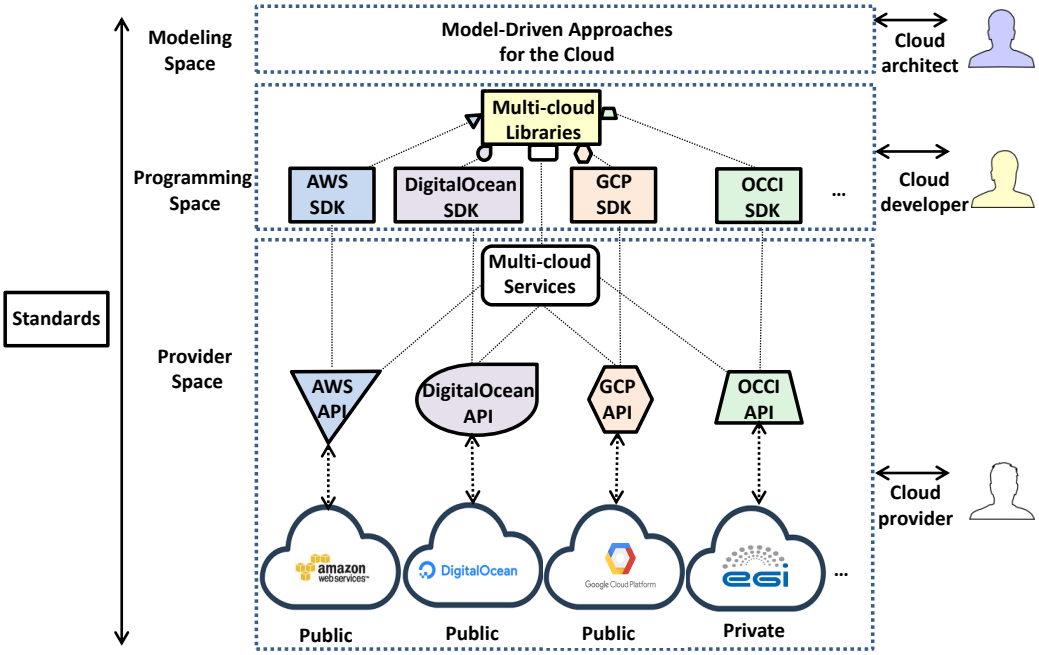


Figure 2.1: Multi-Cloud Ecosystem.

- *Cloud Infrastructure Management Interface* (CIMI) [Davis 2012]: the *Distributed Management Task Force* (DMTF)'s CIMI standard defines a RESTful API for managing IaaS resources only.
- OCCI [Edmonds 2012, occa]: the OGF's OCCI proposes a generic resource-oriented model for describing and managing any kind of cloud resources, including IaaS, PaaS, and SaaS.
- *Open Virtualization Format* (OVF) [ovf]: the DMTF's OVF standard defines a packaging format for portable virtual machine images.
- *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [Binz 2012]: the OASIS's TOSCA defines a model to describe and package cloud application artifacts and to deploy them on IaaS and PaaS resources.

Discussion

Using standards for cloud computing is quite advantageous because they result of a collective agreement and they extract the key actions and characteristics of cloud providers. Also, being a standard means that several implementations have been successfully built using this standard. However, standards are usually specific for a particular cloud service model. Moreover, leading cloud providers have unfortunately no interest in adopting a standard API like the one offered by OCCI to ease interoperability with other clouds. Each of the cloud providers would rather have proprietary, closed source implementations with custom APIs. However, OCCI has proven its utility in several contexts. For example, the EGI FC [egi] is based on OCCI to ensure interoperability among twenty cloud providers and over three hundred data centers. Furthermore, OCCI attracts several cloud brokers such as CompatibleOne [Yangui 2014] that aims at ensuring seamless access to the heterogeneous resources of cloud providers. For these reasons among others, I propose in this thesis an OCCI-based approach for interoperability in a multi-cloud context.

In the following, I present the Cloudware spaces and highlight the problem at each space of the cloud ecosystem.

2.1.1 Provider Space

The cloud market counts today a plethora of cloud providers that, as shown in Table 2.1, are heterogeneous in terms of their deployment model, service model and management interface. This heterogeneity leads to vendor lock-in.

Table 2.1: Heterogeneity of Cloud Providers.

Cloud Provider	Deployment Model	Service Model	CRM-API
AWS [aws]	Public	IaaS & PaaS	REST & SOAP
DigitalOcean [dig]	Public	IaaS	REST
EGI FC [egi]	Private	IaaS	REST
FlexiScale [fle]	Public	IaaS	SOAP
GCP [gcp]	Public	IaaS & PaaS	REST
Microsoft Azure [azu]	Hybrid	IaaS	REST
Heroku [her]	Public	PaaS	REST
SalesForce [sal]	Public	SaaS	REST & SOAP
VMware [vmw]	Hybrid	IaaS	REST

Services. To address the providers' heterogeneity problem, the solution in this space would be a service that offers a unique interface to handle the heterogeneity of different APIs. A service is expected to intermediate the relationship between

the cloud providers and users to simplify the process of combining multiple cloud services. In this subsection, I survey different cloud services, whether they are commercial or open source.

- Aeolus [aeo]: is an open source European research project aiming at automating the deployment and reconfiguration of machine pools in the clouds.
- Aneka [Vecchiola 2009]: is a PaaS, that is commercialized by Manjrasoft [man], for building .NET applications and deploying them on either public or private clouds.
- CompatibleOne [Yangui 2014]: is an open source PaaS to automate application deployment on multiple providers. It is based on CDMI and OCCI standards.
- Kaavo [kaa]: is a commercial management interface for configuring and managing applications on the supported cloud providers and platforms.
- mOSAIC [mos, Sandru 2012]: is a European project that offers an open source API for the development and deployment of applications that use multiple clouds.
- Optimis [Ferrer 2012]: is also a European project that offers an open source PaaS that allows cloud service provisioning and the management of the lifecycle of the services.
- RightScale [rig]: is a commercial service for deploying and managing applications across clouds.
- Scalr [scab]: similar to RightScale, Scalr provides deployment of virtual machines in various clouds and includes automated triggers to scale up and down.
- STRATOS [Pawluk 2012]: offers single sign-on and monitors resource consumption and the fulfillment of service level agreements and offers autoscaling mechanisms.

Discussion

A cloud service only masks the heterogeneity problem and does not semantically resolve it. Cloud users would not have a way to understand how their applications and sensitive data are dealt with inside a cloud, thus hampering trust to cloud services. Also, an important limitation in using cloud brokering

services is the user reliance on the broker to be continuously up to date with new cloud technologies, options and offerings.

2.1.2 Programming Space

In order to allow developers to provision cloud services, each cloud offers one or several language-specific *Software Development Kit* (SDK)s to hide technical details of APIs. However, these SDKs are heterogeneous. Therefore, many multi-cloud libraries have emerged to allow developers to add abstraction between the cloud SDKs and enable multi-clouds. In this subsection, I survey different multi-cloud libraries providing a uniform way to access multiple services and resources, as well as facilitating the provisioning of services and resources from multiple clouds.

- Fog [fog]: is a Ruby library that provides an interface, making clouds easier to work with and to switch between providers.
- Gophercloud [gop]: is a Go library that allows cloud developers to connect to their applications on OpenStack clouds.
- jclouds [jcl]: is a Java library that introduces abstractions aiming the portability of applications and supports more than thirty cloud providers.
- libcloud [lib]: is a Python library that controls *Virtual Machine* (VM)s from different cloud providers.
- SimpleCloud [sim]: is a PHP library for accessing storage, queue and database services in the cloud.

Discussion

The multi-cloud libraries are tightly coupled to their programming languages like Ruby, Go, Java, Python and PHP, so the language compiler is able to check the correctness of the developer code but does not know how to perform a verification related to the cloud computing field. The developer needs to ignore implementation details and focus on general properties and characteristics. This will help him/her to avoid premature commitment to implementation choices.

2.1.3 Modeling Space

Meanwhile, there is a need for cloud architects to design their applications for multi-clouds regardless of the implementation details. For this, model-based solutions are becoming increasingly popular in cloud computing as they provide domain-specific modeling languages and frameworks that enable architects to describe/select/adapt

multi-cloud environments. This strategy is summarized as "Model once, generate anywhere". I identify some of the notable model-based solutions for multi-clouds. Unlike programming libraries, they work at a high level of abstraction by focusing on cloud concerns rather than implementation details. I believe that model-driven engineering brings many benefits for multi-clouds [Bruneliere 2010]. Therefore, I focus in the rest of this chapter on detailing the model-driven Cloudware stack and discussing model-based approaches.

2.2 Taxonomy of Model-Driven Approaches for the Cloud

In order to understand MDAC and as shown in Figure 2.2, I propose a taxonomy that presents the classification of MDAC literature in terms of three main aspects I identified:

- *MDAC usages* categorized by the phase of using the approach: *design time*, *deployment time*, and *production time* (Subsection 2.2.1),
- *MDAC concepts* used to satisfy the corresponding MDAC usages. These concepts may belong to IaaS or PaaS domains, or they reflect transverse cloud concerns like SLA, elasticity, etc. (Subsection 2.2.2), and,
- *MDAC characteristics* that represent the characteristics of the language used to implement the MDAC, *i.e.,* the *paradigm*, the *syntax* and the *semantics* (Subsection 2.2.3).

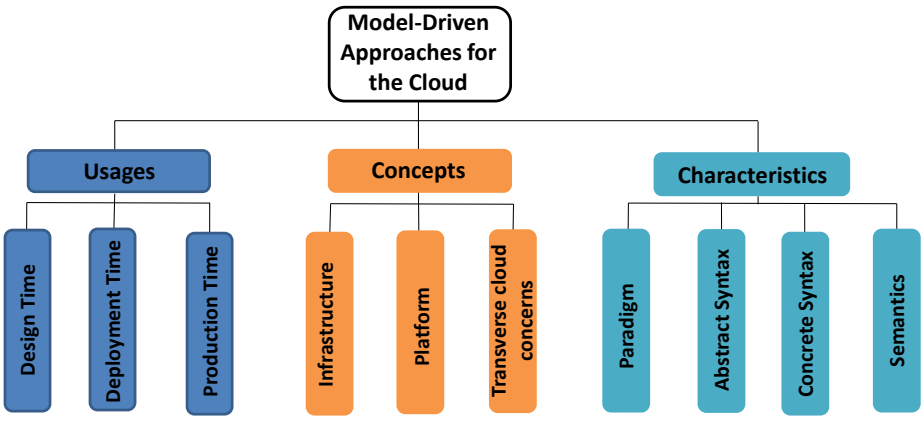


Figure 2.2: Taxonomy Criteria.

2.2.1 Usages

Usually, the process of developing an application for/in the cloud is characterized by three main phases: design, deployment and production (*i.e.,* the runtime). For each one, I have identified a set of recurrent usages that can occur during the lifecycle of a cloud application. They are the models which represent, at a high level of abstraction, concrete concerns of cloud management interfaces.

2.2.1.1 Design Time

This stage regroups the fundamental activities that enact MDAC. It consists for example in migrating a legacy system to the cloud, expressing the client needs by designing and verifying the cloud application requirements, designing the expected cloud environment to focus on cloud concerns rather than the implementation details, selecting the optimum cloud provider that suits the application requirements, refining generic models so they become adapted to represent concrete cloud offerings, exporting cloud models as specifications, documentations, and design artifacts to ease the usage of the cloud systems, etc.

2.2.1.2 Deployment Time

Once the cloud architecture model is designed, MDAC should be capable to generate the code artifacts in any form for the deployment stage. For example, if the model defined is related to Docker technology [Merkel 2014], MDAC should generate the artifacts corresponding to the model in a form of docker-compose file that can be managed by Docker swarm, or YAML configuration files that can be managed by Kubernetes or OpenShift. In addition, MDAC should be able to generate deployment scripts that can be used by a third-part deployment tools. For example MDAC will be able to generate Ansible playbooks (roles, tasks, host_vars, etc.) [ans], Puppet manifests (resources, classes, modules) [pup], and Chef cookbooks (recipes, templates, etc.) [che].

2.2.1.3 Production Time

During this phase, MDAC allow the user to have a model representation, *i.e.,* an abstraction of its cloud running system. Then, MDAC will provide a link between the designed architecture and the deployed cloud artifacts on the executing environment. When modifications occur in an existing architecture, MDAC should update the executing environment. Conversely, when changes occur in the executing environment, they should be reflected in the existing architecture. Finally, MDAC will monitor in a real time all the resources deployed in the executing environments and

will report the status in many forms including updating widgets in the designed architecture, or visualizing the monitored facts in a specialized graphs or exporting them as CSV, excel, etc.

2.2.2 Concepts

To implement an MDAC, the designers need to define a set of concepts that represent a specific cloud domain such as infrastructure or platform or related to transverse cloud concerns such as elasticity, *Service Level Agreement* (SLA) and simulation. Each domain or concern includes a set of specific concepts. For example, VM, container and network integrate the infrastructure domain, whereas server, application and database integrate the platform domain. Each domain-specific concept defines a set of attributes, actions, and constraints. An attribute represents a specific property of this type. An action defines a business specific behavior that can be triggered by a type instance (named also resource). A constraint associated to a type represents a business condition that must be respected by each conforming resource.

2.2.3 Characteristics

MDAC are defined through the use of a metamodel that formalizes the different concepts of the cloud domain. This metamodel defines the modeling language, *i.e.,* the *Cloud Modeling Language* (CML), which is the bridge between cloud developers and the cloud artifacts. Similar to other languages, a CML is defined in terms of its *paradigm*, *syntax* and *semantics*, which are the three pillars of CML characteristics [Kleppe 2008]. The syntax of a CML may be further divided into an *abstract syntax* and a *concrete syntax*.

2.2.3.1 Paradigm

The paradigm is the manner of thinking when using a language. For example, object-oriented languages involve objects as a paradigm. As for a cloud *Domain-Specific Modeling Language* (DSML), its paradigm may rely on the *application components*, *cloud services*, *cloud resources*, or *Feature Model (FM)s*. On one hand, the paradigm of components is application-oriented since it is used to describe the architecture of the application. The components are thus the entities of an application that the developer needs to deploy on the cloud. On the other hand, services, resources and FMs are cloud-oriented paradigms, *i.e.,* used to describe the cloud offerings. By services, I discuss slightly coupled, shared entities, already deployed in the cloud. A cloud platform provides services, such as computing and storage, and provides

management interfaces for these services. Since they are shared by several users simultaneously, services do not keep any state, *i.e.,* stateless. Beside services, resources, such as VMs and containers, are accessible via *Uniform Resource Identifier* (URI)s through REST or SOAP APIs. Basically, resources are not shared but, they are available on demand by each user. Being able to describe everything you want in data centers, *i.e.,* compute, storage, network, applications, but even lights for example, one can say that the use of resources is generic and not specific to the field of cloud computing. This makes the tour de force of this paradigm. As for FMs, they were introduced in 1990 by Kang et al. [Kang 1990], as part of the *Feature Oriented Domain Analysis* (FODA). They are used to denote *Software Product Line* (SPL)s. In cloud computing, FMs are used to represent variability of cloud providers.

2.2.3.2 Abstract syntax

The abstract syntax represents the concepts available in a language and how they are related. For CMLs, the abstract syntax is described by defining a metamodel, which is itself a model that defines the concepts of the domain and how they interrelate. Metamodeling techniques have been standardized by the *Object Management Group* (OMG) *Meta-Object Facility* (MOF) [MOF 2006] and there are several tools, like the EMF [EMFa], Enterprise Architect [EA], Rational Rose [Rat], $ATOM^3$ [Hußmann 2001] etc. that provide metamodeling capabilities. Regarding the abstract syntax, I identify *Unified Modeling Language* (UML) profiles, Ecore and XML schema.

With UML profiles, we talk about Internal *Domain specific language* (DSL)s which are limited to the basic language. Although they may draw the libraries and other facilities, they suffer from the lack of abstraction and the paucity of available operations. As for Ecore and XML schema, they are external DSLs, which weakness is the need to create their own tools.

2.2.3.3 Concrete syntax

The concrete syntax describes a specific representation of the language used to display models to end users. It can be either a textual syntax or a graphical representation, *i.e.,* displayed with a tree-like or a diagram notation. On one hand, the textual syntax is usually defined using a combination of regular expressions and *Backus-Naur Form* (BNF). On the other hand, the graphical syntax uses a diagram technique with named symbols that represent concepts and lines that connect the symbols and represent relationships. Several tools have been proposed to implement *(i)* textual concrete syntaxes for DSLs like Xtext [xte 2016] and EMF-

Text [emfb], and *(ii)* graphical concrete syntaxes like *Graphical Modeling Framework* (GMF) [gmf], Sirius [sir] and Graphiti [gra].

2.2.3.4 Semantics

The semantics defines well-formedness criteria and gives the meaning of abstract syntax and, indirectly, of concrete syntax. It can be classified into two main categories: static (or structural) semantics and behavioral (or dynamic) semantics. The former defines restrictions on the structure of the designed model, while the latter defines the behavior of the model elements in terms of states, events and interactions. The semantics of CMLs can be explicitly specified using *natural language*, *Object Constraint Language (OCL) constraints* and *ontologies*. However, sometimes the semantic content is not explicitly specified. In this case, domain-specific models are only transformed into artifacts of the implementation or directly executed by a model interpreter that has the potential of facilitating the processing of models at runtime in order to adapt a running application [Sousa 2012], [Fowler 2010]. In this case, the semantics is nothing but the abstraction of the model interpreter.

2.3 Model-Driven Approaches for the Cloud

Many MDAC were recently proposed in order to enable abstraction from different implementation languages and platforms. This way, the focus is shifted from the solution space towards the problem space, and from the low-level implementation details towards the higher-level domain specific concepts. The numerous existing MDAC might be overlapping in some aspects and very different in others. Developers require to have means to compare the existing approaches and to select the most appropriate one that fits their needs. Additionally, the lacks of the existing approaches need to be highlighted in order to carry on future work in this field. Consequently, the need for investigating MDAC becomes quite urgent. Across the literature on MDAC surveys, the authors in [Bergmayr 2018] recently presented the most complete state-of-the-art of cloud modeling languages, so far. They surveyed nineteen approaches, that appeared before 2015, in terms of their purposes, characteristics, capabilities and tooling. In my survey, I study twenty-two existing MDAC in terms of the three criteria elaborated in Section 2.2, *i.e.,* usages, concepts and characteristics.

Blueprinting [Nguyen 2012] provides a language that describes cloud services that are combined from a variety of cloud providers, in order to select the best configuration and easily deploy application components in cloud federations while

crossing SaaS, PaaS and IaaS layers. The current version of Blueprinting is focused on designing blueprints, which are the abstract description of applications assembled in terms of components. As for cloud offerings, they are represented and considered as services, and templates are used to specify the service features. Blueprints are encoded in XML and represented graphically in terms of a *Virtual Architecture Topology* (VAT). The Blueprinting approach aims to include a detailed and automatized deployment plan that abstracts the technical details of the interaction with a cloud, and reconfiguration actions defined in terms of policies within the WS-Policy or the SLAng languages. To my knowledge, these functionalities are not implemented so far.

Brooklyn [bro] is a framework developed by the Apache consortium for modeling and managing applications through autonomic deployment blueprints textually expressed in YAML in terms of components, and which semantics complies with the CAMP standard. Brooklyn also exposes many of the CAMP REST API endpoints and uses sensors and actuators to provide support for runtime management allowing for dynamically monitoring the application when needed. It introduces vocabulary to describe PaaS capacities and requirements of the application (*e.g.,* databases, containers), and allows the user to define and enforce his/her own reconfiguration policies.

***Cloud Application Modeling and Execution Language* (CAMEL) [Kirkham 2014]** enables developers to provision IaaS and PaaS, and to deploy application components in multi-clouds. It takes into account several aspects of the application, namely provisioning and deployment topology, provisioning and deployment requirements, service-level objectives, metrics, scalability rules, providers, execution contexts, etc. Therefore, CAMEL considers three types of models: *(i)* a Configuration Model for selecting the suitable cloud services, *(ii)* a Deployment Model for hosting the application and *(iii)* an Execution Model for managing the deployed application. CAMEL exists as an Eclipse plugin, and does not include a graphical interface, but only a textual editor for designing models. CAMEL integrates and extends existing DSLs, such as CloudML [Brandtzæg 2012, Ferry 2013], SALOON [Quinton 2013], the *Scalability Rules Languages* (SRL), and the organization part of CERIF [Asserson 2002]. I believe that CAMEL could be a source of inspiration for the future efforts in modeling the cloud.

***Cloud Application Modeling Language* (CAML) [Bergmayr 2014]** allows cloud architects to represent multi-cloud applications in UML and to select concrete cloud offerings captured by dedicated UML profiles in order to deploy the application components. As an example, *Google App Engine* (GAE) profile was applied to refine the deployment model of their Petstore reference application, towards concrete cloud offerings provided by the GAE. In this approach, cloud providers that operate at both infrastructure level and platform level are designed. CAML is a UML internal language, presented as a graphical notation, and based on a library, profiles and templates. However, the CAML approach does not include a model interpreter to enact the deployment of multi-cloud applications.

Cloud Adoption Toolkit [Khajeh-Hosseini 2012] is a collection of five tools that provide decision support for the migration of computing services to a cloud environment. It considers a number of factors that may contribute to the impact caused by the migration of an application to the cloud, *i.e.,* cost, energy consumption, stakeholder impact, social and political factors among others. However, their proposal is focused only on the cost model, which includes a number of infrastructure configuration elements, *i.e.,* operating system, server specifications (*e.g.,* CPU clock rate, RAM), storage size, applications, and data already deployed on the VM, among others. The Cost Modeling tool utilizes UML deployment diagrams (*i.e.,* graphical notation), to model an intended architecture for running legacy software in a cloud environment. Later on, price information that enables automated cost estimation for a specific cloud environment is added to the deployment model. These authors work under the assumption that, in most cases, the application deployment is performed on virtual machines. The Cost Modeling tool can model the pricing schemes of multiple cloud providers such as AWS, Microsoft Azure, FlexiScale, etc. However, once the users have created the model, they can select a single cloud provider they wish to use for each of their virtual machines.

Cloud DSL [Silva 2014] is a language that describes infrastructure services from different types of clouds. Then, Cloud DSL maps and adapts entities of the cloud models they propose to platform-specific cloud APIs. Cloud DSL is based on an Ecore metamodel and provides a graphical editor and a textual notation. Cloud DSL has been integrated with TOSCA [Binz 2012]. Using Cloud DSL with TOSCA reduces the workload of creating cloud descriptions in a TOSCA specification.

CloudGenius [Menzel 2012] is a framework mainly used for the selection of appropriate cloud infrastructure services among several ones stored manually and

described textually. The Ecore metamodel, on which CloudGenius relies, allows a multi-criteria decision approach from a set of requirements. The latter are based on numerical functional requirements (network latency, technical parameters such as CPU, RAM and storage size, popularity, etc.) and non-numerical functional requirements (operating system, virtual machine format, licence, etc.). Yet, this approach neglects to consider non-functional concerns like the cost, the availability, the response time, etc. A tool prototype named CumulusGenius, used as a Java library, allows the user to programmatically define the requirements that are given as input to CloudGenius selection framework. Then, whenever a solution is found, virtual machines can be executed on top of Amazon EC2 only.

CloudMIG [Frey 2011] is a framework that facilitates the migration of existing software systems to IaaS and PaaS-based cloud environments, which are Amazon EC2 and Google App Engine, respectively. In this approach, cloud environments are modeled as instances of a *Cloud Environment Model* (CEM) which is an Ecore-based metamodel and for each cloud environment, all possible configurations are modeled. A configuration contains in particular a set of elements and constraints on them. CloudMIG takes as input the legacy software system, and extracts the architectural and utilization models based on the *Architecture-Driven Modernization* (ADM) principles. From this model, a single compatible cloud environment model candidate is selected. Then, CloudMIG relies on its own constraint validators CloudMIG Xpress [Clo] to check the conformance of the legacy software (the extracted models) with the candidate CEM in terms of constraint violations. The CloudMIG framework is then extended to improve the search of well-suited IaaS environments using search-based genetic optimization.

CloudML [Brandtzæg 2012, Ferry 2013, Ferry 2018] is a cloud modeling language that allows both cloud providers and developers to describe cloud services and application components, respectively. Then, it helps to provision cloud resources by a semi-automatic matching between the defined application requirements and the cloud offerings. CloudML is exploited both at design time to describe the application provisioning of cloud resources after performing the necessary orchestration, and at runtime to manage the deployed applications. In fact, the model at design time is automatically handled by the *Cloud Modeling Framework* (CloudMF), which returns a runtime model of the provisioning resources, according to the Models@run.time approach [Blair 2009]. CloudML only provides a JSON and an XML textual syntax to specify deployment and management concerns in IaaS and/or PaaS clouds. CloudML is first introduced in

the REMICS project [Sadovykh 2011] as a UML model and developed later by three projects that differ in their objective, *i.e.,* ARTIST [Bergmayr 2013], MODAClouds [Ardagna 2012], and PaaSage [paa, Jeffery 2017]. On one hand, REMICS and ARTIST mainly support the migration of legacy software towards a cloud-based environment. In this sense, they adopt UML models since they are reverse-engineered and tailored to target cloud systems. In order to extract some semantics, they map the UML models to OpenTOSCA [Binz 2013]. The *Cloud target Selection* (CTS) [Kopaneli 2015] provides a multi-criteria decision making process for the selection of the cloud target. It combines different types of criteria by using the concepts of CloudML@ARTIST. On the other hand, MODAClouds and PaaSage aim at supporting engineers in building and deploying multi-cloud applications. Therefore, they propose Ecore-based models that include dynamic variability to deal with multiple cloud environments and especially runtime changes. Note that CloudML in PaaSage is the first member of the family of DSLs that form CAMEL.

Farokhi [Farokhi 2014] proposes a framework that assists SaaS providers to select suitable IaaS, which best satisfy their requirements while handling SLA issues. The framework includes three main phases: *(1)* SLA Construction, *(2)* Service Selection, and *(3)* SLA Monitoring and Violation Detection. The Service Selection Engine takes a textual input, an XML file precisely, that describes the SaaS provider requirements. Then, it finds the adequate IaaS providers' services. A breached SLA on runtime will question the selection of the cloud provider and will probably lead to some reconfiguration.

Frey et al. [Frey 2013] focus on selecting near-optimal cloud deployment architectures and defining runtime reconfiguration rules. The main purpose of this approach is to support the migration of software components and their deployment on IaaS environments. To do so, the authors define *Cloud Deployment Option* (CDO)s which are UML profiles with graphical syntax and constraints written in English prose. Then, they propose CDOXplorer, a genetic algorithm that takes the CDOs as input and analyzes the configuration space of a given cloud provider. Later on, CDOXplorer finds the best configuration based on the average response times and SLA violations. CDOXplorer is implemented in the scope of an open source tool CloudMIG Xpress, that utilizes models which can almost be automatically extracted. The authors in [Frey 2013] assume that the application deployment is always performed on virtual machines. They don't take the principle of containers into account.

Garcia-Galán et al. [García-Galán 2016] aim to solve the problem of selecting the most suitable configuration among the configuration space offered by a given provider. Their focus is on IaaS. They propose a model that is based on FMs, and apply the automated analysis of FMs as a reasoning technique over the model. Their model can be graphically represented using a tree-like notation, in which features are organized hierarchically. However, their approach did not consider defining a metamodel based on FMs, for the configuration of cloud services. The information to create the FM is automatically extracted from the provider website using an ad-hoc web crawler. This proposal is only applicable to one cloud provider, which is Amazon EC2. However, these authors plan to include different providers in the future. Their model includes cloud configuration elements such as instance type, which determines the configuration of a machine, operating system, storage capability, geographic location, billing information, and customer usage data. Finally, their proposal allows defining constraints on the features and attributes of the model. These constraints are written in English prose. They implemented their proposal and compared their implementation against two commercial tools, Amazon TCO and CloudScreener, and they concluded that their proposal is more expressive and accurate in terms of providing a wider range of configuration options and choosing the most suitable configuration.

Gherardi et al. [Gherardi 2014] claim to present the first paper that combines robotics, cloud computing, and SPLs. It is interested in configuring and deploying complex *Robot as a Service* (RaaS) only on top of Rapyuta [Mohanarajah 2015], an open source robotic PaaS. Decisions regarding what components of Rapyuta to employ and how to compose them (the connections) are taken by exploiting three models via a Resolution Engine. The first two models are a reference architecture which is an Ecore metamodel reflecting the requirements of the application, and an extended FM, *i.e.,* a FM that enriches the features in a model with attributes in order to improve the semantics. The third model is the glue between the first two models and specifies how the variability can be resolved. Feature Selector tool for creating a selection of features reflecting the requirements of their application. Graphical editors are used to design the models, that are described within a textual syntax too.

Holmes [Holmes 2014] proposes three textual languages based on an Ecore metamodel for expressing and capturing IaaS concepts, then provisioning a customized stack of cloud services, via model transformations. From the DSL programs and the supplied Puppet [pup] modules, the entire cloud service stack is automat-

ically built, without further user interaction. Later on, in order to reconfigure the deployment and achieve the new requirements of the system, reverse-engineering is used to capture the differences between models. Therefore, for dealing with differential changes of IaaS models, Holmes [Holmes 2015] proposes a model-based round-trip engineering approach that combines the power of model-driven generation with runtime reflection, *i.e.,* this approach does not only incorporate models from design time but also Models@run.time. This approach allows to compare and migrate infrastructure services between two clouds. They consider a migration from OpenStack 2012.1 to OpenStack 2013.2. For orchestration, they employ Nova API [nov], which is OpenStack native.

MOve to Clouds for Composite Applications **(MOCCA) [Leymann 2011]** is a method for moving legacy applications to the cloud. It introduces an Ecore metamodel for specifying the applications that are modeled in terms of components. The model semantics is described with natural language and can also be deduced by the behavior of the deployment optimizer in use. The authors also propose Cafe [Mietzner 2009], a prototypical tool supporting the MOCCA method and offering graphical and textual modeling of the application architecture and topology. The MOCCA method allows for provisioning infrastructure resources that are described in OVF files which perform the required adaptation for the components deployment. Cafe assumes that an OVF file represents only one component. In case an OVF file contains the virtual image of more than one component (*i.e.,* more than one virtual system), this file must be split into separate OVF files manually. Thus, Cafe does not support the notion of multiple clouds. However, the authors of this method state that a future extension of Cafe will support OVF files with virtual images of multiple components.

MULTICLAPP [Guillén 2013] is a framework for modeling components of multi-cloud applications which are not dependent of any specific cloud provider. This framework is based on a UML profile, with a graphical editor to model components that are expected to be deployed on PaaS cloud environments by applying cloud provider independent stereotypes to them. These stereotypes enable the application developers to select the cloud provider offerings that are best for deploying the application components. Applications that are fully modeled are processed by a deployment engine, which generates each of the cloud artifacts identified in the deployment plan. Each artifact is adapted in order to comply with the specifications of its assigned platform. Once they are generated, the artifacts can be deployed in their cloud platforms.

OpenTOSCA [opeb] is an ecosystem developed by the University of Stuttgart that aims to provide modeling tool support and runtime support for the TOSCA standard [Binz 2012]. Several implementations of OpenTOSCA were developed. For example, *(i)* Winery [Kopp 2013] provides an open source Eclipse-based graphical modeling tool for TOSCA topologies/structures/architectures, *i.e.,* the software components that constitute the application, the physical or virtual nodes on which the components will be deployed, and the relationships between components and nodes, and *(ii)* OpenTOSCA runtime [Binz 2013] provides an open source container for deploying TOSCA-based applications defined in a *Cloud Service ARchive* (CSAR) packaging format. The OpenTOSCA runtime is hence responsible for translating a TOSCA topology into actions to be performed in clouds. These actions are sent to the clouds through their respective APIs. Despite TOSCA language manages to cover the infrastructure and platform service stack, it is only defined as a textual XML document or YAML document so it is complicated to have an overview of the supported cloud entities. Furthermore, TOSCA does not employ the typical cloud vocabulary, such as services and resources. Instead, it defines a set of abstract elements, such as nodes and relationships to respectively designate cloud services and how they interact. Therefore, designing a TOSCA topology requires the effort of a human developer, which is a time consuming and an error-prone activity. The application deployment to the target cloud and its management are provided by orchestration plans written within different workflow languages,*e.g.,* BPMN or BPEL. However, in case some module of the application is migrated to a different target provider, the topology and the orchestration plan should be modified which makes the management of a TOSCA-compliant deployment a complex task.

RESERVOIR-ML [Chapman 2012] offers a language for the description of requirements that providers must fulfill when the developers deploy a multi-component application on federated IaaS clouds. Among these requirements, it takes into account non-functional requirements such as quality of service. The RESERVOIR-ML language encodes the OVF standard within XML and its semantics is described within OCL constraints. Beside describing the requirements, the main focus of this approach is also to perform reconfiguration tasks and address the scaling requirements of the application components, *i.e.,* to ensure elasticity and provision IaaS resources on demand. To do so, the RESERVOIR-ML project has also developed UCL-MDA tools, a graphical framework implemented as a plugin for the Eclipse *Integrated Development Environment* (IDE) for the manipulation of the XML models and the OCL constraints.

SALOON [Quinton 2013] is a graphical framework for cloud environments selection and configuration purpose. SALOON is an EMF-based framework that relies on extended FMs to represent clouds variability, as well as on ontology concepts to model the various semantics of cloud systems. It mainly comprises functional elements such as the language used to develop the cloud-based application, the number of application servers, the RAM, the CPU, etc. This proposal also allows to translate the ontology concepts into a *Constraint Satisfaction Problem* (CSP) in order to select the adequate cloud environment. In order to extract the information to create the models for each cloud provider, the authors suggest the use of reverse engineering on the web configurator of each cloud provider as a solution. They implemented their proposal and tested the performance of their implementation. They concluded that their proposal was well suited to handle large configuration spaces, with a number of features and constraints that would make it overwhelming for a human user to perform the selection by hand. Despite that the authors state that SALOON supports the discovery and selection multiple providers, in practice it does not. In the contrary, it deals with one provider at a time. SALOON targets ten cloud environments (IaaS and PaaS).

soCloud [Paraiso 2014] is an approach that aims at developing multi-cloud applications by defining a PaaS platform based on FraSCAti, a Service Component Architecture platform. soCloud defines its concepts within an XML schema. It provides a textual syntax and its semantics is written in English Prose in the context of the SCA specification that is implemented in FraSCAti. soCloud targets fifteen cloud environments (IaaS and PaaS), where it allows deploying and reconfiguring application components after achieving the necessary orchestration.

Sousa et al. [Sousa 2017] aim to generate reconfiguration plans that satisfy the requirements of a multi-cloud computing system. To do so, the authors propose an Ecore metamodel to model FMs and capture the variability of cloud configurations. The multi-cloud constraints that arise during the cloud reconfiguration are defined by *Linear Temporal Logic* (LTL) formulas to express temporal properties. The authors applied their approach only to Heroku cloud PaaS and they manually built their FM by going through the Heroku documentation.

StratusML [Hamdaqa 2015] is a layered modeling language and a modeling framework for cloud applications. StratusML provides a user-friendly interface that allows the cloud developers to specify their application components, configure them, estimate cost under diverse cloud services, select a cloud provider, use templates to

transform and adapt the model into platform specific artifacts, and manage the application behaviour at runtime through a set of rules. It is built as an extension of Microsoft Visual Studio 2012, *i.e.,* the Microsoft DSL toolkit is used to design the StratusML graphical editor and to define the validation constraints. The latter are required to ensure that the specified model satisfies the application requirements and provides the information required to generate the target platform specific artifacts. The validation constraints can be classified into hard constraints, *i.e.,* that the user can never violate, and soft constraints, *i.e.,* that are allowed to be violated, but still create warnings and errors to guide the user to the correct decisions. In order to capture the application deployment configuration, the StratusML metamodel integrates five different models to address five different, but interleaved functional and non-functional cloud concerns. It includes the service model, performance model, adaptation model, availability model, and provider model. StratusML uses layers to view the different cloud application concerns, facilitating visual modeling of adaptation rules, and using template-based transformation to deal with platforms heterogeneity. StratusML has established a connector only with the Windows Azure IaaS.

2.4 Discussion

Conclusion 1. Primary focus on design time aspects

As depicted in Table 2.2, most of the MDAC only provide the possibility to set the resources (CPU, memory, disk, network, etc.) limits at design time. However, they lack of resources management at runtime. The management is necessary because in the cloud environment, the resources consumption fluctuates according to the workload. In order to provision the appropriate resources, if the workload grows or shrinks, the resources should be reconfigured, *i.e.,* increased or decreased as required at runtime. Thus, a major challenge is how to synchronize the predefined architecture of resources with the resources provisioned in the execution environment. When modifications occur in an existing architecture, the update should be done in the executing environment. Conversely, when changes like the increase of the disk storage or the addition of a virtual machine occur in the executing environment, they should affect the existing architecture. It is thus required that an MDAC reduces the gap between design and runtime activities and provides the same model for both of them.

We tackle this problem in Chapter 3 by providing a complete tool chain to handle cloud resources during their whole lifecycle, from the design till the management.

Table 2.2: MDAC Usages.

MDAC	Design Time	Deployment Time	Production Time
Blueprinting	✓		
Brooklyn	✓	✓	✓
CAMEL	✓	✓	✓
CAML	✓		
Cloud Adoption Toolkit	✓		
Cloud DSL	✓		
CloudGenius	✓		
CloudMIG	✓	✓	
CloudML	✓	✓	✓
Farokhi	✓	✓	✓
Frey et al.	✓	✓	✓
Garcia-Galán et al.	✓		
Gherardi et al.	✓		
Holmes	✓	✓	✓
MOCCA	✓	✓	
MULTICLAPP	✓		
OpenTOSCA	✓	✓	
RESERVOIR-ML	✓	✓	✓
SALOON	✓		
soCloud	✓	✓	✓
Sousa et al.	✓	✓	✓
StratusML	✓	✓	

Conclusion 2. Primary focus on IaaS

As depicted in Table 2.3, the largest amount of researchers attention has been focused on IaaS clouds. An efficient MDAC should allow to handle infrastructure, platform and software resources. There is a strong separation between these three types of resources since each of them is managed by a particular resource manager. These managers do not know how to cooperate. Thus it is extremely difficult to implement policies for the management of multi-level resources. However, in order to manage the elasticity of a system for example, the cloud developer needs to manage simultaneously resources at IaaS, PaaS and SaaS levels. Therefore, there is a need for a single MDAC that includes concepts and mechanisms that support both IaaS and PaaS clouds, enabling their management.

