A Formal and Tooled Framework for Managing Everything as a Service

www.occiware.org

Deliverable D2.2.1

OCCI Structural Model

OCCIware project is supported by Fonds national pour la Société Numérique (FSN) and the following competitive clusters Systematic, Minalogic, PICOM, Images & Réseaux and Solutions Communicantes Sécurisées.
Abstract
Cloud computing has been adopted as the dominant delivery model for computing resources, aka Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). In this context, there is a plethora of cloud resource management interfaces for provisioning, supervising, and managing cloud resources. Thereby standards are required to cope with heterogeneity, interoperability, integration, and portability issues in cloud computing. To this end, Open Cloud Computing Interface (OCCI) proposes one of the first widely accepted, community-based, open standards for managing any kinds of cloud resources. But as it is specified in natural language, OCCI is imprecise, ambiguous, incomplete, and needs a precise definition of its core concepts. Indeed, the OCCI Core Model has conceptual drawbacks: an imprecise semantics of its type classification system, a nonextensible data type system for OCCI attributes, a vague and limited extension concept and the absence of a configuration concept. To tackle these issues, this deliverable proposes a precise metamodel for OCCI. This metamodel defines rigourously the static semantics of the OCCI core concepts, of a precise type classification system, of an extensible data type system, and of both extension and configuration concepts. This metamodel is based on the Eclipse Modeling Framework (EMF), its structure is encoded with Ecore and its static semantics is rigourously defined with Object Constraint Language (OCL). As a consequence, this metamodel provides a concrete language to precisely define and exchange OCCI models. The validation of our metamodel is done on the first world-wide dataset of OCCI extensions already published in the literature, and addressing inter-cloud networking, infrastructure, platform, application, service management, cloud monitoring, and autonomic computing domains, respectively. This validation highlights simplicity, consistency, correctness, completeness, and usefulness of the proposed metamodel.
### Document versioning

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Initial version</td>
<td>Philippe Merle</td>
</tr>
<tr>
<td>0.2</td>
<td>Updated document template</td>
<td>Philippe Merle</td>
</tr>
<tr>
<td>0.3</td>
<td>Minor cosmetic textual improvements</td>
<td>Philippe Merle</td>
</tr>
<tr>
<td>0.4</td>
<td>Minor cosmetic textual improvements</td>
<td>Noël Plouzeau</td>
</tr>
<tr>
<td>0.5</td>
<td>Bibliography updates</td>
<td>Olivier Barais</td>
</tr>
<tr>
<td>1.0</td>
<td>Final version</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>1.1</td>
<td>Updated final version</td>
<td>Philippe Merle</td>
</tr>
</tbody>
</table>

### Document reviews

<table>
<thead>
<tr>
<th>Review</th>
<th>Date</th>
<th>Ver.</th>
<th>Reviewer(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>June 22nd, 2015</td>
<td>0.1</td>
<td>Philippe Merle</td>
<td></td>
</tr>
<tr>
<td>Brouillon</td>
<td>July 21st</td>
<td>0.4</td>
<td>Noël Plouzeau</td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td>July 22nd</td>
<td>0.4</td>
<td>Noël Plouzeau</td>
<td></td>
</tr>
<tr>
<td>BAT</td>
<td>July 24th</td>
<td>1.0</td>
<td>Jean Parpaillon</td>
<td></td>
</tr>
<tr>
<td>Deliverable</td>
<td>Status</td>
<td>Expected</td>
<td>Dissemination</td>
<td>Version</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>----------</td>
<td>---------------</td>
<td>---------</td>
</tr>
<tr>
<td>D2.2.1</td>
<td>Final</td>
<td>May 31, 2015</td>
<td>Public</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>January 31, 2016</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contents

1 Introduction ................................................................. 7

2 The OCCI Core Model ......................................................... 9
   2.1 Background ................................................................... 9
   2.2 Drawbacks of the OCCI Core Model .................................. 10
       2.2.1 Informal model ................................................. 11
       2.2.2 Imprecise type classification system .................... 11
       2.2.3 Nonextensible data type system ......................... 11
       2.2.4 Vague and incomplete extension concept ............. 11
       2.2.5 Configuration concept undefined ...................... 11

3 OCCIware Metamodel ......................................................... 13
   3.1 Precise metamodel ..................................................... 13
   3.2 Precise type classification system ............................... 16
   3.3 Extensible data type system ....................................... 17
   3.4 Extension concept ..................................................... 17
   3.5 Configuration concept ................................................ 19
   3.6 Summary .................................................................. 20

4 Validation ........................................................................ 21
   4.1 Methodology .............................................................. 21
   4.2 Analysis of the results .................................................. 21
       4.2.1 Simplicity ......................................................... 22
       4.2.2 Consistency ...................................................... 22
       4.2.3 Correctness ....................................................... 23
       4.2.4 Completeness ................................................... 23
       4.2.5 Usefulness ......................................................... 23
   4.3 Threats to Validity ....................................................... 23

