A Precise Metamodel for Open Cloud Computing Interface

Philippe Merle, Olivier Barais, Jean Parpaillon, Noël Plouzeau, Samir Tata

A Precise Metamodel for Open Cloud Computing Interface

Philippe Merle
Inria Lille - Nord Europe
Villeneuve d'Ascq, France
Email: philippe.merle@inria.fr

Olivier Barais, Jean Parpaillon, Noël Plouzeau
IRISA - Inria Rennes - Bretagne Atlantique
Rennes, France
Email: firstname.name@irisa.fr

Samir Tata
Institut Mines-Telecom,
Telecom SudParis,
UMR CNRS Samovar, Evry, France
Email: samir.tata@mines-telecom.fr

***Abstract*—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 paper 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.**

***Index Terms*—Cloud Computing; Cloud Modeling; Open Cloud Computing Interface; Metamodeling;**

I. Introduction

Cloud computing has been adopted as a dominant delivery model for computing resources [1]. 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) [2]. Other XaaS terms are used nowadays to name different resources provided as services in the clouds [3]. Cloud management poses different challenges [4]. 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 [5].

OCCI is supported by a large community that includes providers of open source cloud software stacks such as Eucalyptus, OpenNebula, CloudStack, OpenStack, and CompatibleOne [6], and users such as the European Grid Infrastructure (EGI), to cite a few[1]. Several OCCI runtime frameworks exist, *e.g.*, erocci, rOCCI, pySSF, pyOCNI, and OCCI4Java, and rely on Erlang, Ruby, Python, Java programming languages, respectively. OCCI has already been used successfully for managing inter-cloud networking [7], building reliable storage virtualizations [8], IaaS resources description [9], PaaS resources description [10], SaaS resources description [10], SLA negotiation and enforcement in data management [11], service management [12], resource management in federated clouds [13], cloud monitoring [14] and reconfiguration [15], and autonomic computing [16]. A common denominator of these recent usages is the use of the REST architecture style for managing cloud computing resources.

The kernel of OCCI is a generic resource-oriented metamodel called the OCCI Core Model and defined in [17]. The OCCI Core Model can be interacted with using renderings (including associated behaviours) and expanded through extensions. For instance, the OCCI HTTP Rendering [18] defines how OCCI resources are accessible as REST resources over the HTTP network protocol. The OCCI Infrastructure Extension [9] defines OCCI-compliant compute, network and storage IaaS resources.

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 limita-

[1] http://occi-wg.org/community/implementations/

tions: 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 the configuration concept.

To tackle these issues, this paper contributes a precise metamodel for OCCI. This metamodel defines rigourously 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 rigourously defined using the 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 has been done on the first worldwide dataset of OCCI extensions already published in the literature. This dataset is composed of seven OCCI extensions addressing inter-cloud networking [7], IaaS [9], PaaS [10], SaaS [10], service management [12], cloud monitoring [14], and autonomic computing [16] domains, respectively. This validation highlights simplicity, consistency, correctness, completeness, and usefulness of the proposed metamodel.

This paper is organized as follows. Section II gives background on the OCCI Core Model and identifies five conceptual drawbacks of this model. Section III describes our precise metamodel for OCCI. Section IV validates our metamodel on seven already published OCCI extensions. Finally, Section V concludes on future perspectives.

II. Background and Drawbacks on the OCCI Core Model

As illustrated by Fig. 1, the OCCI Core Model [17] is a simple resource-oriented model composed of eight concepts[2]:

- `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.
- `Entity` is the abstract base class of all resources and links. Each resource/link has a unique `id`entifier, 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, 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, which defines no additional resource capabilities, aka no `attributes` and no `actions`. Templates allow to apply predefined values to attributes of OCCI types. They are implemented using `Mixin` instances and they associate at entity instantiation 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`.