We also tackle this problem in Chapter 3. In fact, our proposed tool chain for the cloud computing complies to OCCI, the only generic and extensible standard that handles every kind of cloud resources, *i.e.,* IaaS, PaaS, SaaS and even RaaS and *Container as a Service* (CaaS).

Table 2.3: MDAC Concepts.

MDAC	Service Model
Blueprinting	XaaS
Brooklyn	PaaS
CAMEL	XaaS
CAML	IaaS & PaaS
Cloud Adoption Toolkit	IaaS
Cloud DSL	IaaS
CloudGenius	IaaS
CloudMIG	IaaS & PaaS
CloudML	IaaS & PaaS
Farokhi	IaaS & SaaS
Frey et al.	IaaS
Garcia-Galán et al.	IaaS
Gherardi et al.	RaaS
Holmes	IaaS
MOCCA	IaaS
MULTICLAPP	PaaS
OpenTOSCA	IaaS & PaaS
RESERVOIR-ML	IaaS
SALOON	IaaS & PaaS
soCloud	IaaS & PaaS
Sousa et al.	PaaS
StratusML	IaaS

Conclusion 3. Fuzziness of the CML concepts

Most of the MDAC are built from scratch; the designer of the CML goes through the provider or the application documentation, and then manually defines the concepts that he/she considers important to be included to the provider or the application metamodel. This methodology results in the fuzziness of the CML which might be unrepresentative of the concrete cloud environment. Also, the MDAC I reviewed in this chapter describe a part of the cloud domain that was relevant only at the moment of the definition of the modeling language. However, the designers of each MDAC require changing their modeling language, *i.e.,* the CML, at each time they want to support more cloud concepts. As for the user, he/she is unable to add the missing concepts that he/she needs.

I tackle this problem in Chapter 4 where I propose the first advanced approach for automatically inferring a cloud model that properly represents the cloud concepts and operations. This model can be updated to follow up with the cloud API and since it conforms to the generic OCCIware Metamodel, this model can be extended to support new concepts. It also helps analyzing the cloud API and enhances its specification so the developer can correctly use its services.

Conclusion 4. Little attention paid to the semantics

We observe in Table 2.4 that the semantics of the CMLs is, in most cases, either *informal*, namely written in English prose or within OCL constraints, or *implicit* in the model interpreter behaviour. None of these ways of defining the semantics is sufficiently precise. Natural language might be confusing due to its built-in ambiguity; although words with multiple meanings give English a linguistic richness, they also create ambiguity. OCL is semi-formal, *i.e.,* its syntax is well-defined but its semantics is only partially formalized, with many aspects being just described in natural language in the standard document specifications. Also, OCL is efficient for only specifying the static semantics of the CMLs. Dynamic semantics remains defined within natural language. Finally, the model interpreter is the engine that is fed the deployment, configuration, adaptation models in order to execute them. Deriving the semantics of these models from the behaviour of the model interpreter might be erroneous. It is crucial then to propose CMLs with well-formed semantics, *i.e.,* defined within formal methods which are mathematical techniques that allow the cloud stakeholders to reason and describe without ambiguity the structure metamodel and the behavior of its concepts.

I tackle this problem in Chapter 5 where I propose the first formal framework for precisely specifying cloud APIs and reasoning over them. Consequently, the developer can verify the correctness of his/her cloud models and their required behaviour.

2.5 Summary

The wide number of available cloud providers, their high heterogeneity and semantic differences make it complicated to exploit multi-cloud assets. In this chapter, I provided a classification of Cloudware engineering solutions. I showed that the solutions at the provider and the programming spaces are also heterogeneous and their provided features are often incompatible. This diversity hinders the proper exploitation of the full potential of cloud computing, since it prevents interoperability and promotes vendor lock-in, as well as it increases the complexity of development and administration of multi-cloud systems.

To deal with this heterogeneity, I introduced the solutions at the modeling space and explained the major role that models play in the software development for the cloud computing. I discussed the idea of "Modeling the cloud computing" by leveraging MDE to easily build cloud-native applications. I discussed the usages, concepts and characteristics of MDAC. Finally, I reviewed the most relevant approaches in the research area that is closely related to this thesis, *i.e.,* model-driven engineering

Table 2.4: CML Characteristics.

MDAC	Paradigm	Abstract Syntax	Concrete Syntax	Semantics
Blueprinting	Services & Components	XML schema	Graphical	Natural language
Brooklyn	Components	YAML document	Textual	Natural language & CAMP specification
CAMEL	Services & Components	Ecore	Textual	Natural language & ExecutionWare
CAML	Services & Components	UML profile	Graphical	Natural language
Cloud Adoption Toolkit	Services	UML profile	Graphical	Natural language
Cloud DSL	Services & Components	Ecore	Graphical	Mapping to TOSCA
CloudGenius	Services	Ecore	Textual	Natural language & Selection Framework
CloudMIG	Components	Ecore	Textual	Natural language & CloudMIG Xpress (Deployment Optimizer)
CloudML	Services & Components	Ecore	Textual	Natural language & CloudMF
Farokhi	Services & Components	XML schema	Textual	Service Selection Engine
Frey et al.	Services & Components	UML profile	Graphical	Natural language & CDO Xpress (Deployment Optimizer)
Garcia-Galán et al.	Feature Models	-	Graphical	Natural language & Automated Analysis of Feature Models (AAFM)
Gherardi et al.	Feature Model	Ecore	Graphical & Textual	Resolution Engine
Holmes	Services & Components	Ecore	Textual	Natural language
MOCCA	Components	Ecore	Graphical & Textual	Natural language & Deployment Optimizer
MULTICLAPP	Services & Components	UML profile	Graphical	Natural language & Deployment Engine
OpenTOSCA	Services & Components	XML schema or YAML document	Graphical & Textual	Natural language & TOSCA specification
RESERVOIR-ML	Components & Resources	XML schema	Graphical	Natural language & OpenNebula
SALOON	Feature Models	Ecore	Graphical	An Ontology & Translation into a constraint solver
soCloud	Services & Components	XML schema	Textual	Natural language & FraSCAti
Sousa et al.	Feature Models	Ecore	Textual	Temporal constraints
StratusML	Services & Components	Microsoft DSL	Graphical	Natural language

for cloud computing. Among the prolific research in this area, there is a lack of solutions which:

- support design, deployment and management usages,
- allow handling XaaS systems,
- define the appropriate cloud concepts with possibility of extension if needed, and,
- define a precise semantics of these concepts.

Based on this study, I describe in the next parts of this dissertation the OCCIware background and the two contributions of this thesis respectively. I mainly propose to leverage MDE and formal methods to help cloud stakeholders taking better advantage of cloud services.

Part III

Background

In the end of Part II, I discussed the need of precisely describing but also efficiently managing every kind of cloud resources. To accomplish this purpose, I detail in this part our OCCIware *approach.*

Chapter 3

Modeling, Verifying, Generating and Managing Cloud Resources with OCCIware

This chapter corresponds to our article "Model-Driven Cloud Resource Management with OCCIware" [Zalila 2018] submitted to the Future Generation Computer Systems (FGCS) journal, which extends our paper "A Model-Driven Tool Chain for OCCI" [Zalila 2017a] published in the 25th International Conference on Cooperative Information Systems (CoopIS).

Contents

Several cloud computing standards have been proposed to resolve the heterogeneity of cloud providers and promote multi-clouds, as discussed in Chapter 2.

However, the main drawback of these standards is their specificity for a particular cloud service model, *i.e.,* IaaS or PaaS.

OCCI has been proposed as the first and only open standard for managing any cloud resources [Edmonds 2012]. OCCI provides a general purpose model for cloud computing resources and a RESTful API for efficiently accessing and managing any kind of these cloud resources. This will ease interoperability between clouds, as providers will be specified by the same resource-oriented model called the OCCI Core Model [Nyrén 2016b], that can be expanded through extensions and accessed by a common REST [Fielding 2000] API.

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

To overcome the issues presented above, I present in this chapter our OCCIware approach, which can be summarized as:

- **Model-Driven Managing Everything as a Service with OCCIware.** OCCIware is a model-driven vision to manage XaaS. It allows one to model any type of resources. It provides OCCI users with facilities for designing, editing, validating, generating, implementing, deploying, executing, managing, and supervising XaaS with OCCI.

- **Generating Cloud Domain-Specific Modeling Frameworks with OCCIware.** OCCIware is a factory of cloud domain-specific modeling frameworks. Each generated *Cloud Domain-Specific Modeling Studio* (CDSMS) is dedicated for a particular cloud domain. Each CDSMS can be used to design configurations conforms to its related domain and hides the generic concepts of OCCI.

This work has been done in the context of the OCCIware research and devel-

opment project[1] funded by the French PIA. The contribution of the academic and industrial partners has certainly promoted the progress of this project. A special gratitude is due to **Faiez Zalila** and **Christophe Gourdin** who implemented the OCCIware approach.

This chapter is structured as follows. Section 3.1 explains the motivations behind OCCIware. Section 3.2 gives a background on the OCCI standard. Section 3.3 presents an overview of the OCCIware approach. It details the different processes to use the OCCIware approach. Section 3.4 presents the OCCIware Metamodel by detailing its static semantics defined in Ecore and OCL. Section 3.5 provides an overview of OCCIware Studio and its different implemented features. Section 3.6 presents OCCIware Runtime and details its architecture. Section 3.7 validates OCCIware by presenting the different OCCI extensions defined by the standard and implemented using OCCIware. We follow up with the evaluation of OCCIware by discussing different five use cases implemented with the OCCIware approach. Finally, Section 3.8 concludes with future work and perspectives.

3.1 Motivations

Currently, cloud architects and developers have a lot of hope for the multi-cloud computing paradigm as an alternative to avoid the vendor lock-in syndrome, to improve resiliency during outages, to provide geo-presence, to boost performance and to lower costs. However, semantic differences between cloud provider offerings, as well as their heterogeneous CRM-APIs make migrating from a particular provider to another a very complex and costly process. We assume for example that a cloud developer would like to build a multi-cloud system spread over two clouds, AWS and GCP. AWS are accessible via a SOAP-based API, whereas GCP is based on a REST API, which leads to an incompatibility between these two different APIs. To use them, cloud consumers should be inline with the concepts and operations of each API, which is quite frustrating. The cloud developer would like a single API for both clouds to seamlessly access their resources.

For this, OCCI is an open standard that defines a generic extensible model for any cloud resources and a RESTful API for efficiently accessing and managing cloud resources. This will facilitate interoperability between clouds, as cloud provider's offerings will be specified by the same resource model, and accessed by a common REST API. However, cloud developers cannot currently take advantage of this standard. Although there are several implementations of OCCI, there is no tool that allows them to design and verify their configurations, neither to generate and deploy

[1] www.occiware.org

corresponding artifacts. This leads to several challenges:

1. Cloud architects, who are supposed to design the expected multi-cloud platform, are facing on one side to heterogeneity at different levels such as CRM-APIs heterogeneity (REST APIs vs SOAP APIs), service models heterogeneity (IaaS, PaaS, SaaS, etc.), deployment model heterogeneity (public, private and hybrid), and service providers heterogeneity. On the other side, cloud developers, who create and deploy running cloud systems, are focused on implementation details rather than cloud concerns, with the risk of misunderstandings for the concepts and the behavior that rely under cloud APIs. They need a customized cloud framework dedicated to each cloud domain.

2. The only way to be sure that the designed configurations will run correctly is to deploy them in the clouds. In this context, when errors occur, a correction is made and the deployment task can be repeated several times before it becomes operational. This is quite painful and expensive.

3. Cloud developers need to provide various forms of documentation of their cloud configurations, as well as deployment artifacts. However, these tasks are complex and usually made in an ad-hoc manner with the effort of a human developer, which is error-prone and amplifies both development and time costs.

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

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

Recently, we are witnessing several works that take advantage of MDE for the cloud [Bruneliere 2010, Bergmayr 2018]. Therefore, to address the identified challenges, we believe that **there is a need for a tooled model-driven approach for OCCI** in order to:

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

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

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

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

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

3.2 Background on OCCI

OCCI is an open cloud standard [Edmonds 2012] specified by the OGF. OCCI defines a RESTful Protocol and API for all kinds of management tasks on any kind of cloud resources, including IaaS, PaaS and SaaS. In order to be modular and extensible, OCCI is delivered as a set of specification documents divided into the four following categories as illustrated in Figure 3.1:

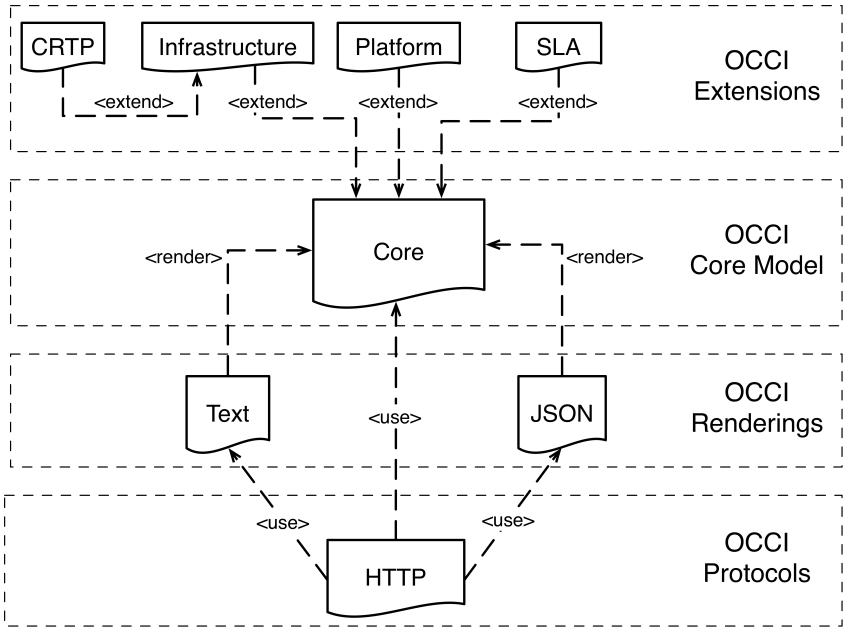


Figure 3.1: OCCI Specifications.

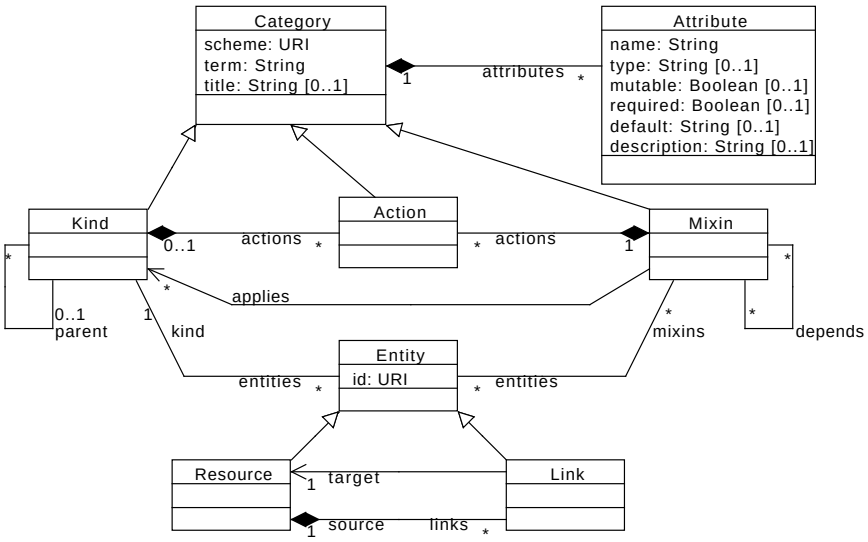


Figure 3.2: UML Class Diagram of the OCCI Core Model (from [Nyrén 2016b]).

OCCI Core Model. It defines the OCCI Core specification [Nyrén 2016b] proposed as a general purpose RESTful-oriented model. It is shown in Figure 3.2 and represented as a simple resource-oriented model composed of eight concepts: `Resource` represents any cloud computing resource, *e.g.,* a virtual machine, a network, an application container, an application. `Link` is a relation between two

`Resource` instances, *e.g.,* a computer connected to a network, an application hosted by a container. `Entity` is the abstract base class of all resources and links. `Kind` is the notion of class/type within OCCI, *e.g.,* `Compute`, `Network`, `Container`, `Application`. `Mixin` is used to associate additional cross-cutting features, *e.g.,* location, price, user preference, ranking, to resource/link instances. `Action` represents an action that can be executed on entities, *e.g.,* start a virtual machine, stop an application container, restart an application, resize a storage. `Category` is the abstract base class inherited by `Kind`, `Mixin`, and `Action`. `Attribute` represents the definition of a customer visible property, *e.g.,* the hostname of a machine, the IP address of a network, or a parameter of an action.

OCCI Protocols. Each OCCI Protocol specification describes how a particular network protocol can be used to interact with the OCCI Core Model. Multiple protocols can interact with the same instance of the OCCI Core Model. Currently, only the OCCI HTTP Protocol [Nyrén 2016a] has been defined. But other OCCI protocols would be proposed in the future such as *Advanced Message Queuing Protocol* (AMQP).

OCCI Renderings. Each OCCI Rendering specification describes a particular rendering of the OCCI Core Model. Multiple renderings can interact with the same instance of the OCCI Core Model and will automatically support any OCCI extension. Currently, both OCCI Text [Edmonds 2016] and JSON[2] [Nyrén 2016d] renderings have been defined. Other OCCI renderings would be specified in the future, such as an XML rendering for instance.

OCCI Extensions. Each OCCI Extension specification describes a particular extension of the OCCI Core Model for a specific application domain, and thus defines a set of domain-specific kinds and mixins. OCCI Infrastructure [Nyrén 2016c] is dedicated to IaaS. Additional OCCI extensions are defined such as OCCI *Compute Resource Templates Profile* (CRTP) [Drescher 2016], OCCI Platform [Metsch 2016] and OCCI SLA [Katsaros 2016].

3.3 OCCIware Approach

3.3.1 Managing Everything as a Service with OCCIware

The OCCIware funded project [occc, Parpaillon 2015] aims to provide a formal comprehensive, coherent, modular, model-driven tool chain for managing any kind

[2] JavaScript Object Notation

of cloud computing resources. The OCCIware approach relies on MDE, a software engineering paradigm that proposes to reason on high-level artifacts, called *models*, rather than the code implementation. As MDE allows us to raise the level of abstraction, a *model* is an abstract representation of a system. It allows us to understand the designed system and answer the related queries. A model conforms to a *metamodel*, which defines the modeling language.

The OCCIware approach is composed of two main components as depicted in Figure 3.3: *(i)* OCCIware Studio implemented by Faiez Zalila, and *(ii)* OCCIware Runtime implemented by Christophe Gourdin.

Figure 3.3: OCCIware Studio and OCCIware Runtime.

OCCIware Studio, detailed in Section 3.5 is an OCCI model-driven tool chain that enables to design, verify, simulate, and develop every kind of resources as a service. Usually, a model-driven approach is based on, at least, a metamodel. The OCCIware approach is designed and developed based on a metamodel, called OCCIware Metamodel and detailed in Section 3.4. This metamodel implements and extends the OCCI Core Model.

OCCIware Runtime detailed in Section 3.6 is a generic OCCI-compliant Models@run.time support and includes a resources container, and tools for deployment, execution, and supervision of XaaS.

To benefit from the OCCIware approach, a proposed process must be followed (cf. Figure 3.4). This process has three steps: the *design step*, the *engineering step*, and the *use step*.

3.3.1.1 Design step

The *design step* (the top of Figure 3.4) consists in defining a new OCCI extension that extends the OCCI `Core` extension (an extension-like representation of the OCCI Core Model), and/or other OCCI extensions already defined. This step is ensured by the **Cloud Architect** who aims to have a tooled model-driven framework for his/her cloud domain such as infrastructure, platform, etc. An OCCI extension model conforms to OCCIware Metamodel. It can be designed textually and/or graphically. Once the extension is designed and validated, a generation process of the `Extension Tooling` may be triggered. It consists to generate, from an OCCI extension model, a set of artifacts that meet the needs of the cloud developers.

The set of artifacts generated from an OCCI extension model can be summarized as:

1. **Extension Documentation** represents a comprehensive documentation of the designed extension. It serves as the reference document to describe for the cloud developer the different notions designed in the extension.

2. **Extension Formal Specification** defines formally the specification of the designed extension. This artifact can be later analyzed using a dedicated tool to check rigorously its correctness and its conformance to the OCCI specifications. This extension is detailed in Chapter 5.

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

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

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

6. **Extension Designer** is a dedicated model-driven graphical designer to the domain of the extension. It gives a suitable framework for the cloud developer to graphically design the different resources of a running system, which instantiate the extension concepts.

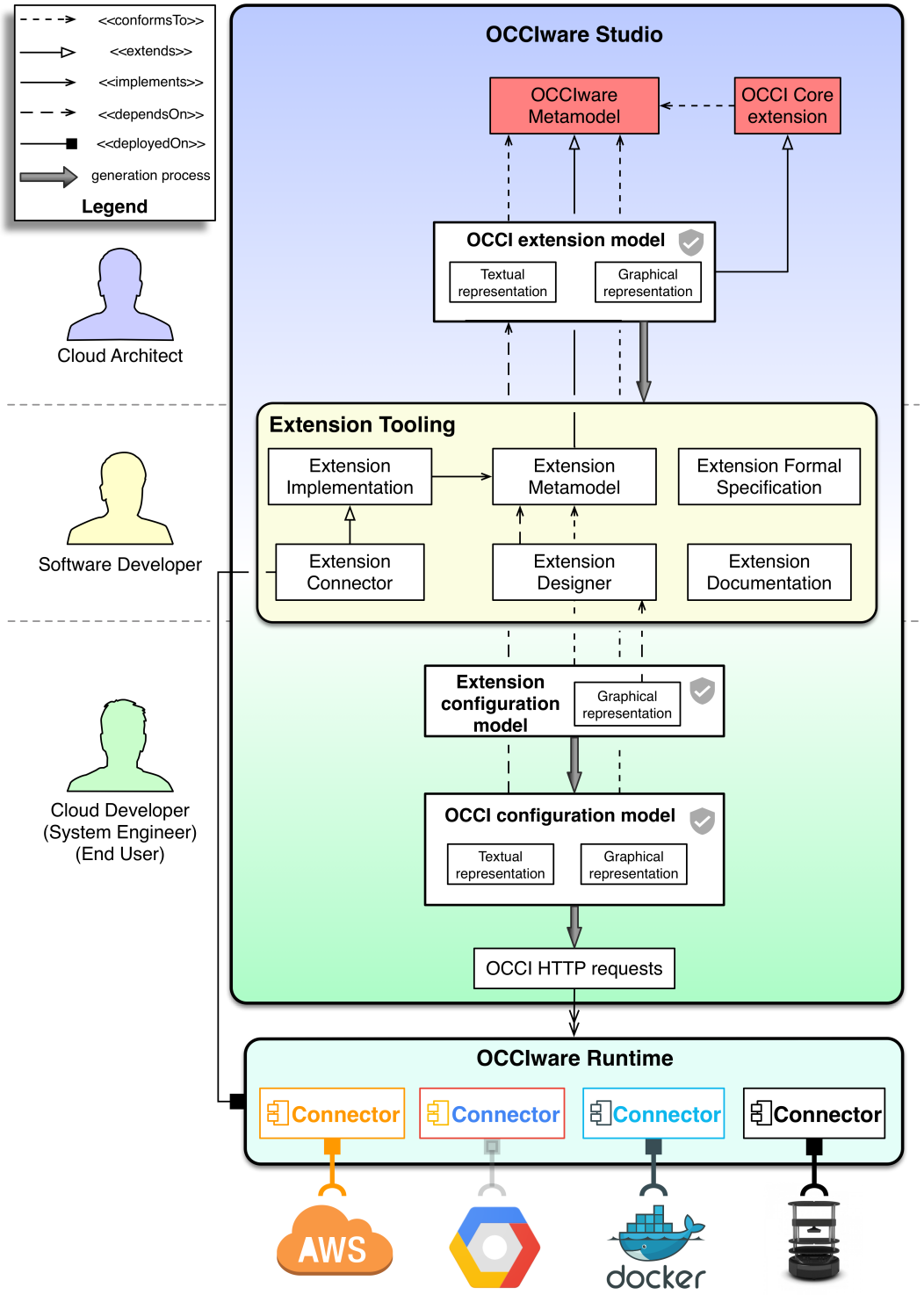


Figure 3.4: Model-Driven Managing Everything as a Service with OCCIware.

3.3.1.2 Engineering step

Once the previous step is achieved, the **Software Developer** can complete the generated `Extension Connector`. It consists of implementing the business code required to handle each concept of the extension. Later, the completed connector must be deployed on OCCIware Runtime. Finally, he/she customizes the generated `Extension Designer` to adapt it to the dedicated domain and to abstract the different concepts of the extension. From now on, we can consider that the `Extension Tooling` is able to be used to manage conforming configurations.

3.3.1.3 Use step

Thanks to OCCIware Studio enriched with the `Extension Tooling` provided during the previous steps, the cloud developer, who is the **System Engineer** (end user from our perspective), can design using the dedicated `Extension Designer`, an `Extension` configuration model conforms to the `Extension Metamodel`. For the system engineer, the different OCCIware tooling is entirely hidden because he/she only deals with domain concepts such as VMs, containers, networks, etc.

To benefit from the OCCI-compliant tools of OCCIware (*i.e.,* OCCIware Runtime), the `Extension` configuration model must be translated into an OCCI configuration model that conforms to the OCCIware Metamodel. Both models are semantically identical, but the first one instantiates the concepts of the generated `Extension Metamodel`, and the second instantiates the concepts of OCCIware Metamodel.

In order to deploy and manage this generated configuration, the cloud developer can interact with the cloud by sending OCCI HTTP requests to OCCIware Runtime extended with the deployed `Extension Connector`. These scripts can be generated from the OCCI configuration model. To execute them, OCCIware Runtime invokes the appropriate `Extension Connector` to create the instance in the cloud. Finally, the created resource is deployed in the cloud.

3.3.2 Generating Cloud Domain-Specific Modeling Studios with OCCIware

As previously explained in Subsection 3.3.1, the OCCIware approach provides a set of tools to design, edit, validate, generate, and manage OCCI artifacts. Concretely, the main goal of OCCIware Studio consists in designing, at the end, a correct OCCI configuration model that conforms to the OCCIware Metamodel. Moreover, the ultimate goal for the cloud developer, the end user of the OCCIware approach, consists in executing this model that represents an eventual running system

in the cloud. In the OCCIware approach, executing a configuration model invokes the OCCIware Runtime to create the different designed entities. During model-driven development, two strategies to execute models are possible [Brambilla 2012]: *Code Generation* and *Model Interpretation*.

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

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

In the OCCIware approach, as shown on the left part of Figure 3.5, both strategies have been implemented. Code generation process allows us to integrate a generator of HTTP requests from OCCI configuration models. These requests can be later sent to OCCIware Runtime to create and deploy OCCI entities. The extensibility of OCCIware Studio lets software engineers implement additional generators to target other existing tools for other purposes such as the generation of deployment plans. Model interpretation is implemented by defining a `Runtime Connector`. Using this connector, we can *(i)* discover a configuration model by mapping a running system from OCCIware Runtime to OCCIware Studio, *(ii)* edit the obtained configuration model, *(iii)* send these modifications to the running system, and finally *(iv)* bring back the changes triggered by the runtime to the model.

OCCIware Studio represents the first model-driven framework to design OCCI artifacts. In addition, thanks to the different proposed generators, OCCIware Studio can be considered as a factory to build cloud domain-specific modeling studios, each one is specific to a particular cloud domain. As shown on the right part of Figure 3.5, once an OCCI extension model is defined, we can proceed to the generation of a Cloud Domain-Specific Modeling Studio (CDSMS) dedicated to a particular cloud domain. This specific studio provides *(i)* an `Extension Metamodel`, *(ii)* an `Extension Validator`, *(iii)* an `Extension Implementation`, and *(iv)* an `Extension Designer`. These generated tools allow the cloud developer to design configurations conform to a specific cloud domain. As OCCIware Studio, a specific studio can support both strategies to execute the designed `Extension` config-

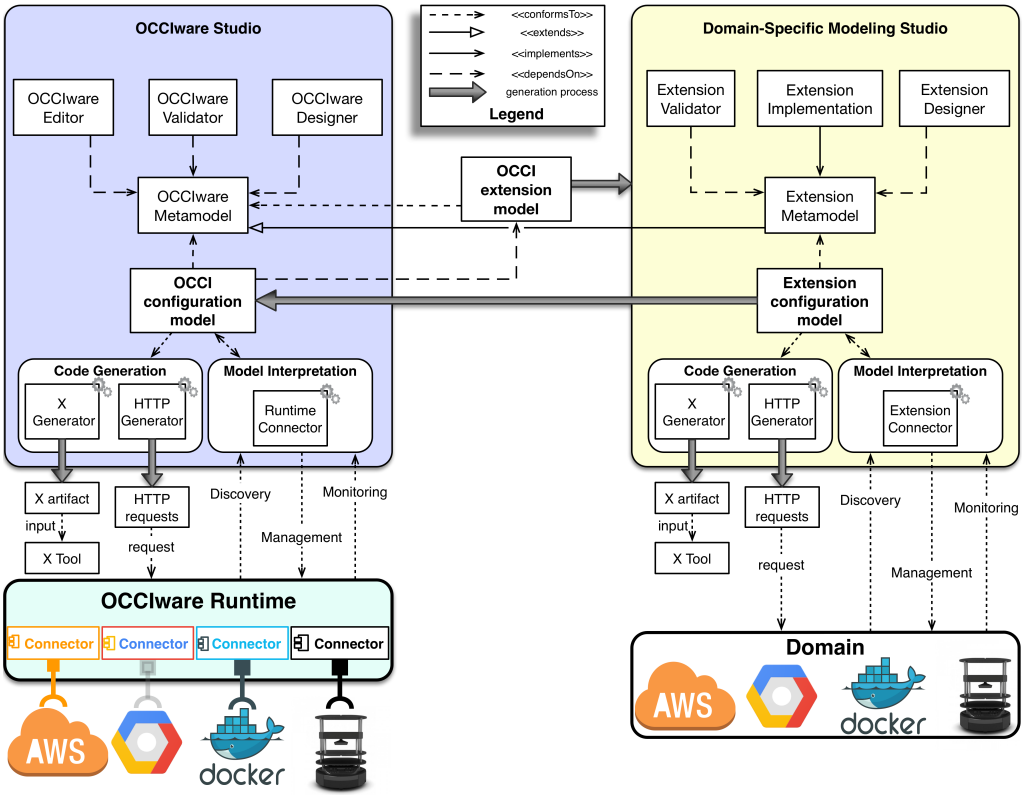


Figure 3.5: Generating Cloud Domain-Specific Modeling Studios with OCCIware.

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

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

3.4 OCCIware Metamodel

Designing is the key activity that must, at first, be addressed to later resolve other encountered challenges such as verifying, generating, and deploying. Therefore, in order to assist OCCI users in modeling different OCCI artifacts, a metamodel for OCCI named OCCIware Metamodel[3] is proposed, as shown in Figure 3.6.

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

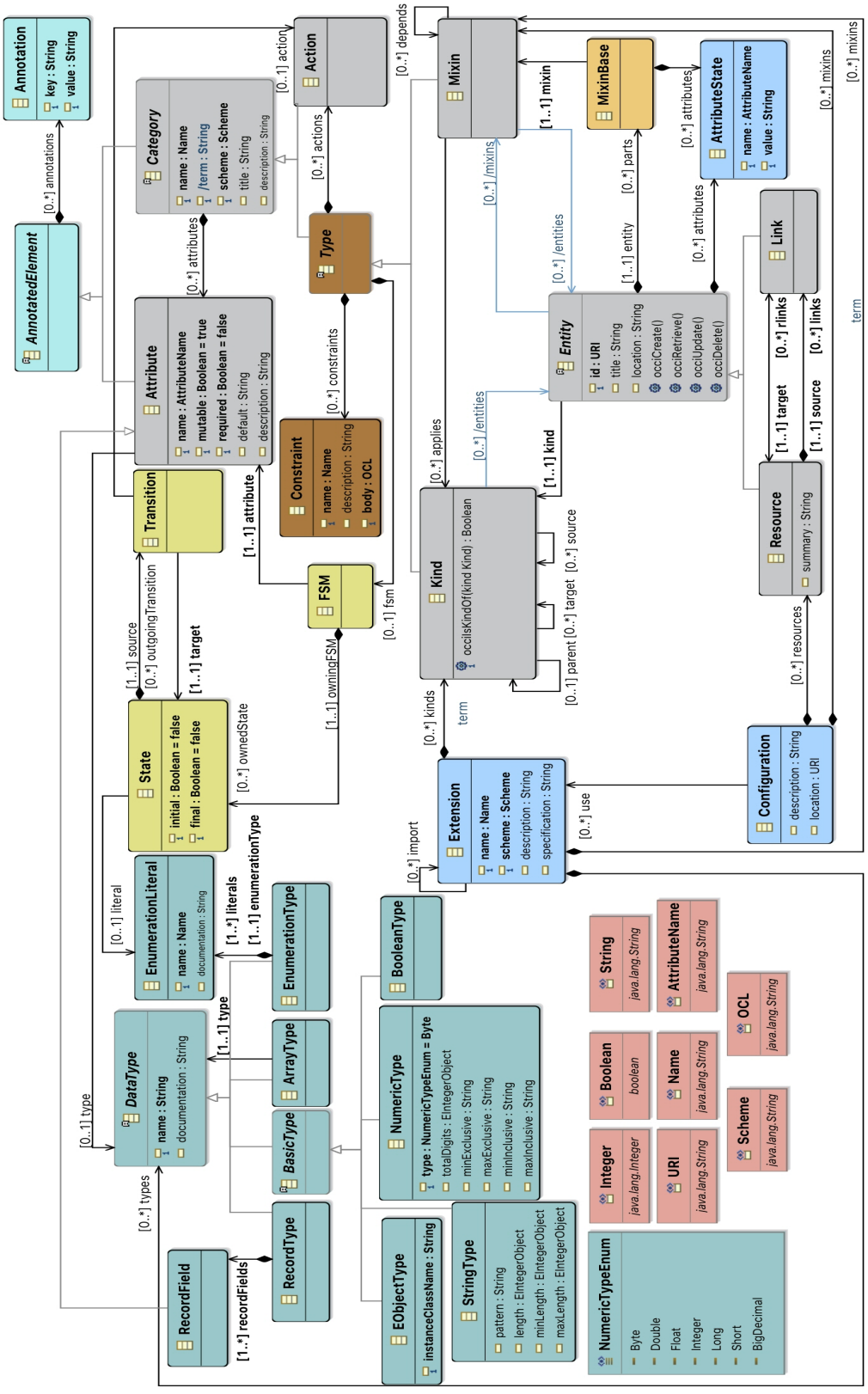


Figure 3.6: Ecore diagram of OCCIware Metamodel.

The entry point to define OCCIware Metamodel was the OCCI Core Model [Nyrén 2016b]. The gray-colored classes in Figure 3.6 show the eight concepts of OCCI Core Model. The blue-colored classes show the added concepts during our previous work [Merle 2015a]. Since then, we continued to extend OCCIware Metamodel in order to meet different needs appeared during its use. The brown-colored classes introduce the added concepts needed to express business constraints related to a particular domain. The orange-colored class represents the required concept to instantiate the mixins in a configuration model. The yellow-colored classes define the concepts used to express the behavior of an OCCI kind/mixin. The cyan-colored classes provide the required concepts to express non-OCCI core information needed to perform several activities such as code generation and model visualization. The green-colored classes define the OCCIware data type system. Finally, the red-colored classes provides a set of Ecore data types required to create correct OCCI artifacts.

In the following, the different concepts of the OCCIware Metamodel with a subset of their associated OCL invariants defining the static semantics are detailed:

- `Extension` represents an OCCI extension, *e.g.,* inter-cloud networking extension [Medhioub 2013], infrastructure extension [Nyrén 2016c], platform extension [Yangui 2013, Yangui 2016, Metsch 2016], application extension [Yangui 2016], SLA negotiation and enforcement [Katsaros 2016], cloud monitoring extension [Ciuffoletti 2016], and autonomic computing extension [Mohamed 2013, Mohamed 2014b, Mohamed 2014a, Mohamed 2015]. `Extension` has a `name`, has a `scheme`, has a `description`, has a `specification`, owns zero or more `kinds`, owns zero or more `mixins`, owns zero or more data `types`, and can `import` zero or more extensions. Each designed extension must, at least, extend the OCCI `Core` extension, the extension-like representation of the OCCI Core Model. The OCCI `Core` extension is composed of three kinds: a root `Entity` kind, and two children kinds, `Resource` and `Link`.

  Definition 1 *Each `Extension` instance must have a unique `scheme` among all `Extension` instances.*

```
context Extension
invariant UniqueScheme:
    Extension.allInstances()->isUnique(scheme);
```

  Definition 2 *The `scheme` of all `kinds` must be equal to the `scheme` of the owning `Extension` instance.*

```
context Extension
    invariant KindsSchemeValid:
        kinds->forAll(k | k.scheme = self.scheme);
```

- `Kind` is an OCCI Core Model concept representing the immutable type of OCCI entities and defines allowed attributes and actions. Single inheritance, using the `parent` relation between `Kind`s, allows us to factorize attributes and actions common to several kinds. The `source` and `target` references specify the sense of relations between link-oriented kinds.

  Definition 3 *Each `Kind` instance must inherit from the `entity kind` instance directly or transitively. The `entity kind` instance is the root of the hierarchy of `Kind` instances.*

```
context Kind
    invariant EntityKindIsRootParent:
        self->closure(parent)->exists(k | k.term = 'entity' and k.
    scheme = 'http://schemas.ogf.org/occi/core#' and k.parent =
    null);
```

  Definition 4 *A `Kind` instance must not overload an inherited attribute.*

```
context Kind
    invariant AttributesNameNotAlreadyDefinedInParent:
        attributes.name->excludesAll(parent->closure(parent).
    attributes.name);
```

- `Mixin` is an OCCI Core Model concept representing cross-cutting attributes and actions that can be dynamically added to an OCCI entity. `Mixin` can be applied to zero or more kinds and can depend on zero or more other `Mixin` instances.