5 Conclusion and Perspectives ............................................... 25

A OCCIware Ecore Package .................................................. 31
List of Figures

2.1 UML class diagram of the OCCI Core Model [from [1]]. .......................... 9

3.1 Ecore diagram of OCCIWARE METAMODEL ................................. 14
List of Tables

4.1 Summary of the OCCIWARE MODEL DATASET ...................................... 22
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D2.2.1</th>
<th>Status</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>May 31, 2015</td>
<td>Dissemination</td>
<td>Public</td>
</tr>
<tr>
<td>Delivered</td>
<td>January 31, 2016</td>
<td>Version</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Cloud computing has been adopted as a dominant delivery model for computing resources [2]. This model defines three well discussed layers of services known as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [3]. Other XaaS terms are used nowadays to name different resources provided as services in the clouds [4]. Cloud management poses different challenges [5]. Provisioning, supervising, and managing these outsourced, on-demand, pay as you go, elastic resources require cloud resource management interfaces (CRM-API). However, there is a plethora of CRM-API, proposed by Amazon, Eucalyptus, Microsoft, Google, OpenNebula, CloudStack, OpenStack, CloudBees, OpenShift, Cloud Foundry, to name a few. Even if there are several client-side API for interacting with multiple popular cloud service providers, these API are all linked to a specific programming language: Apache Libcloud for Python, Apache jclouds for Java, Gophercloud for Go, etc.

Thereby cloud computing standards are required to cope with four main issues: heterogeneity of cloud offers, interoperability between CRM-API, integration of CRM-API for building multi-cloud systems, and portability of cloud management applications. To this end, Open Cloud Computing Interface (OCCI) is an Open Grid Forum (OGF) community-based effort to create one of the first open extensible standards for managing any kind of cloud computing resources [6].

OCCI is supported by a large community that includes providers of open source cloud software stacks such as Eucalyptus [7], OpenNebula [8], CloudStack, OpenStack, and Compati-bOne [9], and users such as the European Grid Infrastructure (EGI), to cite a few. Several OCCI runtime frameworks exist, e.g., erocci, rOCCI, pySSF, pyOCNI, and OCCI4Java, and they rely on Erlang, Ruby, Python, Java programming languages, respectively. OCCI has already been used successfully for managing inter-cloud networking [10], building reliable storage virtualizations [11], IaaS resources description [12], PaaS resources description [13] [14][15][16], SaaS resources description [14][16], SLA negotiation and enforcement in data management [17], service management [18], resource management in federated clouds [19], cloud monitoring [20] and reconfiguration [21], and autonomic computing [22][23][24]. A common denominator of these recent usages is the use of the REST architecture style for managing cloud computing resources [25].

The kernel of OCCI is a generic resource-oriented model called the OCCI Core Model and defined in [1]. The OCCI Core Model can be interacted with using renderings (including associated behaviours) and expanded through extensions. For instance, the OCCI HTTP Rendering [26] defines how OCCI resources are accessible as REST resources over the HTTP network protocol.

---

1 [http://libcloud.apache.org/]
2 [http://jclouds.apache.org/]
3 [http://gophercloud.io/]
4 A complete list is available at [http://occi-ug.org/community/implementations/][1]

---

[1] OCCIware
Other OCCI renderings are being specified by the OCCI working group (OCCI-WG) such as a JSON rendering and a XMPP network transport. The OCCI Infrastructure Extension defines OCCI-compliant compute, network and storage IaaS resources. Other drafts address PaaS resources and cloud monitoring.

Nevertheless, OCCI lacks a precise definition of its core concepts. Indeed, OCCI specifications are informal documents written in natural language and illustrated by UML diagrams. This informal definition of the OCCI Core Model can be interpreted in various different ways, which can lead to interoperability issues between OCCI implementations. Moreover, the OCCI Core Model has conceptual drawbacks and limitations: an imprecise semantics of its built-in type classification system, a nonextensible data type system for OCCI attributes, a vague and limited extension concept, and the absence of a configuration concept.

To tackle these issues, this deliverable contributes a precise metamodel for OCCI. This metamodel defines rigorously the static semantics of the OCCI core concepts. Our metamodel proposes a precise type classification system, an extensible data type system, and both extension and configuration concepts. This metamodel is based on the Eclipse Modeling Framework (EMF), structured as an Ecore package, and its static semantics is rigorously defined using the Object Constraint Language (OCL). As a consequence, this metamodel provides a modeling language to precisely define and exchange OCCI models. The validation of our metamodel has been done on the first world-wide dataset of OCCI extensions already published in the literature. This dataset is composed of seven OCCI extensions addressing: inter-cloud networking, IaaS, PaaS, SaaS, service management, cloud monitoring, and autonomic computing, domains, respectively. This validation highlights simplicity, consistency, correctness, completeness, and usefulness of the proposed metamodel.

This deliverable is an extended version of the paper published in the proceedings of the 8th IEEE International Conference on Cloud Computing (IEEE CLOUD 2015), June 27 - July 2, 2015, New York, USA.