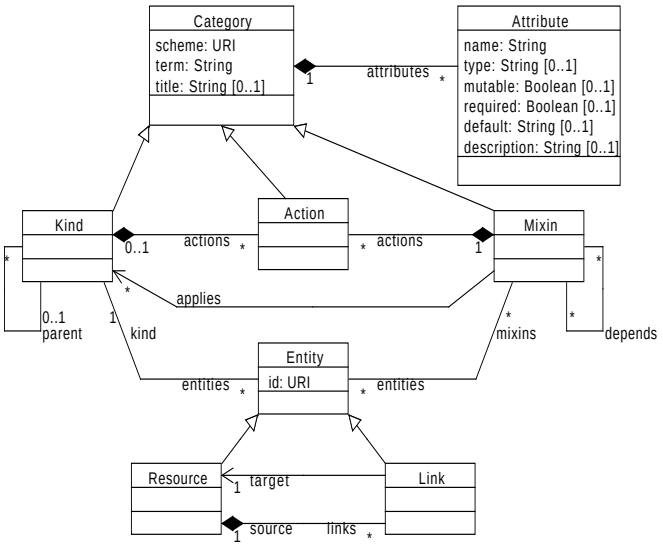

Fig. 1. UML class diagram of the OCCI Core Model (from [17])

Even if the OCCI Core Model is simple, it is generic enough to model any cloud computing resource such as inter-cloud networking [7], building reliable storage virtualizations [8], IaaS resources [9], PaaS resources [10], SaaS resources [10],

[2]Our work is based on the last revised draft of the OCCI Core Model [17], and not on the officially published and publicly available version of this document.

service management [12], SLA negotiation and enforcement in data management [11], resource management in federated clouds [13], cloud monitoring [14] and reconfiguration [15], and autonomic computing [16]. Section IV validates our precise metamodel for OCCI on seven of these already published works.

Through our study of OCCI in funded research projects (CompatibleOne, EASI-CLOUDS, OpenPaaS and OCCIware) we have identified five conceptual drawbacks/limitations on the OCCI Core Model:

*1) **Informal model**:* The structure of the OCCI Core Model is illustrated by the UML class diagram shown in Fig. 1. This diagram contains at least three semantic errors. In [17], `Category` and `Entity` are explicitly described as abstract types, *i.e.*, no instance of these types can exist. Then their type name must be in *italic* in the UML class diagram. URI is not a base data type in UML, it must be defined. Both static and dynamics 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, [17] 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) **Imprecise type classification system**:* [17] 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`s 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.

*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 [9], as this extension uses other scalar data types like 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. Therefore the OCCI Core Model needs an open and extensible data type system for OCCI attributes.

*4) **Vague and incomplete extension concept**:* While the concept of extension exists in OCCI's specification, it is too vaguely and incompletely defined. Informally, each OCCI extension is a set of `Kind` and `Mixin` 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.

*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 requiring to deploy them actually in the clouds. Therefore the OCCI Core Model needs a precise and rigorous definition of the notion of OCCI configurations.

The next section presents how these five limitations are addressed in our precise metamodel for OCCI.

III. OCCIware Metamodel

Our precise metamodel for OCCI, named OCCIware Metamodel, is based on the Eclipse Modeling Framework (EMF). 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 to encode the semantics of metamodels, to cite a few. The static semantics of our OCCIware 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 explained.

A. Precise metamodel

This section addresses Drawback II-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 OCCIware Metamodel is encoded as an Ecore package as it is illustrated in Fig. 2. Classes with a white background and their references in black encode the OCCI Core Model as defined in Section II and illustrated by Fig. 1. The static semantics of the OCCI built-in type classification system is rigourously defined in Section III-B. 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