  Definition 5 *The inheritance relation `depends` between `Mixin` instances must form a direct acyclic graph. A mixin instance must not inherit from itself directly or transitively.*

```
context Mixin
    invariant NoCyclicInheritance:
        depends->closure(depends)->excludes(self);
```

- `Type` is an added concept to represent an abstract type inherited by `Kind` and `Mixin` classes. Each type can own zero or more `actions`, zero or more `constraints` and a *`Finite State Machine`* (`FSM`) describing its behavior.

  Definition 6 *Each* `action` *instance must have a unique* `scheme` *among all* `action` *instances in a* `Type` *instance.*

```
context Type
invariant ActionTermUnicity: actions->isUnique(term);
```

  Definition 7 *Each* `constraint` *instance must have a unique* `name` *among all* `constraints` *instances in a* `Type` *instance.*

```
context Type
invariant ConstraintNameUnique: constraints->isUnique(name);
```

- `Action` is an OCCI Core Model concept representing business specific behaviors, such as start/stop a virtual machine, and up/down a network, etc.

- `Constraint` is an added concept to represent a detailed aspect related to a particular cloud computing domain. In fact, each extension targets a concrete cloud computing domain, *e.g.,* IaaS, PaaS, SaaS, pricing, etc. Therefore, there are certainly business constraints related to each domain, which must be respected by configurations that use the extension. For example, all IP addresses of all network resources must be distinct. A `Constraint` has a `name`, a `description` and a `body` that can be defined with Object Constraint Language (OCL) [OMG 2014].

- `Category` is an OCCI Core Model concept and the abstract base class inherited by `Type` and `Action`. Each instance of kind, mixin or action is uniquely identified by both a `scheme` and a `term`, has a human-readable `title`, a `description`, and owns a set of `attributes`.

  Definition 8 *The* `scheme` *of each* `Category` *instance must end with a sharp.*

```
context Category
invariant SchemeEndsWithSharp:
    scheme.substring(scheme.size(), scheme.size()) = '#';
```

- `Attribute` is an OCCI Core Model concept and represents the definition of a customer-visible property, *e.g.,* the hostname of a machine, the IP address of a network, or a parameter of an action. An attribute has one `name`, can have a data `type`, can be (or not) `mutable` (*i.e.,* modifiable by customers), can be (or not) `required` (*i.e.,* value is provided at creation time), can have a `default` value and a human-readable `description`.

- `AnnotatedElement` is an added concept to represent the abstract base class inherited by `Attribute` and `Category`. Each attribute/kind/mixin/action can own zero or more `annotations`.

- `Annotation` is an added concept and represents an additional information that can be attached to an `AnnotatedElement` instance. This mechanism is usually used to limit changing the metamodel. It allows us adding an information that may not be related to the core of the OCCI specifications, but important to some related processes like code generation, model visualization, etc.

- `FSM` is an added concept to model the behavior of OCCI concepts such as state diagrams of OCCI `Kind` instances used in both the Infrastructure [Nyrén 2016c] and Platform [Metsch 2016] extensions. `FSM` describes the behavior of a kind/mixin instance, the current state is stored in a specific `attribute` and can own a set of `states` (`State`).

  Definition 9 *The `type` of a `FSM` `attribute` must be `EnumerationType`.*

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

  Definition 10 *The `attribute` of a `FSM` instance must belong to the `attributes` of the owner `Type` instance.*

```
context FSM
invariant AttributeMustBeDefined:
    self.oclContainer().oclAsType(Type).attributes->includes(self.
    attribute)
```

- `State` is an added concept to model a FSM state of a kind/mixin instance. It can be an `initial` and/or a `final` one. It refers to a `literal` and can own a set of transitions (`outgoingTransitions`).

Definition 11 *The `enumerationType` of a `State literal` is equals to the `type` of the `attribute` of the owner `FSM` instance.*

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

- `Transition` is an added concept to represent a FSM transition from a `source` to a `target` FSM state. When a transition is triggered, an associated `action` is executed.

Definition 12 *The `action` of a `Transition` instance must belong to the `actions` of the owner `Type` instance.*

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

- `DataType` is an added concept to represent an abstract type defining OCCI data types. A `DataType` instance has a `name` and a `documentation`.
- `BasicType` is an added concept to represent an abstract type extending `DataType` meta-class and defining OCCI primitive data types.
- `BooleanType` is an added concept to represent a type extending `BasicType` meta-class and defining OCCI boolean data types.
- `NumericType` is an added concept to represent a type extending `BasicType` meta-class and defining OCCI numeric data types. A numeric type has a concrete `type`, a `totalDigits` value defining the maximal number of digits of a number, a minimal exclusive `minExclusive` value, a maximal exclusive `maxExclusive` value, a minimal inclusive `minInclusive` value, and a maximal inclusive `maxInclusive` value.
- `NumericTypeEnum` is an added concept to represent an enumeration type defining the different OCCIware concrete numeric types. A `NumericType` can be `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`, or `BigDecimal`.

- `StringType` is an added concept to represent a type extending `BasicType` meta-class and defining OCCIware string data types. A `StringType` instance can have a `pattern` value defining a pattern constraint expressed as a regular expression, a `length` value defining a length constraint, a `minLength` value defining the minimum length constraint, and a `maxLength` defining the maximum length constraint.

- `EObjectType` is an added concept to represent a type extending `BasicType` meta-class and defining OCCIware Java data types such as `URI`, `Date`, etc.

- `ArrayType` is an added concept to represent array data types. An `ArrayType` instance has a `type` defining its type.

- `EnumerationType` is an added concept to represent enumeration data types. An `EnumerationType` instance owns a set of `literals`.

- `EnumerationLiteral` is an added concept to represent an enumeration literal. An `EnumerationLiteral` instance has a `name` and a `documentation`.

- `RecordType` is an added concept to represent record data types. A `RecordType` instance owns a set of record fields (`recordFields`).

- `RecordField` is an added concept to represent a record field. It extends the `Attribute` meta-class.

- `Integer` is an Ecore data type defining the primitive `Integer` type.

- `Boolean` is an added concept to represent an Ecore data type defining the primitive `Boolean` type.

- `String` is an added concept to represent an Ecore data type defining the primitive `String` type.

- `URI` is an added concept to represent an Ecore string-based data type extended with a pattern constraint that conforms to the *Uniform Resource Name* (URN) syntax [Moats 1998].

- `Name` is an added concept to represent an Ecore string-based data type extended with the following pattern
  ‘‘`[a-zA-Z][a-zA-Z0-9_-]*`’’. This data type allows us to initialize a valid `name` attribute to the OCCI constructs and, thus, deduce a term conforms to the OCCI RESTful HTTP Rendering specification [Nyrén 2016a].

- `AttributeName` is an added concept to represent an Ecore string-based data type extended with the following pattern ‘‘`[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)+`’’. It allows us to create `AttributeName` instances conform to the OCCI Text Rendering specification [Edmonds 2016].
- `Scheme` is an added concept to represent an Ecore string-based data type extended with a pattern constraint conforms to the Uniform Resource Identifier (URI) syntax [Berners-Lee 1998].
- `OCL` is an added concept to represent an Ecore string-based data type allowing us to create correct OCL expressions.
- `Configuration` is an added concept to represent a running OCCI system. `Configuration` owns zero or more `resources` (and transitively `links`), and `use` zero or more extensions. For a given configuration, the kind and mixins of all its entities (resources and links) must be defined by `used` extensions only. This avoids a configuration to transitively reference a type defined we do not know where.

  Definition 13 *The `kind` of all `resources` of a configuration must be defined by an `extension` that is explicitly `used` by this configuration.*

```
context Configuration
invariant AllResourcesKindsInUse:
    use->includesAll(resources.kind.oclContainer());
```

  Definition 14 *The `target` resource of all `links` of all `resources` of a configuration must be a resource of this configuration.*

```
context Configuration
invariant AllResourcesLinksTargetsInConfiguration:
    resources.links.target->forAll(r | r.oclContainer() = self);
```

- `Resource` is an OCCI Core Model concept and represents any cloud computing resource, such as a virtual machine, a network, and an application. A `Resource` owns a set of `links`.

  Definition 15 *The `kind` of a `Resource` instance must inherit from the `resource kind` instance directly or transitively.*

```
context Resource
invariant ResourceKindIsInParent:
    kind->closure(parent)->exists(k | k.term = 'resource' and k.
    scheme = 'http://schemas.ogf.org/occi/core#');
```

- `Link` is an OCCI Core Model concept and represents a relation between two resources, such as a virtual machine connected to a network and an application hosted by a virtual machine. A `Link` instance refers to both a `source` and `target` resource.

  Definition 16 *The `kind` of a `Link` instance must inherit from the `link kind` instance directly or transitively.*

```
context Resource
invariant ResourceKindIsInParent:
    kind->closure(parent)->exists(k | k.term = 'link' and k.scheme
    = 'http://schemas.ogf.org/occi/core#');
```

- `Entity` is an OCCI Core Model abstract concept. Each OCCI entity (resource or link) owns zero or more `attributes`, such as its unique `id`entifier, the host name of a virtual machine, the Internet Protocol address of a network. In addition, each OCCI entity is strongly typed by a `Kind` and a set of `Mixin` instances. As OCCI is a REST API, it gives access to cloud resources via classical CRUD operations (*i.e., Create*, *Retrieve*, *Update*, and *Delete*).

  Definition 17 *The `kind` of an `Entity` instance must be compatible with one `applies` kind instance of each mixin `parts` of this entity.*

```
context Resource
invariant KindCompatibleWithOneAppliesOfEachMixin:
    parts.mixin->forAll(m | m.applies->notEmpty() implies m.
    applies->exists(k | kind->closure(parent)->includes(k)));
```

- `AttributeState` is an added concept to represent an instantiated OCCI attribute. An `AttributeState` instance has a `name` and a `value`.

- `MixinBase` is an added concept strongly typed by a `mixin`. It represents an instantiated mixin and allows us to instantiate the attributes of the referenced `mixin` outside the owner `entity` in order to separate the `entity` attributes from the `mixin` ones. A `MixinBase` can own zero or more `attributes`.

3.5 OCCIware Studio

Once the OCCIware Metamodel was defined, it has been tooled with OCCIware Studio, which is a set of plugins for the Eclipse [ecl] IDE. Figure 3.7 shows all the main features of OCCIware Studio:

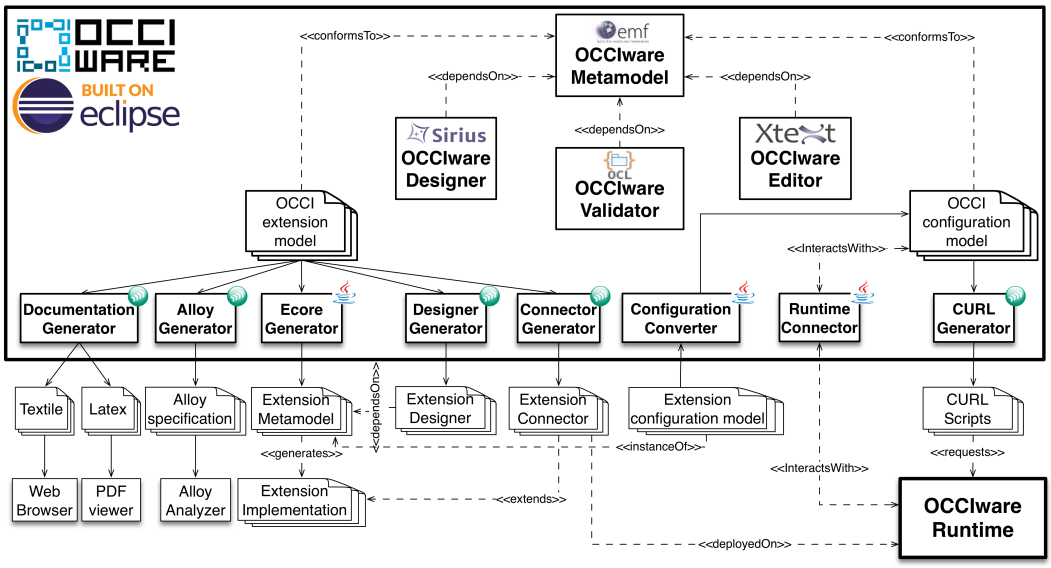


Figure 3.7: OCCIware Studio Features.

- **OCCIware Designer** is a graphical modeler to create, modify, and visualize both OCCI extensions and configurations. The OCCI standard does not define any standard notation for the graphical or textual concrete syntax. This tool is implemented on top of the Eclipse Sirius framework [sir].

- **OCCIware Editor** is a textual editor for both OCCI extensions and configurations. Our OCCI textual syntax is described in [Merle 2015b]. This tool is implemented on top of the Eclipse Xtext framework [xte 2016].

- **OCCIware Validator** is a tool to validate both OCCI extensions and configurations. This tool checks all the constraints defined in the OCCIware Metamodel, *i.e.,* both Ecore and OCL ones.

- **Textile Documentation Generator** is a tool to generate a Textile documentation from an OCCI extension model. Textile is a Wiki-like format used for instance by GitHub projects. This tool is implemented on top of the Eclipse Acceleo framework [acc].

- **Latex Documentation Generator** is a tool to generate a LaTeX documentation from an OCCI extension model. It allows us to later generate a portable document describing the OCCI extension. This tool is implemented on top of the Eclipse Acceleo framework.

- **Ecore Generator** is a tool to generate the Ecore metamodel and its associated Java-based implementation code from an OCCI extension. As shown in the left part of Figure 3.8, designing a new OCCI extension consists in extending the OCCI `Core` extension. Designing an OCCI configuration consists in defining an instance of an OCCI extension and represents a cloud architecture already deployed or to deploy.

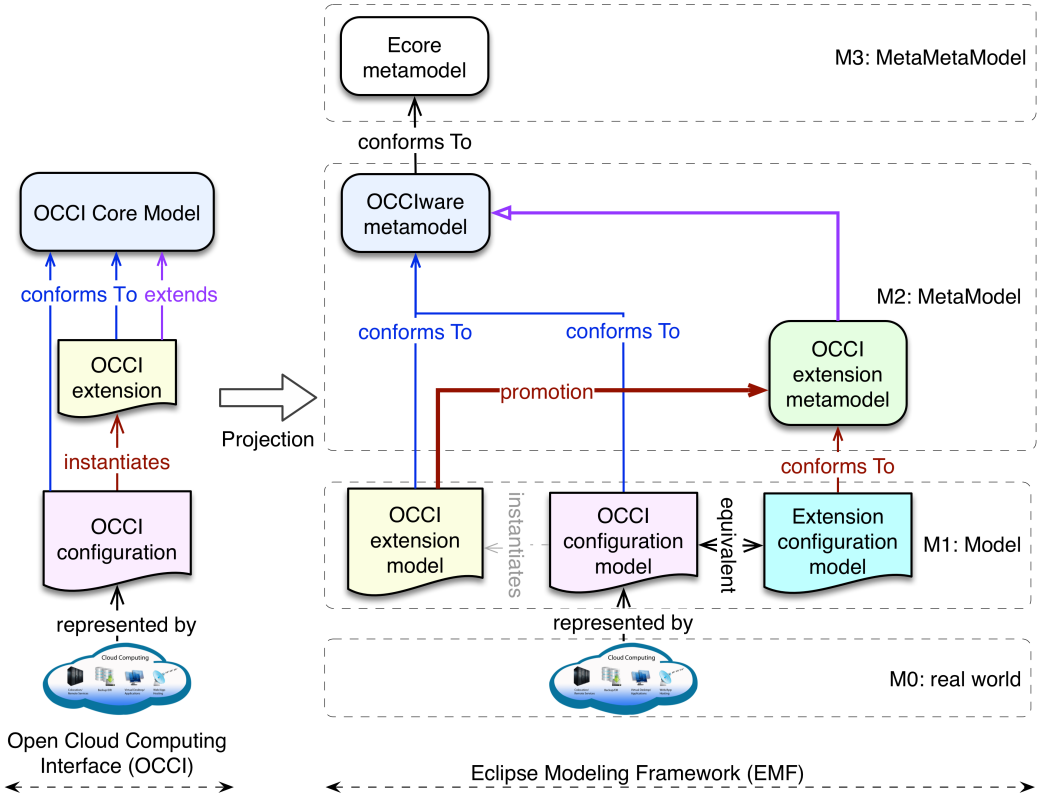


Figure 3.8: Projection of OCCI to EMF.

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

To do that, it was necessary to map different OCCI concepts into a modeling framework to benefit from the available facilities for building tools based on a metamodel (the right part of Figure 3.8). EMF was chosen to embed OCCI and, thus, the OCCIware Metamodel was proposed as a precise meta-

model for OCCI [Merle 2015a]. Therefore, we can define either an OCCI extension model or an OCCI configuration model that conform to the OCCIware Metamodel. However, the current tooling in EMF does not allow us to encode that: an OCCI configuration is an *"instantiation"* of an OCCI extension. For that, OCCIware proposes, using the **Ecore Generator** tool, to promote the OCCI extension model by translating it into an Ecore metamodel, extending the OCCIware Metamodel. Consequently, we can design an Extension configuration model, instance of this generated metamodel. Later, we can deduce a semantically equivalent OCCI configuration model, instance of the OCCIware Metamodel.

This tool is directly implemented in Java. In the following, the generation process of OCCIware Metamodel concepts into the EMF concepts is detailed:

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

Table 3.1 outlines the mapping process of OCCIware Metamodel concepts into EMF concepts.

- **Alloy Generator** is a tool to generate an Alloy specification from an OCCI extension model. Alloy is a lightweight formal specification language based on the first-order relational logic [Jackson 2012]. We are able to analyze OCCI

Table 3.1: The Mapping Process of OCCI Concepts into EMF Concepts.

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

extensions formally with the Alloy analyzer. The Alloy analyzer is a solver that takes the constraints of a model and finds structures that satisfy them. We used it to explore the model by generating sample OCCI configurations, and also to check properties of the OCCI extension by generating counterexamples. The generated OCCI configurations can be displayed graphically with the OCCIware Designer. Alloy Generator is the core of the fclouds approach, my contribution that I detail in Chapter 5. This tool is implemented on top of the Eclipse Acceleo framework.

- **Connector Generator** is a tool to generate the OCCI connector implementation associated to an OCCI extension. This generated connector code extends the generated Ecore implementation code. This connector code must be completed by software developers to implement concretely how OCCI CRUD operations and the specific actions must be executed on a real cloud infrastructure. Later, this generated connector will be deployed on OCCIware Runtime. This tool is implemented on top of the Eclipse Acceleo framework.

- **Designer Generator** is a tool to generate a graphical extension-specific designer from an OCCI extension. This designer can be later customized to be able to represent the concepts related to the extension domain. This tool is implemented on top of the Eclipse Acceleo framework and generate Eclipse Sirius models.

- **CURL Generator** is a tool to generate a CURL-based script from an OCCI configuration model. These generated scripts contain HTTP requests to instantiate OCCI entities into any OCCI-compliant runtime. These scripts are used for offline deployment. This tool is implemented on top of the Eclipse

Acceleo framework.

- **Runtime Connector** is a tool to synchronize OCCI configuration models with running OCCI configurations hosted by any OCCI-compliant runtime. This connector allows cloud developers to introspect an OCCI runtime in order to build the corresponding OCCI configuration model, then update this model and send changes back to the OCCI runtime. This tool integrates the jOCCI API[4], a Java library implementing transport functions for rendered OCCI queries.

- **Configuration Converter** is a tool to translate an Extension configuration model into an OCCI configuration model (the **equivalent** relation in Figure 3.8). This tool allows us to reuse the tools specific to OCCI artifacts such as the CURL Generator to deploy later the configuration into an OCCI-compliant runtime. This tool is directly implemented in Java.

3.6 OCCIware Runtime

To enact OCCI configuration models, we adopt the Models@run.time approach [Blair 2009] that extends the use of modeling techniques beyond the design and implementation phases. It seeks to extend the applicability of models and abstractions to capture the behavior of the executing environment. End users of OCCIware Studio require interaction with cloud APIs to create, retrieve, update and delete cloud resources. Therefore, OCCIware Runtime is implemented as a generic Java implementation of OCCI, available as an open-source project[5]. The OCCIware Runtime can be deployed as a standalone server including an embedded Jetty server or as a Java library that is based on Java Servlet API specification.

The OCCIware Runtime is composed of four main parts, as illustrated in Figure 3.9.

- **OCCI Server** that implements the following OCCI specifications: *(i)* OCCI HTTP Protocol [Nyrén 2016a], *(ii)* OCCI Text Rendering [Edmonds 2016], OCCI JSON Rendering [Nyrén 2016d] based on the Jackson[6] library, and *(iii)* OCCI Core Model [Nyrén 2016b] based on the OCCIware Metamodel.

- **OCCI Extensions** that represent the OCCIware-based modeling of concrete cloud domains such as OCCI `Infrastructure` [Nyrén 2016c], OCCI

[4] https://github.com/EGI-FCTF/jOCCI-api
[5] https://github.com/occiware/MartServer
[6] https://github.com/FasterXML/jackson

`Platform` [Metsch 2016], OCCI `SLA` [Katsaros 2016], Docker [Paraiso 2016], cloud mobile robotics [Merle 2017], etc.

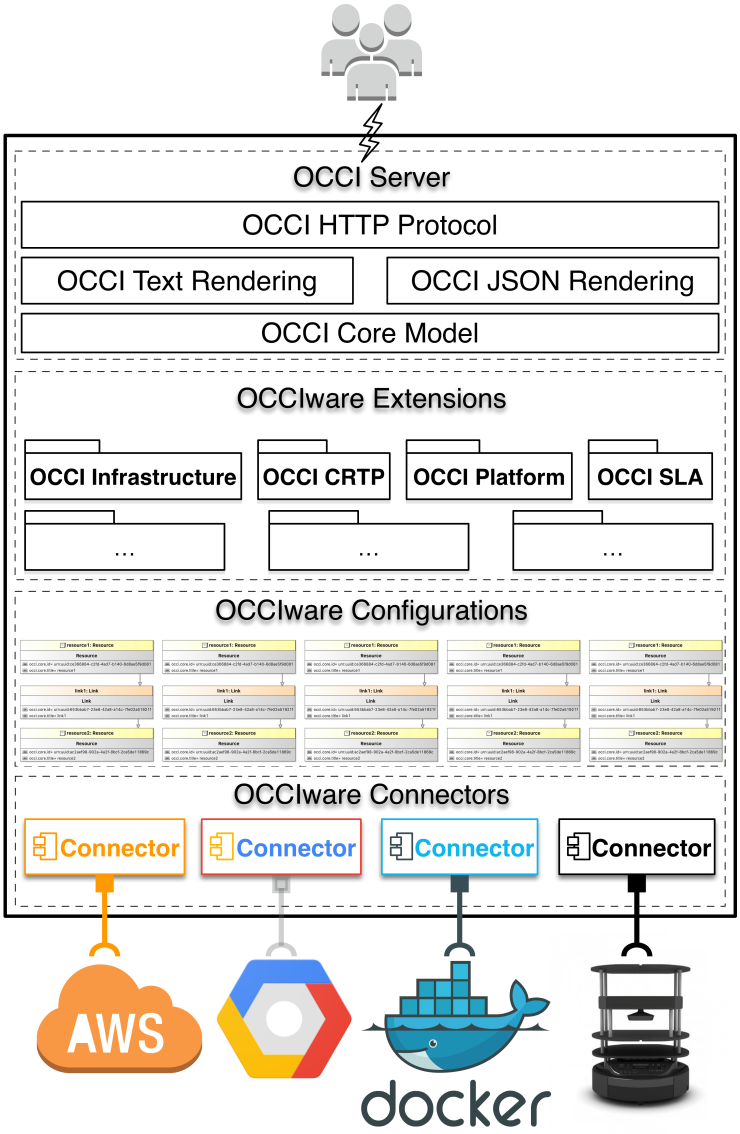


Figure 3.9: OCCIware Runtime Architecture.

- **OCCI Configurations** that are instances of OCCIware configuration, such as a configuration using OCCI `Infrastructure` and containing a running virtual machine with 4 cores, 3.2 GHz and 16 GiB.
- **OCCI Connectors** that consist in the pivot between OCCIware extensions/configurations and CRM-APIs. Each connector is dedicated to a specific cloud domain, *e.g.,* AWS, GCP, Docker, or mobile robotics. It implements the

corresponding OCCI extension. It executes CRUD (Create, Retrieve, Update and Delete) operations and extension-specific actions, such as *start compute* for the OCCI `Infrastructure` extension. The OCCIware Runtime is extensible by design: supporting a new kind of cloud resources consists of adding a new connector.

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

3.7 Evaluation of OCCIware Studio

This section validates the OCCIware approach. We discuss how OCCIware addresses the different requirements listed in Section 3.1. At first, we show the different OCCI extensions defined by the OGF's OCCI working group and implemented with OCCIware. We particularly focus on the `Infrastructure` extension by showing how the cloud developer leverages the generated tooling around this extension to create/manage his/her configuration models with OCCIware Studio and deploy them in the cloud. Then, we illustrate the different OCCIware approach usages presented in Section 3.3 by presenting five major use cases that apply OCCIware.

3.7.1 Implementation of a Catalog of Standard OGF's OCCI Extensions

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

3.7.1.1 The OCCI `Infrastructure` Extension

OCCI Infrastructure [Nyrén 2016c] defines compute, storage and network resource types and associated links. To design the `Infrastructure` extension, the OCCI-

ware architect can use OCCIware Designer and/or OCCIware Editor.

This extension defines five kinds (`Network`, `Compute`, `Storage`, `StorageLink` and `NetworkInterface`), six mixins (`Resource_Tpl`, `IpNetwork`, `Os_Tpl`, `SSH_key`, `User_Data`, and `IpNetworkInterface`), and around twenty data types (`Vlan` range, `Architecture` enumeration, various status enumerations, etc.). Figure 3.10 shows a subset of this extension. The complete one is available here[7].

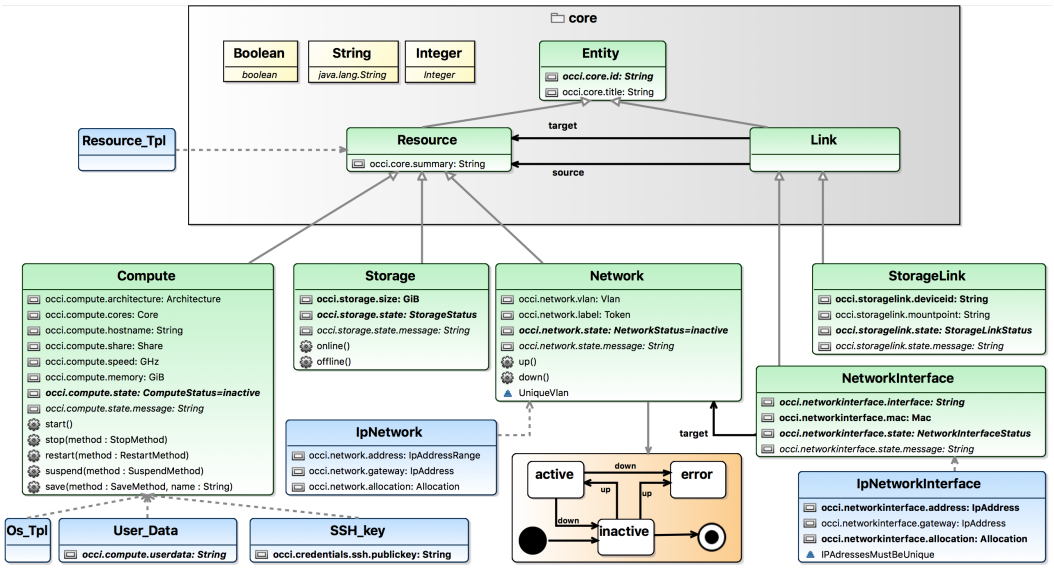


Figure 3.10: OCCI Infrastructure Extension Model.

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

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

The orange-colored box in Figure 3.10 illustrates the state diagram of a `Network` instance and describes its behavior. As shown previously in Section 3.4, the OCCIware Metamodel provides the required concepts to describe the behavior of each kind/mixin. In addition, it allows us defining extension-specific constraints. For

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

example, the following OCL constraint specifies that each `Network` instance must have a unique `VLAN`.

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

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

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

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

Once the extension is defined, the generation process of `Infrastructure` Tooling may be triggered. It generates four main elements: *(i)* the `Infrastructure Metamodel`, *(ii)* the Java-based `Infrastlandscructure Implementation`, *(iii)* `Infrastructure Connector`, and *(iv)* `Infrastructure Designer`.

Listing 3.1 shows a subset of the generated `Network` connector class. It extends the `NetworkImpl` class generated by the EMF tooling and contains the OCCI specific callback methods for the CRUD operations and all `Network` kind-specific actions (*i.e.,* `up` and `down`). The generated code of specific actions is deducted from the defined FSM on the `Network` kind.

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

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

```
        if ( true ) {
          // TODO: Transition active -down-> inactive
          setState(NetworkStatus.INACTIVE);
        } else {
          // TODO: Transition active -down-> error
          setState(NetworkStatus.ERROR);
        }
      }
    }
}
```

Listing 3.1: The Generated `Network` Connector Class.

Once the generation step is achieved, the software developer can complete the generated connector classes by updating their methods implementations (`TODO` sections in Listing 3.1) with business code related to targeted API. For the `NetworkConnector` class, the software developer completes the code to trigger that the OCCI `Network` resource was created (`occiCreate`), will be retrieved (`occiRetrieve`), was updated (`occiUpdate`) and will be deleted (`occiDelete`). In addition, he/she completes the generated methods (`up` and `down`) related to specific actions defined in the `Network` kind. The completed connector code must be later deployed on the OCCIware Runtime. Then, the software developer can proceed to the customization of the generated `Infrastructure Designer` which will be used to create `Infrastructure` configuration models as shown in Figure 3.11.

Figure 3.11: An Infrastructure Configuration Model.

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

Using OCCIware Studio enriched with the `Infrastructure Tooling`, cloud

developers can design an OCCI `Infrastructure` configuration model conforms to the `Infrastructure Metamodel`. Figure 3.11 illustrates a small infrastructure configuration composed of a compute (`vm1`) connected to a network (`network1`), via an OCCI link (green-colored box), the network interface (`ni1`). As this configuration uses an IP-based network, the `Network` resource and the `NetworkInterface` link have an `IpNetwork` and `IpNetworkInterface` mixin, respectively. Each OCCI entity is configured by its attributes, *e.g.,* `vm1` has the `vm1` hostname, an `x64`-based architecture, `4` cores, and `4` GiB of memory.

To benefit from the OCCI-compliant tools defined in the OCCIware Studio, an `Infrastructure` configuration model must be translated into an OCCI configuration model that conforms to the OCCIware Metamodel. Figure 3.12 shows a generated OCCI configuration model from the Infrastructure configuration model. As shown in the palette of the Infrastructure Designer (right part of Figure 3.11), the Infrastructure Designer allows cloud developers to create an instance of Infrastructure Metamodel such as `Compute`, `Network`, and `Storage`. However, in the palette of the OCCIware Designer (right part of Figure 3.12), the cloud developer can only create instances of OCCIware Metamodel `Resource` and `Link` classes.

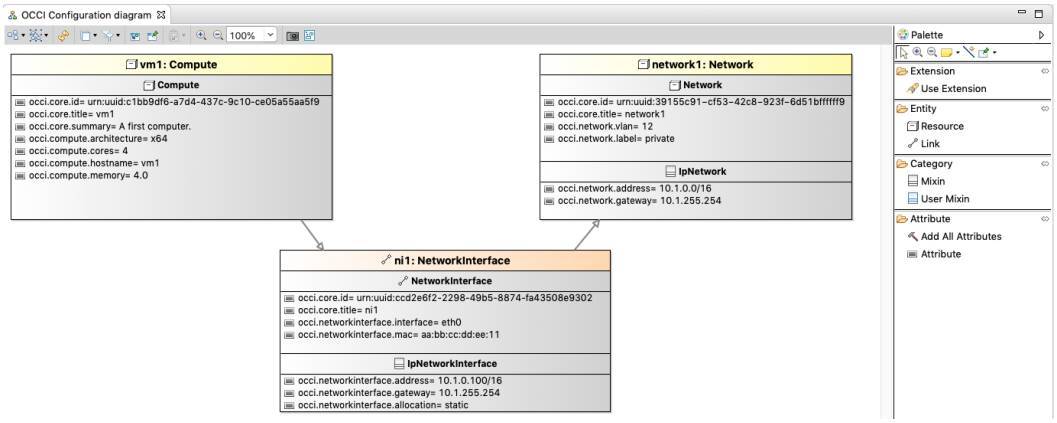


Figure 3.12: An OCCI Configuration Model.