This deliverable is organized as follows. Chapter gives background on the OCCI Core Model and identifies five conceptual drawbacks of this model. Chapter describes our precise metamodel for OCCI. Chapter validates our metamodel on seven already published OCCI extensions. Chapter concludes on future work and perspectives. Finally, Appendix provides a formal definition of the OCCIware Ecore Package.

---

Chapter 2

The OCCI Core Model

2.1 Background

The OCCI Core Model \[1\] is a simple resource-oriented model composed of eight concepts\[1\] as illustrated in Figure 2.1.

- **Resource** represents any cloud computing resource, *e.g.*, a virtual machine, a network, an application container, an application. **Resource** owns a set of **links**.

- **Link** is a relation between two **Resource** instances, *e.g.*, a computer connected to a network, an application hosted by a container. **Link** references to both **source** and **target** resources.

\[1\]Our work is based on the last revised draft of the OCCI Core Model \[1\], and not on the officially published and publicly available version of this document.
- **Entity** is the abstract base class of all resources and links. Each resource or link has a unique identifier, and is strongly typed by a kind type and zero or more mixins types.

- **Kind** is the notion of class/type within OCCI, *e.g.*, Compute, Network, Container, Application. A Kind instance owns a set of actions, can have one parent kind, and lists its entities, which are instances of this kind. There will be at least three instances of Kind: entity kind, resource kind and link kind instances.

- **Mixin** is used to associate additional features, *e.g.*, location, price, user preference, ranking, to resource/link instances. Each Mixin instance owns a set of actions, can inherit from zero or more depends mixins, defines on which kinds it can be applied (applies reference), and references a set of entities on which the mixin is applied. Tagging of OCCI resource instances is supported through the association of Mixin instances (called Tags). A tag is simply a Mixin instance that defines no additional resource capabilities, i.e. no attributes and no actions. Templates allow to apply predefined values to attributes of OCCI types. They are implemented using Mixin instances and at entity instantiation they associate time certain pre-populated attributes. A template is a Mixin instance with all attributes having a default value.

- **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. Each instance of kind, mixin or action is uniquely identified by both a scheme and a term, has a human readable title, and owns a set of attributes.

- **Attribute** represents the definition of a client 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 scalar data type, can be (or not) mutable (i.e., modifiable by clients), can be (or not) required (i.e., value is provided at creation time), can have a default value and a human readable description.

Even if the OCCI Core Model is simple, it is generic enough to model any cloud computing resource such as inter-cloud networking [10], building reliable storage virtualizations [11], IaaS resources [12], PaaS resources [13 14 15 16], SaaS resources [14 16], service management [18], SLA negotiation and enforcement in data management [17], resource management in federated clouds [19], cloud monitoring [20] and reconfiguration [21], and autonomic computing [22 23 24]. Chapter 4 validates our precise metamodel for OCCI on seven of these already published works.

## 2.2 Drawbacks of the OCCI Core Model

Through our study of OCCI in funded research projects (CompatibleOne[2] EASI-CLOUDS[3] OpenPaaS[4] and OCCIware[5]), we have identified five conceptual drawbacks or limitations on the OCCI Core Model:

2.2.1 Informal model

The structure of the OCCI Core Model is illustrated by the UML class diagram shown in Figure 2.1. This diagram contains at least three semantic errors. In [1], Category and Entity are explicitly described as abstract types, i.e., no instance of these types can exist. Therefore their type name must be shown in italic in the UML class diagram. URI is not a base data type in UML, it must be defined. Both static and dynamic semantics of the OCCI Core Model are described by sentences and tables in natural language. Some of these sentences are imprecise and/or ambiguous and can be interpreted in various different ways, which can lead to interoperability issues between OCCI implementations. For instance, [1] does not explicitly state that two distinct attributes with the same name must not belong to the same category instance. Therefore the OCCI Core Model needs a precise and rigorous definition of its core concepts.

2.2.2 Imprecise type classification system

The OCCI Core Model [1] does not explicitly state that the built-in type classification system of OCCI — parent and depends relations — must form a directed acyclic graph, i.e., a kind must not inherit from itself and a mixin must not depend from itself, both directly or transitively, else the OCCI type classification system would have a unusual semantics! A kind instance must not overload an attribute inherited from its parent directly or transitively. A mixin instance must not overload an attribute inherited from its depends directly or transitively. Entities of a mixin must have a kind compatible with an applies kind of this mixin at least. Therefore the OCCI Core Model needs a precise and rigorous definition of its type classification system.

2.2.3 Nonextensible data type system

The data type system for OCCI attributes (attribute type defined as a string) is not general enough. Currently only string, number, and boolean types are supported. This is insufficient for describing complex OCCI extensions such as OCCI Infrastructure Extension [12], as this extension uses other scalar data types, such as IP address, float and enumeration types. OCCI should provide some mechanisms to extend or restrict data types such as enumeration, string pattern, number range, decimal precision, etc. The OCCI data type system must be compatible with already existing and widely accepted data type systems like W3C XSD [29] or OMG IDL [30]. Therefore the OCCI Core Model needs an open and extensible data type system for OCCI attributes.