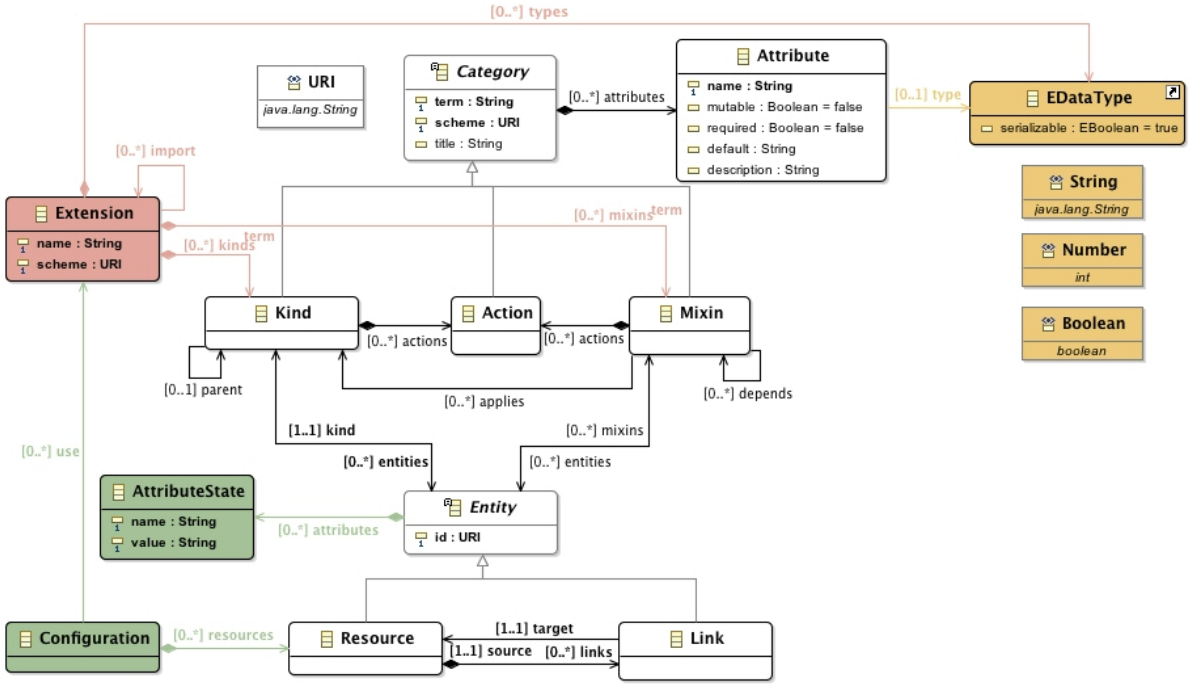

Fig. 2. Ecore diagram of OCCIware Metamodel

for OCCI attributes and are discussed in Section III-C. The `Extension` class and its references drawn in red model the concept of OCCI extensions. Its static semantics is rigourously defined in Section III-D. Both `Configuration` and `AttributeState` classes and their references drawn in green model the concept of OCCI configurations. Their static semantics is rigourously defined in Section III-E.

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. *The `identity` of a `Category` instance, aka 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. *Each `Category` instance must have a unique `identity`.*

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

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

```
context Category inv: scheme.substring(scheme.size()
    ,scheme.size()) = '#'
```

Definition. *The `category` of an `Action` instance is the `Category` instance owning this action*[3].

```
context Action def: category : Category =
    oclContainer().oclAsType(Category)
```

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

```
context Action inv: scheme = category.scheme.
    substring(1,category.scheme.size()-1) + '/' +
    category.term + '/action#'
```

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

```
context Category inv: attributes->isUnique(name)
```

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

```
context Mixin def: isTag : Boolean = attributes->
    isEmpty() and actions->isEmpty() and depends->
    forAll(isTag)
```

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

[3] `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.

```
context Mixin def: isTemplate : Boolean = attributes
    ->forAll(default <> null) and depends->forAll(
    isTag or isTemplate)
```

Definition. *Each `Entity` instance must have a unique `id`.*

```
context Entity inv: Entity.allInstances()->isUnique(
    id)
```

B. Precise type classification system

This section addresses Drawback II-2, aka imprecise type classification system, by defining the static semantics of the OCCI built-in type classification system with eight OCL invariants.

Definition. *The inheritance relation `parent` between `Kind` instances must form a direct acyclic graph. A kind instance must not inherit from itself directly or transitively*[4].

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

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

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

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

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

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

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

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

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

[4]In OCL, `a->closure(b)` returns the set { a, a.b, a.b.b, ... }.

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

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

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

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

C. Extensible data type system

This section addresses Drawback II-3, aka 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 `EEnumType`, 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 II-3.

In the OCCIware 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 Fig. 2. Each OCCI extension can define its own data `types`.