In order to deploy and manage the generated OCCI configuration models, cloud developers interact with the cloud by sending OCCI HTTP requests to OCCIware Runtime. These requests can be automatically generated as CURL scripts using the CURL Generator tool. Listing 3.2 shows the CURL script that requests OCCIware Runtime via both OCCI HTTP Protocol [Nyrén 2016a] and OCCI Text Rendering [Edmonds 2016] to create the `network1` instance. Then, OCCIware Runtime invokes the `occiCreate()` method of the `NetworkConnector` class, which implements how to create the considered network instance in the cloud. Finally, the

created `Network` resource is deployed in the cloud.

```
OCCI_SERVER_URL=$1

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

Listing 3.2: The generated CURL script to create a `Network` instance

The proposed tooling around the OCCI `Infrastructure` extension allows us, with our industrial partner Scalair [scaa], to implement a Java connector for VMware API. The whole tooling around Infrastructure extension is available here[8]. In the future, we will target additional CRM-APIs such as AWS, OpenStack, GCP, etc.

3.7.1.2 The OCCI `CRTP` Extension

OCCI CRTP [Drescher 2016] defines a set of preconfigured instances of the OCCI compute resource type. It extends the Infrastructure extension. Figure 3.13 shows the OCCI CRTP extension designed with OCCIware Studio. Its tooling is available here[9].

3.7.1.3 The OCCI `Platform` Extension

OCCI Platform [Metsch 2016] defines application and component resource types and associated links. Figure 3.14 shows the OCCI Platform extension designed with OCCIware Studio. Its tooling is available here[10].

[8] https://github.com/occiware/OCCI-Studio/tree/master/plugins/org.eclipse.cmf.occi.infrastructure

[9] https://github.com/occiware/OCCI-Studio/tree/master/plugins/org.eclipse.cmf.occi.CRTP

[10] https://github.com/occiware/OCCI-Studio/tree/master/plugins/org.eclipse.cmf.occi.platform

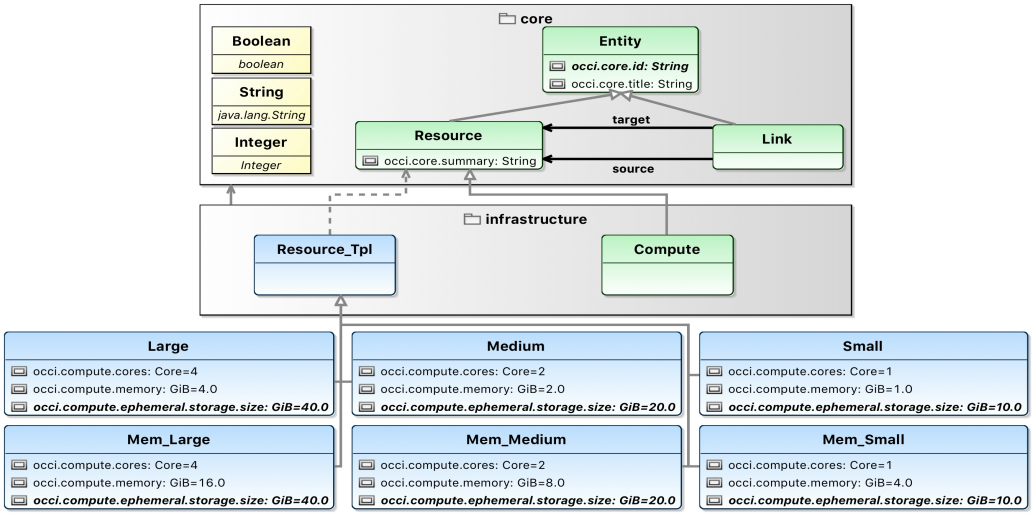


Figure 3.13: OCCI CRTP Extension Model.

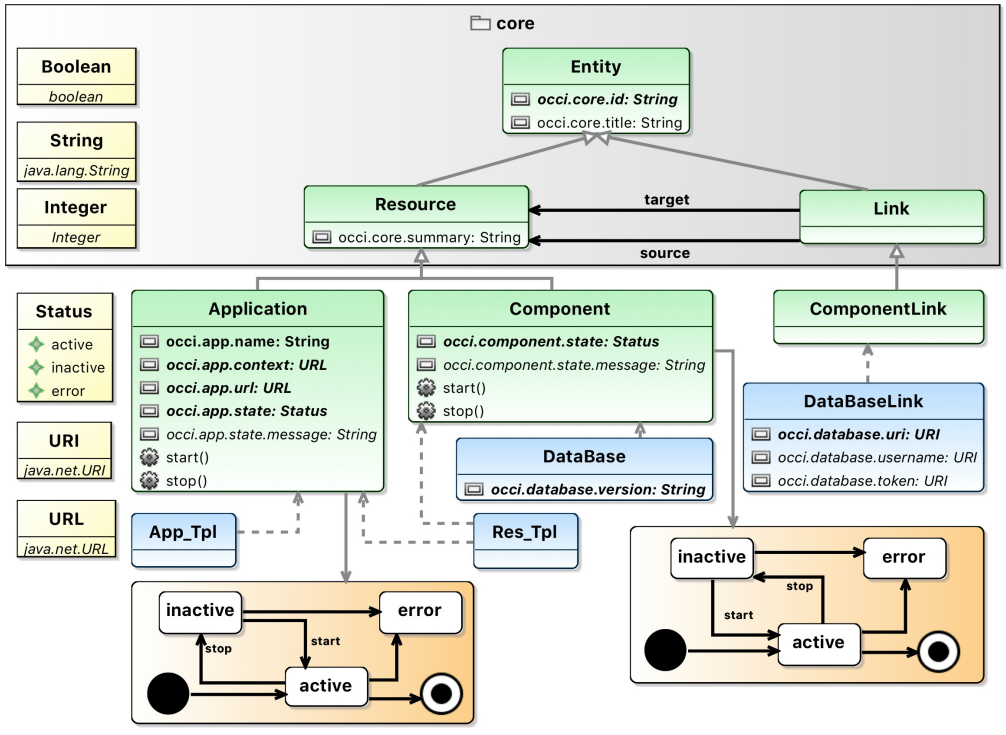


Figure 3.14: OCCI Platform Extension Model.

3.7.1.4 The OCCI SLA Extension

OCCI SLA [Katsaros 2016] defines OCCI types for modeling service level agreements. Figure 3.15 shows the OCCI SLA extension designed with OCCIware Studio. Its tooling is available here[11].

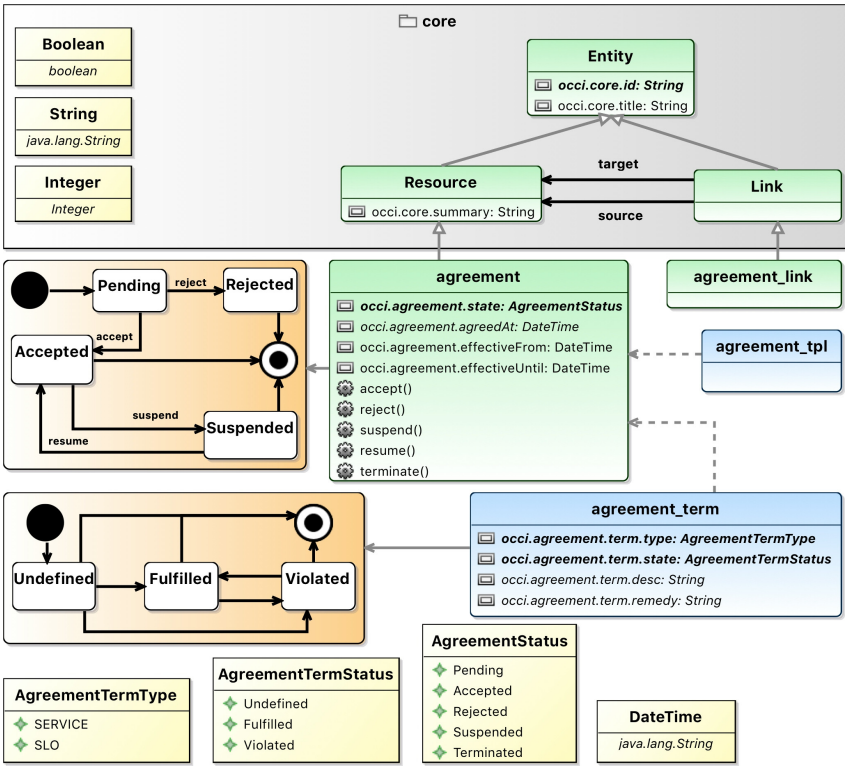


Figure 3.15: OCCI SLA Extension Model.

3.7.1.5 The OCCI Monitoring Extension

OCCI Monitoring [Ciuffoletti 2016] is a draft specification and defines sensor and collector types for monitoring cloud systems. Figure 3.16 shows the OCCI Monitoring extension designed with OCCIware Studio. Its tooling is available here[12].

[11] https://github.com/occiware/OCCI-Studio/tree/master/plugins/org.eclipse.cmf.occi.sla

[12] https://github.com/occiware/OCCI-Studio/tree/master/plugins/org.eclipse.cmf.occi.monitoring

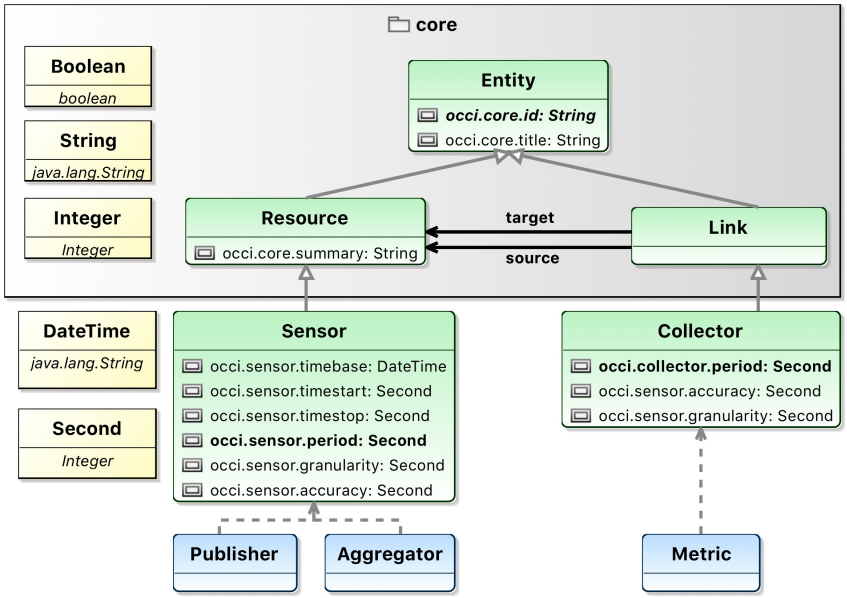


Figure 3.16: OCCI Monitoring Extension Model.

3.7.2 Five OCCIware Use Cases

In this subsection, five major use cases of the OCCIware approach are illustrated.

3.7.2.1 Cloud Simulation with OCCIware

This work [Ahmed-Nacer 2016a, Ahmed-Nacer 2017] provides a methodology for the simulation of OCCI configurations. The simulation technology has increasingly become popular as it allows users to evaluate their algorithms and applications before deploying them on a real cloud environment. This work reuses CloudSim [Calheiros 2011], a generalized and extensible simulation framework that allows seamless modeling, simulation, and experimentation of emerging cloud computing infrastructures and application services. CloudSim allows users to test the performance of a newly developed application service in a controlled environment. Moreover, CloudSim allows a user to model and simulate all the cloud infrastructure resources. The main idea of this use case consists in defining an OCCI extension named `Simulation` which extends the `Infrastructure` extension. The extension defines two notions: *a resource to simulate* represents the resource to be simulated, and *a simulation resource* represents the resource which performs the simulation activity. Two main artifacts are generated for this use case: `Simulation Metamodel` and `Simulation Designer`. Once a `Simulation` configuration model is defined, it can be later verified, and analyzed by the CloudSim tool. The simulation activity evaluates the configuration using some metrics such as the percentage of used

memory, the percentage of used disk, and the average of the CPU utilization.

3.7.2.2 Cloud Mobile Robotics with OCCIware

This use case [Merle 2017] illustrates the convergence of cloud computing and robotics platforms. It introduces *Open Mobile Cloud Robotics Interface* (OMCRI), a Robot-as-a-Service vision based platform, which offers a unified easy access to remote heterogeneous mobile robots. OMCRI is a new OCCI extension for mobile cloud robotics. This extension defines three kinds of heterogeneous robots: Lego Mindstorm NXT 2, Turtlebot, and Parrot AR.Drone. It introduces a set of mixins that customize the sensors and the actuators of each robot. An OMCRI connector has been developed and deployed on the OCCIware Runtime. End users can create OMCRI configuration models to manage the mobile robots. They interact with mobile robots by sending OCCI HTTP requests to the OCCIware Runtime. These requests may be: creating a mobile robot resource, updating its attributes, and executing an action of a mobile robot. In this work, an evaluation to measure the performance, stability, and scalability of the OMCRI has been performed. Evaluation tests have shown that OMCRI overhead remains low and that OMCRI does not add any latency. Figure 3.17 shows an OMCRI configuration modeled using the customized OMCRI Designer.

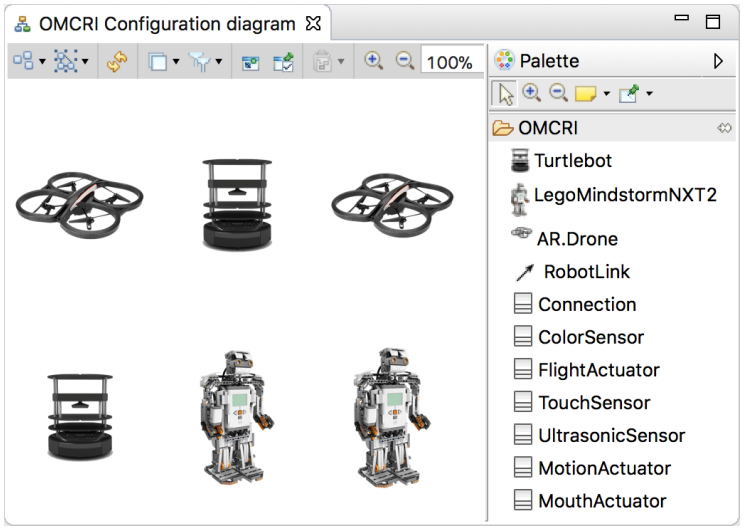


Figure 3.17: OMCRI Designer.

3.7.2.3 Docker Management with OCCIware

In this work [Paraiso 2016], authors present a model-driven management of Docker [doc] containers based on OCCIware Studio. Docker is a technology

used for developing, deploying and executing applications packaged into containers. Docker lacks of deployability verification tool for containers at design time. In addition, the synchronization between the designed containers and those deployed was still a major challenge for the Docker technology. Finally, Docker did not provide a mechanism to reconfigure the container resources at runtime. To resolve these issues, authors refer to the OCCIware approach by proposing Docker Studio[13], an OCCI-based studio for the Docker technology. This work defines a Docker extension using OCCIware Studio. To resolve the lack of verification challenge in Docker, this work reuses the `Constraint` concept added in the OCCIware Metamodel to define constraints specific to the Docker domain such as the not permitted bidirectional or closed loop link between Docker *Containers*. By defining this kind of information and validating it using the OCCIware Validator, the end user is now sure that his/her designed configuration is correct and it is ready to be deployed. Then, a Docker Designer is generated and customized to provide a high-level abstraction for Docker containers that is used for reasoning and managing large container deployments in the cloud. As shown in Figure 3.18, Docker Designer allows representing a human understandable description of some aspects of a running Docker system. In addition, to resolve the synchronization issue, a specific generator for the Docker technology has been provided in order to generate Docker artifacts (Docker *Command-Line Interface* (CLI) commands, Docker Compose file, Docker Swarm configurations). These generated artifacts are used for online deployment. Moreover, to ensure the synchronization from the running system to the Docker model, a *Docker Connector* has been developed. This connector updates the model elements according to the running system changes. Finally, to ensure the resource management at runtime, the developed connector implements the observer/listener design pattern. Therefore, the connector relies on a notification mechanism that reports events when a model element has been changed.

3.7.2.4 Managing Cloud Applications with OCCIware

In this work [Korte 2018], authors focus on modeling cloud applications with OCCI. OGF defines extensions that target the requirements of different cloud service levels, such as IaaS and PaaS. However, no concrete use cases and implementations have been provided around the OCCI Platform extension. This behavior is due to several issues. At first, the lifecycle for the `Component` and `Application` resources as defined in the OCCI Platform specification is incomplete (P1 issue). Moreover, the OGF provides two separate OCCI extensions for the Infrastructure and Platform domains, but it misses to define the connection between them (P2 issue). In addition,

[13] Available here https://github.com/occiware/Docker-Studio

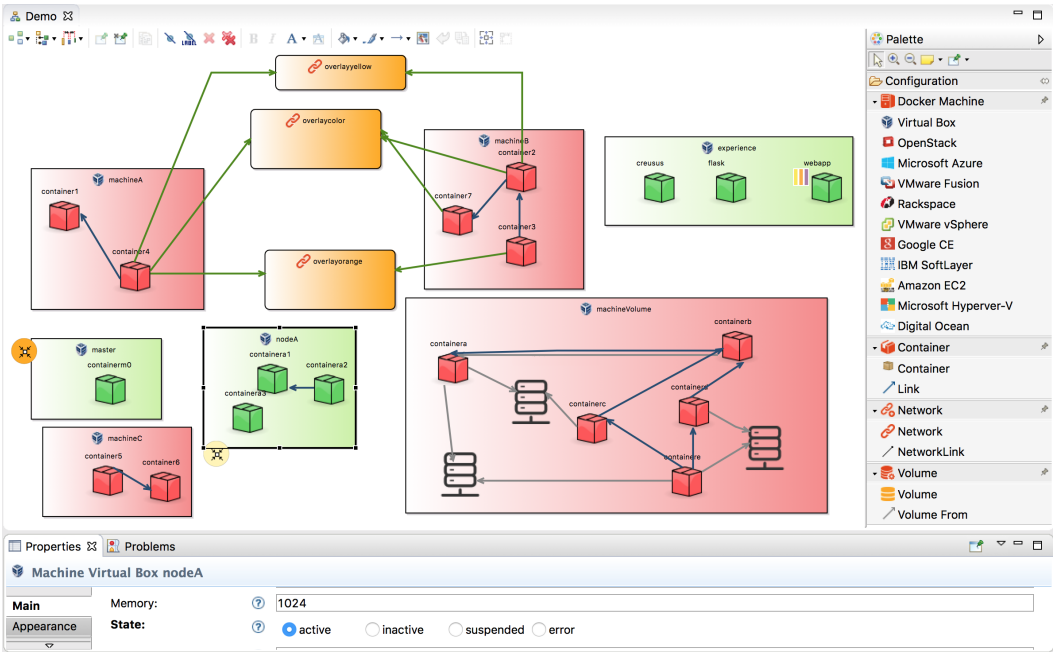


Figure 3.18: Docker Designer.

the current version of the OCCI specification does not define how `Components` and `Applications` can be managed throughout their lifecycle and if and how additional tooling, *e.g.,* configuration management tools, can be integrated (P3 issue). Finally, the current version of the OCCI specification lacks of any real-world use case for the application of the Platform extension (P4 issue). To tackle these issues, authors define the *Model-Driven Configuration Management of Cloud Applications with OCCI* (MoDMaCAO) framework. It addresses the first issue P1 by enhancing the OCCI Platform extension via additional lifecycle `States` and `Actions`. Furthermore, they resolve the P2 issue by introducing a new `Link` kind to be able to connect `Components` of the OCCI Platform extension to `Compute` resources of the OCCI Infrastructure extension. In addition, they define a new OCCI extension to be able to model application components that are managed with help of a configuration management tool (addressing the P3 issue). They demonstrate the feasibility of the defined extension by modeling five different distributed cloud applications (a Client/Server application, a distributed MongoDB database, the popular LAMP web-application stack, a distributed Cassandra database, and an Apache Spark cluster) and finally provide a framework for implementing model-driven configuration management with different configuration management tools such as Ansible and Roboconf, thereby addressing P4. Figure 3.19 shows a LAMP configuration modeled using the LAMP designer.

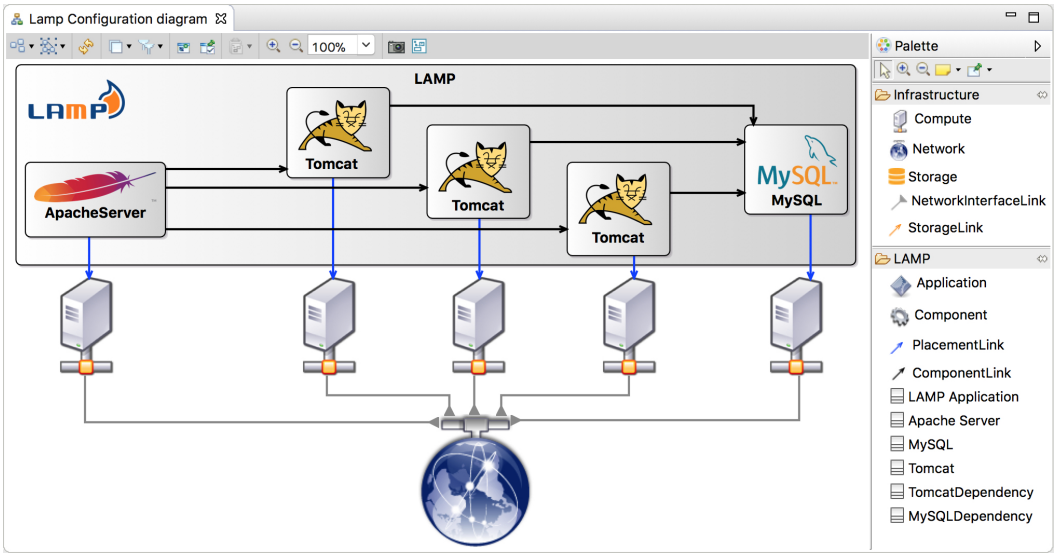


Figure 3.19: LAMP Designer.

3.7.2.5 Modeling *Google Cloud Platform* with OCCIware

In this work [Challita 2018a], I define a precise model, a.k.a an OCCI extension, that describes GCP API. For the previous use cases, the entry point of each one is designing an OCCI extension that describes a specific cloud domain. The particularity of this work consists in analyzing of the textual documentation of the GCP API in order to infer an OCCI extension for GCP. This extension represents a precise model for the GCP API. It will allow end users to graphically design their GCP configurations and deploy them later. This use case is detailed next in Chapter 4.

3.7.3 Synthesis on the OCCIware Approach

The OCCIware approach has been successfully applied in different use cases and domains. Each one implements some required features in a cloud modeling framework. Table 3.2 states an overview on the implementation of these usages in different use cases.

For the `Infrastructure` use case (Subsection 3.7.1.1), cloud architects have taken advantage of the OCCIware Studio tools to design the structural and behavioral aspects of the extension, verify it according to the requirements specific to the domain, and generate its documentation[14]. To deploy designed configurations, `Infrastructure` use case implements the code generation strategy to pro-

[14] Available here `https://github.com/occiware/OCCI-Studio/tree/master/plugins/org.eclipse.cmf.occi.infrastructure/documentation`

Table 3.2: OCCIware Use Cases

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

duce CURL scripts from an OCCI configuration. In addition, model interpretation strategy is also implemented. Indeed, an OCCI extension for the `VMware` technology has been defined[15]. `VMware` extension extends the `Infrastructure` extension. A `VMware` connector has been developed. It supports the deployment of a designed `VMware` configuration in the cloud environment. In addition, it ensures the reconfiguration of a running system at runtime (management) and the synchronization between the design and the execution environment by affecting changes occurred in the executing environment to the existing architecture (monitoring).

For the `Cloud Simulation` use case (Subsection 3.7.2.1), it illustrates several usages of the OCCIware approach. At first, it represents the capability to create a specific designer for the `Simulation` domain. In addition, it shows how we can come back from the `Simulation` Studio to the OCCIware Studio by generating an OCCI configuration model from a `Simulation` configuration model. Finally, this work implements the model interpretation strategy by simulating an OCCI configuration model using the CloudSim API. This use case illustrates how we can plug an external API, here CloudSim, into the OCCIware framework. In fact, once a configuration has been designed, it can be reused for several activities such as simulation, deployment, and cost analysis.

OMCRI use case (Subsection 3.7.2.2) endorses the fact that we can manage XaaS with the OCCIware approach, even mobile robots. OMCRI use case validates the genericity of the OCCIware Runtime. In addition, the evaluation shows that adopting the OCCIware approach into another domain, other than cloud computing, does not cause a latency that damages the responsiveness of the system.

Docker use case (Subsection 3.7.2.3) is considered as the first OCCI-based studio implemented with OCCIware technology. It illustrates our approach to consider

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

OCCIware as a factory to build cloud modeling frameworks (right part of Figure 3.5). Docker Studio represents a concrete use case which implements both strategies, code generation, and model interpretation, to execute designed models specific to a particular domain. The most important feature developed in the Docker use case is the mapping of a running architecture from the execution environment to the modeling framework. It allows end users to benefit from the Docker Studio while the configuration was not initially designed using it.

MoDMaCAO use case (Subsection 3.7.2.4) validates the applicability of the OCCIware approach on different cloud layers. This work is the result of a fruitful collaboration between the OCCIware team in Inria, Spirals, Fabian Korte and Jens Grabowski from the University of Goettingen. MoDMaCAO can even be applied to connect two layers in the cloud such as IaaS and PaaS. Model interpretation strategy has been experimented by developing a connector which allows us to deploy and manage different designed cloud applications using Ansible [ans], a flexible configuration management system. Code generation strategy may also be experimented by generating Ansible playbook artifacts from MoDMaCAO configurations.

GCP use case (Subsection 3.7.2.5) represents the first use case that applies the OCCIware approach on a big cloud provider, a.k.a GCP. I present the GCP use case in the next chapter. GCP use case is a major part of my first contribution in this thesis. In this work I target to integrate both code generation and model interpretation strategies. With the code generation approach, they aim to use GCP configurations to generate GCP artifacts, such as JSON files that contain the needed structured information for creating a VM for example. In addition, this use case aims to experiment the model interpretation approach, by defining the business logic of GCP connector which defines the relationship between GCP configurations and their executing environment.

To summarize, the OCCIware approach provides a software product line, named OCCIware Studio Product Line as shown in Figure 3.20. OCCIware Studio is considered as a factory to create other studios. Each one targets a particular cloud domain. The different generated studios share a common base, which is OCCIware Metamodel. Therefore, composing the different generated studios can be an interesting perspective to create an *Internet of Everything* (IoE) Studio that allows us to compose heterogeneous concepts and domains in the same modeling framework.

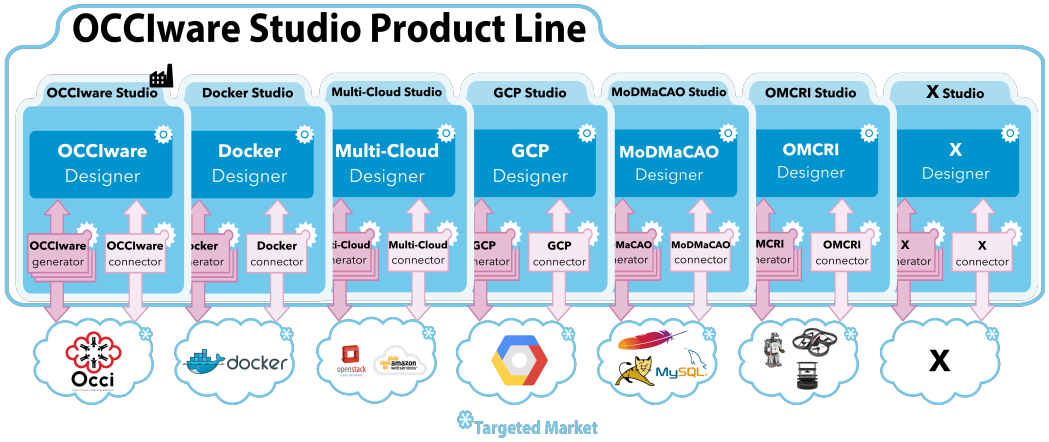


Figure 3.20: OCCIware Studio Product Line.

3.8 Summary

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

The OCCIware approach is proposed as a framework to manage XaaS with OCCI. Moreover, the OCCIware approach is considered as a factory of cloud domain-specific modeling languages and studios due to its capability to generate a complete framework to manage resources specific to a particular cloud domain. The OCCIware approach has been validated via several use cases, which target different domains (IaaS, PaaS, RaaS and CaaS). Each use case illustrates a specific usage of the OCCIware approach and demonstrates its genericity and extensibility.

In the next part of this dissertation, I present my contributions that are based on the OCCIware approach. In the next chapter, I present my approach for representing in an enhanced and automated way the concrete set of cloud concepts and the behaviour of cloud operations.

Part IV

Contributions

In the end of Part II, I discussed the need of automatically building cloud models and formally reasoning over them. In this part, I detail my solution to make it reality.

Chapter 4

Inferring Precise Models from Cloud APIs Textual Documentations

This chapter is an extended version of our paper "A Precise Model for Google Cloud Platfom" [Challita 2018a] published in the 6th IEEE International Conference on Cloud Engineering (IC2E).

Contents

Cloud documentations are the first agreement between cloud developers and cloud providers on how exactly cloud APIs should operate. Even with documentations, cloud developers tend to build cloud configurations or send HTTP requests that are inconsistent with the legal use of the cloud API. This upsetting situation is due to the impreciseness of the cloud textual documentations and to the

lack of verification process that ensures the correctness of cloud configurations before their deployment. Cloud models play an important role to capture the expectations of a cloud API and to a priori validate the correctness of its cloud configurations. These models are manually designed so far, which is prohibitively labor intensive, time consuming and error-prone. To address this issue, I propose a novel approach to infer model-driven specifications from natural language text of cloud API documentations. My approach is applied on a concrete cloud provider, GCP, which is today one of the most important and growing provider in the cloud market. GCP provides developers several products to build a range of programs from simple websites to complex applications. Although it was established only five years ago, GCP has gained notable expansion due its suite of public cloud services that it based on a huge, solid infrastructure. Actually, GCP offers hosting services on the same supporting infrastructure that Google uses internally for end-user products like Google Search and YouTube. This outstanding reliability results in GCP being adopted by eminent organizations such as Airbus, Coca-Cola, HTC, Spotify, etc. In addition, the number of GCP partners has also increased substantially, most notably Equinix, Intel and Red Hat.

To use GCP services, expert developers refer at first to the GCP API documentation provided at GCP website. By going through the GCP documentation, I realize that it contains wealthy information about GCP services and operations, such as the semantics of each attribute and the behaviour of each operation. However, GCP documentation is described through HTML pages at its website[1] and is written in natural language, a.k.a. English prose, which results in human errors and/or semantic confusions. Also, the current GCP documentation lacks of visual support, hence the developer will spend considerable time before figuring out the links between GCP resources. To avoid confusion and misunderstandings, the cloud developers obviously need a precise specification of the knowledge and activities in GCP.

After presenting the general approach we promote to obtain formal cloud models, this chapter presents a precise model for GCP. It describes GCP resources and operations, reasons about this API and provides corrections to its current drawbacks, such as *Informal Heterogeneous Documentation*, *Imprecise Types*, *Implicit Attribute Metadata*, *Hidden Links*, *Redundancy* and *Lack of Visual Support* that I detail in Section 4.2. This is a work of reverse engineering [Rugaber 2004], which is the process of extracting knowledge from a man-made documentation and reproducing it based on the extracted information. In order to formally encode the GCP API without ambiguity, I choose to automatically infer a GCP Model as the

[1] https://cloud.google.com

target documentation format. In fact, my approach leverages the use of MDE to provide a precise and homogeneous specification, and reduce the cost of developing complex systems. MDE allows to rise in abstraction from the implementation level to the model level, and to provide a graphical output and a formal verification of GCP structure and operations. My GCP Model conforms to the OCCIware Metamodel presented in Chapter 3.

The contributions of this chapter can be summarized in three categories:

1. an automated approach to infer models from textual documentations,
2. a concrete use case of our approach: a precise GCP Model that consists in a formal specification of GCP. This model, automatically built, also provides corrections for the drawbacks that I identified in GCP documentation,
3. an analysis of GCP documentation because it is as important as analyzing the API itself. This is done thanks to my model which is a clearer representation of GCP compared to the original one and hence easier to reason over it, and,
4. a validation of the preciseness of my GCP model.

This chapter is structured as follows. Section 4.1 argues for the need to analyze cloud textual documentations and proposes a protocol to automatically infer models from them. Section 4.2 identifies six general drawbacks of GCP documentation that motivate this work. Next, Section 4.3 describes my model-driven approach for a better description of GCP API and gives an overview of some background concepts I use in my GCP Model. Section 4.4 presents and discusses my results, which validate my approach. Finally, Section 4.5 concludes the chapter with future work.

4.1 Inferring Precise Cloud Models

Modeling cloud APIs is an important way for capturing their requirements and succeeding a thorough understanding of their behavior. Modeling allows the cloud developer to keep the cloud concepts clean from the details of implementation classes and the HTTP requests. Moreover, modeling by using MDE principles allows the developer to generate from the cloud model the corresponding Java implementation classes. The model can also be used to generate different outputs, *e.g.,* HTML pages, or it can be interpreted at runtime by software. However, modeling the cloud requires knowledge in modeling techniques which is not necessarily available among cloud experts. For this reason, modeling experts are employed to formalize models within an iterative way in collaboration with cloud experts. This procedure involves long meetings due to ambiguities or misunderstandings among the involved actors,

so the acquisition of cloud models consumes a lot of time and cannot follow-up with the extensive documentations provided by the cloud providers. These documentations are the most relevant source of information for the construction of cloud models. Such textual documents can be stored in a structured way like HTML pages, JSON, YAML or XML files or in an unstructured way like reports or manuals. However, the textual documentations may contain syntactical or semantic errors that distort the real semantics of cloud APIs and that require analysis and rectifications before gathering the information. Therefore, the acquisition process of cloud models remains relatively error-prone and costly.

To address this challenge, I propose to automatically provide a precise model of the cloud API from the documentation supplied by the cloud provider. This way, the cloud developer can easily access the model-based documentation of the cloud that he/she uses. Model-based documentations allow the developer to thoroughly understand the insights of the cloud API. I combined both web crawling and *Natural Language Processing* (NLP) in order to correctly infer models from the cloud documentations. The global approach is summarized as follows. First, using a dedicated web crawler, I extract information that allows me to build a model. Web crawlers are bots that visit the internet to extract information from web pages. Second, I apply NLP [Chowdhury 2003] with rules to extract properties, semantics and constraints on the API. NLP is a branch of the artificial intelligence field that automates the study and extraction of useful knowledge from texts written by humans, in order to better interface with computers. In this context, programs implementing NLP take as input a text flow and exploit information to achieve several tasks such as relationship extraction, automatic summarization, terminology extraction, etc. NLP has two main approaches: rule-based and statistical. The former refers to the idea of using hand-coded set of rules to exploit information. For instance, to recognize email addresses, IP addresses, I use specific regular expressions. This approach is easy to implement and suitable for specific cases. The latter relies on machine learning to achieve more complex tasks. My approach exploited the rule-based NLP which successfully achieved knowledge extraction. My web crawler takes into assumption the structure of the provided documentation, whether it takes the form of HTML pages, JSON files or others. Since most of cloud documentations are provided as HTML pages at the provider's website, I specifically propose in this chapter to automate the extraction of precise models from HTML cloud documentations.

4.1.1 Approach Overview

This section discusses the design of our approach, which is represented in Figure 4.1, and also outlines the foundations behind it, namely web crawling, cloud modeling

and NLP. First, my proposed system takes a documentation as input, and then uses the documentation *Specific Parser*, *Model Generator*, *Text Analysis Engine* and *Model Validator* to correctly generate the model of the cloud API.

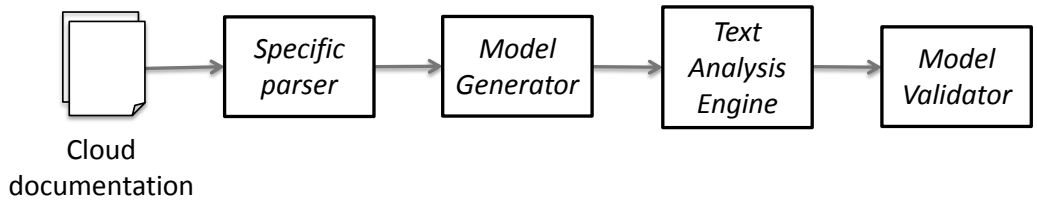


Figure 4.1: My Model Extraction Approach Overview.

- *Specific Parser:* First, I connect to the main page of the cloud documentation and the specific parser takes the documentation in a structured HTML format as input and extracts resources from the HTML tags. More specifically, using an HTML parser, I retrieve the table or the list containing the cloud resources. In fact, the retrieval of information from HTML pages requires recognition rules that are specific to the documentation format. Sometimes, the information is structured in tables with specific IDs. Other documentations could be organized within lists and therefore crawling must be adapted. Afterwards, for each resource, the parser retrieves the resource elements such as its name, description, list of actions and list of attributes. For each attribute, I retrieve its properties like its name, description and type.

- *Model Generator:* I define exhaustive and systematic mapping between the parsed cloud API concepts and the used cloud metamodel. For example, for each row of the documentation HTML table (`<tr> </tr>`), I add a new resource instance to the target cloud model. This resource instantiates the corresponding concept of the cloud metamodel, in terms of the mapping rules I defined. For each cell (`<td> </td>`) that contains the definition of an element of this resource, I add a new attribute instance to the target cloud model.

- *Text Analysis Engine:* Later on, the text analysis engine performs some natural language post-processing actions. I mainly define a set of rules on the description of the attributes to detect implicit properties that are not clearly stated in specific cells and to elucidate domain-specific metadata. I also apply on the description a set of rules based on keywords of each attribute to refine its type if necessary. Sometimes, the type and the description of an attribute do not match, and thus the cloud developer may define incorrect types in

their HTTP requests. I also use NLP to generate a tree structure to identify relations between concepts.

- *Model Validator:* Finally, the cloud model I generated is passed to the model validator to ensure that it is consistent, conforms to the metamodel and behaves as expected. At each time an error is detected, I iterate on the model to fix it by defining additional rules or correcting the existing ones. Our approach does not provide automated correction mechanisms for the errors raised at the validation, the developer has to correct errors by himself/herself.

This contribution is original and striking, and I estimate that it will leave an impact in the cloud domain. It might seem simple but it involves many complicated details. In the remaining of this chapter, I apply my model inferring approach to a real and significant cloud provider and I cover the whole modeling of his documentation. I concretely exploit this idea by automatically inferring from GCP textual documentation, a precise model that conforms to the OCCIware Metamodel. This use case might be applied to other source platforms, like Amazon, towards other target platforms, like CloudML. I chose GCP because it is one of the major cloud providers with an informal documentation and OCCIware because it consists in the context of this thesis and the OCCIware Metamodel is extensible in a way to support all kinds of cloud resources.

4.1.2 Related Work

The literature of model-driven approaches for the cloud shows that researchers do not discuss the details of how they obtain their models for the cloud. This is due to cloud models being manually constructed. Even though researchers might have already thought about automating the process of designing cloud models, this idea has not been put into practice. To the best of our knowledge, we provide the first work that investigates and formalizes a cloud API documentation. In [Petrillo 2016], Petrillo et al. have focused on studying three different cloud APIs and proposed a catalog of seventy-three best practices for designing REST APIs. In contrast to our work, this work is limited to analyzing the documentations of these APIs and does not propose any corrections. Two recent works were interested in studying REST APIs in general. [Haupt 2017] provides a framework to analyze the structure of REST APIs to study their structural characteristics, based on their Swagger documentations. [Cao 2017] presents AutoREST, an approach and a prototype to automatically infer an OpenAPI specification from a REST API HTML documentation. Our work can be seen as a combination of these two previous works [Haupt 2017, Cao 2017], since we infer a rigorous model-driven specification

from GCP HTML documentation and we provide some analysis of its corresponding API. However, in contrast to these two works, our work is specifically applied on a cloud REST API and proposes corrections to the detected deficits of its documentation. Moreover, given that it is an important but very challenging problem, analyzing natural language documents from different fields has been studied by many previous works. In [Zhai 2016], Zhai et al. apply NLP techniques to construct models from Javadocs in natural language. These models allow one to reason about library behaviour and were implemented to effectively model 326 Java API functions. [Pandita 2012] presents an approach for inferring formal specifications from API documents targeted towards code contract generation. [Zhong 2009] develops an API usage mining framework and its supporting tool called *Mining API usage Pattern from Open source repositories* (MAPO) for mining API usage patterns automatically. [Sinha 2010] proposes abstract models of quality use cases by inspecting information in use case text.

4.2 GCP Use Case: Motivation & Drawbacks

The object of my study is the GCP documentation. GCP is a proprietary cloud platform that consists of a set of physical assets (*e.g.,* computers and hard disk drives) and virtual resources (*e.g.,* virtual machines, a.k.a. VMs) hosted in Google's data centers around the globe. I especially target this API because it belongs to a well-known cloud provider and because I believe it can be represented within a better formal specification.

GCP documentation is available in the form of HTML pages online. The URL[2] is the starting point of my study and the base for building my GCP Model. This page exhaustively lists the resources supported by the deployment manager, and provides a hyperlink to each of these resources. Normally, the developer will use the deployment manager to deploy his/her applications. The deployment manager will then provision the required resources. Therefore, I adopt this page to study the documentation of each GCP resource that could be provisioned by the developer.

Through my study of GCP, I have identified six main conceptual drawbacks/limitations on GCP documentation, which are detailed below.

Informal heterogeneous documentation. Enforcing compliance to documentation guidelines requires specialized training and a strongly managed documentation process. However, often due to aggressive development schedules, developers

[2] https://cloud.google.com/deployment-manager/docs/configuration/supported-resource-types

neglect these extensive processes and end up writing documentations in an ad-hoc manner with some little guidance. This results in poor quality documentations that are rarely fit for any downstream software activities.

By going through the HTML pages of GCP documentation, it was not long before I realized that it has two different formats to describe the attributes of each resource (cf. Figure 4.2). This is an issue because it may disturb and upset the reader, *i.e.,* the cloud developer.

Property name	Value	Description	Notes
IPv4Range	string	The range of internal addresses that are legal on this network. This range is a CIDR specification, for example: 192.168.0.0/16. Provided by the client when the network is created.	

Available at
https://cloud.google.com/compute/docs/reference/latest/networks

Vs.

Fields	
projectId	string Required. The Google Cloud Platform project ID that the cluster belongs to.

Available at
https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters

Figure 4.2: Different Documentation Formats.

Imprecise types. GCP documentation is represented by a huge number of descriptive tables written in natural language. Thus it is a syntactically and semantically error-prone documentation; it may contain human-errors and its static and dynamic semantics are not well-formed, *i.e.,* does not describe without ambiguity the API and its behavior. In fact, some of the written sentences are imprecise and can be interpreted in various different ways, which can lead to confusions and misunderstandings when the user wants to provision cloud resources from GCP API. For each resource attribute, I checked the corresponding type and description to assess whether the information is accurate. Figure 4.3 shows that the current GCP documentation states explicitly that string types are supported. But later on, further details in the description explain how to set such strings. For example, the effective type of the attribute is a *URL* in *(1)*, an *email address* in *(2)*, an *enumer-*

ation in *(3)*, and an *array* in *(4)*. The cloud developer may define non-valid string formats for his/her application. The bugs will be detected during the last steps of the provisioning process and fixing them becomes a tricky and time consuming task.

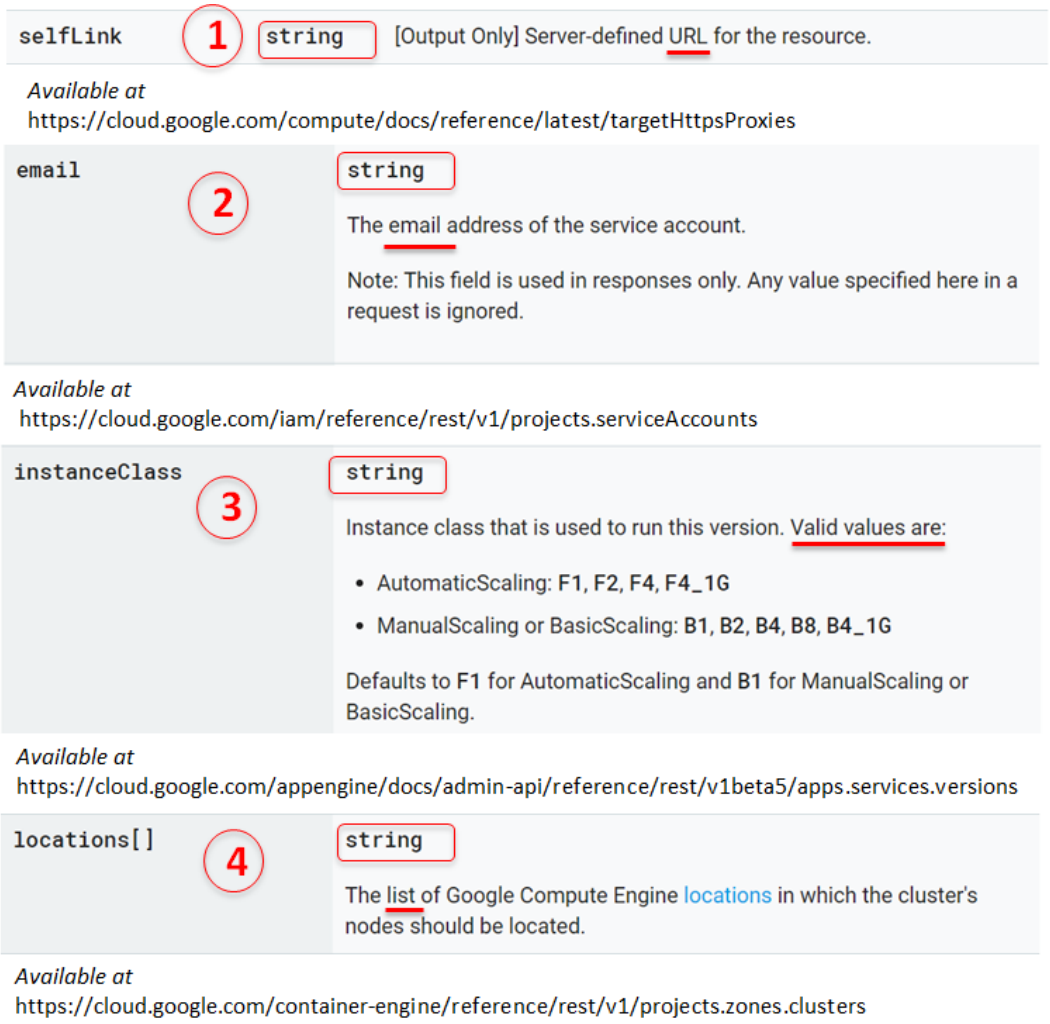


Figure 4.3: Imprecise String Types.

In addition, Figure 4.4 shows that GCP documentation employs several ways to denote an enumeration type. Sometimes, the enumeration literals are listed in the description of the attribute, and sometimes they are retrievable from another HTML page.

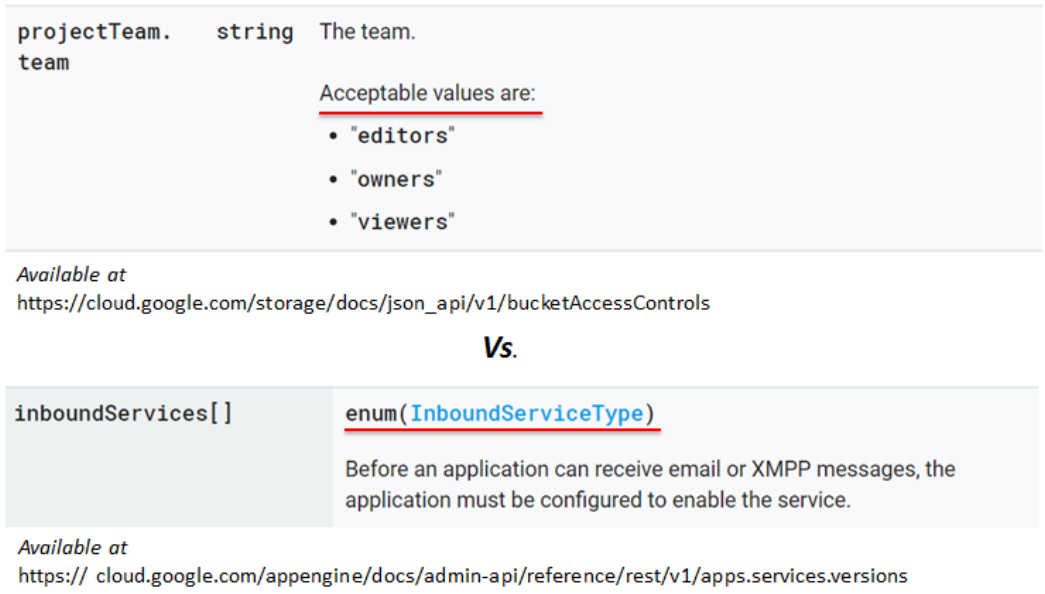


Figure 4.4: Informal Enumeration Types.

As for Figure 4.5, it represents the documentation of the `kind` attribute in four different resources. I notice that (4) shows a formatting error, which induces to the fact that **GCP documentation is written by hand.**

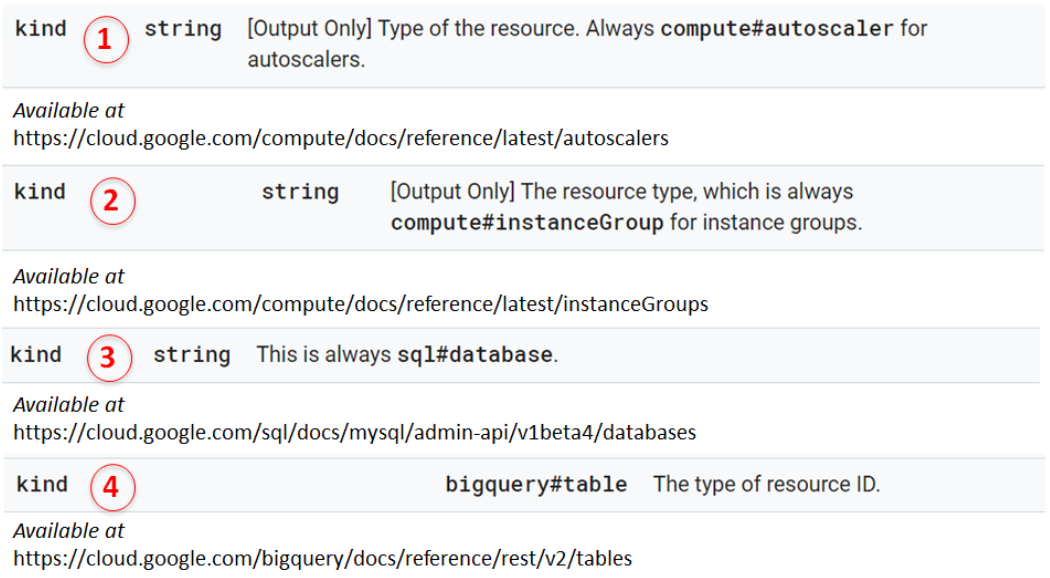


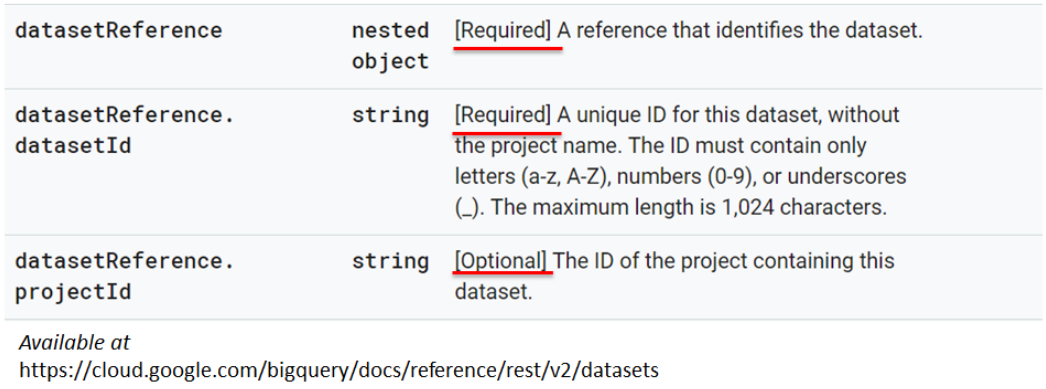
Figure 4.5: Error in Describing the "Kind" Attribute.

Therefore, GCP documentation lacks of a precise and rigorous definition of its

data types.

Implicit attribute metadata. I notice that **GCP documentation contains implicit information in the attribute description.** For example, it contains some information that specifies if an attribute:

- is optional or required (cf. Figure 4.6),

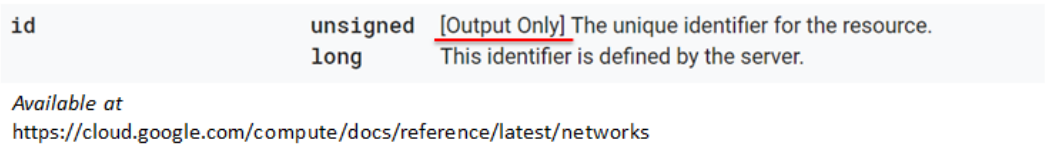


`datasetReference`	`nested object`	[Required] A reference that identifies the dataset.
`datasetReference.datasetId`	`string`	[Required] A unique ID for this dataset, without the project name. The ID must contain only letters (a-z, A-Z), numbers (0-9), or underscores (_). The maximum length is 1,024 characters.
`datasetReference.projectId`	`string`	[Optional] The ID of the project containing this dataset.

Available at
https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets

Figure 4.6: "Optional/Required" Attribute Constraint.

- is mutable or immutable (cf. Figure 4.7),

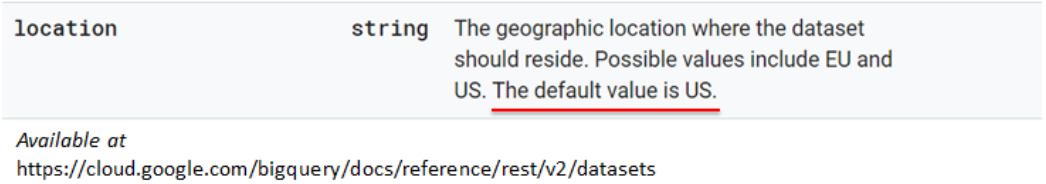


`id`	`unsigned long`	[Output Only] The unique identifier for the resource. This identifier is defined by the server.

Available at
https://cloud.google.com/compute/docs/reference/latest/networks

Figure 4.7: "Immutable Attribute" Constraint.

- has a default value (cf. Figure 4.8).



`location`	`string`	The geographic location where the dataset should reside. Possible values include EU and US. The default value is US.

Available at
https://cloud.google.com/bigquery/docs/reference/rest/v2/datasets

Figure 4.8: "Default Value" Constraint.

These constraints are only explained in the description of each attribute, but lacks of any verification process. The developer will not be able to ensure, before the deployment phase, that his/her code meets these constraints.

Hidden links. A link is the relationship between two resource instances: a source and a target. These links are implicit in GCP documentation but they are important for proper organization of GCP resources. They are represented by a nested hierarchy, where a resource is encompassed by another resource and where an attribute defines the link between these resources, either directly or indirectly. Figure 4.9 shows an example of a deducible link, a.k.a. `networkInterface` because the description of this attribute is a URL pointing to the target resource, a.k.a. `network`. Therefore, I can say that `networkInterface` is a link that connects an `instance` to a `network`. If graphical support exists, this link would definitely be more explicit.

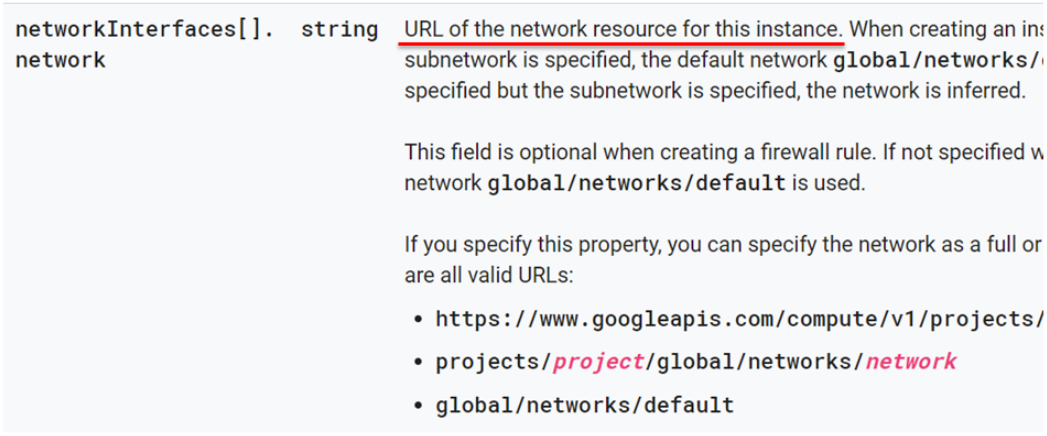


Figure 4.9: Hidden Link between `Instance` and `Network`.

Redundancy. In addition to this, I observe from my study that **GCP documentation is redundant.** According to my observation, it contains a set of attributes and actions in common, *i.e.,* with the same attribute name and type, and the same action name and type respectively. Among this set, I especially notice a redundancy of the attributes `name`, `id`, `kind`, `selfLink`, `description`, etc., as well as of the actions `get`, `list`, `delete`, `insert`, etc.

Lack of visual support. Finally, the information in GCP documentation is only descriptive, which involves a huge time to be properly understood and

analyzed. In contrast to textual descriptions, visual diagrams help to avoid wastage of time because it easily highlights in short but catchy view the concepts of the API. Consequently, logical sequence and comparative analysis can be undertaken to enable quick understanding and attention. Cloud developers can view the graphs at a glance to understand the documentation very quickly, which is more complicated through descriptive format.

Overall, these six drawbacks above are calling for more analysis of GCP documentation and for corrections. Once the development has begun, corrections can be exponentially time consuming and expensive to amend. Therefore, the cloud developer firstly needs a clear detailed specification, with no ambiguous information, in order to:

1. make the development faster and meet expectations of the cloud API,
2. avoid the different interpretations of a functionality and minimize assumptions, and,
3. help the developer to move along more smoothly with the API updates for maintainability purpose.

4.3 GCP Model Extraction Approach

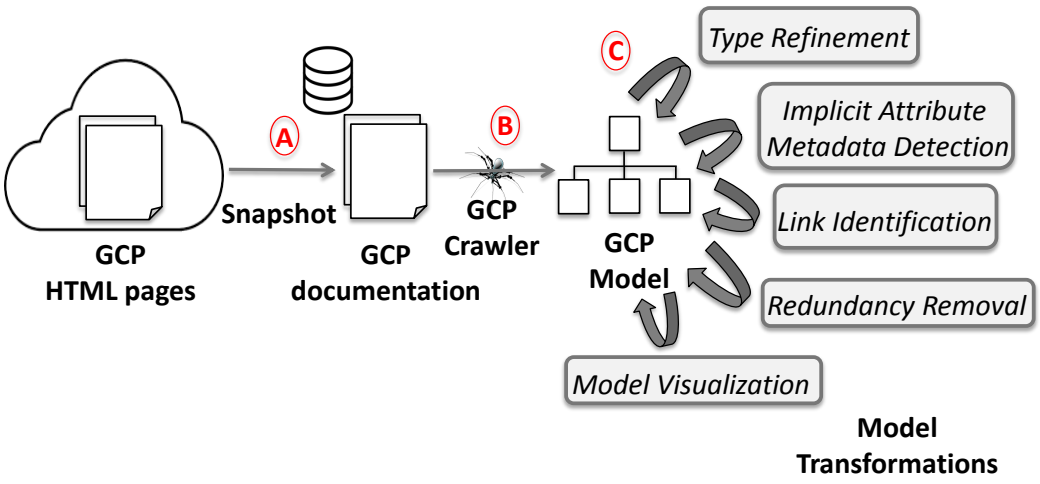


Figure 4.10: GCP Model Extraction Approach Overview.

This section presents my approach that takes advantage of MDE techniques to precisely, textually and graphically, describe GCP API. In fact, MDE is emerging

and emphasizing on the use of models and model transformations to raise the level of abstraction and automation in the software development.

To understand the concepts that rely under the architecture of my approach, I begin by giving an illustration of it in Figure 4.10. This architecture is composed of three main parts: a Snapshot of GCP HTML pages, a GCP Crawler and a GCP Model increased by *Model Transformations* for GCP Refinement. Each of these four parts is detailed in the following.

4.3.1 GCP Snapshot

Google is the master of its cloud API and its documentation, which means that GCP engineers could update/correct GCP documentation, whenever they are requested to or they feel the urge to. But since continuously following up with GCP documentation is crippling and costly, I locally save the HTML pages of GCP documentation in order to have a snapshot of GCP API at the moment of crawling its documentation. This snapshot is built on July 27, 2017.

4.3.2 GCP Crawler

In order to study and understand GCP documentation, the main step of my approach is to extract all GCP resources, their attributes and actions and to save them in a format that is very simple and easily readable by a human. In this sense, extracting knowledge by hand from this documentation is not reliable nor representative of reality; if the documentation changes, extracted knowledge should also evolve through an automated process. Therefore, I have set up an automatic crawler to infer my GCP specification from the natural language documentation.

4.3.3 GCP Model

For a better description of the GCP resources and for reasoning over them, I propose to represent the knowledge I extracted into a model that formally specifies these resources, while providing a graphical concrete syntax and processing with transformations. This addresses the drawbacks of GCP documentation identified in Section 4.2. Choosing the adequate metamodel when developing a model is crucial for its expressiveness [Fowler 2010]. In this context, a language tailored for cloud computing domain will bring us the power to easily and finely specify and validate GCP API. Therefore, I choose to adopt the OCCIware Metamodel because it is a precise metamodel dedicated to describe any kind of cloud resources. As I detailed in Chapter 3, the OCCIware Metamodel is encoded in EMF [Steinberg 2008] and it defines its own data type classification system. Therefore, it easily allows

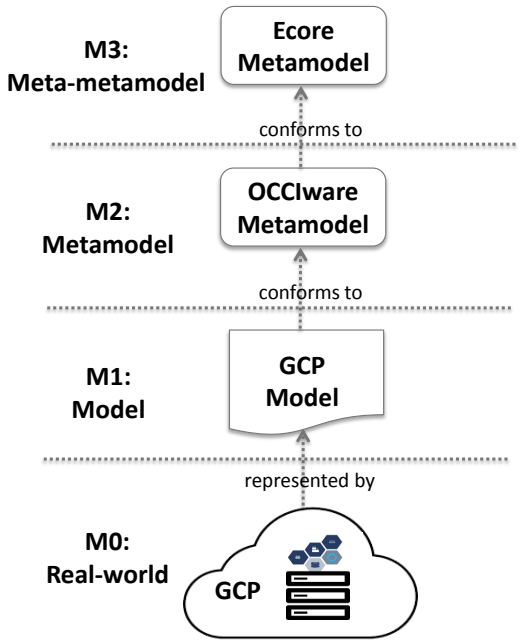


Figure 4.11: Metamodeling Stack for GCP Model.

to define *primitive types* such as booleans, numerics and strings, and *complex types* such as arrays, enumerations and records. In addition, thanks to its `Extension` concept, OCCIware Metamodel allows us to define a set of resource instances targeting a concrete cloud computing domain such as GCP.

In my approach, I exploit these two advantages and I build a GCP Model, which is an expressive model and an appropriate abstraction of the GCP API. GCP Model conforms to OCCIware Metamodel, which conforms to Ecore Metamodel, as illustrated in Figure 4.11.

To go further, I present my approach with an algorithm, as illustrated in Figure 4.12. First, using the OCCI API, I create a model, *i.e.,* an OCCI extension (cf. line 1). Then, I connect to the GCP documentation using the jsoup library that provides an API to parse HTML pages from a URL (cf. line 2). Second, for each resource in the HTML page, I apply the following procedure (cf. line 3): I connect to the dedicated documentation page of the resource (cf. line 4). I create an OCCI resource (cf. line 5) and I use the information inside the HTML page to set the correct values (cf. line 6). Third, for each attribute of the resource detailed in the HTML page (cf. line 7), I extract information from the documentation and add them to the OCCI attribute (cf. line 9). The newly created attribute is added to the set of attributes of the OCCI resource (cf. line 10). Eventually, I add the new OCCI resource to the model under construction and so on (cf. line 11).

Algorithm: Crawler and model generator

Input: Documentation's URL: *url*
Output: Model conforms to OCCI: *model*