2.2.4 Vague and incomplete extension concept

While the concept of extension exists in OCCI specification, it is too vaguely and incompletely defined. Informally, each OCCI extension is a set of Kind andMixin types targeting a concrete cloud computing domain, e.g. IaaS, PaaS, SaaS, pricing, cloud monitoring, etc. An extension can use or extend other extensions, e.g., SaaS running on PaaS deployed on IaaS implying that the SaaS extension uses the PaaS extension, which needs the IaaS extension. To make a precise and correct use of these informal use/extend dependencies in OCCI architectures, the OCCI Core Model needs a precise and rigorous definition of the notion of OCCI extensions.

2.2.5 Configuration concept undefined

The concept of OCCI configurations is not explicitly defined in OCCI specifications. A configuration is an abstraction of an OCCI-based running system, and is composed of resource and link
instances. A configuration must explicitly state which extensions it uses. Modeling a configuration offline could allow designers to think about and analyse their cloud systems without having to deploy them concretely in the clouds. Therefore the OCCI Core Model needs a precise and rigorous definition of the notion of OCCI configurations.

The next chapter explains how the five limitations described above are addressed in our precise metamodel for OCCI.
Chapter 3

OCChware Metamodel

Our precise metamodel for OCCI, named OCChware Metamodel, is based on the Eclipse Modeling Framework (EMF) [31]. We have chosen EMF as this is the leading metamodeling framework, offering a plethora of metamodeling technologies such as Ecore to encode the structure of metamodels, OCL [27] to encode the semantics of metamodels, XMI to exchange (meta)models, Xtext to build metamodel-based textual editors with advanced features such as syntax highlighting, completion, outline, hyperlinks, Sirius to build graphical modelers, Acceleo to implement model-to-text transformations, to cite a few.

The static semantics of our OCChware Metamodel is rigourously defined using OCL, which is widely accepted for modeling semantics in UML and EMF, and is complete in order to help reproducibility of our contribution (see Section Availability). In the following OCL snippets are contained into simple frames and are as concise as possible to help comprehension and readability. We only use five OCL keywords: context gives the context in which an OCL snippet applies, def introduces a shortcut for an OCL expression, inv defines an invariant that must be always true, let introduces a local shortcut, and self refers to the current object on which an OCL expression is evaluated. The name of invoked OCL operations is self-explanatory.

3.1 Precise metamodel

This section addresses Drawback 2.2.1 aka OCCI’s model being informal, by introducing our Ecore package and defining the static semantics of OCCI basic core concepts with five OCL invariants and four OCL definitions.

The structure of OCChware Metamodel is encoded as an Ecore package as it is illustrated in Figure 3.1. A complete formal definition of this Ecore package is given in Appendix A. Classes with a white background and their references in black encode the OCCI Core Model as defined in Chapter 2 and illustrated by Figure 2.1. The static semantics of the OCCI built-in type classification system is rigourously defined in Section 3.2. The EDataType class, the three String, Number, and Boolean data types, and the type reference in the orange color model our extensible data type system for OCCI attributes and are discussed in Section 3.3. The Extension class and its references drawn in red model the concept of OCCI extensions. Its static semantics is rigourously defined in Section 3.4. Both Configuration and AttributeState classes and their

---

2. [Object Constraint Language](http://www.eclipse.org/emf/)
3. [XML Metadata Interchange](http://www.eclipse.org/emf/)
references drawn in green model the concept of OCCI configurations. Their static semantics is rigourously defined in Section 3.5.

In the following we provide basic definitions related to the OCCI Core Model: unicity of Category identity, constraints related to the Category.scheme attribute, unicity of the name of attributes of Category, Mixin tags and templates, and unicity of Entity.id.

**Definition 1.** The identity of a Category instance, i.e. a kind, a mixin, or an action, is equal to the concatenation of its scheme and term attributes.

```
context Category
def : identity : String = scheme + term
```

**Definition 2.** Each Category instance must have a unique identity.

```
context Category
inv IdentityUnique:
    Category.allInstances() -> isUnique(identity)
```

**Definition 3.** The scheme of each Category instance must end with a sharp.

```
context Category
inv SchemeEndsWithSharp:
scheme.substring(scheme.size(), scheme.size()) = '#'
```
Definition 4. The category of an Action instance is the Category instance owning this action.

\[
\text{context Action}
\text{def : category : Category = oclContainer().oclAsType(Category)}
\]

Definition 5. The scheme of an Action instance must be the concatenation of both scheme and term of its category plus the suffix /action#.

\[
\text{context Action}
\text{inv CorrectScheme :}
\text{scheme = category.scheme.substring(1, category.scheme.size() - 1) + ' '/
\text{ + category.term + '/action#')}
\]