D. Extension concept

This section addresses Drawback II-4, aka 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. *`Extension` represents an OCCI extension, e.g., inter-cloud networking extension [7], infrastructure extension [9], platform extension [10], application extension [10], SLA negotiation and enforcement [11], cloud monitoring extension [14], and autonomic computing extension [16]. As encoded in the Ecore package shown in Fig. 2, `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. *Each `Extension` instance must have a unique `scheme` among all `Extension` instances.*

```
context Extension inv: Extension.allInstances()->
    isUnique(scheme)
```

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

```
context Extension inv: kinds->forAll(k | k.scheme =
    scheme)
```

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

```
context Extension inv: mixins->forAll(m | m.scheme.
    substring(1,scheme.size()-1) = scheme.substring
    (1,scheme.size()-1))
```

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

```
context Category def: extension : Extension =
    oclContainer().oclAsType(Extension)
```

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

```
context Extension def: isDefinedOrImported(category
    : Category) : Boolean = let e = category.
    extension in e = self or import->includes(e)
```

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

```
context Extension inv: kinds.parent->forAll(k |
    isDefinedOrImported(k))
```

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

```
context Extension inv: mixins.depends->forAll(m |
    isDefinedOrImported(m))
```

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

```
context Extension inv: mixins.applies->forAll(k |
    isDefinedOrImported(k))
```

E. Configuration concept

This section addresses Drawback II-5, aka 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. `Configuration` *represents a running OCCI system. As encoded in the Ecore package shown in Fig. 2,* `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. *The* `kind` *of all* `resources` *of a configuration must be defined by an* `extension` *that is explicitly* `used` *by this configuration.*

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

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

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

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

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

Definition. *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.*

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

Definition. *The* `configuration` *of a* `Resource` *instance is the* `Configuration` *instance owning this resource.*

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

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

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

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

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

F. Summary

The OCCIware Metamodel addresses the five drawbacks identified in Section II. This metamodel encodes all the eight core concepts of OCCI, gives a precise semantics of these concepts (*cf* Drawback II-1) including the type classification system (*cf* Drawback II-2), reuses an extensible data type system for typing OCCI attributes (*cf* Drawback II-3), and introduces two new concepts: extension (*cf* Drawback II-4), and configuration (*cf* Drawback II-5). The static semantics is expressed in OCL using twenty five invariants and seven definitions.

IV. VALIDATION

This section 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. Subsection A details the methodology we applied to build the dataset. Subsection B analyses the dataset quantitatively and qualitatively. Subsection C discusses the threats to the validity of our evaluation.

A. 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 [7], infrastructure [9], [11], platform [10], application [10], service management [12], cloud monitoring [14], and autonomic computing [16] 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, `e.g.`, 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 world-wide 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.

B. Analysis of the results

Due to space limitations, Table I presents only a summary of our OCCIWARE MODEL DATASET (see Section Availability for more information). 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 to 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 II and corrected by our metamodel discussed in Section III. The first eight classes are related to both informal model (II-1) and imprecise type classification system (II-2) drawbacks corrected in III-A and III-B respectively. The nonextensible data type system drawback (II-3) is addressed by the use of the `EDataType` class (III-C). Let's note that all the seven extensions needed to define new data types. The drawback of incomplete extension concept (II-4) was corrected by the addition of the `Extension` class (III-D). Finally, the configuration concept undefined drawback (II-5) was corrected by the addition of both `Configuration` and `AttributeValue` classes (III-E). Thus our dataset validates our metamodel quantitatively as all the classes and OCL constraints of our metamodel are covered by the dataset.

TABLE I
SUMMARY OF THE OCCIWARE MODEL DATASET

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

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

1) Simplicity: Is our metamodel simple to model cloud systems? The response 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 [19] or TOSCA [20] define a huge set of concepts, and only addressed IaaS and cloud applications, respectively.

2) Consistency: Is our precise semantics of OCCI consistent for modeling any OCCI systems? The response 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 equals to the number of instances multiplied by the number of invariants).

3) Correctness: Is our metamodel of OCCI correct for modeling any OCCI system? The response to this question is **yes** as our metamodel allows to correctly model all OCCI extensions in all observable aspects.

