Specifying Semantic Interoperability between Heterogeneous Cloud Resources with the FCLOUDS Formal Language

Stéphanie Challita, Faiez Zalila, Philippe Merle

Specifying Semantic Interoperability between Heterogeneous Cloud Resources with the FCLOUDS Formal Language

Stéphanie Challita, Faiez Zalila and Philippe Merle
Inria Lille - Nord Europe / University of Lille / CRIStAL UMR CNRS 9189, France
Email: firstname.lastname@inria.fr

***Abstract*—With the advent of cloud computing, different cloud providers with heterogeneous services and Application Programming Interfaces (APIs) have emerged. Hence, building an interoperable multi-cloud system becomes a complex task. Our idea is to design FCLOUDS framework to achieve semantic interoperability in multi-clouds, i.e., to identify the common concepts between cloud APIs and to reason over them. In this paper, we propose to take advantage of the Open Cloud Computing Interface (OCCI) standard and the Alloy formal specification language to define the FCLOUDS language, which is a formal language for specifying heterogeneous cloud APIs. To do so, we formalize OCCI concepts and operational semantics, then we identify and validate five properties (*consistency, sequentiality, reversibility, idempotence* and *safety*) that denote their characteristics. To demonstrate the effectiveness of our cloud formal language, we present thirteen case studies where we formally specify infrastructure, platform, Internet of Things (IoT) and transverse cloud concerns. Thanks to the Alloy analyzer, we verify that these heterogeneous APIs uphold the properties of FCLOUDS and also validate their own specific properties. Then, thanks to formal transformation rules and equivalence properties, we draw a precise alignment between our case studies, which promotes semantic interoperability in a multi-cloud system.**

***Index Terms*—Multi-Clouds; OCCI; Formal Language; Formal Verification; Alloy; Interoperability**

I. Introduction

The "Multi-cloud" computing, which aims to combine different offerings or migrate applications between different cloud providers, is becoming trendy. Multi-clouds improve cloud application performance and costs, and ensure their resiliency in case of outages [1]. But with the advent of cloud computing, different cloud providers with *heterogeneous services and Application Programming Interfaces (APIs)* have emerged. This is quite challenging for the implementation of multi-cloud systems. Several multi-cloud interoperability solutions have emerged to address this challenge. We mainly identify four strategies from the literature [2]: *i) services* such as Kaavo and RightScale, *ii) programming libraries* such as jclouds and fog, *iii) modeling languages* such as CloudML [3] and SALOON [4] and *iv) standards* such as Open Cloud Computing Interface (OCCI[1]) and Topology and Orchestration Specification for Cloud Applications (TOSCA[2]). Regardless of its abstraction level, a solution for multi-cloud interoperability must achieve a compromise between defining the common cloud principles and supporting any kind of cloud resources. This frustrating situation calls for more depth about the cloud providers' semantics to reason about the common principles that interoperability solutions must adhere to.

Our vision is to build FCLOUDS, a framework for semantic interoperability in a multi-cloud context [2]. By semantic interoperability we mean to identify the similarities and differences between cloud APIs concepts and to reason over them. Our 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 our FCLOUDS framework, we 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 desired and/or imposed cloud properties and finally define formal transformation rules between cloud concepts. We adopt the concepts of the OCCI common standard to define the formal language of the FCLOUDS framework. We choose to formalize OCCI with Alloy, a lightweight promising formal specification language designed by Daniel Jackson from the Massachusetts Institute of Technology (MIT) [5].

The key contribution of this paper is specifying semantic interoperability between heterogeneous cloud resources using the FCLOUDS formal language. The remainder of this paper is structured as follows. In Section II we explain the motivations behind our contribution. In Section III, we present our contribution, the FCLOUDS language that specifies OCCI core concepts and operational semantics, and verifies properties on how OCCI should work. In Section IV, we illustrate the use of our formal language with a series of thirteen examples. Finally, we discuss related work in Section V and we conclude in Section VI.

II. Motivations

Multi-cloud applications exist for several reasons. Among them we mention the optimization of the resource provisioning costs and the enhancement of the application quality of service. Apart from these two goals, the cloud developer should also comply to the different requirements of each service of the application. For instance, a web server interface should be

[1] http://occi-wg.org/
[2] https://www.oasis-open.org/committees/TOSCA

made available 24/7 for users, so it is highly recommended that the cloud developer provisions cloud resources from at least two heterogeneous providers to maintain the availability of the web server interface in case of an outage. Unfortunately, this is not a straightforward task. To illustrate the complexity faced when building a multi-cloud system, we introduce an example scenario of a developer who would like to provision virtual machines from two clouds, Google Cloud Platform (GCP) and DigitalOcean. Each of these two cloud providers is based on its own REpresentational State Transfer (REST) API, so the developer is faced to two heterogeneous APIs implementing different concepts. For instance, GCP refers to its compute service as *"instance"*, whereas DigitalOcean calls it *"droplet"*. To address the problem of interoperability in multi-clouds, we identified four strategies that were used in the literature:

1) *services* to offer a unique interface that handles the heterogeneity of different APIs, i.e., that only masks the problem but does not semantically resolve it,
2) *programming libraries* to allow developers to achieve interoperability at their code level, without necessarily understanding the underlying concepts of the API,
3) *modeling languages* to describe with an appropriate abstraction, a part of the cloud domain that was relevant at the moment of the definition of the modeling language. However, the designers require changing their modeling language, i.e., the metamodel, 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,
4) and *standards* to provide some concepts to be commonly used by cloud providers. However, standards may forget to define some useful concepts or may be overcrowded by useless concepts. Moreover, standards are accompanied by ambiguous documentations written in natural language.

Regardless of their strategy, the heterogeneity and the lack of formalization of multi-cloud interoperability solutions complicate the understanding of the cloud common principles. Our vision is to build a framework for semantic interoperability in a multi-cloud context. We refer to this framework as the FCLOUDS framework and we previously introduced it in [2]. To implement FCLOUDS, we advocate the use of a formal language that provides the formal specifications of the FCLOUDS cloud APIs. Formal methods are techniques that are based on mathematical notations and they will allow us 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. Formal methods allows us 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 focuses on *what* a system should do rather than *how* to accomplish it. Being precisely specified, verified and correctly understood, the cloud APIs can be correctly compared. For this, we 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.

We 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 European Grid Infrastructure Federated Cloud (EGI FC) and it was successfully extended to support heterogeneous aspects of the cloud domain, through OCCI Infrastructure [6], OCCI Platform [7], 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 schools 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. It allows us 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, we choose to formalize OCCI using a lightweight promising formal language, the Alloy specification language [5]. We refer to this formal specification as the FCLOUDS language.

III. The FCLOUDS Formal Language

The present contribution aims to define the formal language of the FCLOUDS framework. Therefore, we propose the FCLOUDS language, an Alloy-based formal language which makes explicit *OCCI core concepts* [8] and *OCCI REST operations* [9], as well as the underlying properties. In this section, we 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 [10] and in the OCCIware GitHub repository (see AVAILABILITY section).

A. *Specifying* FCLOUDS *static semantics*

The static semantics of FCLOUDS correspond to the formalization of the OCCI core concepts [8] in Alloy. We use a strategy based on *Time* dimension [5]. It allows us 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. Table I presents a summary of the FCLOUDS language concepts and due to space limitations, we detail in the following only four of these concepts 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 }
```

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 three fields: **id**, **kind** and **mixins**. Each entity 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** keyword signifies that the relation between a tuple from *id* and a tuple from *String* has a 1..1 cardinality. **Kind** 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 **id** and **kind** are immutable. As for the **mixins** field, it is mutable and it identifies the association between MIXIN atoms and their *Time*. The arrow product *Mixin* $\rightarrow$ *Time* is the relation we get by taking every combination of a tuple from MIXIN and a tuple from *Time* and concatenating them. Due to space restrictions in this paper, please refer to [10] for the specification of KIND and MIXIN concepts.

- RESOURCE represents any cloud computing resource, which refers to any entity hosted in a cloud, e.g., *Compute, Network, Storage,* etc. RESOURCE owns a set of mutable **links**.

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

The keyword *extends* 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.

- 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. LINK contains two mutable fields: **source** and **target**.

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

- 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 [11]. The **use** field, which is the set of extensions used in a configuration, is immutable because the extensions cannot be added on removed at runtime. Please refer to [10] for the specification of the EXTENSION concept. The **resources** field is mutable.

TABLE I: FCLOUDS Static Semantics.

FCLOUDS Concepts	Description
EXTENSION	a concrete cloud computing domain, *such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), cloud robotics, etc.*
CONFIGURATION	a running OCCI system
ENTITY	an abstract type defining the set of all resources and links
RESOURCE	represents any cloud computing resource, *such as a virtual machine*
LINK	a relation between two resources
ATTRIBUTE	a resource property, *such as the hostname of a virtual machine*
DATATYPE	an abstract type defining enumerations, lists, records, etc.
CATEGORY	the abstract base class inherited by KIND, MIXIN and ACTION
KIND	immutable type of OCCI entities
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*

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

B. *Specifying* FCLOUDS *operational semantics*

The operational semantics of FCLOUDS correspond to the formalization of the informal OCCI behavioural specification detailed in [9]. 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 [12]. 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. The following predicate shows how we formally specify the creation of a resource.