Definition 6. Each Attribute instance of a Category instance must have a distinct name.

\[
\text{context Category}
\text{inv AttributesNameUnique :}
\text{attributes ->isUnique(name)}
\]

Definition 7. A tag is a Mixin instance with no attributes and no actions. Its depends must be tags also.

\[
\text{context Mixin}
\text{def : isTag : Boolean = attributes->isEmpty() and actions->isEmpty()}
\text{and depends->forAll(isTag)}
\]

Definition 8. A template is a Mixin instance with all attributes must have a default value, and all its depends mixins must be tags or templates.

\[
\text{context Mixin}
\text{def : isTemplate : Boolean = attributes->forAll(default <> null) and}
\text{depends->forAll(isTag or isTemplate)}
\]

Definition 9. Each Entity instance must have a unique id.

\[
\text{context Entity}
\text{inv IdUnique :}
\text{Entity.allInstances() ->isUnique(id)}
\]

\footnote{oclContainer() is a standard OCL operation returning the owner of the invoked object, here the object owning the action, and oclAsType() is a standard OCL operation to cast an object to a given class.}
3.2 Precise type classification system

This section addresses Drawback 2.2.2, aka imprecise type classification system, by defining the static semantics of the OCCI built-in type classification system with eight OCL invariants.

**Definition 10.** *The inheritance relation* parent *between Kind instances must form a direct acyclic graph. A kind instance must not inherit from itself directly or transitively.*

```
context Kind
inv NoCyclicInheritance:
parent->closure(parent)->excludes(self)
```

**Definition 11.** *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
inv EntityKindIsRootParent:
self->closure(parent)->exists(k | k.identity = 'http://schemas.ogf.org/occi/core#entity' and k.parent = null)
```

**Definition 12.** *A Kind instance must not overload an inherited attribute.*

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

**Definition 13.** *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
inv NoCyclicInheritance:
depends->closure(depends)->excludes(self)
```

**Definition 14.** *A Mixin instance must not overload an inherited attribute.*

```
context Mixin
inv AttributeNotAlreadyDefinedInDepends:
attributes.name->excludesAll(depends->closure(depends).attributes.name)
```

**Definition 15.** *The kind of an Entity instance must be compatible with one applies kind instance of each mixins of this entity.*

```
context Entity
inv KindCompatibleWithOneAppliesOfEachMixin:
mixins->forall(m | m.applies->notEmpty()) implies m.applies->exists(k | kind->closure(parent)->includes(k))
```

8In OCL, a->closure(b) returns the set { a, a.b, a.b.b, ... }.
Definition 16. The kind of a Resource instance must inherit from the resource kind instance directly or transitively.

```context Resource
inv ResourceKindIsInParent:
kind->closure(parent)->exists(k | k.identity = 'http://schemas.ogf.org/occi/core#resource')```

Definition 17. The kind of a Link instance must inherit from the link kind instance directly or transitively.

```context Link
inv LinkKindIsInParent:
kind->closure(parent)->exists(k | k.identity = 'http://schemas.ogf.org/occi/core#link')```

3.3 Extensible data type system

This section addresses Drawback 2.2.3, i.e., OCCI’s nonextensible data type system, by reusing the extensible data type system provided by EMF.

EMF provides an open and extensible data type system. This system is composed of two classes: EDataType to model scalar data types and EEnum, extending EDataType, to model enumerations. An EMF data type can be restricted with metadata annotations to define regular expressions, the maximal number of digits of a number, the minimal and maximal length, minimal and maximal values, or custom constraints implemented in Java or OCL. All XSD data types are already modeled as EMF data types. Thus the EMF data type system meets all the requirements defined in Drawback 2.2.3.

In the OCClware Metamodel, the type of Attribute is modeled by the reference type from Attribute to EDataType, and the three OCCI base data types are modeled by String, Number, and Boolean data types, as shown in Figure 3.1. Each OCCI extension can define its own data types.

3.4 Extension concept

This section addresses Drawback 2.2.4, i.e., the vague and incomplete extension concept, by introducing the Extension class and defining the static semantics of the extension concept with six OCL invariants and two OCL definitions.

Definition 18. Extension represents an OCCI extension, e.g., inter-cloud networking extension [10], infrastructure extension [12], platform extension [13, 14, 15, 16], application extension [17, 10], SLA negotiation and enforcement [17], cloud monitoring extension [20], and autonomic computing extension [21, 22, 23, 24]. As encoded in the Ecore package shown in Figure 3.1, Extension has a name, has a scheme, owns zero or more kinds, owns zero or more mixins, owns zero or more types, and can import zero or more extensions.

Definition 19. Each Extension instance must have a unique scheme among all Extension instances.
Definition 20. The scheme of all kinds must be equal to the scheme of the owning Extension instance.

Definition 21. The scheme of all mixins must start with the scheme of the owning Extension instance.

Definition 22. The extension of a Category instance is the Extension instance owning this category.

Definition 23. The following OCL function checks if a category is defined or is imported by this extension.