```
1:  model ← OCCIFactory.createModel()
2:  document ← connect(url)
3:  for resource in document do
4:      resourceHTML ← connect(resource.url)
5:      resourceOCCI ← OCCIFactory.createResource()
6:      resourceOCCI.properties ← extract(resourceHTML)
7:      for attributeHTML in resourceHTML do
8:          attributeOCCI ← OCCIFactory.createAttribute()
9:          attributeOCCI.properties ← extract(attributeHTML)
10:         resourceOCCI.attributes.add(attributeOCCI)
11:     model.append(resourceOCCI)
12: return model
```

Figure 4.12: The Algorithm of the Model Extraction Approach.

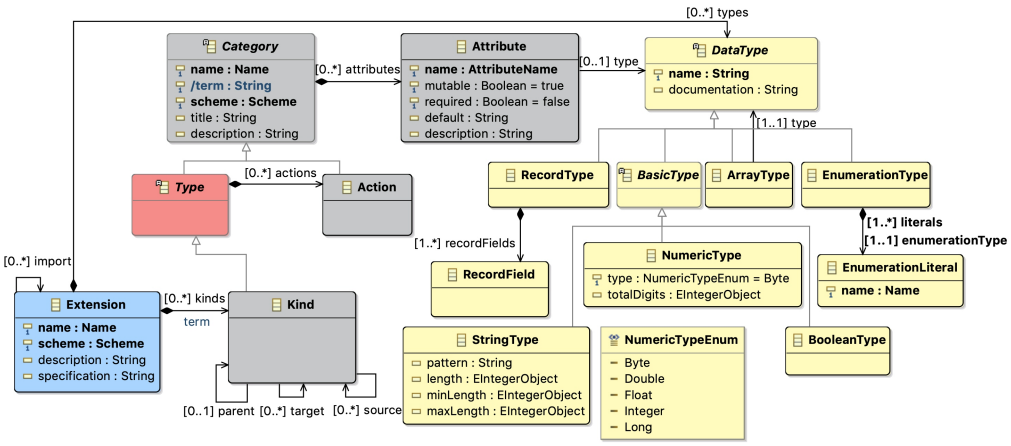


Figure 4.13: A Subset of OCCIware Metamodel.

In order to ease the understanding of this work, I briefly present in the following the main concepts of OCCIware Metamodel used to design my GCP Model (cf. Figure 4.13). For more details on the OCCIware Metamodel, readers can refer to Chapter 3.

- `Extension` represents concrete cloud computing domains, *e.g.,* IaaS, PaaS, SaaS. In my work, `Extension` represents the GCP Model. It contains several kinds and data types.

- `Kind` represents a GCP entity type, such as `version`, `firewall`, `image`, `instance`, `network`, `cluster`, `database`, etc. Each `Kind` has a set of attributes and actions. A `Kind` can have as parent another `Resource` or `Link`. A `Link` is

a relation between two `Resource` instances. For example, `networkInterface` connects a `virtual machine` instance to a `network` instance.

- `Attribute` represents a property of a GCP resource or a link, such as its `id`, `name`, `description`, `selfLink`, `timestamp`, etc. An `Attribute` instance has a name, a description, may have a default value and may be mutable or required.

- `Action` represents an operation that can be executed on GCP API, *i.e.,* on its `Kind` instances, such as `create` an instance, `get` a database, `delete` a cluster, etc.

- `BooleanType` represents the Boolean type. For example, the `vm` attribute expects a boolean value to indicate whether to deploy a version on a virtual machine or in a container and the `autoDelete` attribute to denote whether the disk will be auto-deleted when the instance is deleted.

- `NumericType` represents numeric types such as `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`, etc., as well as their minimal and maximal values. For example, the `currentDiskSize` attribute, which is the current disk usage of a Cloud SQL resource, expects a long type value and the `targetUtilization` attribute, which is the target CPU utilization to maintain when scaling, expects a float type value between 0 and 1.

- `StringType` represents string types, as well as their regular expressions if they exist. A regular expression is used to define the specific textual syntax for representing patterns that matching text needs to be conform to. For example, the `networkIP` attribute, which is an IPv4 internal network address to assign to a network interface of an `instance` resource, expects a string type value with an appropriate regular expression to check for an IP address.

- `ArrayType` represents a complex DataType to define lists and their types. For example, the `serverNames` attribute, which is list of server names that are delegated to a managed zone, is an array of strings.

- `EnumerationType` represents a complex DataType to define enumerations. Each `EnumerationType` has at least one literal (`EnumerationLiteral`). For example, the `status` attribute of an `image` resource is an enumeration, *i.e.,* it can take a value among its enumeration literals only.

- `EnumerationLiteral` represents a value that an attribute can have. For example, the status of an `image` instance can be `Failed`, `Pending` or `Ready`.

- `RecordType` represents a complex DataType to define structures. A `RecordType` instance appears as a row in the database table. It contains some data about one particular attribute. For example, in GCP, the `settings` attribute which defines the user settings of a Cloud SQL resource is a record. Each `RecordType` has at least one `RecordField`.

- `RecordField` represents a field of a record. For example, `pricingPlan`, which is the pricing plan of a resource, and `replicationType`, which is the replication type of a resource, are record fields of the `settings` attribute and expect an enumeration type value.

4.3.4 GCP Refinement

Thanks to OCCIware Metamodel, my GCP Model provides a homogeneous specification language for GCP, which tackles the **Informal Heterogeneous Documentation** drawback, identified in Section 4.2. It also carries out five in-place *Model Transformations* that propose corrections to face and address the other drawbacks discussed in Section 4.2. They aim for several objectives, especially *Type Refinement*, *Implicit Attribute Metadata Detection*, *Link Identification*, *Redundancy Removal* and *Model Visualization*. I highlight in the following these correcting and/or enhancing transformations.

- *Type Refinement* is done by adopting the data type system proposed by OCCIware Metamodel, defining regular expressions, and using the EMF validator to check the type constraints that are attached to the attributes. For instance, among the constraints defined for the GCP Model, one constraint states that if the type of an attribute in the documentation is string and the description explains that this is an email address, my GCP Model will apply the email validation constraint for refinement purpose. This kind of information is translated into a `StringType` containing the following regular expression:

```
^[A–Z0–9._%+–]+@[A–Z0–9.–]+\\.[A–Z]{2,6}$
```

- *Implicit Attribute Metadata Detection* to explicitly store information into additional attributes defined in the `Attribute` concept of my GCP Model. To do so, I apply NLP which has made great progress and has proven to be efficient in acquiring the semantics of sentences in API documentation. Among NLP techniques, I use the *Word Tagging/Part-of-Speech* (PoS) [Klein 2003] one. It consists in marking up a word in a text as corresponding to a particular part of speech, based on both its definition and its context. For this, I

declare my pre-defined tags for some GCP specific attribute properties. Some pre-defined tags are as follows:

- `mutable = true` if `[Input-Only]`.
- `mutable = false` if `[Output-only]/read only`.
- `required = true` if `[Required]`.
- `required = false` if `[Optional]`.

- *Link Identification* to deduce logical connections between resources. Therefore, I also refer to the idea of applying NLP techniques. This time, I use *Syntactic Parsing* [Jurafsky 2000] to acquire the semantics of sentences in GCP documentation. The parse tree in Figure 4.14 describes the sequential patterns that allow us to identify the semantics of a link between two resources, namely between an `instance` and a `network` in this example. The syntactic parsing is achieved by using Stanford parser [sta], which is a library based on neural network.

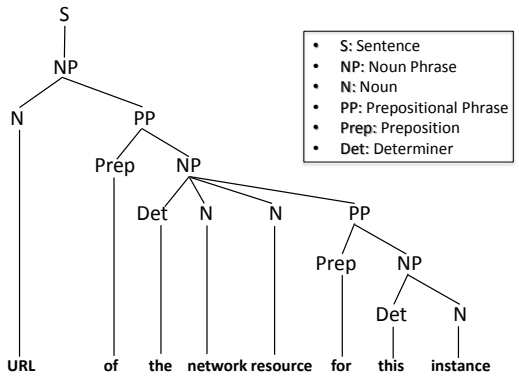


Figure 4.14: Syntactic Parse Tree for Identifying a Hidden Link in a Sentence.

- *Redundancy Removal* in order to offer the cloud developers more compact, intuitive and explicit representation of GCP resources and links. To do so, I propose to have some `abstractKind` instances. An `abstractKind` is an abstract class from which inherit a group of Kind instances. It allows to factorize their common attributes and actions and to reuse them. This is known as *Formal Concept Analysis* (FCA) technique [Priss 2006], which is a conceptual clustering technique mainly used for producing abstract concepts and deriving implicit relationships between objects described through a set of attributes.

- *Model Visualization* for an easier analysis of the API, even if the model is not as sophisticated as the original documentation. In fact, when we visualize information, we understand, retain and interpret them better and quicker because the insights become obvious [Moody 2009]. Unfortunately, as discussed in Section 4.2 (lack of visual support), GCP does not currently provide such a visual model. I provide hence, using OCCIware tooling, a model visualization of GCP documentation. Figure 4.15 shows the graphical output of a subset of GCP Model. This diagram also shows an example of *Redundancy Removal*, where I introduce an `abstractKind` instance and I factorize the attributes and actions of two versions of the same kind. Further information regarding the `abstractKind` instance is given in Section 4.4.1.2.

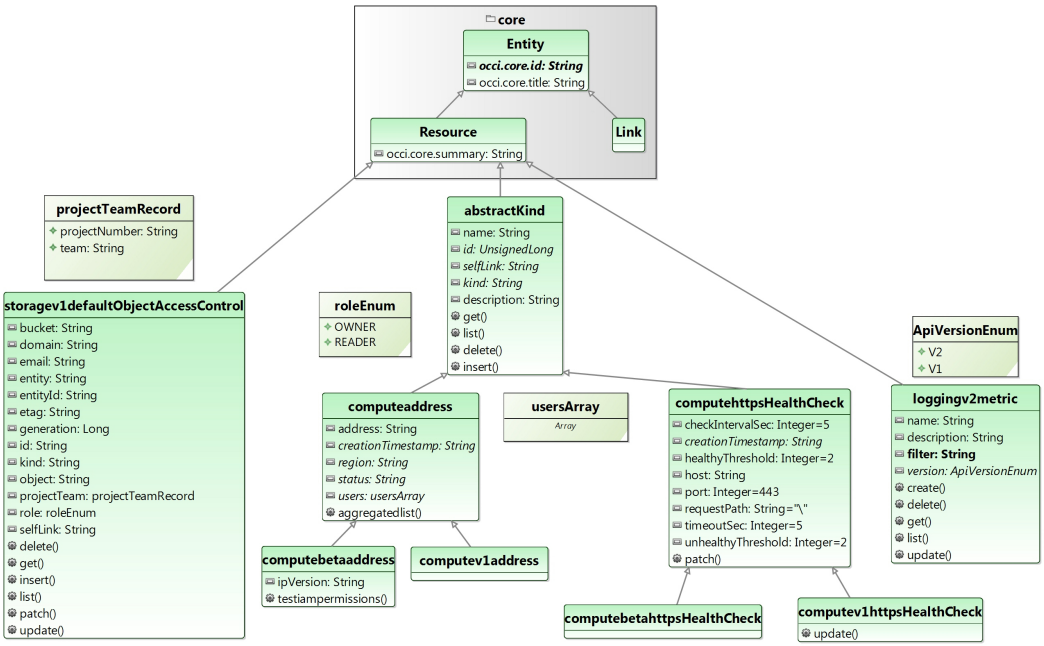


Figure 4.15: A Subset of GCP Extension Diagram.

I have implemented a prototype of my approach in Java. I used jsoup library[3] for building the Snapshot of GCP HTML pages and GCP Crawler, and the Eclipse-based OCCIware Studio for building GCP Model. Readers can find the snapshot of GCP documentation built on July 27, 2017, as well as my precise GCP Model and its code here[4].

Once my model is built, GCP configurations, which represent GCP Instances that conform to GCP Model, can be designed. Then, I aim to elaborate use cases

[3] https://jsoup.org
[4] https://github.com/occiware/GCP-Model

for my model-based GCP configurations as a way of checking them. To do so, I identify the *code generation* and *model interpretation* techniques which are two of the advantages of model-driven engineering [Schmidt 2006]. First, with the *code generation* approach, I aim to use GCP Instances to generate artifacts, such as:

- JSON files that contain the needed structured information for creating a VM for example, through GCP deployment manager,
- CURL scripts that allow us to create a VM for example via the POST action,
- Shell scripts for GCP CLI, and,
- Java or Python code for GCP SDKs to aid in identifying bugs prior to runtime.

Second, I aim to experiment the *model interpretation* approach, by defining the business logic of GCP Connector. The latter defines the relationship between GCP Instances and their executing environment. For this, the connector provides tools that are not only used to generate the necessary artifacts corresponding to the behavior of GCP actions (create, get, insert, list, patch, update, etc.), but also to efficiently make online updates for the GCP Instances elements according to the changes in the executing environment and to the models@run.time approach [Bencomo 2014]. The generated artifacts will be seamlessly executed in the executing environment thanks to MDE principles [Paraiso 2016].

This validation process is entitled "validation by test", because it aims at verifying whether GCP Instances can be executed and updated in the real world. By validating a broad spectrum of GCP Instances, I validate the efficiency of my GCP Model.

4.3.5 Challenges

This contribution of inferring models from APIs might seem simple but it encompasses many challenges like the fact of the documentation is spread over pages. Therefore, one must deeply explore the documentation to completely define all the required elements. For example, when an attribute is an enumeration, we must define the corresponding OCCIware `EnumerationType` before adding the attribute to the correspondent OCCIware kind. However, as we can see in Figure 4.16, the definition of this enumeration is in another HTML page, and thus the mining of this page is mandatory and preliminary to designing the model.

Another challenge of this contribution is the effort and time to toughly analyze the structure of the documentation, which of course contains format inconsistencies making the mining very complicated. This task is necessary in order to design a

`inboundServices[]`	`enum(InboundServiceType)` Before an application can receive email or XMPP messages, the application must be configured to enable the service.

Figure 4.16: Recursive Parsing Example.

correct and resilient crawler. Also, the metadatas are described in natural language in the description, and do not have specific placeholder. So the extraction of this knowledge requires to carefully design the rules that will detect these metadata and put them forward.

Finally, the need to manually check the validity of the extracted model is also a challenge. So far, it is complicated to set up an automatic verification support, and I manually verify the automatically extracted values. For example, if blank spaces were found, I iterate and refine the rules of knowledge extraction.

4.4 Evaluation of GCP Model

Thanks to my approach, I perform a qualitative analysis of the GCP documentation and I check if it satisfies three properties: the *uniformity*, the *conciseness* and the *consistency & comprehensiveness*. Then, to evaluate the effectiveness of my model, I report quantitative results that validate the preciseness of my GCP Model.

4.4.1 Qualitative Evaluation

4.4.1.1 Uniformity

GCP documentation is not uniform, *i.e.,* it is not written by following the same documentation guideline. In fact, the crawling process has been sophisticated and I ended up implementing two parsing filters to capture the two different formats of the HTML pages (cf. Section 4.2 (heterogeneous documentation format)). For each `Kind` instance, I noted the filter I used to parse the attributes and the one I used for the actions. This information allows us to reconstruct and understand the map of the GCP development teams. I arrive at the following hypothesis: **GCP documentation is developed by two clusters of development teams**, as shown in Figure 4.17.

In addition, this classification helps us to predict/learn what teams collaborate together, and to identify the tight links between the GCP development teams and the GCP products. Since the clusters of `Kind` instances are completely disjoint, I conclude that **the two GCP development teams are completely separated**.

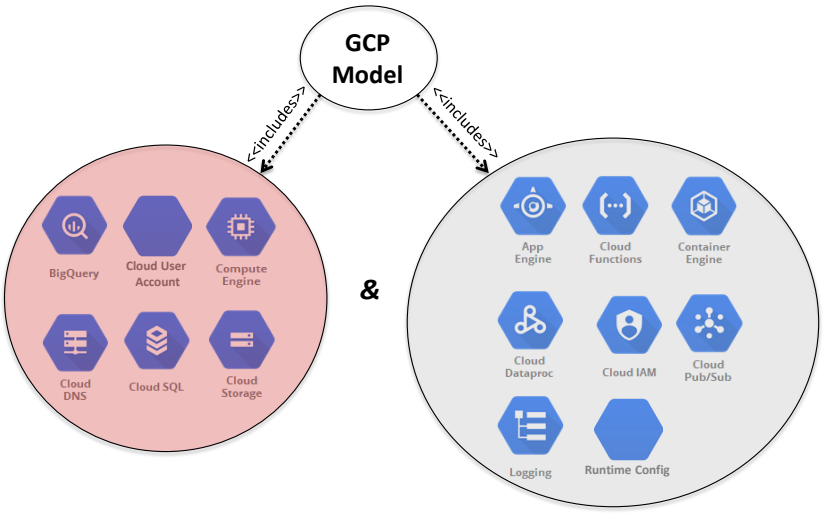


Figure 4.17: Two Clusters of Development Teams.

Table 4.1: Redundant Attributes and Actions among Kinds.

	Before Abstraction		After Abstraction	
Redundant Attributes	**# of occurrences**	**% of redundancy**	**# of occurrences**	**% of redundancy**
`name`	92	64,79%	26	18,31%
`id`	80	56,34%	14	9,86%
`selfLink`	79	55,63%	13	9,15%
`kind`	79	55,63%	13	9,15%
`description`	75	52,82%	9	6,34%
Average		57,04%		10,56%
Redundant Actions	**# of occurrences**	**% of redundancy**	**# of occurrences**	**% of redundancy**
`get`	142	100,00%	76	53,52%
`list`	142	100,00%	76	53,52%
`delete`	140	98,59%	74	52,11%
`insert`	76	53,52%	10	7,04%
Average		88,02%		41,54%

4.4.1.2 Conciseness

A cloud API documentation should be concise, *i.e.,* describing all the concepts clearly but briefly and without redundancy. However, this is not the case for GCP documentation. As depicted in Table 4.1, I prove that GCP documentation contains several redundant attributes and actions. Then I show how my approach minimizes the redundancy by introducing the `abstractKind` concept. Column **"Before Abstraction"** encompasses the # of occurrences and the % of redundancy for each redundant attribute and action in the official GCP documentation. As for column **"After Abstraction"**, it studies the same aspects but after introducing simply one `abstractKind` instance in my GCP Model. This instance contains all the attributes and actions of Table 4.1. Columns **"Redundant Attributes"** and **"Re-**

dundant Actions" list respectively the names of the attributes and actions that I observed in more than 50% of the resources offered by GCP. For each attribute/action, I show the number of occurrences in columns **"# of occurrences"**, as well as the percentage of redundancy in columns **"% of redundancy"**. Rows **"Average"** give the average of columns **"% of redundancy"**.

From the results in Table 4.1, I have the following observations. First, I calculate the % of redundancy for all attributes and actions of GCP documentation, which indicates the effectiveness of my approach. Second, these percentages are relatively high, which prove that **GCP documentation is redundant.** Third, **after factorizing the attributes and actions, I succeed to have 66 less occurrences and 46,48% decrease of the redundancy**. These results are noticeable and satisfying even with my simple example that consists in only one `abstractKind` instance included to GCP Model.

4.4.1.3 Consistency & Comprehensiveness

I have just discussed that some attributes and actions of the GCP resources are redundant. It is not strange to have all these attributes and actions in common. In fact, it makes quite sense that almost each of the `Kind` instances has an `id` as an identifier to guarantee that a resource is unique, a `kind` as this is the type of the resource, a `selfLink` as a URL that can be used to access the resource again, a `description` to report and explain the use of this resource, etc. In addition, it is normal that every `Kind` instance has a list of operations so it can be created, updated, retrieved, deleted, etc.

However, the main problem and what is strange here is why among the 142 resources, the GCP documentation has left 50 without a `name` or 62 without an `id` or 63 without a `kind` or a `description`, or 2 without `delete` or 66 without `insert`, etc. I went further in this research and I checked for each redundant attribute/action the set of `Kind` instances that are missing it. For example, {`compute.beta.regionInstanceGroup`, `compute.v1.regionInstanceGroup`} is the set of `Kind` instances that does not contain the `delete` action. In this case, it is comprehensible that the developer has no right to delete, for both `compute` versions, the `regionInstanceGroup` concept, which refers to the virtual machines in a particular region.

Another example is regarding `dataproc.v1.cluster Kind` instance, which describes the identifying information, configuration, and status of a cluster of Google Compute Engine instances. `dataproc.v1.cluster` belongs to the set of kinds that miss the `name` attribute and also to the set of kinds that miss the `description` attribute. In the first set, *i.e.,* the one that misses the `name` attribute,

`dataproc.v1.cluster` substitutes the `name` attribute by the `clusterName` attribute. Although these two have the same semantics, using different vocabulary to express it will lead to nothing but a confusion for the developer. Therefore, I deduce from this example that **GCP documentation is inconsistent**, *i.e.,* does not employ the same words to define the same characteristics of its services. In the second set, the `description` attribute is effectively missing and this is not justifiable. I can then state that **GCP documentation is not comprehensive**, *i.e.,* it contains some oversights or incompleteness.

4.4.2 Quantitative Evaluation

To evaluate the preciseness of my model, I conduct using my GCP Crawler an automated and recursive analysis of the GCP official documentation. First, I perform a global investigation to identify, quantify and sort the provided services. This analysis allows us to declare that **GCP API contains 142 resources packaged into 14 products** as listed in Table 4.2. Each product offers a service like *Big Data*, *Compute*, *Network*, *Management*, *Storage & Databases*, etc.

Table 4.2: GCP Products.

Product Name	Number of Resources	Offered Service
App Engine	48	Compute
BigQuery	2	Big Data
Cloud Functions	1	Compute
Cloud User Account	2	Management
Compute Engine	67	Compute
Container Engine	1	Compute
Cloud Dataproc	4	Big Data
Cloud DNS	1	Network
Cloud IAM	2	Identity & Security
Logging	2	Management
Cloud Pub/Sub	2	Big Data
Runtime Config	3	Management
Cloud SQL	2	Storage & Databases
Cloud Storage	5	Storage & Databases
Total	142	6

Besides the 142 types of resources, **GCP documentation contains 2124 attributes that describe the static aspect of the API, and 985 actions that describe its dynamic aspect.** Table 4.3 presents a summary of my GCP Model. This dataset covers all the information presented in GCP official documentation and formalized by GCP Model. For each class of my model, Table 4.3 provides the number of instances present in my dataset. The last line provides the total of GCP objects present in the dataset.

Table 4.3: Summary of the GCP Model Dataset.

GCP Metaclass Name	GCP Model Instances
`Kind`	142
`Attribute`	2124
`Action`	985
`BooleanType`	90
`NumericType`	375
`StringType`	1402
`ArrayType`	218
`EnumerationType`	21
`EnumerationLiteral`	80
`RecordType`	143
`RecordField`	613
Total	5437

In addition to these precise statistics that I provide, my approach allows to clearly specify GCP attributes thanks to the *Type Refinement* and the *Implicit Attribute Metadata Detection*. Also, unlike GCP official documentation, my model uses a DataType validator at **design time** to validate the types defined before the provisioning. This validation guarantees the coherence and preciseness of the GCP configurations that must be conform to GCP Model, which is an efficient abstraction of the GCP API.

4.5 Summary

In this chapter, I proposed the first approach that analyses cloud API documentations and applies NLP techniques to extract model-driven specifications. A major use case was carried out on GCP. I highlighted six main drawbacks of GCP documentation and I argued for the need of inferring a formal specification from the current natural language documentation. To address the problem of *informal heterogeneous documentation*, I present my model-driven approach which consists in a GCP Model that conforms to the OCCIware Metamodel presented in Chapter 3. Using my GCP Crawler, my model is automatically populated by the GCP resources that are documented in plain HTML pages. I also proposed five *Model Transformations* to correct the remaining five drawbacks. Finally, my approach allowed us to deduce some facts regarding the *uniformity*, *conciseness*, *consistency* and *comprehensiveness* of GCP API. To conclude, I validated the preciseness of my model by providing quantitative results on GCP API.

In the next chapter of this dissertation, I present my approach for reasoning on cloud APIs in order to verify their behaviour and understand their similarities.

Chapter 5

Specifying Heterogeneous Cloud Resources and Reasoning over them with FCLOUDS

This chapter is a combined and extended version of our papers "Specifying Semantic Interoperability between Heterogeneous Cloud Resources with the FCLOUDS Formal Language" [Challita 2018b] published in the 11th IEEE International Conference on Cloud Computing (CLOUD) and "Towards Formal-Based Semantic Interoperability in Multi-Clouds: The FCLOUDS Framework" [Challita 2017b] published in the 10th IEEE International Conference on Cloud Computing (CLOUD).

Contents

CHAPTER 2 reviewed the solutions for multi-cloud interoperability. To be effective, these solutions must achieve a compromise between defining the common cloud principles and supporting any kind of cloud resources, regardless of their abstraction level. This frustrating situation calls for more depth about the cloud providers' semantics to reason about the common principles that interoperability solutions must adhere to.

In this chapter, I present my vision for reasoning on cloud solutions via FCLOUDS, my framework for semantic interoperability in a multi-cloud context. By semantic interoperability I mean to identify the similarities and differences between cloud APIs concepts and to reason over them. My framework contains a catalog of cloud APIs that are precisely described. It will help the cloud customer to understand how to migrate from one API to another, thus to promote semantic interoperability. To implement the formal language that will encode all the APIs of my FCLOUDS framework, I advocate the use of formal methods, *i.e.,* techniques based on mathematical notations. They will allow me to rigorously encode cloud concepts and behaviour, validate desired and/or imposed cloud properties and finally define formal transformation rules between cloud concepts. For more reliability, I adopt the concepts of the OCCI common standard to define the formal language of the FCLOUDS framework. I choose to formalize OCCI with Alloy, a lightweight promising formal specification language designed by Daniel Jackson from the MIT [Jackson 2012].

The key contributions of this chapter are:

1. the FCLOUDS framework, my formal approach for semantic interoperability in multi-clouds,

2. the FCLOUDS language, the formal language of the FCLOUDS framework, which consists in a formalization of OCCI concepts and operational semantics in Alloy,

3. the identification of five properties (*consistency, sequentiality, reversibility, idempotence* and *safety*) that reflect OCCI RESTful operational semantics, and their validation in FCLOUDS, which ensures the language correctness,

4. a catalog of thirteen formal specifications of cloud APIs from different application domains, encoded with FCLOUDS language, which prove the language expressiveness, and,

5. formal transformation rules between heterogeneous concepts with similar semantics.

This chapter is structured as follows. Section 5.1 outlines the need to reason on the "Cloudware engineering" solutions, explains the motivations behind exploring their semantics and positions my contribution in relation to the related works. Section 5.2 presents my framework FCLOUDS, its components and a usage scenario. Section 5.3 presents the FCLOUDS language that specifies OCCI core concepts and operational semantics, and verifies properties on how OCCI should work. Section 5.4 illustrates the use of my formal language with a series of thirteen examples. Finally, I conclude in Section 5.5.

5.1 Exploring the Semantic Space

A set of common principles that all interoperability solutions adhere to must be agreed on. Accordingly, I argue in the following for the need to explore the *Semantic space*. The latter will allow the cloud architect to rise in abstraction and reason about cloud APIs through the use of formal methods, as shown in Figure 5.1.

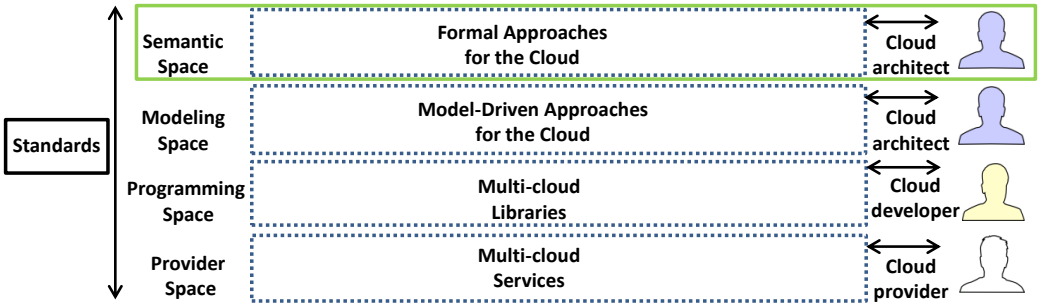


Figure 5.1: Semantic Space.

5.1.1 Formal methods and their benefits

Formal methods are techniques that are based on mathematical notations and they will allow me to rigorously encode the underlying semantics of cloud APIs concepts through *formal specifications*. Formal specifications remove ambiguities, since unlike natural language statements, mathematical specifications are only interpreted in one way, the correct one. This focuses on *what* a system should do rather than *how* to accomplish it. Formal methods also allow me to effectively reason on the structure and behaviour of the encoded concepts, by using a model checker verifying cloud

properties, *i.e.,* constraints denoting characteristics of cloud configurations and/or operations. This is quite advantageous to guarantee the accuracy and correctness of the multi-cloud solutions, and because the earlier a defect is removed the cheaper it will be to correct it. Being precisely specified, verified and correctly understood, the cloud APIs can be correctly compared. For this, I will define formal transformation rules between their concepts, and verify equivalence properties. The developers will hence be able to achieve semantic interoperability in a multi-cloud system.

For the above reasons, I propose in the current work to rise in abstraction by heading towards using formal models and verification for cloud computing.

5.1.2 Related Work

Only few works from the literature applied *formal methods for the cloud*, which proves the novelty of this domain. Benzadri et al. [Benzadri 2013] proposed a formal model for cloud computing using *Bigraphical Reactive Systems* (BRS). AWS [Newcombe 2015] used TLA+ specification language in their complex systems such as S3 and DynamoDB. Bobba et al. [Bobba 2017] specified and validated Google's Megastore, Apache Cassandra, Apache ZooKeeper, and RAMP using Maude language and model checker. Besides using different techniques to reason over the cloud, these three works differ in their objectives too. [Benzadri 2013] reasons over cloud concepts for deployment and adaptation purpose. [Newcombe 2015] aims at finding subtle bugs in their internal distributed algorithms, which helps correcting and optimizing their systems. [Bobba 2017] verifies the performance and correctness of cloud storage systems. My work focuses on the interoperability concern by formalizing the static and operational semantics of the cloud domain.

From the literature, I found out that semantic interoperability between cloud domains has been also achieved with techniques different from formal methods. For example, the authors in [Yongsiriwit 2016] proposed an ontology-based framework for semantic interoperability in multi-clouds. They define translations between IaaS concepts of three standards. However, they do not consider any mapping between API operations. Also, *PaaS Semantic Interoperability Framework* (PSIF) [Loutas 2011], which was implemented in the context of Cloud4SOA project, proposes common PaaS models that describe structural, functional and behavioural semantics. FCLOUDS goes beyond PSIF and aims to verify properties of the models thanks to the usage of formal methods.

5.2 The FCLOUDS Framework

This section presents the **FCLOUDS** formal-based framework, which is our vision for semantic interoperability in multi-clouds. I begin by giving a scenario that motivates my approach, then I describe how I model FCLOUDS structure and behaviour, and how I reason over them.

5.2.1 Usage Scenario

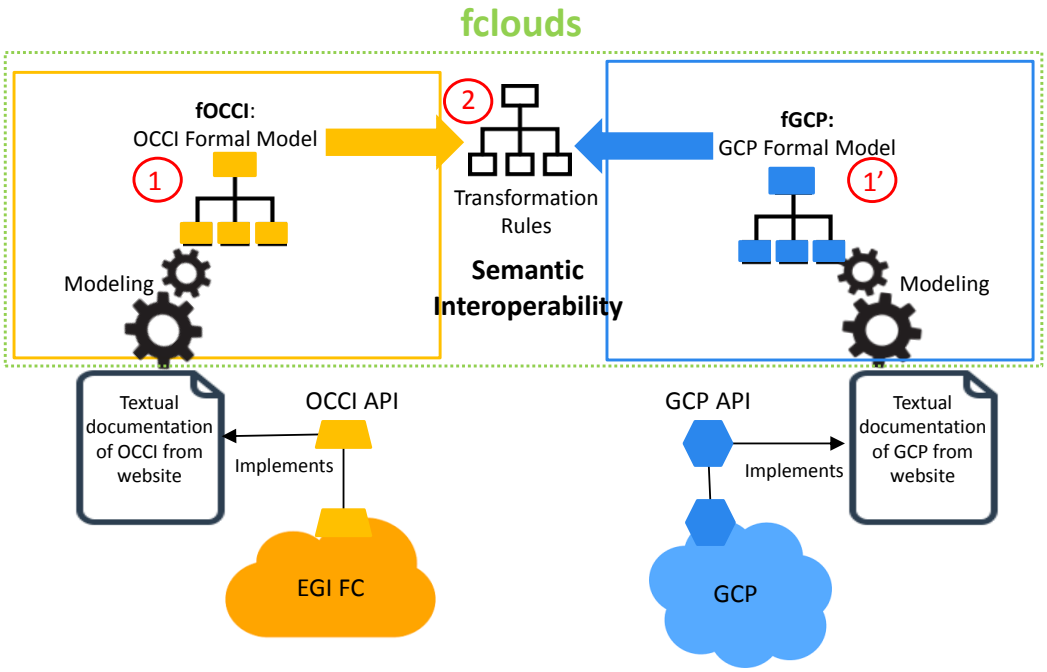


Figure 5.2: FCLOUDS Usage Scenario.

I assume that a developer would like to build a multi-cloud system spread over two clouds, the private EGI FC and the public GCP. EGI is based on OCCI REST API and GCP on its own REST API, so the developer is faced to two heterogeneous APIs implementing different concepts and paradigms. To provision a virtual machine, the HTTP request has a different format for each API, which is quite frustrating. The developer would like a single API for both clouds to seamlessly access their resources. However, the GCP API will not work on OCCI and if OCCI wanted to provide equivalent services to GCP, it should not only adopt the same type of API but also the same concepts with the risk of misunderstandings, inconsistencies and incompleteness. Conflicts and misunderstandings about the semantics

of cloud providers can be solved, or at least identified at an earlier stage, if aspects of structure and operations are conveyed through the use of formal models.

As depicted in Figure 5.2, *the first stage* of FCLOUDS requires extracting from websites, in a manual or automated way, knowledge regarding the services offered by cloud providers. Then, I proceed by a precise modeling to understand and validate the behaviour of cloud APIs, in order to overcome their semantic heterogeneity. *In the second stage* of FCLOUDS, I proceed with transformation models, which explain how knowledge collected against one cloud provider can be transformed to fit another. This stage allows a rigorous comparison and semantic connections between cloud providers.

5.2.2 Overall Architecture

FCLOUDS framework is based on several cloud formal models that can be composable. It consists of a formal model for OCCI (FOCCI), a formal model for GCP (FGCP), a formal model for Docker (FDOCKER), etc. (see the rectangles in Figure 5.3). Later on, I will define *transformation rules* that find equivalence and specialization relationships between them, thus I can seamlessly achieve semantic interoperability (see the ovals in Figure 5.3). The green shapes represent some of the APIs and the transformation rules that are specified in my FCLOUDS framework (see Section 5.4 for the entirety of FCLOUDS APIs) and the orange shapes represent an API and transformation rules that are considered for future work.

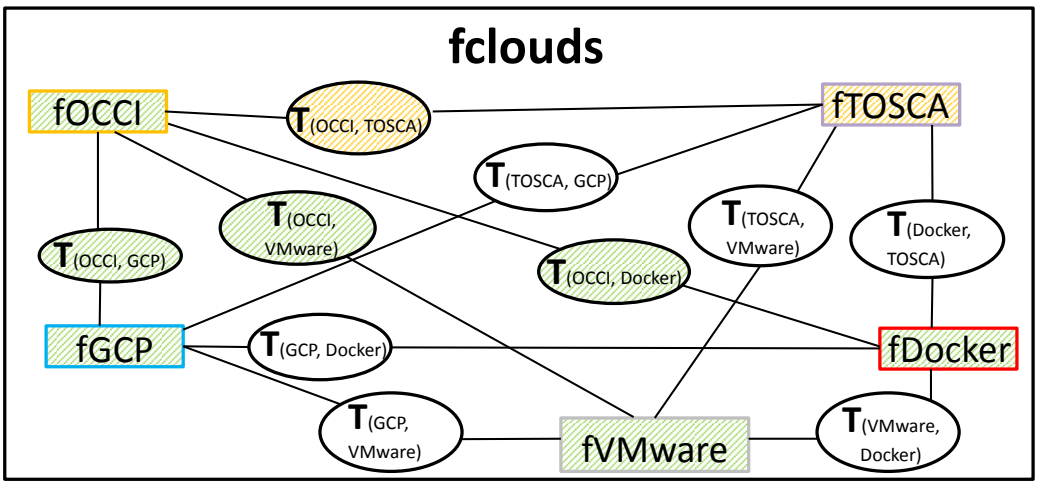


Figure 5.3: FCLOUDS Framework Overview.

In the following, I detail the process of formalizing cloud APIs, which is at the

basis of FCLOUDS framework. As shown in Figure 5.4, it is developed in two main steps: **Modeling** and **Reasoning**.

5.2.2.1 Modeling

Modeling using formal methods is the process of providing a precise specification of a cloud model, *i.e.,* defining and validating:

- **Cloud structure and constraints**, as represented in Frame *(1)* in Figure 5.4, to denote the types of cloud concepts such as virtual machines, containers, storage, operating systems, servers, applications, etc. and describe configurations of these types. FCLOUDS models support cloud computing concepts specification while simplifying irrelevant details to focus on the most important characteristics.

- **Cloud API operations**, as represented in Frame *(2)* in Figure 5.4, to denote the operations that the developer uses to provision, manage or release cloud services through the cloud API.

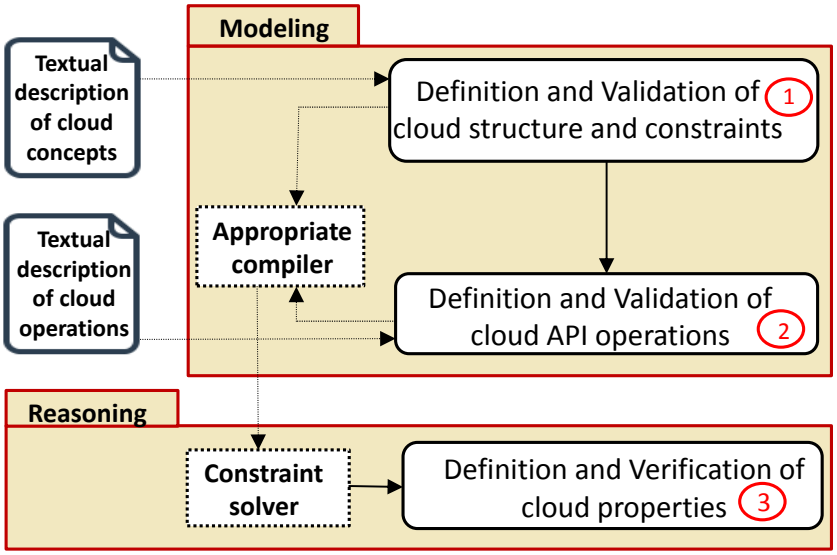


Figure 5.4: Formalization Process.

In my work, I choose the Alloy formal language to provide a concise specification of FCLOUDS models, with both a graphical output and a textual output, so it can be easy to analyze and reason over them. More details about this language, called the FCLOUDS language are given in Subsections 5.3.2 and 5.3.3.

5.2.2.2 Reasoning

Using formal models has the advantage of allowing reasoning over concepts and operations for a better understanding of their semantics and how they work. Therefore, once FCLOUDS models have been specified, I proceed by the **definition and verification of cloud structural and behavioural properties** as shown in Frame (3) in Figure 5.4. This step allows to ensure the correctness of all cloud formal models in the FCLOUDS framework, so I can draw later inferences between them. More details about cloud properties are given in Subsection 5.3.4.

5.3 The FCLOUDS Language

The formal language on which is based the FCLOUDS framework and that I propose, is called the FCLOUDS language. It is an Alloy-based formal language which makes explicit *OCCI core concepts* [Nyrén 2016b] and *OCCI REST operations* [Nyrén 2016a], as well as the underlying properties.

5.3.1 Notations

I argue it is more advantageous and reliable to adopt an open standard to define the formal specification language of all FCLOUDS APIs, instead of writing a language from scratch. The most popular standard is OCCI since it is used by the private EGI FC and it was successfully extended to support heterogeneous aspects of the cloud domain, through OCCI Infrastructure [Nyrén 2016c], OCCI Platform [Metsch 2016], etc. In fact, OCCI defines **a generic and extensible model for cloud resources** and **a RESTful API** for efficiently accessing and managing resources. This facilitates interoperability between clouds that are implemented as OCCI `extensions`, *i.e.,* specified by the same OCCI resource model, and accessed by the same REST API. Today, there are several trends for formal methods like Petri nets, languages based on logic, semantics programs, automata theory, etc. Choosing the appropriate notation is critical in order to find the right compromise between formalization and complexity. Meanwhile, Alloy is becoming increasingly popular among formal methods, as it is a relational, first-order logic language, with well-thought out syntax and model visualization features. Alloy is a lightweight promising formal specification language designed by Daniel Jackson from the MIT [Jackson 2012]. It allows to specify complex systems in a streamlined way, by describing concepts and constraints. The specifications are translated into first-order logic expressions that can be automatically solved by the Alloy analyzer, a model checker using SATisfiability (SAT) solvers. The latter allows automatic verification of a system model, to

ensure its consistency and other desired properties, thus to guarantee its correctness. Therefore, I choose to formalize OCCI using a lightweight promising formal language, the Alloy specification language [Jackson 2012]. I refer to this formal specification as the FCLOUDS language.

In the following, I present a subset of the FCLOUDS static and operational semantics. For more details, the entirety of this language is available in the OCCIWARE official website [Ahmed-Nacer 2016b] and in this GitHub repository[1].

5.3.2 Specifying FCLOUDS Static Semantics

The static semantics of FCLOUDS corresponds to the formalization of the OCCI core concepts [Nyrén 2016b] in Alloy. I use a strategy based on `Time` dimension [Jackson 2012]. It allows to distinguish between mutable fields, *i.e.,* those that are related to Time, and immutable ones, *i.e.,* those that are not related to Time. I detail in the following the twelve concepts of FCLOUDS in Alloy:

- `Entity` represents an abstract type defining the set of all resources and links.

```
abstract sig Entity {
        id : one String,
        kind : one Kind,
        mixins : set Mixin -> Time,
        attributes : set AttributeState -> Time
}
```

A signature (`sig`) in Alloy defines a set of atoms. An atom is an *indivisible, immutable* and *uninterpreted* unity. Signature declarations can introduce `fields`. A field represents a relation among signatures. For example, `Entity` declares four fields: `id`, `kind`, `mixins` and `attributes`. Each entity instance has a unique identifier `(id)`. A declaration of the form `id : one String` can be read as declaring a feature of the set Entity; formally, it declares a binary relation between the set of entities, `Entity`, and the set of Strings, `String`. The **one** multiplicity keyword signifies that the relation between a tuple from `id` and a tuple from `String` has a 1..1 cardinality. The `kind` field is the `Entity` type, for example the kind of a resource can be `Compute`, `Application`, etc. The `mixins` field is used to add additional features such as location and price. The **set** multiplicity keyword signifies that `mixins` can contain any number of elements. The `id` and `kind` are immutable. As for the `mixins` and `attributes`

[1] https://github.com/occiware/fclouds-Framework

fields, they are mutable and they identify the association between `Mixin` atoms and their `Time`, and the association between `AttributeState` atoms and their `Time` . `mixins` and `attributes` are ternary relations. `mixins` relation associates entities, mixins and times, whereas `attributes` relation associates entities, attributes and times.

- `Kind` is the `Entity` type. For example, the kind of a resource can be `Compute`, `Network`, `Application`, etc.

```
sig Kind extends Category {
        parent : lone Kind,
        actions : set Action,
        entities : set Entity -> Time
}
```

The **extends** keyword in Alloy indicates that a set is declared as a subset of another one and that it will form, with other subsets similarly declared, a partition of the set it extends. A `Kind` instance can have zero or one `parent` kind. The **lone** keyword is an example of a relation multiplicity. In our case, it signifies that a given `Kind` is to be associated to at most one `parent`. A `Kind` instance can also have zero or many `actions` and zero or many `entities`. The `parent` and `actions` fields are immutable. Only the `entities` field is mutable because entities (resources or links) can be created/added or deleted/removed at runtime. The entities associated to a certain kind cannot have the same id. This is defined through the following constraint, which must always hold, *i.e.,* an invariant.

```
all t : Time | no disj e1, e2 : entities.t | e1.id = e2.id
```

This invariant takes the form of a basic first-order logical formula, and it says that for every time, there is no pair of distinct entities that have the same id. The **all** keyword denotes the universal quantifier, where a declaration such as `t : Time` denotes an arbitrary element t of the set Time. The **no disj** keyword means that kinds do not have overlapping entities. Likewise, the dot notation `e1.id` or `e2.id` is the standard notation for accessing a feature, or attribute, of an instance of a signature. In this case, the dot serves to access the id feature of an entity instance.

- `AttributeState` represents an instantiated OCCI `Attribute`.

```
sig AttributeState {
        name : one String,
        value : one String
}
```

An `AttributeState` instance has exactly one `name` and one `value`.

- `Attribute` is the property of an entity like machine hostname, IP address of a network, parameter of an action, etc.

```
sig Attribute {
        name : one String,
        mutable : lone Boolean,
        required : lone Boolean,
        default : lone String,
        description : lone String,
        type : lone DataType,
        multiple_values : lone Boolean
}
```

An `Attribute` instance has exactly one `name` and can have zero or one information whether it is `mutable` or not, `required` or not and has `multiple values` or not. However, Boolean type is not supported by Alloy for declarations. Therefore, in order to express these fields, I use *util/boolean* library that defines a boolean type `sig Boolean { }` with `one sig True, False extends Boolean { }`. An `Attribute` instance can also have zero or one `default` value and zero or one `description`. All the attributes fields are immutable.

- `DataType` is the abstract class used to extend the non-extensible data type system of the OCCI specification. Since attributes can have scalar data types (IP address, float, etc.) and enumeration, the classical data types defined as strings, booleans and integers are insufficient and require to be extended.

```
abstract sig DataType {
        name: one String
}
```

A `Datatype` instance has exactly one immutable name such as `array`, `enumeration`, `record`, etc.

- `Mixin` is a concept that adds additional features to OCCI entities.

```
sig Mixin extends Category {
        actions : set Action,
        depends : set Mixin,
        applies : set Kind,
        entities : set Entity -> Time
}
```

A `Mixin` instance can contain zero or many actions, it can `depend` from zero or many of mixins, *i.e.,* if Mixin A depends from Mixin B, any entity associated to Mixin B will inherit the capabilities (attributes and actions) of both Mixin B and Mixin A. `Mixin` can be also `applied` to zero or many kinds, *i.e.,* add new capabilities to the kind instances. A `Mixin` instance has zero or many `entities` that are associated to it. Only the `entities` field is mutable. A `Mixin` instance must not inherit/depend from itself directly or transitively. This constraint is ensured as follows:

```
no d : this.^@depends | d = this
```

This constraint specifies that the relation between mixins is acyclic, *i.e.,* the relation between mixins do not form a ring. It denotes that there is no mixin that belongs to the set of targets reachable from the same mixin itself. The **no** quantifier means that this constraint is true when for `d = this` is true, there is no bindings of the `d` variable. In other words, `d` does not have a value if it belongs to the `this.^@depends` relation and is equal to the mixin instance itself. The "^" operator denotes the transitive closure of the `depends` relation, *i.e.,* the smallest enclosing relation that is transitive. The "@" symbol is used to prevent the `depends` field from being expanded. Without the @ symbol, the constraint would instead be short for:

```
no d : this.^(this.depends) | d = this
```

which does not even type-check.

- `Action` represents an operation that can be executed on an entity instance such as `start` virtual machine, `stop` virtual machine, `restart` an application, `resize` a storage, etc.

```
sig Action extends Category {
}
```

- `Category` is the abstract class of all the `Action`, `Kind` and `Mixin` instances.

```
abstract sig Category {
        term : one String,
        scheme : one String,
        title : lone String,
        attributes : set Attribute
}
```

A `Category` instance has exactly one `term`, one `scheme` and may have or not one `title`. A `Category` instance also contains a set of `attributes`. The attribute name must be unique, as defined by the following constraint:

```
no disj a1, a2 : attributes | a1.name = a2.name
```

This constraint means that there is no pair of distinct attributes that have the same name. All the `Category` fields are immutable.

- `Resource` represents a concrete cloud computing resource, which refers to any entity hosted in a cloud, *e.g.,* `compute1` is a resource that belongs to `Compute` kind, `network3` is a resource that belongs to `Network` kind, `storage2` is a resource that belongs to `Storage` kind, etc.

```
sig Resource extends Entity {
        links : set Link -> Time
}
```

A `Resource` instance owns a set of mutable `links`.

- `Link` is the relationship between two `Resource` instances. For example, `NetworkInterface` connects a `Compute` instance to a `Network` instance, and `StorageLink` connects a `Compute` instance to a `Storage` instance.

```
sig Link extends Entity {
        source : Resource one -> Time,
        target : Resource one -> Time
}
```

`Link` contains two mutable fields: `source` and `target`. Each sourced link must have a target, as defined by the following constraint:

```
one source implies one target
```

The **one** quantifier applied before the "source" expression means that the set of sources has exactly one tuple, *i.e.,* for a `Link` l, `l.source` is the resource that l is currently sourced to. Similarly, the **one** quantifier applied before the "target" expression means that the set of targets has exactly one tuple, *i.e.,* for a `Link` l, `l.target` is the resource that l currently targets. This constraint means that the *one source* constraint implies the *one target* constraint.

- `Extension` is a set of `kind` and `mixin` instances targeting a concrete cloud domain (IaaS, PaaS, SaaS, pricing, SLA, cloud monitoring, etc.).

```
sig Extension {
        name : one String,
        scheme : one String,
        import : set Extension,
        kinds : set Kind,
        mixins : set Mixin,
        types : set DataType
}
```

An `Extension` instance has one `name` and one `scheme`. It can `use` or extend zero or many extensions. It owns zero or many `kinds`, `mixins` and `datatypes`. All the `Extension` fields are immutable. The scheme of all kinds must be equal to the scheme of the owning `Extension` instance, as defined by the following constraint:

```
all k : kinds | k.@scheme = scheme
```

This constraint means that for every kind of an extension, the scheme of this kind is equal to the scheme of the extension.

- `Configuration` is the abstraction of an OCCI-based running system. Modeling a configuration offline allows designers to think and analyze their cloud systems without having to deploy them concretely in the clouds [Merle 2015a].

```
sig Configuration {
        use : set Extension,
        resources : set Resource -> Time,
        mixins : set Mixin -> Time
}
```

The `use` field, which is the set of extensions used in a configuration, is immutable because the extensions cannot be added or removed at runtime. The `resources` and `mixins` fields are mutable. The kind of all resources of a configuration instance must be defined by an extension that is explicitly used by this configuration. This constraint can be expressed as follows:

```
all t : Time | resources.t.kind.extension in use
```

This constraint means that for every time, the extensions containing the kinds of the configuration resources are contained in the extensions of this configuration. In Alloy, the `in` keyword denotes the subset relation.

Table 5.1: FCLOUDS Static Semantics.

FCLOUDS Concepts	Description
`Entity`	an abstract type defining the set of all resources and links
`Kind`	an immutable type of OCCI entities
`AttributeState`	an instantiated OCCI attribute
`Attribute`	an entity property, *such as the hostname of a virtual machine*
`DataType`	an abstract type defining enumerations, lists, records, etc.
`Mixin`	represents crosscutting attributes and actions that can be dynamically added to an OCCI entity
`Action`	domain specific behavior, *such as start/stop a virtual machine*
`Category`	the abstract base class inherited by `Kind`, `Mixin` and `Action`
`Resource`	represents any cloud computing resource, *such as a virtual machine*
`Link`	a relation between two resources
`Extension`	a concrete cloud computing domain, *such as IaaS, PaaS, SaaS, cloud robotics, etc.*
`Configuration`	a running OCCI system

Table 5.1 presents a summary of the FCLOUDS language concepts.

5.3.3 Specifying FCLOUDS Operational Semantics

The operational semantics of FCLOUDS corresponds to the formalization of the informal OCCI behavioural specification detailed in [Nyrén 2016a]. It mainly includes different REST operations, *i.e.,* CREATE, RETRIEVE, UPDATE, DELETE. In this idiom, these operations are modeled as predicates that specify the relationship between pre-state, *i.e.,* the state before the operation is called and post-state, *i.e.,* the state after the operation is completed. To do so, Time is added at the end of each mutable field to represent the state concept. To be more specific, an operation *op* will be specified using a predicate: `pred op[...,t,t':Time] ...`,

with two special parameters t and t' denoting, respectively, the pre- and post-states [Garis 2012]. The core of each predicate is carried out by defining explicitly pre- and post-conditions, which are constraints that must be satisfied before executing the operation and after the operation is finished respectively. Eight operations were defined to the `Configuration` concept of FCLOUDS, as depicted in Table 5.2. I present in the following the four REST operations applied to resources.

5.3.3.1 Create semantics

The following predicate shows how I formally specify the creation of a resource.

```
pred CreateResource[config : Configuration, resourceId : String,    1
    kind : Kind, mixins : set Mixin, attributes : set
    AttributeState, t, t' : Time] {
 // preconditions at instant t                                      2
    no resource : config.resources.t | resource.id = resourceId     3
    kind in config.use.kinds                                        4
    mixins in config.use.mixins                                     5
 // postconditions at instant t'                                    6
    one resource : Resource {                                       7
      resource.id = resourceId                                      8
      resource.kind = kind                                          9
      resource.mixins.t' = mixins                                   10
      resource.attributes.t' = attributes                           11
      config.resources.t' = config.resources.t + resource           12
    }                                                               13
    config.mixins.t' = config.mixins.t'                             14
}                                                                   15
```

At time t, I specify that a configuration, passed as argument to the predicate, does not have a resource with the id passed as a predicate argument (cf. line 3). The **no** quantification keyword expresses the null cardinality of the empty resource set that satisfies the constraint `resource.id = resourceId`. As well, I specify that the kind and the mixins of the resource I want to create, are contained in the configuration extensions (cf. lines 4 and 5). At time t', I add to the configuration resources, one resource with the id, kind, mixins and attributes, passed as argument to the predicate (cf. lines 7 to 12). The "+" operator denotes set union. I also explicitly specify that the mixins of the configuration remain unchanged (cf. line 14).

5.3.3.2 Retrieve semantics

The following predicate shows how I formally specify the retrieval of a resource.

```
pred RetrieveResource[config : Configuration, resourceId :
    String, t, t' : Time] {
 // preconditions at instant t
    one resource : config.resources.t {
      resource.id = resourceId
    }
 // postconditions at instant t'
    one resource : config.resources.t' {
      resource.id = resourceId
      resource.mixins.t' = resource.mixins.t
      resource.attributes.t' = resource.attributes.t
      resource.links.t' = resource.links.t
    }
    config.resources.t' = config.resources.t
    config.mixins.t' = config.mixins.t
}
```

At time *t*, I verify that a configuration has one resource with the id passed as a predicate argument (cf. lines 4 and 5). At time *t'*, I specify that the id, mixins, attributes and links of the retrieved resource remain unchanged (cf. lines 8 to 12). I also specify that the resources and the mixins of the configuration remain unchanged (cf. lines 14 and 15).

5.3.3.3 Update semantics

The following predicate shows how I formally specify the update of a resource.