4) Completeness: Is our precise semantics of OCCI complete for modeling any OCCI systems? The response 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 II.

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

C. 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.

V. Conclusion and Perspectives

OCCI proposes a generic model, APIs and protocols for managing any cloud computing resources. We argue in this paper 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 validate them on seven OCCI extensions published in the literature previously. Based on the standard OCCI Core Model [17], resource providers ambiguously described their OCCI extensions in natural language [7]–[16]. 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.

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/

Acknowledgment

This work is supported by the OCCIware (www.occiware.org) research and development project funded by French Programme d'Investissements d'Avenir (PIA).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *NIST Special Publication*, vol. 800, no. 145, Sep. 2011.

[3] P. Banerjee, C. Bash, R. Friedrich, P. Goldsack, B. A. Huberman, J. Manley, C. Patel, P. Ranganathan, and A. Veitch, "Everything as a Service: Powering the new information economy," *IEEE Computer*, vol. 44, no. 3, pp. 36–43, 2011.

[4] J. Martin-Flatin, "Challenges in Cloud Management," *IEEE Cloud Computing*, vol. 1, no. 1, pp. 66–70, 2014.

[5] A. Edmonds, T. Metsch, A. Papaspyrou, and A. Richardson, "Toward an Open Cloud Standard," *IEEE Internet Computing*, vol. 16, no. 4, pp. 15–25, 2012.

[6] S. Yangui, I.-J. Marshall, J.-P. Laisne, and S. Tata, "CompatibleOne: The Open Source Cloud Broker," *Journal of Grid Computing*, vol. 12, no. 1, pp. 1–17, 2013.

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

[8] B. Nagarajan and J. Suguna, "A Review on Cloud Data Storage in Virtual Perspective," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 5, 2014.

[9] T. Metsch and A. Edmonds, "Open Cloud Computing Interface – Infrastructure," Open Grid Forum, OCCI-WG, Specification Document GFD-P-R.184, Oct. 2010.

[10] S. Yangui and S. Tata, "An OCCI Compliant Model for PaaS Resources Description and Provisioning," *The Computer Journal*, 2014, in press. [Online]. Available: http://comjnl.oxfordjournals.org/content/early/2014/11/19/comjnl.bxu132.abstract

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

[12] A. Ghrab, S. Skhiri, H. Kœner, and G. Leduc, "Towards a standards-based cloud service manager," in *3rd International Conference on Cloud Computing and Services Science (CLOSER 2013)*, 2013.

[13] M. Mosch, S. Groß, and A. Schill, "User-controlled resource management in federated clouds," *Journal of Cloud Computing*, vol. 3, no. 1, pp. 1–18, 2014.

[14] A. Ciuffoletti, "A Simple and Generic Interface for a Cloud Monitoring Service," in *4th International Conference on Cloud Computing and Services Science (CLOSER 2014)*, Apr. 2014, pp. 143–150.

[15] M. Mohamed, D. Belaïd, and S. Tata, "Monitoring and Reconfiguration for OCCI Resources," in *5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2013)*, vol. 1, Dec. 2013, pp. 539–546.

[16] M. Mohamed, M. Amziani, D. Belaid, S. Tata, and T. Melliti, "An autonomic approach to manage elasticity of business processes in the cloud," *Future Generation Computer Systems*, Available online 22 October 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X14002106

[17] R. Nyrén, A. Edmonds, A. Papaspyrou, and T. Metsch, "Open Cloud Computing Interface – Core," Open Grid Forum, OCCI-WG, Specification Document GFD-P-R.183, Apr. 2011, errata update available at http://redmine.ogf.org/projects/occi-wg/repository/show?rev=core-errata.

[18] T. Metsch and A. Edmonds, "Open Cloud Computing Interface – HTTP Rendering," Open Grid Forum, OCCI-WG, Specification Document GFD-P-R.185, Apr. 2011.

[19] D. Davis and G. Pilz, "Cloud Infrastructure Management Interface (CIMI) Model and REST Interface over HTTP," *vol. DSP-0263*, May 2012.

[20] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – A Runtime for TOSCA-based Cloud Applications," in *Service-Oriented Computing*. Springer, 2013, pp. 692–695.