Definition 24. The parent of all the kinds of an extension must be defined or imported by this extension.

Definition 25. All the depends of all the mixins of an extension must be defined or imported by this extension.
Definition 26. *All the applies of all the mixins of an extension must be defined or imported by this extension.*

```ocaml
context Extension
inv MixinAppliesLocalOrImported:
  mixins.applies -> forall (k | isDefinedOrImported(k))
```

### 3.5 Configuration concept

This section addresses Drawback 2.2.5, i.e. the absence of a configuration concept, by introducing the `Configuration` class and defining the static semantics of the configuration concept with six OCL invariants and one OCL definition.

Definition 27. *Configuration represents a running OCCI system. As encoded in the Ecore package shown in Figure 3.1, 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 28. *The kind of all resources of a configuration must be defined by an extension that is explicitly used by this configuration.*

```ocaml
context Configuration
inv AllResourcesKindInUse:
  use -> includesAll(resources.kind.extension)
```

Definition 29. *All the mixins of all resources of a configuration must be defined by an extension that is explicitly used by this configuration.*

```ocaml
context Configuration
inv AllResourcesMixinsInUse:
  use -> includesAll(resources.mixins.extension)
```

Definition 30. *The kind of all links of all resources of a configuration must be defined by an extension that is explicitly used by this configuration.*

```ocaml
context Configuration
inv AllResourcesLinksKindInUse:
  use -> includesAll(resources.links.kind.extension)
```

Definition 31. *All the mixins of all links of all resources of a configuration must be defined by an extension that is explicitly used by this configuration.*

```ocaml
context Configuration
inv AllResourcesLinksMixinsInUse:
  use -> includesAll(resources.links.mixins.extension)
```
Definition 32. The configuration of a Resource instance is the Configuration instance owning this resource.

context Resource
def : configuration : Configuration = oclContainer().oclAsType(Configuration)

Definition 33. The target resource of all links of all resources of a configuration must be a resource of this configuration.

context Configuration
inv AllResourcesLinksTargetsInConfiguration:
resources.links.target->forall(r | r.configuration = self)

Definition 34. The name of all attributes of any Entity instance must be unique.

context Entity
inv AttributesNameUnique:
attributes->isUnique(name)

3.6 Summary

The OCCIware Metamodel addresses the five drawbacks identified in Section 2.2. This metamodel encodes all the eight core concepts of OCCI, gives a precise semantics of these concepts (cf. Drawback 2.2.1) including the type classification system (cf. Drawback 2.2.2), reuses an extensible data type system for typing OCCI attributes (cf. Drawback 2.2.3), and introduces two new concepts: extension (cf. Drawback 2.2.4), and configuration (cf. Drawback 2.2.5). The static semantics is expressed in OCL using twenty-five invariants and seven definitions.
Chapter 4
Validation

This chapter presents a quantitative and qualitative evaluation of the proposed metamodel. We have set up an experimental protocol to build a dataset of already published OCCI extensions, and to analyze this dataset in respect to five qualitative criteria: Simplicity, Consistency, Correctness, Completeness, Usefulness. Section 4.1 details the methodology we applied to build the dataset. Section 4.2 analyses the dataset quantitatively and qualitatively. Section 4.3 discusses the threats to the validity of our evaluation.

4.1 Methodology

To evaluate our metamodel, we have surveyed the literature to find all the already published OCCI extensions. We have identified seven distinct works related to inter-cloud networking [10], infrastructure [12, 17], platform [13, 14, 15, 16], application [14, 16], service management [18], cloud monitoring [20], and autonomic computing [21, 22, 23, 24] domains. As a working hypothesis, we have assumed that all these extensions are correct as they were already accepted through a peer-to-peer reviewing process.

Secondly, we have encoded these seven correct extensions as instances of the OCCIWARE METAMODEL. When a work does not provide all the information required by the OCCI core model, e.g., a scheme, then we have set up this information correctly, i.e., a well-formed scheme respecting our OCL invariants. When a work is composed of several publications, e.g., autonomic computing, we have considered the last publication as the most relevant and up-to-date one. These seven encodings form what we name the OCCIWARE MODEL DATASET. To our best knowledge, this is the first worldwide dataset of OCCI extensions.

Thirdly, we have passed this dataset through the EMF Validation Framework (EMF-VF), which checks all the structural constraints of our Ecore metamodel, e.g., cardinality of Ecore attributes and references, as well as all our OCL invariants. Then we have analysed our dataset and the validation results produced by EMF-VF.

4.2 Analysis of the results

Table I presents only a summary of our OCCIWARE MODEL DATASET. For each class of our metamodel, Table I provides the number of instances of this class present in the dataset,
the number of OCL invariants defined in our metamodel, and the number of OCL invariants validated as correct by EMF-VF. The last line provides the total of OCCIware objects present in the dataset, of our OCL invariants, and of OCL invariants evaluated to true by EMF-VF.