```
pred UpdateResource[config : Configuration, resourceId : String,
   attribute1 : AttributeState, attribute2 : AttributeState, t,
    t' : Time] {
 // preconditions at instant t
    one resource: config.resources.t | resource.id = resourceId
         && resource.attributes.t = attribute1
    attribute1 != attribute2
 // postconditions at instant t'
    one resource : config.resources.t {
      resource.attributes.t' = resource.attributes.t ++
          attribute2
```

```
    }                                                                    9
    config.resources.t' = config.resources.t                            10
    config.mixins.t' = config.mixins.t                                   11
}                                                                        12
```

At time *t*, I specify that a configuration, passed as argument to the predicate, has a resource with the id passed as a predicate argument and with the attribute, *attribute1* which is also passed as a predicate argument (cf. line 4). The "&&" operator denotes conjunction of constraints. I also specify that the *attribute1*, passed as argument to the predicate, is different from *attribute2*, also passed as argument to the predicate (cf. line 5). The "!=" operator is the negation operator "!" associated to the comparison operator "=" and is equivalent to `not attribute1 = attribute2`. At time *t'*, I update the existing attribute state of my resource, *i.e., attribute1*, with the value of the new attribute state, *i.e., attribute2* (cf. lines 7 and 8). The "++" operator is used for the override, which means that the tuples of *attribute2* replace the tuples of *attribute1*. I also explicitly specify that the resources and the mixins of the configuration remain unchanged. Only the attributes of one resource change (cf. lines 10 and 11).

5.3.3.4 Delete semantics

The following predicate shows how I formally specify the deletion of a resource.

```
pred DeleteResource[config : Configuration, resourceId : String,     1
    t, t' : Time] {
 // preconditions at instant t                                       2
    one resource : config.resources.t | resource.id = resourceId     3
 // postconditions at instant t'                                     4
    one resource : config.resources.t {                              5
      resource.id = resourceId                                       6
      config.resources.t' = config.resources.t - resource            7
    }                                                                8
    config.mixins.t' = config.mixins.t                               9
}                                                                    10
```

At time *t*, I specify that a configuration, passed as argument to the predicate, has a resource with the id passed as an argument (cf. line 3). At time *t'*, I remove from the configuration resources, one resource with the id passed as argument to the predicate (cf. lines 5 to 7). The "-" operator denotes set difference. I explicitly specify that the mixins of the configuration remain unchanged (cf. line 9).

5.3.4 Identifying & Validating FCLOUDS Properties

Using formal languages has the advantage of allowing reasoning over concepts and operational semantics for a better understanding of their semantics and how they work. Therefore, once the FCLOUDS formal language has been specified, I proceed by the definition of some structural and behavioural properties to ensure its correctness and to express its desired/required behaviour. I formally encode the *consistency*, *sequentiality*, *reversibility*, *idempotence* and *safety* behavioural properties. The last two properties are classified into a broader property which is the *conformance to HTTP 2 protocol* [Belshe 2015]. Then, using the Alloy analyzer, I validate that these properties adequately hold in my FCLOUDS static and operational semantics. To express these properties, `assertions` are written and are expected to hold as consequence of the specified constraints. In order to be confident that an assertion holds, I check it within a reasonable scope, *i.e.,* I bound the number of atoms allowed for each signature to 10. If no counterexamples are returned by the Alloy analyzer with such a scope, I can be confident that my language reflects the desired semantics.

5.3.4.1 Consistency

FCLOUDS language is consistent if it does not contain any contradictory constraints, so its concepts can be instantiated and each cloud API operation can be executable. I can also analyze what could not be instantiated, thus can't be deployed in real-world. In these cases, my formal language might be over-constraining so I deem necessary to relax some constraints. The `CreateResourceIsConsistent` assertion below can't be shown to have a counterexample. Hence, it asserts the existence of a valid configuration that meets the pre- and post-conditions of `Create Resource`, *i.e.,* it is consistent and expresses the desired behaviour.

```
assert CreateResourceIsConsistent {
 all config : Configuration, resourceId : String, resourceKind :
    Kind, mixins : Mixin, attributes : AttributeState, t : Time |
  CreateResource [config, resourceId, resourceKind, mixins,
     attributes, t, t.next]
  implies {
   no resource : config.resources.t | resource.id = resourceId
   resourceKind in config.use.kinds
   and one resource : config.resources.(t.next) |
     resource.id = resourceId
     and resource.kind = resourceKind
  }
}
```

Table 5.2: Properties of the FCLOUDS Language.

Properties	Consistency	Idempotence	Safety
Static Semantics	+	N/A	N/A
Operational Semantics (OCCI REST Operations)			
`Create Resource`	+	+	-
`Retrieve Resource`	+	+	+
`Update Resource`	+	-	-
`Delete Resource`	+	+	-
`Create Link`	+	+	-
`Retrieve Link`	+	+	+
`Update Link`	+	-	-
`Delete Link`	+	+	-

Properties	Sequentiality	Reversibility
Pairs of OCCI REST Operations		
`Create Resource & Retrieve Resource`	+	-
`Retrieve Resource & Create Resource`	-	-
`Retrieve Resource & Update Resource`	+	-
`Update Resource & Retrieve Resource`	-	-
`Update Resource & Delete Resource`	-	-
`Delete Resource & Update Resource`	-	-
`Delete Resource & Create Resource`	-	+
`Create Resource & Delete Resource`	+	+
`Create Link & Retrieve Link`	+	-
`Retrieve Link & Create Link`	-	-
`Retrieve Link & Update Link`	+	-
`Update Link & Retrieve Link`	-	-
`Update Link & Delete Link`	-	-
`Delete Link & Update Link`	-	-
`Delete Link & Create Link`	-	+
`Create Link & Delete Link`	+	+

Note that to model finite execution traces, Alloy defines a library *util/ordering* that provides useful relations to manipulate the total order of `Time` concept, namely *first* to denote the first time, and *next*, a binary relation that, given a time returns the following time in the order.

The FCLOUDS static semantics and all OCCI REST operations were proven to be consistent, as shown in Table 5.2. However, the notion of consistency is basic and does not suffice in order to validate my FCLOUDS language. There are other examples of reliable verification and validation tasks that can be performed on pairs of operations such as sequentiality and/or reversibility of operations.

5.3.4.2 Reversibility

Two cloud API operations are reversible when they contain inverse mathematical logic. For example, de-provisioning a virtual machine reverses the operation of provisioning it. In OCCI, `Create Resource` and `Delete Resource`, `Create Link` and `Delete Link` are reversible. The following assertion, which is checking that `Create Resource` is reversed by `Delete Resource`, was proven to be valid.

```
assert DeleteResourceReverseCreateResource {
  all config : Configuration, resourceId : String, kind : Kind,
     mixins : Mixin, attributes : AttributeState,  t : Time {
    CreateResource[config, resourceId, kind, mixins, attributes,
       t, t.next]
    implies DeleteResource[config, resourceId, t.next, t]
   }
}
```

5.3.4.3 Sequentiality

Two cloud API operations are sequential when one cannot happen if the other one did not happen at the time before. For example, the developer can adapt the performance of a virtual machine only if it was created before.

It is explicitly stated in the informal specification of OCCI that `Update Resource` operation should be preceded by `Retrieve Resource` operation: *"Before updating a resource instance it is RECOMMENDED that the client first retrieves the resource instance"* [Nyrén 2016a]. I also explicitly specify in FCLOUDS that `Retrieve Resource` operation must be preceded by `Create Resource` operation (as shown in the following assertion), `Update Link` by `Retrieve Link` and `Retrieve Link` by `Create Link`.

```
assert CreateResourceThenRetrieveResource {
  all config : Configuration, resourceId : String, kind : Kind,
     mixins : set Mixin, attributes : set AttributeState, t :
     Time  |
    CreateResource[config, resourceId, kind, mixins, attributes,
       t, t.next]
    and RetrieveResource[config, resourceId, t.next, t.next.next]
    implies one resource : config.resources.(t.next.next) {
      resource.id = resourceId
      and resource.kind = kind
      and resource.mixins.(t.next.next) = mixins
    }
}
```

5.3.4.4 Conformance to HTTP 2 Protocol

As OCCI is a REST architecture that conforms to the HTTP protocol, it must conform to its specification too. Therefore, there are some imposed properties I must have in any REST-based systems so they must be checked in the FCLOUDS

language. According to the *Request For Comments* (RFC) HTTP 2 [Belshe 2015], I identified two properties of the HTTP methods and I verified in my formal language that the appropriate pairs of operations respect these properties.

1. **Idempotence:** a method is idempotent when it always produces the same server external state even if applied several times [Belshe 2015]. In HTTP, GET, PUT and DELETE methods are *idempotent*. In OCCI, `Retrieve` operation is associated with a GET HTTP method, `Create` operation is a PUT HTTP method and `Delete` operation is a DELETE HTTP method. So I verify in my formal language that `Retrieve Resource`, `Retrieve Link`, `Create Resource`, `Create Link`, `Delete Resource` and `Delete Link` are idempotent. For example, as a result, the following assertion is valid:

```
assert CreateResourceIsIdempotent {
  all config : Configuration, resourceId : String, kind :
     Kind, mixins : Mixin, attributes : AttributeState,  t :
      Time |
    CreateResource [config, resourceId, kind, mixins,
       attributes, t, t.next]
    and CreateResource [config, resourceId, kind, mixins,
       attributes, t.next, t.next.next]
    implies config.resources.(t.next) =
    config.resources.(t.next.next)
}
```

   This assertion checks if creating a resource induces the same configuration at times *t.next* and *t.next.next*. In contrast, an `Update` operation, referred to as a POST in HTTP, is not idempotent. Therefore, the accuracy of my FCLOUDS language is maintained if the following assertion is not valid:

```
assert UpdateResourceIsIdempotent
```

2. **Safety:** a method is safe when it does not change the external server state [Belshe 2015]. It mainly concerns the retrieval of information. A *safe* method is necessarily an *idempotent* method, but not the reverse way. In HTTP, a GET method is *safe*. Therefore, in FCLOUDS, I check that `Retrieve Resource` and `Retrieve Link` respect this property, so they do not change the cloud configuration. An example of a *safe* operation is detailed below:

```
assert RetrieveResourceIsSafe {
  all config : Configuration, resourceId : String, t : Time |
    RetrieveResource[config, resourceId, t, t.next]
    implies config.resources.t = config.resources.(t.next)
    and config.mixins.t = config.mixins.(t.next)
    and one resource : config.resources.(t.next) {
      resource.id = resourceId
      resource.attributes.t = resource.attributes.(t.next)
          resource.links.t = resource.links.(t.next)
    }
}
```

This assertion checks if a configuration remains the same at time *t*, *i.e.,* before retrieving the resource, and at time *t.next*, *i.e.,* after retrieving the resource.

Table 5.2 lists all the operations that I have modeled, as well as all the properties that have been checked. The "+" symbol represents the operations or the pairs of operations that should fulfill a property, while the "-" represents the operations or the pairs of operations that they should not fulfill this property. By using the Alloy analyzer, I check that the FCLOUDS language, the core language of my FCLOUDS framework, correctly reflects these properties, so I guarantee that it is valid and that I implemented the desired behaviour.

5.4 Evaluation of FCLOUDS

To validate the effectiveness of my formal language, I demonstrate how it can be easily adapted to different concerns by providing formal specifications in Alloy for OCCI extensions from different cloud application domains. Therefore, I have surveyed the literature to find all the already published OCCI extensions. I have identified thirteen works that belong to IaaS, PaaS and *Internet of Things* (IoT) domains, as well as to transverse cloud concerns. As a working hypothesis, I have assumed that all these extensions are correct as they were already accepted through a peer review process. My validation allows me to confirm:

1. the power of expression of my FCLOUDS language (Subsection 5.4.1),
2. the validity of the FCLOUDS behaviour I defined, on all of the OCCI extensions (Subsection 5.4.3),
3. the ability of my language to define domain-specific properties (Subsection 5.4.4), and,

4. its ability to encode equivalence predicates, *i.e.,* transformation rules between heterogeneous concepts, and to define properties of the equivalence (Subsection 5.4.5).

5.4.1 Catalog of Cloud Formal Specifications

Thanks to my proposed formal language, I succeeded to precisely encode thirteen heterogeneous APIs, as shown in Table 5.3 that gives a summary of my FCLOUDS framework dataset. For each concept of my language, Table 5.3 provides the number (#) of instances of this concept present in the dataset. The last line provides the total of FCLOUDS concepts present in the dataset. For brevity, I give in the following excerpts of the formal APIs' specifications, implemented beforehand as OCCI extensions. Full specifications for each of these thirteen extensions can be found in the supplemental material here[2].

1. OCCI INFRASTRUCTURE [Nyrén 2016c] is an OCCI-based extension for IaaS application domain. It defines compute, storage and network resource types and associated links. It defines five kinds such as `Compute`, six mixins such as `IpNetworkInterface`, and around twenty data types such as `Vlan` range. The `Compute` kind represents a generic information processing resource, *e.g.,* a virtual machine or container. It inherits the `Resource` kind defined in the OCCI Core Model. `Compute` has a set of OCCI attributes that I declare as fields to my `Compute` signature, such as `occi.compute.architecture` to specify the CPU architecture of the instance, `occi.compute.core` to define the number of virtual CPU cores assigned to the instance, `occi.compute.memory` to define the maximum RAM in gigabytes allocated to the instance, etc. The **lone** keyword signifies that the relation between two tuples from two sets, such as `occi.compute.architecture` and `Architecture`, has a 0..1 cardinality.

```
sig Compute extends fclouds/Resource {
    occi_compute_architecture : lone Architecture,
    occi_compute_cores : lone Core,
    occi_compute_hostname  : lone String,
    occi_compute_share : lone Share,
    occi_compute_speed : lone GHz,
    occi_compute_memory : lone GiB,
    occi_compute_state : one ComputeStatus,
    occi_compute_state_message : lone String
}
```

[2] https://github.com/occiware/fclouds-Framework

Table 5.3: Summary of the FCLOUDS Framework Dataset.

Extension	#Kind	#Mixin	#Attribute	#Action	#DataType
IaaS					
OCCI INFRASTRUCTURE	5	6	31	9	20
OCCI CRTP	0	6	18	0	0
DOCKER	24	0	251	7	2
GCP	150	0	2348	985	398
VMWARE	6	7	19	0	1
PaaS					
OCCI PLATFORM	3	4	11	4	3
MODMACAO	1	31	9	0	2
IoT					
OMCRI	5	9	20	15	2
COT	6	4	21	0	3
Transverse cloud concerns					
OCCI SLA	2	2	8	5	4
OCCI MONITORING	2	3	9	0	2
CLOUD SIMULATION	8	14	53	0	0
CLOUD ELASTICITY	2	4	23	4	5
Total	214	90	2821	1029	442

2. OCCI CRTP [DRESCHER 2016] is an OCCI-based extension that defines a set of preconfigured instances of the OCCI `Compute` resource type.

3. DOCKER is a lightweight container for deploying and managing applications. Docker is implemented as an extension of OCCI in [Paraiso 2016]. It defines generic and specific container and machine resource types and associated links.

4. GCP is one of the leaders among cloud providers. It offers several service such as *Compute*, *Storage*, *Network*, *Management*, *Big Data* and *Security*. GCP was implemented as OCCI extension in [Challita 2018a] and as I presented in Chapter 4. I present in the following the formal specification of the `Instance` resource type, which represents a virtual machine in GCP.

```
sig Instance extends fclouds/Resource {
    creationTimestamp : one String,
    name : one String,
    description : one String,
    machineType : one String,
    status : one StatusEnum,
    statusMessage : one String,
    zone : one String
    disks : one DiskRecord,
```

```
    cpuPlatform : one String,
    labels : one Map,
    minCpuPlatform : one String,
    guestAccelerators : one GuestAcceleratorRecord,
    startRestricted : one Boolean,
    deletionProtection : one Boolean,
    kind : one String
}
```

5. VMware is a virtualization and cloud computing software provider and it is implemented as an extension of OCCI in [Zalila 2017b]. It defines VMware instance, storage and network resource types and associated links.

6. OCCI Platform [Metsch 2016] is an OCCI-based extension for PaaS application domain. It defines application and component resource types and associated links.

7. MoDMaCAO [Korte 2018] is an application of OCCI for managing cloud applications. It defines generic and specific application, component, installation dependency and execution dependency resource types.

8. OMCRI [Merle 2017] is an application of OCCI for Robot-as-a-Service domain. The OMCRI extension defines generic and specific robot resource types.

9. CoT is an application of OCCI for seamlessly provisioning cloud and IoT resources [Rachkidi 2017].

10. OCCI SLA [Katsaros 2016] defines OCCI types for modeling service level agreements.

11. OCCI Monitoring [Ciuffoletti 2016] is a draft specification of OCCI that defines sensor and collector types for monitoring cloud systems.

12. Cloud Simulation [Ahmed-Nacer 2016a, Ahmed-Nacer 2017] is an application of OCCI to simulate cloud systems. The Cloud Simulation extension defines two notions: *(i) a resource to simulate* that represents the resource to be simulated, and *(ii) a simulation resource* that represents the resource which performs the simulation activity.

13. Cloud Elasticity [Zalila 2017b] is an application of OCCI that defines a controller resource type to provide strategies for automatically provisioning and de-provisioning compute resources such as memory and cores.

5.4.2 Implementation of FCLOUDS Formal Specifications

To provide the formal specifications of OCCI extensions, I implemented Alloy Generator, which is an Acceleo [acc] generation module added to the OCCIware tool chain. Acceleo is a model-to-text generator, *i.e.,* as illustrated in Figure 5.5, it takes an OCCI extension and generates a text file, which an Alloy specification in our case. This specification conforms to the FCLOUDS specification.

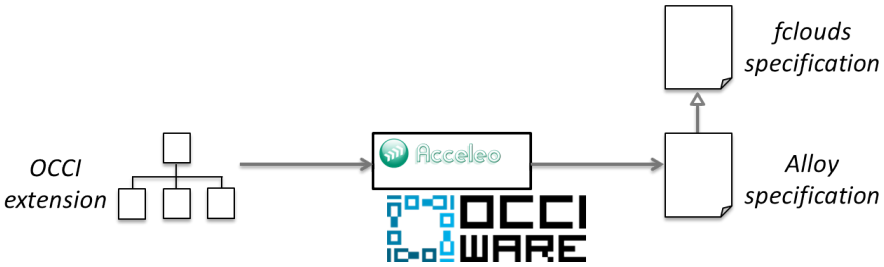


Figure 5.5: Alloy Generator.

Acceleo uses a template-based approach. This approach mixes static parts, which are raw text that will be outputted as it is, and templates, which are text that contains special part which will be filled with information from the model. Let's consider the example in Figure 5.6, which represents the template that is responsible of generating an Alloy signature for each kind of an OCCI extension. “one sig” and “extends Kind” are static parts, whereas the values between brackets will be replaced by values of the extension kinds (cf. line 1). In this template, we can see that there are conditionals on concepts (cf. lines 3, 5 and 7). For instance, if there is no actions, then we print “no actions”. Otherwise, for each action, we use a dedicated template (cf. line 7). In fact, in our Alloy Generator, I created templates for each OCCI concept to create the corresponding Alloy concept.

Template: sample of the “Kind” template.

```
1: one sig [kind.term/] extends Kind  {
2:       scheme = [kind.scheme/]
3:       [if (kind.parent = null)] no parent
4:       [else] parent = [kind.parent/] [/if]
5:       [if (kind.attributes.isEmpty())] no attributes
6:       [else] attributes = [generateAttributes/][/if]
7:       [if (kind.actions.isEmpty())] no actions
8:       [else] actions = [generateActions/] [/if]
9: }
```

Figure 5.6: Acceleo Template.

5.4.3 Verification of FCLOUDS Properties

Being rigorously encoded using the same formal language, *i.e.,* FCLOUDS, my thirteen case studies can be now accessed by the same OCCI RESTful interface. Therefore, it is important to make sure that they correctly reflect the behaviour of FCLOUDS. Using the Alloy analyzer, I verify that my thirteen formal specifications satisfy all the assertions, *i.e.,* properties, that I formulated in my FCLOUDS language (cf. Table 5.2). For instance, I verify that `Create Compute` operation of OCCI INFRASTRUCTURE is idempotent and that `Update Instance` operation of GCP is not safe.

5.4.4 Definition & Validation of Domain-Specific Properties

My framework allows me to verify some domain-specific properties. For instance, in the following listing, I check whether creating a `NetworkInterface` between two OCCI resources only occurs between one `Compute` resource type and one `Network` resource type. This assertion is validated so my formal specification respects and implements the following requirement of OCCI INFRASTRUCTURE specification: *"NetworkInterface connects a Compute instance to a Network instance"* [Nyrén 2016c].

```
assert NetworkInterfaceBetweenComputeAndNetwork {
 all config : Configuration, linkId : String, linkKind :
    networkinterface, linkSource : fclouds/Resource, linkTarget :
     fclouds/Resource, t : Time |
  CreateLink[config, linkId, linkKind, linkSource, linkTarget, t,
      t.next]
  implies one config : Configuration {
   one resourceCompute : config.resources.t {
    resourceCompute.hasKind[compute]
    one link : fclouds/Link {
     link.id = linkId
     link in resourceCompute.links.(t.next)
    }
   }
   one resourceNetwork  : config.resources.t {
    resourceNetwork.hasKind[network]
    one link : fclouds/Link {
     link.id = linkId
     link in resourceNetwork.links.(t.next)
    }
   }
  }
}
```

5.4.5 Transformation Rules for Semantic Interoperability in Multi-clouds

The last step for achieving semantic interoperability between heterogeneous domains is to define predicates that implement bidirectional formal transformation rules to map between resources with similar semantics. In the following example, I show how I can migrate from an OCCI Infrastructure virtual machine at to a GCP virtual machine. I map the attributes of a given `Compute` resource to those of an `Instance` resource. Also, since I learned from the OCCI Infrastructure and GCP documentations that the memory in OCCI is expressed in gigabytes, whereas it is in megabytes in GCP, I applied the multiplication operator for the conversion.