This dataset covers all the five drawbacks presented in Section 2.2 and corrected by our metamodel discussed in Chapter 3. The first eight classes are related to both informal model (cf. Section 2.2.1) and imprecise type classification system (cf. Section 2.2.2) drawbacks corrected in Section 3.1 and 3.2 respectively. The nonextensible data type system drawback (cf. Section 2.2.3) is addressed by the use of the EDataType class (cf. Section 3.3). Let us note that all the seven extensions need to define new data types. The drawback of an incomplete extension concept (cf. Section 2.2.4) was corrected by the addition of the Extension class (cf. Section 3.4). Finally, the configuration concept undefined drawback (cf. Section 2.2.5) was corrected by the addition of both Configuration and AttributeValue classes (cf. Section 3.5). Thus our dataset validates our metamodel quantitatively, as all the classes and OCL constraints of our metamodel are covered by the dataset.

Table 4.1: Summary of the OCCIware Model Dataset

<table>
<thead>
<tr>
<th>OCCIware Class Name</th>
<th>Dataset Instances</th>
<th>OCCIware Invariants</th>
<th>EMF-VF Validations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>123</td>
<td>3</td>
<td>369</td>
</tr>
<tr>
<td>Attribute</td>
<td>147</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Action</td>
<td>56</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>Kind</td>
<td>43</td>
<td>3</td>
<td>129</td>
</tr>
<tr>
<td>Mixin</td>
<td>24</td>
<td>2</td>
<td>48</td>
</tr>
<tr>
<td>Entity</td>
<td>61</td>
<td>3</td>
<td>183</td>
</tr>
<tr>
<td>Resource</td>
<td>37</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>Link</td>
<td>24</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>EDataType</td>
<td>42</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Extension</td>
<td>7</td>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>Configuration</td>
<td>12</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td>AttributeValue</td>
<td>157</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>549</td>
<td>25</td>
<td>948</td>
</tr>
</tbody>
</table>

Analysing these statistics on our dataset provides an answer to the following five validation questions:

4.2.1 Simplicity


Is our metamodel simple to model cloud systems? The answer to this question is yes, as our metamodel contains only twelve concepts – eight from the OCCI Core Model and four new ones – allowing to model seven cloud computing domains, when other concurrent cloud standards like CIMI [32] or TOSCA [33] define a huge set of concepts, and only address IaaS and cloud applications, respectively.

4.2.2 Consistency

Is our precise semantics of OCCI consistent for modeling any OCCI systems? The answer to this question is yes, as there are no contradictions between our twenty five OCL invariants, else EMF-VF will not evaluate our whole dataset as correct (last column in Table I is equal to the number of instances multiplied by the number of invariants).

4.2.3 Correctness

See a definition of correctness at [http://en.wikipedia.org/wiki/Worse_is_better](http://en.wikipedia.org/wiki/Worse_is_better).

Is our metamodel of OCCI correct for modeling any OCCI system? The answer to this question is yes, as our metamodel allows to correctly model all OCCI extensions in all observable aspects.

4.2.4 Completeness

See a definition of completeness at [http://en.wikipedia.org/wiki/Worse_is_better](http://en.wikipedia.org/wiki/Worse_is_better).

Is our precise semantics of OCCI complete for modeling any OCCI systems? The answer to this question is yes, as our metamodel covers all situations encountered in the seven works proposing an OCCI extension. Moreover, our metamodel fully covers all the OCCI core concepts and addresses the five drawbacks presented into Section 2.2.

4.2.5 Usefulness


Is our metamodel useful for modeling all OCCI systems? The answer to this question is yes, as our new introduced classes – Extension, EDataType, Configuration, and AttributeState – are useful for modeling the seven OCCI extensions and fully address the five drawbacks (cf. Section 2.2). Moreover, our extensible data type system is useful in all covered extensions, i.e., new EDataType instances are created in each extension.

4.3 Threats to Validity

Firstly, we could miss some already published OCCI extensions. These extensions could violate structural and/or OCL constraints of our OCCIWARE METAMODEL. This is a threat to validity of the correctness of our metamodel.

Secondly, we could incorrectly encode some already published extensions. This could introduce a potential threat to validity on the correctness of our metamodel. We will contact the authors of these works to validate with them the correctness of our encoding of their OCCI extension.

Thirdly, EMF-VF could be buggy and produce erroneous validations of incorrect OCCIWARE models. However, EMF-VF is in use in industry for several years so we can expect that EMF-VF is a reliable framework.

Fourthly, we are not certain that we have encoded all OCL invariants covering all unexpected OCCI extensions or configurations. This is a threat to validity of the completeness of our metamodel.
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2.2.1</td>
<td>Final</td>
</tr>
<tr>
<td>Expected</td>
<td>Dispensation</td>
</tr>
<tr>
<td>Delivered</td>
<td>January 31, 2016</td>
</tr>
</tbody>
</table>

**Version 1.1**
Chapter 5

Conclusion and Perspectives

OCCI proposes a generic model, APIs and protocols for managing any cloud computing resources. We argue in this deliverable that OCCI suffers from the absence of a precise definition of its core concepts. To address this issue, we propose a precise semantics for OCCI implemented as an Ecore metamodel with OCL invariants, and we validate them on seven OCCI extensions published in the literature previously. Our metamodel can be seen as a modeling language to precisely define and exchange OCCI extensions and configurations between end-users and resource providers. Using the standard OCCI Core Model [1], resource providers ambiguously described their OCCI extensions in natural language [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. With our metamodel, they can now precisely encode their OCCI extensions and verify their consistency automatically.

As future work we will continuously complete the OCCIWARE Model Dataset with missed or new published OCCI extensions. We will submit our metamodel to the OGF’s OCCI working group gathering key world-wide OCCI specialists. We will apply our metamodel to four use cases of the OCCIware project: Datacenter as a Service, Deployment as a Service, Big Data as a Service, and Linked Data as a Service. We will define a concrete textual and graphical language to express OCCI extensions and configurations naturally. This language will be implemented in the OCCIWARE STUDIO, a model-driven tool chain providing a text editor and a graphical modeler to think about, design, model, and analyse OCCI extensions and configurations easily. Model-based generators will help to generate artefacts such as various forms of documentation as well as executable code for existing OCCI runtime frameworks, e.g. as erocci, OCCI4Java, rOCCI, pySSF, pyOCNI, etc. Finally, we will define an execution semantics of OCCI, i.e. an operational semantics for Create, Read, Update, Delete (CRUD) operations of OCCI and a behaviour semantics for OCCI actions. This will make OCCIWARE models executable natively inside a Models@run.time interpreter framework [34].

Availability

Readers can find both our precise OCCIware Metamodel (the Ecore package and all OCL invariants) and OCCIware Model Dataset at the following address:

https://github.com/occiware/ecore/tree/master/metamodel/

Acknowledgments

This work is supported by the OCCIware (www.occiware.org) research and development project funded by French Programme d’Investissements d’Avenir (PIA).
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D2.2.1</th>
<th>Status</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>May 31, 2015</td>
<td>Dissemination</td>
<td>Public</td>
</tr>
<tr>
<td>Delivered</td>
<td>January 31, 2016</td>
<td>Version</td>
<td>1.1</td>
</tr>
</tbody>
</table>

OCCIware
Deliverable D2.2.1
Status Final
Expected May 31, 2015
Delivered January 31, 2016
Dissemination Public
Version 1.1

Bibliography


<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D2.2.1</th>
<th>Status</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>May 31, 2015</td>
<td>Dissemination</td>
<td>Public</td>
</tr>
<tr>
<td>Delivered</td>
<td>January 31, 2016</td>
<td>Version</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Appendix A

OCCIware Ecore Package

The following listing defines the OCCI Ecore Package and its content formally. This formal definition is encoded with the Eclipse OCLInEcore textual syntax.

```ecore

package OCCI = 'http://schemas.ogf.org/occi' {
  abstract class Category {
    attribute term : String;
    attribute scheme : String;
    attribute title : String[?];
    property attributes : Attribute[*] { ordered composes };
  }

  class Attribute {
    attribute name : String;
    attribute mutable : Boolean[?];
    attribute required : Boolean[?];
    attribute default : String[?];
    attribute description : String[?];
    property type : ecore::EDataType[?];
  }

  class Kind extends Category {
    property parent : Kind[?];
    property actions : Action[*] { ordered composes };
    property entities : Entity[*] { ordered derived readonly } {
      derivation: Entity.allInstances()—>select(kind = self);
    }
  }

  class Action extends Category {
```
class Mixin extends Category
{
    property actions : Action[*] { ordered composes };
    property depends : Mixin[*] { ordered };
    property applies : Kind[*] { ordered };
    property entities : Entity[*] { ordered derived readonly };
    
    derivation: Entity.allInstances() → select(mixins→includes(self));
}

abstract class Entity
{
    attribute id : URI;
    property kind : Kind;
    property mixins : Mixin[*] { ordered };
    property attributes : AttributeState[*] { ordered composes };
}

class AttributeState
{
    attribute name : String;
    attribute value : String;
}

class Resource extends Entity
{
    property links#source : Link[*] { ordered composes };
}

class Link extends Entity
{
    property source#links : Resource;
    property target : Resource;
}

class Extension
{
    attribute name : String;
    attribute scheme : String;
    property import : Extension[*] { ordered };
    property kinds : Kind[*] { ordered composes };
    
    key term;
}

property mixins : Mixin[*] { ordered composes };
    
    key term;
property types : ecore::EDataType[*] { ordered composes };

class Configuration
{
    property use : Extension[*] { ordered };
    property resources : Resource[*] { ordered composes };
}

datatype URI : 'java.lang.String' { serializable };
datatype _'String' : 'java.lang.String' { serializable };
datatype Number : 'int' { serializable };
datatype _'Boolean' : 'boolean' { serializable };
}