```
pred ComputeMapInstance[c : one Compute, i : one Instance] {
        i.machinetype.guestCpus = c.occi_compute_cores
        i.name = c.occi_compute_hostname
        i.machinetype.isSharedCpu = c.occi_compute_share
        i.machinetype.memoryMb = mul[1024, c.occi_compute_memory]
        i.status = c.occi_compute_state
        i.statusMessage = c.occi_compute_state_message
}
```

Such formal equivalence rules and properties are capable of gaining huge time and development costs. The cloud developer can now verify a priori the feasibility of his/her multi-cloud system, before embarking on error-prone implementations. Thanks to my transformation rules and to MDE principles, I can later imagine a model that factorizes common attributes for re-usability between OCCI Infrastructure and GCP.

5.5 Summary

This chapter presented the fclouds framework, my approach that relies on formal specification techniques, specifically on the Alloy language, to rigorously and clearly describe the requirements of cloud APIs. I formalize the OCCI standard in order to implement my formal language for the clouds. My formal specification of OCCI was checked for validity thanks to the Alloy analyzer that provides a verification backbone for OCCI properties. To demonstrate the usefulness of my approach, I conducted thirteen case studies to show how my approach is applied on OCCI extensions and that these thirteen APIs with different functionality verify the OCCI properties so they correctly comply to the OCCI standard. Also, applying Alloy to these APIs allowed me to reflect the proper behavior of each API by identifying

and validating its specific properties. Finally, having rigorously specified the static and operational semantics of each cloud API, I defined formal transformation rules between their formal specifications, thus ensure semantic interoperability between them.

This chapter concludes the fourth part of this dissertation, *i.e.,* the contributions. In Chapter 4 I described how I automatically extracted a precise model from GCP documentation to provide an accurate description of GCP API. Afterwards, Chapter 5 described how I formally verified that cloud APIs reflect the desired behaviour and how I achieved semantic interoperability between their concepts with the FCLOUDS framework.

In the following part, I summarize the main contributions of this thesis, present the conclusions of the research work, and define a set of perspectives for future work.

Part V

Conclusion

The last part of the manuscript presents a summary, future perspectives and final remarks of this thesis.

Chapter 6

Conclusions and Perspectives

Contents

This last chapter summarizes the contributions of this thesis and discusses future research lines of the work presented in this dissertation. It is structured as follows. Section 6.1 recapitulates the OCCIware research project that supports this thesis. In Section 6.2, I summarize the contributions of this thesis. Section 6.3 states short-term and long-term perspectives to extend this research. Section 6.4 concludes this manuscript.

6.1 Background Summary

This thesis is supported by the French OCCIware project and promotes its advancement. OCCIware addresses:

- **RQ#1:** *Is it possible to have a solution that allows to represent all kinds of cloud resources despite their heterogeneity, and a complete framework for managing them?*

OCCIware platform is summarized as follows:

A complete modeling, verification, generation and management support. MDE approaches have proven to be advantageous to address the heterogeneity across cloud providers. However, the existing MDE approaches for the cloud, are limited to designing cloud infrastructures. Cloud developers need support to deploy and manage, not only design all the kinds of cloud resources. Automating

the deployment of cloud artifacts and managing different types of cloud resources at runtime are not straightforward. To tackle these issues, I introduced the OCCIware approach in Chapter 3. This solution is based on well accepted and defined standards and technologies. In particular, OCCIware relies on the OCCI standard that proposes a generic model and API for managing any kind of cloud computing resources. Also, OCCIware exploits the principles of MDE and leverage them to provide modeling, verification, generation and management support for cloud extensions and configurations. The OCCIware approach especially provides the OCCIware Metamodel, which is based on an Ecore syntax and allows to define cloud concepts with OCL constraints over them. The OCCIware Metamodel can be seen as a domain-specific modeling language to define and exchange OCCI extensions and configurations between end-users and resource providers. For tooling purpose, the OCCIware Studio is a tool chain built on top of the OCCIware Metamodel. The OCCIware Studio allows both cloud architects and users can encode OCCI extensions and configurations, respectively, graphically via the **OCCIware Designer** tool, and textually via the **OCCIware Editor** tool. They can also automatically verify the consistency of these extensions and configurations via the **OCCIware Validator** tool, generate dedicated model-driven tooling via both **Ecore Generator** and **Connector Generator** tools, generate a deployment script via the **CURL Generator** tool, and manage their configurations at runtime via the generated connectors deployed in OCCIware Runtime.

6.2 Contributions Summary

The two contributions of this thesis are made as part of my work around the OCCIware platform. The first contribution addresses:

- **RQ#2:** *Is it possible to automatically extract precise models from cloud APIs and to synchronize them with the cloud evolution?*

The second contribution addresses:

- **RQ#3:** *Is it possible to reason on cloud APIs and identify their similarities and differences?*

The two contributions are respectively summarized as follows:

An automated knowledge extraction support. As cloud environments evolve over time, their models have to evolve as well to be kept up-to-date. However, nearly all the existing models that represent the cloud environment are manually

built, which is tedious and error-prone. In addition, the defined vocabulary of the existing cloud models is not rich enough to cover the heterogeneity of all existing resources. It only considers the lowest common denominator of the cloud providers. It should also be noted that the defined vocabulary is not flexible enough and there is no information provided on how the developer can extend this vocabulary. In Chapter 4, I thus introduce my approach of inferring precise models from cloud textual documentations. To experiment this approach, I studied the documentation of GCP that presents various drawbacks. Later on, I implemented a crawler that automatically extracts GCP resource types, their attributes and operations. These resources are stored in GCP Model that conforms to the OCCIware Metamodel and is built as an OCCI extension using the OCCIware Studio. I showed that using MDE to specify GCP API improve the specification of GCP, especially via the model transformations like refining the types of the attributes and detecting implicit metadata. A precise specification, accompanied by a validator at design time, helps the developer to ensure correctness of his/her GCP configurations before their deployment. I also showed that using such a model-driven specification leads to a redundancy reduction, even with a simple transformation, *i.e.,* by introducing a single abstract kind to the specification. My approach also allowed me to deduce some facts regarding the *uniformity*, *conciseness*, *consistency* and *comprehensiveness* of GCP API.

A formal specification support. Cloud solutions (APIs, services, standards and model-driven approaches) are numerous and heterogeneous. Moreover, there is no clear consensus on how these solutions work. And although MDE approaches allow the developer to validate cloud configurations before their deployment through OCL constraints, the developer needs to logically think and understand the cloud solutions. Developers need a formal specification of cloud solutions, fclouds is the first approach ensuring this support. As explained in Chapter 5, fclouds encapsulates the OCCIware Metamodel and helps developers understand without ambiguity the static semantics of cloud environments and the behavioural semantics of their operations. From the developer's perspective, fclouds acts a black box: the developer defines OCCI extensions using OCCIware Studio as entry point of fclouds, and retrieves a formal specification of each extension. This specification is written in Alloy which is a lightweight relational formal language based on the first-order logic. The Alloy syntax is simple and easy to use, and through my experimentation I demonstrated that it is expressive enough to specify different cloud concerns. In addition, by using the Alloy analyzer, developers can check the consistency of their extensions, the sequentiality of certain couples of

cloud operations, the reversibility of others, as well as the idempotence and safety of certain operations. If counterexamples are found, the developers can detect inconsistencies in their extensions. Finally, I describe transformation rules for mapping concepts from a cloud solution to another and thus, I make cloud solutions more semantically interoperable.

Together, these two contributions support the automated construction of formal cloud specifications by handling the complexities linked to the natural language documentations of cloud providers. They provide means to specify cloud concepts and operations and a way for verifying their consistency and various other properties. In addition, they provide means for defining transformation rules, thus ensuring semantic interoperability among cloud providers.

6.3 Perspectives

In this dissertation, I presented my work that successfully covers the needs of precisely modeling and verifying cloud resources and reasoning over them. However, there is still a lot of work that can be done to improve my research. In this section I thus discuss some short-term and long-term perspectives that should be considered in the continuation of this work.

6.3.1 Short-term Perspectives

Broadening the validation scope of OCCIware. The OCCIware approach was successfully validated by designing and managing, using the OCCIware Studio and the OCCIware Runtime respectively, five OCCI extensions and three academic uses cases. Hereafter, we target industrial validation for the OCCIware approach. Therefore, an ongoing work aims to get this approach tested and adopted within Scalair [scaa], a hybrid cloud provider. Also, in order to cover the whole cloud market, the Xscalibur [xsc] start-up is currently developing the Multi-Cloud Studio[1] that supports two other cloud providers: AWS and OpenStack. Both OCCI extensions for AWS and OpenStack are under development[2] in order to provide a modeling studio to design both AWS and OpenStack configurations.

Generating a new textual documentation from GCP Model and evaluating it. I aim to generate from GCP Model, thanks to the OCCIware Stu-

[1] https://github.com/occiware/Multi-Cloud-Studio

[2] Available here https://github.com/occiware/Multi-Cloud-Studio/tree/master/plugins/org.eclipse.cmf.occi.multicloud.aws.ec2

DIO facilities, a new textual documentation of GCP API. Then, I aim to strengthen the validation of this documentation by conducting a survey to be taken by developers that are using GCP API. This survey will help us to verify how accurate the processed documentation is and if it actually saves their development time. Also, for ultimate measurement of our approach, we will contact Google employees who are in charge of GCP API, because we believe that their expertise is the most efficient for reviewing this work.

Providing a complete tool chain for GCP. For the moment, GCP MODEL is an enhanced and accurate specification of the GCP API that allows the developer to analyze and correctly understand its services. The developer would therefore make more effective use of GCP API and tend to write more efficient code or REST requests. GCP STUDIO, which is a dedicated model-driven environment for graphically and textually designing configurations that conform to GCP MODEL, is a work in progress. I aim to associate GCP STUDIO to a GCP connector in order to allow an effective provisioning of GCP resources and their management at runtime.

Extending the catalog of formal cloud APIs. As described in Chapter 5, I successfully specified thirteen cloud APIs by using my proposed FCLOUDS language. I aim to extend my catalog of formal cloud APIs in order to achieve my vision of building the first comprehensive framework for semantic interoperability in multi-clouds. Therefore, using the FCLOUDS language, I aim to formally specify AWS, OpenStack, TOSCA, etc.

6.3.2 Long-term Perspectives

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

Following the evolution of GCP API. I plan to update my approach so it would automatically handle the evolution of GCP API. At the moment, this evolution is manually ensured. For automating the process, it is more practical if my crawler is less related to the structure of GCP HTML pages, because in reality the latter are constantly updated. This can be done by experimenting artificial intelligence algorithms to extract knowledge from GCP documentation, then studying

whether the inferred GCP MODEL in this case will not be missing some information. Also, my model needs to incrementally detect streaming modifications, by calculating and modifying only the differences between the initially processed version and the newly modified one.

Dealing with additional types of properties. FCLOUDS allows to verify that the appropriate cloud operations satisfy these five properties: consistency, sequentiality, reversibility, idempotence and safety. For verifying different aspects of the cloud APIs, I aim to enrich FCLOUDS with additional properties such as *Reachability*, *i.e.,* when executing operations on cloud resources through APIs, there is always a transition from a resource state to another.

Considering further formal techniques. I proposed in this dissertation to use the Alloy formal language and its analyzer to formally specify cloud APIs and reason about them. However, being a SAT solver, Alloy is not effective for resolving numerical constraints which aim for example to minimize the cloud application cost. Therefore, I intend to use adequate heuristics like SMT solvers, which are obviously better than SAT solvers for such scenarios. For example, the TLA+ specification language and TLC, its model checker [Lamport 2002] are recognized and used to verify the reachability problem. Furthermore, although model checkers verify that the properties are valid within a big scope of research, I need to prove it in the absolute through a convincing argument. Hence, I will use automated proof assistants [Loveland 2016], namely Coq [Barras 1997], which implements algorithms and heuristics to build a proof describing the sequence of needed moves in order to solve a property.

Working on real-world interoperability. I presented in Chapter 5, a scenario that showed the usage of fclouds for ensuring *semantic interoperability*. It mainly consisted in two stages: modeling and reasoning. For the future, I aim to introduce *a third stage*, where my formal framework is incorporated in the development and maintenance of a bridge with a unified API, to promote *real-world interoperability*, while formal semantics is properly reflected in its behaviour. This third stage is depicted in (3) of Figure 6.1.

Improving the management of cloud applications. In this thesis, I focused on OCCI standard, which is developed by the OGF and aims to standardize an API for the management of any kind of cloud resources. Besides OCCI, TOSCA currently receives more attention by both the industry and research community, but their focus is different and they can be used complementarily. For managing

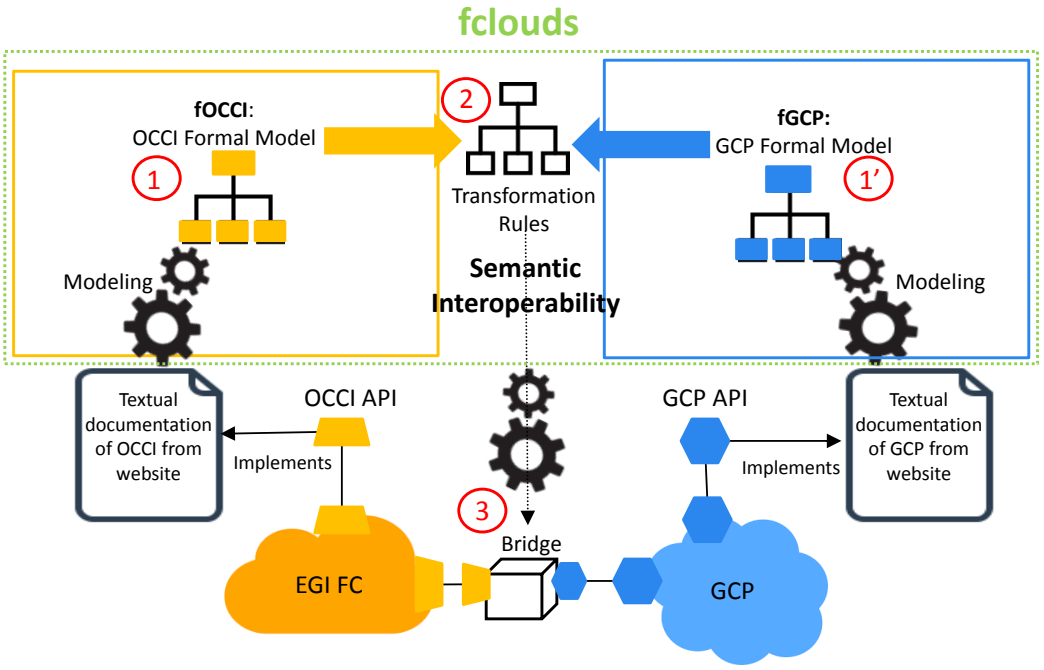


Figure 6.1: Formal Real-World Bridge.

cloud applications, I aim to implement a model-driven cloud orchestrator based on two complementary standards, TOSCA and OCCI. Therefore, I am refining the mapping between the concepts of these two standards [Glaser 2017], and building TOSCA Studio, a dedicated model-driven environment for designing applications with TOSCA. This approach will allow TOSCA to have a complementary tool to take better advantage of deployed applications in production environments. At runtime, TOSCA Studio will be able to: *(i)* communicate with an OCCI Infrastructure such as the EGI FC to provision virtual machines for example, or *(ii)* be exposed via the OCCI Plaform API in order to create, retrieve, update and delete any kind of cloud application resources.

6.4 Final Conclusion

To close this manuscript, two quotes synthesize the main idea of this thesis:

> *"No problem can be solved from the same consciousness that created it."*
>
> —Albert Einstein

This first quote remarks the value of changing the consciousness to resolve a challenge in life. In other terms, what got you to a problem is not going to get you out of

it. Therefore, proposing new cloud APIs to resolve the heterogeneity of the existing ones, will only worsen the problem. However, MDE brings new opportunities to improve the cloud solutions. This thesis evidences the value of rising in abstraction to face the heterogeneity in the cloud domain.

"Concision in style, precision in thought, decision in life."

—Victor Hugo

This second quote remarks the value of concision and precision to successfully make a decision. Eliminating redundancy while conveying the ideas without ambiguity, will lead to a better conclusion. By using formal methods to provide a concise and precise specification mechanism, this thesis evidences a better understanding of cloud APIs. This is crucial for taking efficient use of the cloud ecosystem and for making better decisions regarding the offers selection.

Bibliography

[acc] *Acceleo Website*. http://www.eclipse.org/acceleo/ (accessed on July 25, 2018). (Cited on pages 71 and 147.)

[aeo] *Aeolus ANR Project Website*. http://aeolus-project.org/ (accessed on May 27, 2018). (Cited on page 25.)

[Ahmed-Nacer 2016a] Mehdi Ahmed-Nacer and Samir Tata. *Simulation Extension for Cloud standard OCCIware*. In 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pages 263–264. IEEE, 2016. (Cited on pages 85 and 146.)

[Ahmed-Nacer 2016b] Mehdi Ahmed-Nacer, Samir Tata, Walid Gaaloul, Philippe Merle, Jean Parpaillon, Noël Plouzeau and Stéphanie Challita. *OCCI Behavioural Model*. OCCIware Deliverable 2.2.2, December 2016. (Cited on page 129.)

[Ahmed-Nacer 2017] Mehdi Ahmed-Nacer, Walid Gaaloul and Samir Tata. *OCCI-Compliant Cloud Configuration Simulation*. In IEEE International Conference on Edge Computing (EDGE), pages 73–81. IEEE, 2017. (Cited on pages 85 and 146.)

[ans] *Ansible Website*. https://www.ansible.com/ (accessed on June 17, 2018). (Cited on pages 28 and 91.)

[Ardagna 2012] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria*et al.* *MODAClouds: A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds*. In 4th International Workshop on Modeling in Software Engineering, pages 50–56. IEEE Press, 2012. (Cited on page 35.)

[Armbrust 2010] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica*et al.* *A View of Cloud Computing*. Communications of the ACM, vol. 53, no. 4, pages 50–58, 2010. (Cited on page 3.)

[Asserson 2002] Anne Asserson, Keith G Jeffery and Andrei Lopatenko. *CERIF: past, present and future: an overview*. 2002. (Cited on page 32.)

[aws] *Amazon Web Services Website*. https://aws.amazon.com/ (accessed on June 3, 2018). (Cited on page 24.)

[azu] *Microsoft Azure Website*. https://azure.microsoft.com/en-us/ (accessed on June 3, 2018). (Cited on page 24.)

[Barras 1997] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy*et al.* The Coq Proof Assistant Reference Manual: Version 6.1. 1997. (Cited on page 158.)

[Baryannis 2013] George Baryannis, Panagiotis Garefalakis, Kyriakos Kritikos, Kostas Magoutis, Antonis Papaioannou, Dimitris Plexousakis and Chrysostomos Zeginis. *Lifecycle Management of Service-based Applications on Multi-Clouds: A Research Roadmap*. In International workshop on Multi-cloud applications and federated clouds, pages 13–20. ACM, 2013. (Cited on page 21.)

[Belshe 2015] Mike Belshe, Martin Thomson and Roberto Peon. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. 2015. (Cited on pages 139 and 142.)

[Bencomo 2014] Nelly Bencomo, Robert B France, Betty HC Cheng and Uwe Aßmann. Models@run.time: Foundations, Applications, and Roadmaps, volume 8378. Springer, 2014. (Cited on page 115.)

[Benzadri 2013] Zakaria Benzadri, Faiza Belala and Chafia Bouanaka. *Towards a Formal Model for Cloud Computing*. In International Conference on Service-Oriented Computing, pages 381–393. Springer, 2013. (Cited on page 124.)

[Bergmayr 2013] Alexander Bergmayr, Hugo Bruneliere, Javier Luis Canovas Izquierdo, Jesus Gorronogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela*et al. Migrating legacy software to the cloud with ARTIST*. In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pages 465–468. IEEE, 2013. (Cited on page 35.)

[Bergmayr 2014] Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer and Gerti Kappel. *UML-based Cloud Application Modeling with Libraries, Profiles, and Templates**. In Proc. Workshop on CloudMDE, pages 56–65, 2014. (Cited on page 33.)

[Bergmayr 2018] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel and Frank Ley-

mann. *A Systematic Review of Cloud Modeling Languages*. ACM Computing Surveys (CSUR), vol. 51, no. 1, page 22, 2018. (Cited on pages 31 and 52.)

[Berners-Lee 1998] Tim Berners-Lee, Roy Fielding and Larry Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. Technical report, 1998. (Cited on page 69.)

[Binz 2012] Tobias Binz, Gerd Breiter, Frank Leymann and Thomas Spatzier. *Portable Cloud Services Using TOSCA*. IEEE Internet Computing, no. 3, pages 80–85, 2012. (Cited on pages 23, 33 and 38.)

[Binz 2013] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak and Sebastian Wagner. *OpenTOSCA-A Runtime for TOSCA-based Cloud Applications*. In Service-Oriented Computing, pages 692–695. Springer, 2013. (Cited on pages 35 and 38.)

[Blair 2009] Gordon Blair, Nelly Bencomo and Robert B France. *Models@run.time*. Computer, vol. 42, no. 10, pages 22–27, 2009. (Cited on pages 34 and 75.)

[Bobba 2017] Rakesh Bobba, Jon Grov, Indranil Gupta, Si Liu, José Meseguer, Peter C Olveczky and Stephen Skeirik. *Design, Formal Modeling, and Validation of Cloud Storage Systems Using Maude*. Technical report, 2017. (Cited on page 124.)

[Brambilla 2012] Marco Brambilla, Jordi Cabot and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering, vol. 1, no. 1, pages 1–182, 2012. (Cited on page 60.)

[Brandtzæg 2012] Eirik Brandtzæg, Sébastien Mosser and Parastoo Mohagheghi. *Towards CloudML, a Model-Based Approach to Provision Resources in the Clouds*. In 8th European Conference on Modelling Foundations and Applications (ECMFA), pages 18–27, 2012. (Cited on pages 32 and 34.)

[bro] *Brooklyn Website*. `https://brooklyn.apache.org/` (accessed on September 14, 2018). (Cited on page 32.)

[Bruneliere 2010] Hugo Bruneliere, Jordi Cabot and Frédéric Jouault. *Combining Model-Driven Engineering and Cloud Computing*. In Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010), 2010. (Cited on pages 21, 27 and 52.)

[Buyya 2009] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg and Ivona Brandic. *Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility*. Future Generation Computer Systems, vol. 25, no. 6, pages 599–616, 2009. (Cited on page 3.)

[Calheiros 2011] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose and Rajkumar Buyya. *CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms*. Software: Practice and Experience, vol. 41, no. 1, pages 23–50, 2011. (Cited on page 85.)

[cam] *OASIS CAMP specification*. http://docs.oasis-open.org/camp/camp-spec/v1.2/camp-spec-v1.2.pdf (accessed on June 6, 2018). (Cited on page 22.)

[Cao 2017] Hanyang Cao, Jean-Rémy Falleri and Xavier Blanc. *Automated Generation of REST API Specification from Plain HTML Documentation*. In the 15th International Conference on Service-Oriented Computing (ICSOC), pages 453–461. Springer, 2017. (Cited on page 100.)

[Carlson 2012] Mark Carlson, Martin Chapman, Alex Heneveld, Scott Hinkelman, Duncan Johnston-Watt, Anish Karmarkar, Tobias Kunze, Ashok Malhotra, Jeff Mischkinsky, Adrian Otto*et al. Cloud Application Management for Platforms*. Specification document, OASIS, 2012. (Cited on page 22.)

[cdm] *Storage Networking Industry Association (SNIA) Website*. https://www.snia.org/cdmi (accessed on June 3, 2018). (Cited on page 22.)

[Challita 2017a] Stéphanie Challita, Fawaz Paraiso and Philippe Merle. *A Study of Virtual Machine Placement Optimization in Data Centers*. In 7th International Conference on Cloud Computing and Services Science (CLOSER), 2017. (Cited on page 17.)

[Challita 2017b] Stéphanie Challita, Fawaz Paraiso and Philippe Merle. *Towards Formal-based Semantic Interoperability in Multi-Clouds: the fclouds Framework*. In 10th IEEE International Conference on Cloud Computing (CLOUD), pages 710–713. IEEE, 2017. (Cited on pages 17 and 121.)

[Challita 2018a] Stéphanie Challita, Faiez Zalila, Christophe Gourdin and Philippe Merle. *A Precise Model for Google Cloud Platform*. In 6th IEEE International Conference on Cloud Engineering (IC2E), pages 177–183. IEEE, 2018. (Cited on pages 16, 89, 95 and 145.)

[Challita 2018b] Stéphanie Challita, Faiez Zalila and Philippe Merle. *Specifying Semantic Interoperability between Heterogeneous Cloud Resources with the fclouds Formal Language*. In 11th International Conference on Cloud Computing (CLOUD), pages 367–374. IEEE, 2018. (Cited on pages 16 and 121.)

[Chapman 2012] Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman and Alex Galis. *Software Architecture Definition for On-Demand Cloud Provisioning*. Cluster Computing, vol. 15, no. 2, pages 79–100, 2012. (Cited on page 38.)

[che] *Chef Website*. https://www.chef.io/chef/ (accessed on June 17, 2018). (Cited on page 28.)

[Chowdhury 2003] Gobinda G Chowdhury. *Natural Language Processing*. Annual review of information science and technology, vol. 37, no. 1, pages 51–89, 2003. (Cited on page 98.)

[Ciuffoletti 2016] Augusto Ciuffoletti. *Open Cloud Computing Interface - Monitoring Extension*. Specification Document 1.2, Open Grid Forum, January 2016. (Cited on pages 63, 84 and 146.)

[Clo] *CloudMIG Xpress Website*. http://www.cloudmig.org/ (accessed on June 11, 2018). (Cited on page 34.)

[Cohen 2009] Reuven Cohen. *Examining Cloud Compatibility, Portability and Interoperability*. ElasticVapor: Life in the Cloud, 2009. (Cited on page 9.)

[Davis 2012] Doug Davis and Gilbert Pilz. *Cloud Infrastructure Management Interface (CIMI) Model and REST Interface over HTTP*. vol. DSP-0263, May 2012. (Cited on page 23.)

[dig] *DigitalOcean Website*. https://www.digitalocean.com/ (accessed on June 3, 2018). (Cited on page 24.)

[doc] *Docker Website*. https://www.docker.com/ (accessed on July 22, 2018). (Cited on page 86.)

[Drescher 2016] Michel Drescher, Boris Parák and David Wallom. *Open Cloud Computing Interface - Compute Resource Template Profile*. Specification Document GFD.222, Open Grid Forum, February 2016. (Cited on pages 55, 82 and 145.)

[EA] *Enterprise Architect Website*. http://www.sparxsystems.com/products/ea/ (accessed on May 27, 2018). (Cited on page 30.)

[ecl] *Eclipse Website*. http://www.eclipse.org/ (accessed on July 25, 2018). (Cited on page 71.)

[Edmonds 2012] Andy Edmonds, Thijs Metsch, Alexander Papaspyrou and Alexis Richardson. *Toward an Open Cloud Standard*. IEEE Internet Computing, vol. 16, no. 4, pages 15–25, 2012. (Cited on pages 10, 23, 50 and 53.)

[Edmonds 2016] Andy Edmonds and Thijs Metsch. *Open Cloud Computing Interface - Text Rendering*. Specification Document GFD-R-P.229, Open Grid Forum, 2016. (Cited on pages 55, 69, 75 and 81.)

[egi] *EGI FC Website*. https://www.egi.eu/ (accessed on June 13, 2018). (Cited on page 24.)

[EMFa] *Eclipse Modeling Framework (EMF) Website*. http://www.eclipse.org/modeling/emf/ (accessed on May 27, 2018). (Cited on page 30.)

[emfb] *EMFText Website*. http://www.emftext.org/ (accessed on May 27, 2018). (Cited on page 31.)

[ero] *erocci Website*. http://erocci.ow2.org (accessed on August 13, 2018). (Cited on page 50.)

[Farokhi 2014] Soodeh Farokhi. *Towards an SLA-Based Service Allocation in Multi-Cloud Environments*. In 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 591–594. IEEE, 2014. (Cited on page 35.)

[Ferrer 2012] Ana Juan Ferrer, Francisco HernáNdez, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, RaüL Sirvent, Jordi Guitart, Rosa M Badia, Karim Djemame*et al.* *OPTIMIS: A Holistic Approach to Cloud Service Provisioning*. Future Generation Computer Systems, vol. 28, no. 1, pages 66–77, 2012. (Cited on page 25.)

[Ferry 2013] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin and Arnor Solberg. *Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-Cloud Systems*. In IEEE 6th International Conference on Cloud Computing (CLOUD), pages 887–894. IEEE, 2013. (Cited on pages 32 and 34.)

[Ferry 2018] Nicolas Ferry, Franck Chauvel, Hui Song, Alessandro Rossini, Maksym Lushpenko and Arnor Solberg. *CloudMF: Model-Driven Management of Multi-Cloud Applications*. ACM Transactions on Internet Technology (TOIT), vol. 18, no. 2, pages 16:1–16:24, 2018. (Cited on page 34.)

[Fielding 2000] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. (Cited on pages 8 and 50.)

[fle] *FlexiScale Website*. `http://www.flexiscale.com/` (accessed on June 6, 2018). (Cited on page 24.)

[fog] *Fog Website*. `https://fog.io/` (accessed on June 3, 2018). (Cited on page 26.)

[Fowler 2010] Martin Fowler. Domain-Specific Languages. Pearson Education, 2010. (Cited on pages 31 and 108.)

[Frey 2011] Sören Frey and Wilhelm Hasselbring. *The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications*. International Journal on Advances in Software, vol. 4, no. 3 and 4, pages 342–353, 2011. (Cited on page 34.)

[Frey 2013] Sören Frey, Florian Fittkau and Wilhelm Hasselbring. *Search-Based Genetic Optimization for Deployment and Reconfiguration of Software in the Cloud*. In International Conference on Software Engineering, pages 512–521. IEEE Press, 2013. (Cited on page 35.)

[García-Galán 2016] Jesús García-Galán, Pablo Trinidad, Omer F Rana and Antonio Ruiz-Cortés. *Automated Configuration Support for Infrastructure Migration to the Cloud*. Future Generation Computer Systems, vol. 55, pages 200–212, 2016. (Cited on page 36.)

[Garis 2012] Ana Garis, Ana CR Paiva, Alcino Cunha and Daniel Riesco. *Specifying UML Protocol State Machines in Alloy*. In International Conference on Integrated Formal Methods, pages 312–326. Springer, 2012. (Cited on page 136.)

[gcp] *Google Cloud Platform Website*. `https://cloud.google.com/` (accessed on June 3, 2018). (Cited on page 24.)

[Gherardi 2014] Luca Gherardi, Dominique Hunziker and Gajamohan Mohanarajah. *A Software Product Line Approach for Configuring Cloud Robotics Applications*. In 7th IEEE International Conference on Cloud Computing (CLOUD), pages 745–752. IEEE, 2014. (Cited on page 36.)

[Glaser 2017] Fabian Glaser, Johnannes Erbel and Jens Grabowski. *Model Driven Cloud Orchestration by Combining TOSCA and OCCI*. In 7th International

Conference on Cloud Computing and Services Science (CLOSER), pages 644–650, 2017. (Cited on page 159.)

[gmf] *Graphical Modeling Framework (GMF) Website.* https://www.eclipse.org/gmf-tooling/ (accessed on May 27, 2018). (Cited on page 31.)

[gop] *Gophercloud Website.* http://gophercloud.io/ (accessed on June 3, 2018). (Cited on page 26.)

[gra] *Graphiti Website.* https://www.eclipse.org/graphiti/ (accessed on May 27, 2018). (Cited on page 31.)

[Guillén 2013] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo and Carlos Canal. *A UML Profile for Modeling Multicloud Applications.* In Service-Oriented and Cloud Computing, pages 180–187. Springer, 2013. (Cited on page 37.)

[Hamdaqa 2015] Mohammad Hamdaqa and Ladan Tahvildari. *StratusML: A Layered Cloud Modeling Framework.* In IEEE International Conference on Cloud Engineering (IC2E), pages 96–105, 2015. (Cited on page 39.)

[Haupt 2017] Florian Haupt, Frank Leymann, Anton Scherer and Karolina Vukojevic-Haupt. *A Framework for the Structural Analysis of REST APIs.* In the International Conference on Software Architecture (ICSA), pages 55–58. IEEE, 2017. (Cited on page 100.)

[her] *Heroku Website.* https://www.heroku.com/ (accessed on June 3, 2018). (Cited on page 24.)

[Holmes 2014] Taíd Holmes. *Automated Provisioning of Customized Cloud Service Stacks using Domain-Specific Languages.* CloudMDE 2014, pages 46–55, 2014. (Cited on page 36.)

[Holmes 2015] Taíd Holmes. *Facilitating Migration of Cloud Infrastructure Services-A Model-Based Approach.* 3rd International Workshop on Model-Driven Engineering on and for the Cloud in conjunction with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, pages 7–12, 2015. (Cited on page 37.)

[Hußmann 2001] H Hußmann. *Fundamental Approaches to Software Engineering (FASE).* In 5th International Conference held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS). Springer, 2001. (Cited on page 30.)

[Jackson 2012] Daniel Jackson. Software Abstractions: logic, language, and analysis. MIT press, 2012. (Cited on pages 10, 12, 73, 122, 128 and 129.)

[jcl] *Apache jclouds Website*. http://www.jclouds.org/ (accessed on May 27, 2018). (Cited on page 26.)

[Jeffery 2017] Keith Jeffery and Lutz Schubert. *PaaSage*. IEEE Cloud Computing, vol. 4, no. 3, pages 60–60, 2017. (Cited on page 35.)

[Jurafsky 2000] Daniel Jurafsky. *Speech and Language Processing: An Introduction to Natural Language Processing*. Computational linguistics, and speech recognition, 2000. (Cited on page 113.)

[kaa] *Kaavo Website*. http://www.kaavo.com/ (accessed on May 27, 2018). (Cited on page 25.)

[Kang 1990] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak and A Spencer Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990. (Cited on page 30.)

[Katsaros 2016] Gregory Katsaros. *Open Cloud Computing Interface - Service Level Agreements*. Specification Document GFD.228, Open Grid Forum, October 2016. (Cited on pages 55, 63, 76, 84 and 146.)

[Khajeh-Hosseini 2012] Ali Khajeh-Hosseini, David Greenwood, James W Smith and Ian Sommerville. *The Cloud Adoption Toolkit: Supporting Cloud Adoption Decisions in the Enterprise*. Software: Practice and Experience, vol. 42, no. 4, pages 447–465, 2012. (Cited on page 33.)

[Kirkham 2014] Tom Kirkham, Brian Matthews, Vasily Bunakov and Keith Jeffery. *CAMEL and the Modelling of Cloud Lifecycles*. In 2014 Conference, eChallenges e-2014, pages 1–6. IEEE, 2014. (Cited on page 32.)

[Klein 2003] Dan Klein and Christopher D Manning. *Accurate Unlexicalized Parsing*. In Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1, pages 423–430. Association for Computational Linguistics, 2003. (Cited on page 112.)

[Kleppe 2008] Anneke Kleppe. Software Language Engineering: Creating Domain-Specific Languages using Metamodels. Pearson Education, 2008. (Cited on page 29.)

[Ko 2013] SSGL Ryan Ko, Stephen Lee and Veerappa Rajan. *Cloud Computing Vulnerability Incidents: A Statistical Overview*. Cloud Security Alliance, 2013. (Cited on page 5.)

[Kopaneli 2015] Aliki Kopaneli, George Kousiouris, Gorka Echevarria Velez, Athanasia Evangelinou and Theodora Varvarigou. *A Model Driven Approach for Supporting the Cloud Target Selection Process*. Procedia Computer Science, vol. 68, pages 89–102, 2015. (Cited on page 35.)

[Kopp 2013] Oliver Kopp, Tobias Binz, Uwe Breitenbücher and Frank Leymann. *Winery-A Modeling Tool for TOSCA-based Cloud Applications*. In International Conference on Service-Oriented Computing, pages 700–704. Springer, 2013. (Cited on page 38.)

[Korte 2018] Fabian Korte, Stéphanie Challita, Faiez Zalila, Philippe Merle and Jens Grabowski. *Model-Driven Configuration Management of Cloud Applications with OCCI*. In 8th International Conference on Cloud Computing and Services Science (CLOSER), pages 100–111, 2018. (Cited on pages 16, 87 and 146.)

[Lamport 2002] Leslie Lamport. Specifying Systems: the TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., 2002. (Cited on page 158.)

[Leymann 2011] Frank Leymann, Christoph Fehling, Ralph Mietzner, Alexander Nowak and Schahram Dustdar. *Moving Applications to the Cloud: an Approach Based on Application Model Enrichment*. International Journal of Cooperative Information Systems, vol. 20, no. 03, pages 307–356, 2011. (Cited on page 37.)

[lib] *Apache Libcloud Website*. `http://libcloud.apache.org/` (accessed on May 27, 2018). (Cited on page 26.)

[Loutas 2011] Nikolaos Loutas, Eleni Kamateri and Konstantinos Tarabanis. *A Semantic Interoperability Framework for Cloud Platform as a Service*. In 3rd International Conference on Cloud Computing Technology and Science (CloudCom), pages 280–287. IEEE, 2011. (Cited on page 124.)

[Loveland 2016] Donald W Loveland. Automated Theorem Proving: a logical basis. Elsevier, 2016. (Cited on page 158.)

[man] *Manjrasoft Website*. `http://www.manjrasoft.com/` (accessed on May 27, 2018). (Cited on page 25.)

[Medhioub 2013] Houssem Medhioub, Bilel Msekni and Djamal Zeghlache. *OCNI – Open Cloud Networking Interface*. In 22nd International Conference on Computer Communications and Networks (ICCCN), pages 1–8. IEEE, 2013. (Cited on page 63.)

[Mell 2011] Peter Mell and Tim Grance. *The NIST Definition of Cloud Computing*. 2011. (Cited on page 4.)

[Menzel 2012] Michael Menzel and Rajiv Ranjan. *CloudGenius: Decision Support for Web Server Cloud Migration*. In 21st International Conference on World Wide Web, pages 979–988. ACM, 2012. (Cited on page 33.)

[Merkel 2014] Dirk Merkel. *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. Linux Journal, vol. 2014, no. 239, page 2, 2014. (Cited on page 28.)

[Merle 2015a] Philippe Merle, Olivier Barais, Jean Parpaillon, Noël Plouzeau and Samir Tata. *A Precise Metamodel for Open Cloud Computing Interface*. In 8th International Conference on Cloud Computing (CLOUD), pages 852–859. IEEE, 2015. (Cited on pages 63, 73 and 134.)

[Merle 2015b] Philippe Merle, Jean Parpaillon and Olivier Barais. *OCCI Specific Language - Structural Part*. OCCIware Deliverable 2.3.1, May 2015. (Cited on page 71.)

[Merle 2017] Philippe Merle, Christophe Gourdin and Nathalie Mitton. *Mobile Cloud Robotics as a Service with OCCIware*. In 2nd IEEE International Congress on Internet of Things (ICIOT), pages 710–713. IEEE, 2017. (Cited on pages 76, 86 and 146.)

[Metsch 2016] Thijs Metsch and Mohamed Mohamed. *Open Cloud Computing Interface - Platform*. Specification Document GFD.227, Open Grid Forum, February 2016. (Cited on pages 55, 63, 66, 76, 82, 128 and 146.)

[Mietzner 2009] Ralph Mietzner, Tobias Unger and Frank Leymann. *Cafe: A Generic Configurable Customizable Composite Cloud Application Framework*. On the Move to Meaningful Internet Systems: OTM 2009, pages 357–364, 2009. (Cited on page 37.)

[Moats 1998] R. Moats. *URN Syntax*. Technical report, 1998. (Cited on page 68.)

[MOF 2006] OMG MOF. *2.0 Core Specification*. OMG Document, January, 2006. (Cited on page 30.)

[Mohamed 2013] Mohamed Mohamed, Djamel Belaïd and Samir Tata. *Monitoring and Reconfiguration for OCCI Resources.* In 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), volume 1, pages 539–546. IEEE, 2013. (Cited on page 63.)

[Mohamed 2014a] Mohamed Mohamed. *Generic Monitoring and Reconfiguration for Service-based Applications in the Cloud.* PhD thesis, INT, Evry, France, 2014. (Cited on page 63.)

[Mohamed 2014b] Mohamed Mohamed, Djamel Belaïd and Samir Tata. *Autonomic Computing for OCCI Resources.* Technical report, Telecom Sud Paris, January 2014. (Cited on page 63.)

[Mohamed 2015] Mohamed Mohamed, Mourad Amziani, Djamel Belaid, Samir Tata and Tarek Melliti. *An Autonomic Approach to Manage Elasticity of Business Processes in the Cloud.* Future Generation Computer Systems, vol. 50, pages 49–61, 2015. (Cited on page 63.)

[Mohanarajah 2015] Gajamohan Mohanarajah, Dominique Hunziker, Raffaello D'Andrea and Markus Waibel. *Rapyuta: A Cloud Robotics Platform.* IEEE Transactions on Automation Science and Engineering, vol. 12, no. 2, pages 481–493, 2015. (Cited on page 36.)

[Moody 2009] Daniel Moody. *The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering.* IEEE Transactions on Software Engineering, vol. 35, no. 6, pages 756–779, 2009. (Cited on page 114.)

[mos] *mOSAIC Project Website.* `http://www.mosaic-project.eu/` (accessed on June 14, 2018). (Cited on page 25.)

[Newcombe 2015] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker and Michael Deardeuff. *How Amazon Web Services Uses Formal Methods.* Communications of the ACM, vol. 58, no. 4, pages 66–73, 2015. (Cited on page 124.)

[Nguyen 2012] Dinh Khoa Nguyen, Francesco Lelli, Mike P Papazoglou and Willem-Jan Van Den Heuvel. *Blueprinting Approach in Support of Cloud Computing.* Future Internet, vol. 4, no. 1, pages 322–346, 2012. (Cited on page 31.)

[nov] *Nova Documentation.* `https://docs.openstack.org/nova/latest/` (accessed on June 14, 2018). (Cited on page 37.)

[Nyrén 2016a] Ralf Nyrén, Andy Edmonds, Thijs Metsch and Boris Parák. *Open Cloud Computing Interface - HTTP Protocol*. Specification Document GFD.223, Open Grid Forum, February 2016. (Cited on pages 55, 68, 75, 81, 128, 135 and 141.)

[Nyrén 2016b] Ralf Nyrén, Andy Edmonds, Alexander Papaspyrou, Thijs Metsch and Boris Parák. *Open Cloud Computing Interface - Core*. Specification Document GFD.221, Open Grid Forum, February 2016. (Cited on pages xv, 50, 54, 63, 75, 128 and 129.)

[Nyrén 2016c] Ralf Nyrén, Andy Edmonds, Alexander Papaspyrou, Thijs Metsch and Boris Parák. *Open Cloud Computing Interface - Infrastructure*. Specification Document GFD.224, Open Grid Forum, February 2016. (Cited on pages 55, 63, 66, 75, 77, 128, 144 and 148.)

[Nyrén 2016d] Ralf Nyrén, Florian Feldhaus, Boris Parák and Zdenek Sustr. *Open Cloud Computing Interface - JSON Rendering*. Specification Document GFD-R-P.226, Open Grid Forum, 2016. (Cited on pages 55 and 75.)

[occa] *OCCI-WG: OCCI Working Group Website*. http://occi-wg.org/ (accessed on May 27, 2018). (Cited on pages 10 and 23.)

[occb] *OCCI4Java GitHub Repository*. https://github.com/occi4java/occi4java (accessed on August 13, 2018). (Cited on page 50.)

[occc] *OCCIware Project Website*. http://www.occiware.org/ (accessed on June 13, 2018). (Cited on pages 6 and 55.)

[OMG 2014] OMG. *Object Constraint Language, Version 2.4*. OMG Specification OMG Document Number: formal/2014-02-03, Object Management Group, February 2014. (Cited on page 65.)

[opea] *OpenStack Website*. https://www.openstack.org/ (accessed on June 6, 2018). (Cited on page 4.)

[opeb] *OpenTOSCA Ecosystem Website*. http://www.iaas.uni-stuttgart.de/OpenTOSCA/ (accessed on June 11, 2018). (Cited on page 38.)

[ovf] *DMTF Website*. https://www.dmtf.org/standards/ovf (accessed on June 6, 2018). (Cited on page 23.)

[paa] *PaaSage Project Website*. https://paasage.ercim.eu/ (accessed on June 13, 2018). (Cited on page 35.)

[Pandita 2012] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney and Amit Paradkar. *Inferring Method Specifications from Natural Language API Descriptions*. In the 34th International Conference on Software Engineering (ICSE), pages 815–825. IEEE, 2012. (Cited on page 101.)

[Paraiso 2012] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy and Lionel Seinturier. *A Federated Multi-Cloud PaaS Infrastructure*. In 5th IEEE International Conference on Cloud Computing (CLOUD), pages 392–399. IEEE, 2012. (Cited on page 8.)

[Paraiso 2014] Fawaz Paraiso, Philippe Merle and Lionel Seinturier. *soCloud: A Service-Oriented Component-based PaaS for Managing Portability, Provisioning, Elasticity, and High Availability across Multiple Clouds*. Computing, pages 1–27, 2014. (Cited on page 39.)

[Paraiso 2016] Fawaz Paraiso, Stéphanie Challita, Yahya Al-Dhuraibi and Philippe Merle. *Model-driven Management of Docker Containers*. In 9th IEEE International Conference on Cloud Computing (CLOUD), pages 718–725. IEEE, 2016. (Cited on pages 17, 76, 86, 115 and 145.)

[Parpaillon 2015] Jean Parpaillon, Philippe Merle, Olivier Barais, Marc Dutoo and Fawaz Paraiso. *OCCIware-A Formal and Tooled Framework for Managing Everything as a Service*. In Projects Showcase@ STAF'15, volume 1400, pages 18–25, 2015. (Cited on page 55.)

[Pawluk 2012] Przemyslaw Pawluk, Bradley Simmons, Michael Smit, Marin Litoiu and Serge Mankovski. *Introducing STRATOS: A Cloud Broker Service*. In 5th IEEE International Conference on Cloud Computing (CLOUD), pages 891–898. IEEE, 2012. (Cited on page 25.)

[Petcu 2013] Dana Petcu. *Multi-Cloud: Expectations and Current Approaches*. In International Workshop on Multi-cloud Applications and Federated Clouds, pages 1–6. ACM, 2013. (Cited on page 5.)

[Petrillo 2016] Fabio Petrillo, Philippe Merle, Naouel Moha and Yann-Gaël Guéhéneuc. *Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study*. In the International Conference on Service-Oriented Computing (ICSOC), pages 157–170. Springer, 2016. (Cited on page 100.)

[Priss 2006] Uta Priss. *Formal Concept Analysis in Information Science*. Arist, vol. 40, no. 1, pages 521–543, 2006. (Cited on page 113.)

[pup] *Puppet Website*. https://puppet.com/ (accessed on June 17, 2018). (Cited on pages 28 and 36.)

[pyo] *pyOCNI GitHub Repository*. https://github.com/tmetsch/pyssf (accessed on August 13, 2018). (Cited on page 50.)

[pys] *pySSF GitHub Repository*. https://github.com/tmetsch/pyssf (accessed on August 13, 2018). (Cited on page 50.)

[Quinton 2013] Clément Quinton, Nicolas Haderer, Romain Rouvoy and Laurence Duchien. *Towards Multi-Cloud Configurations Using Feature Models and Ontologies*. In International Workshop on Multi-cloud Applications and Federated Clouds, pages 21–26. ACM, 2013. (Cited on pages 32 and 39.)

[Rachkidi 2017] Elie Rachkidi, Djamel Belaïd, Nazim Agoulmine and Nada Chendeb. *Cloud of Things Modeling for Efficient and Coordinated Resources Provisioning*. In OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", pages 175–193. Springer, 2017. (Cited on page 146.)

[Rat] *Rational Rose Modeler Website*. https://www-01.ibm.com/software/rational/uml/products/ (accessed on May 27, 2018). (Cited on page 30.)

[rig] *RightScale Website*. https://www.rightscale.com/ (accessed on May 27, 2018). (Cited on page 25.)

[roc] *rOCCI Website*. http://gwdg.github.io/rOCCI (accessed on August 13, 2018). (Cited on page 50.)

[Rugaber 2004] Spencer Rugaber and Kurt Stirewalt. *Model-Driven Reverse Engineering*. IEEE software, vol. 21, no. 4, pages 45–53, 2004. (Cited on page 96.)

[Sadovykh 2011] Andrey Sadovykh, Christian Hein, Brice Morin, Parastoo Mohagheghi and Arne J Berre. *REMICS: REuse and Migration of legacy applications to Interoperable Cloud Services*. In 4th European conference on Towards a service-based internet, pages 315–316. Springer-Verlag, 2011. (Cited on page 35.)

[sal] *Salesforce Website*. http://www.salesforce.com/eu/ (accessed on May 27, 2018). (Cited on page 24.)

[Sandru 2012] Calin Sandru, Dana Petcu and Victor Ion Munteanu. *Building an Open-Source Platform-as-a-Service with Intelligent Management of Multiple*

Cloud Resources. In IEEE/ACM 5th International Conference on Utility and Cloud Computing, pages 333–338. IEEE Computer Society, 2012. (Cited on page 25.)

[scaa] *Scalair Website*. https://www.scalair.fr/ (accessed on July 22, 2018). (Cited on pages 82 and 156.)

[scab] *Scalr Website*. https://www.scalr.com/ (accessed on June 3, 2018). (Cited on page 25.)

[Schmidt 2006] Douglas C Schmidt. *Model-Driven Engineering*. COMPUTER-IEEE COMPUTER SOCIETY, vol. 39, no. 2, page 25, 2006. (Cited on page 115.)

[Silva 2014] Gabriel Costa Silva, Louis M Rose and Radu Calinescu. *Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities*. CloudMDE 2014, pages 36–45, 2014. (Cited on page 33.)

[sim] *SimpleCloud Website*. https://www.ibm.com/developerworks/opensource/library/os-simplecloud/os-simplecloud-pdf.pdf (accessed on June 3, 2018). (Cited on page 26.)

[Sinha 2010] Avik Sinha, Stanley M Sutton Jr and Amit Paradkar. *Text2Test: Automated Inspection of Natural Language Use Cases*. In the 3rd International Conference on Software Testing, Verification and Validation (ICST), pages 155–164. IEEE, 2010. (Cited on page 101.)

[sir] *Sirius Website*. http://www.eclipse.org/sirius/ (accessed on May 27, 2018). (Cited on pages 31 and 71.)

[Sousa 2012] Gustavo Sousa, Fábio M Costa, Peter J Clarke and Andrew A Allen. *Model-Driven Development of DSML Execution Engines*. In 7th Workshop on Models@ run. time, pages 10–15. ACM, 2012. (Cited on page 31.)

[Sousa 2017] Gustavo Sousa, Walter Rudametkin and Laurence Duchien. *Extending Dynamic Software Product Lines with Temporal Constraints*. In Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pages 129–139. IEEE Press, 2017. (Cited on page 39.)

[sta] *Stanford Parser Website*. https://nlp.stanford.edu/software/lex-parser.shtml (accessed on July 24, 2018). (Cited on page 113.)

[Steinberg 2008] Dave Steinberg, Frank Budinsky, Ed Merks and Marcelo Paternostro. EMF: Eclipse Modeling Framework. Pearson Education, 2008. (Cited on page 108.)

[Vecchiola 2009] Christian Vecchiola, Xingchen Chu and Rajkumar Buyya. *Aneka: A Software Platform for .NET-based Cloud Computing*. High Speed and Large Scale Scientific Computing, vol. 18, pages 267–295, 2009. (Cited on page 25.)

[vmw] *VMware Website*. `https://www.vmware.com/` (accessed on June 6, 2018). (Cited on page 24.)

[xsc] *XScalibur Website*. `http://www.xscalibur.com/` (accessed on October 15, 2018). (Cited on page 156.)

[xte 2016] *Xtext Website*, 2016. `http://www.eclipse.org/Xtext/` (accessed on May 27, 2018). (Cited on pages 30 and 71.)

[Yangui 2013] Sami Yangui and Samir Tata. *CloudServ: PaaS resources provisioning for service-based applications*. In 27th IEEE International Conference on Advanced Information Networking and Applications (AINA 2013), pages 522–529. IEEE, 2013. (Cited on page 63.)

[Yangui 2014] Sami Yangui, Iain-James Marshall, Jean-Pierre Laisne and Samir Tata. *CompatibleOne: The Open Source Cloud broker*. Journal of Grid Computing, vol. 12, no. 1, pages 93–109, 2014. (Cited on pages 24 and 25.)

[Yangui 2016] Sami Yangui and Samir Tata. *An OCCI Compliant Model for PaaS Resources Description and Provisioning*. The Computer Journal, vol. 59, no. 3, pages 308–324, 2016. (Cited on page 63.)

[Yongsiriwit 2016] Karn Yongsiriwit, Mohamed Sellami and Walid Gaaloul. *A Semantic Framework Supporting Cloud Resource Descriptions Interoperability*. In 9th International Conference on Cloud Computing (CLOUD), pages 585–592. IEEE, 2016. (Cited on page 124.)

[Zalila 2017a] Faiez Zalila, Stéphanie Challita and Philippe Merle. *A Model-Driven Tool Chain for OCCI*. In OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", pages 389–409. Springer, 2017. (Cited on pages 16 and 49.)

[Zalila 2017b] Faiez Zalila, Philippe Merle, Jean Parpaillon, Slim Kallel, Mehdi Ahmed-Nacer, Walid Gaaloul and Christophe Gourdin. *OCCI Extension Models*. OCCIware Deliverable 2.4.1, September 2017. (Cited on page 146.)

[Zalila 2018] Faiez Zalila, Stéphanie Challita and Philippe Merle. *Model-Driven Cloud Resource Management with OCCIware*. Future Generation Computer Systems (FGCS), 2018. under review. (Cited on pages 17 and 49.)

[Zhai 2016] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao and Feng Qin. *Automatic Model generation from Documentation for Java API Functions*. In the 38th International Conference on Software Engineering, pages 380–391. ACM, 2016. (Cited on page 101.)

[Zhong 2009] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei and Hong Mei. *MAPO: Mining and Recommending API Usage Patterns*. ECOOP–Object-Oriented Programming, pages 318–343, 2009. (Cited on page 101.)