```
pred CreateResource[config : Configuration,                    1
    resourceId : String, kind : Kind, mixins :
     set Mixin, t, t' : Time] {
  // preconditions at instant t                                2
     no resource : config.resources.t |                        3
         resource.id = resourceId
     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
      config.resources.t' =                                    11
          config.resources.t + resource } }                    12
```

At time *t*, we specify that our 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** keyword expresses the null cardinality of the empty resource set that

satisfies the constraint *resource.id = resourceId*. The dot in *resource.id* is the standard notation for accessing a feature, or attribute, of an instance of a class. As well, we specify that the kind and the mixins of the resource we want to create, are contained in our configuration extensions (cf. lines 4 and 5). At time *t'*, we add to our configuration resources, one resource with the id, kind and mixins, passed as argument to the predicate (cf. lines 7 to 11). Eight operations were defined to the CONFIGURATION concept of FCLOUDS, as depicted in Table II.

C. 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, we proceed by the definition of some structural and behavioural properties to ensure its correctness and to express its desired/required behaviour. We formally encode the *consistency*, *sequentiality*, *reversibility*, *idempotence* and *safety*. The last two properties are classified into a broader property which is the *conformance to HTTP 2 protocol* [13]. Then, using Alloy analyzer, we validate that these properties adequately hold in our FCLOUDS static and operational semantics. We specify a big scope, i.e., we bound the number of atoms allowed for each signature to 10, in order to be confident that the assertion holds. If no counterexamples are returned with such big scope, we can be confident that our language reflects the desired semantics.

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. We can also analyze what could not be instantiated, thus can't be deployed in real-world. In these cases, our formal language might be over-constraining so we 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 {
  one config : Configuration, set resourceId :
        String, kind : Kind, mixins : Mixin,
        t : Time |
          CreateResource[config, resourceId,
              kind, mixins, t, t.next] }
```

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 II. However, the notion of consistency is basic and does not suffice in order to validate our FCLOUDS language. There are other examples of reliable verification and validation tasks that

TABLE II: 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		+	+

can be performed on pairs of operations such as sequentiality and/or reversibility of operations:

2) 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"* [9]. We 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,
       t : Time |
    CreateResource[config, resourceId, kind,
        mixins, t, t.next]
    and RetrieveResource[config, resourceId,
          t.next, t.next.next]
    implies resource.id = resourceId
    and resource.kind = kind
    and resource.mixins.(t.next.next) = mixins
}
```

3) 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 CreateResourceReverseDeleteResource {
  all config: Configuration, resourceId:
      String, kind : Kind, mixins : Mixin,
      t: Time {
      CreateResource[config, resourceId, kind,
          mixins, t, t.next]
      implies DeleteResource[config,
          resourceId, t.next, t] } }
```

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 we 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 [13], we identified two properties of the HTTP methods and we verified in our formal language that the appropriate pairs of operations respect these properties.

- *Idempotence:* a method is idempotent when it always produces the same server external state even if applied several times [13]. 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 we verify in our 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,
      t : Time |
    CreateResource [config, resourceId,
        kind, mixins, t, t.next]
    and CreateResource [config, resourceId,
        kind, mixins, 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.

- *Safety:* a method is safe when it does not change the external server state [13]. 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, we check that *Retrieve Resource*, *Retrieve Link* and *Retrieve Collection*[3] 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 one resource : config.resources.(t.
      next) {
    resource.id = resourceId } }
```

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 II lists all the operations that we 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, we check that the FCLOUDS language, the core language of our FCLOUDS framework, correctly reflects these properties, so we guarantee that it is valid and that we implemented the desired behaviour.

IV. Validation

To validate the effectiveness of our formal language, we 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, we have surveyed the literature to find all the already published OCCI extensions. We 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, we have assumed that all these extensions are correct as they were already accepted through a peer review process. Our validation allows us to confirm:

1) the power of expression of our FCLOUDS language (Subsection A),
2) the validity of the FCLOUDS behaviour we defined, on all of the OCCI extensions (Subsection B),
3) the ability of our language to define domain-specific properties (Subsection C), and,
4) its ability to encode equivalence predicates, i.e., transformation rules between heterogeneous concepts, and to define properties of the equivalence (Subsection D).

A. Catalog of cloud formal specifications

Thanks to our proposed formal language, we succeeded to precisely encode thirteen heterogeneous APIs, as shown in Table III that gives a summary of our FCLOUDS framework dataset. For each concept of our language, Table III 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. Due to space constraints, we give only 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 (see AVAILABILITY section).

1) OCCI INFRASTRUCTURE [6] 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

[3]A collection is a set of resources with the same kind.

TABLE III: 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

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 we declare as fields to our 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) OCCI CRTP [14] 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 [15]. 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 [16]. We 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 [17]. It defines VMware instance, storage and network resource types and associated links.
6) OCCI PLATFORM [7] is an OCCI-based extension for PaaS application domain. It defines application and component resource types and associated links.
7) Model-Driven Management of Cloud Applications with OCCI (MODMACAO) [18] is an application of OCCI for managing cloud applications. It defines generic and specific application, component, installation dependency and execution dependency resource types.
8) Open Mobile Cloud Robotics Interface (OMCRI) [19] 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 [20].
10) OCCI SLA [21] defines OCCI types for modeling service level agreements.
11) OCCI MONITORING [22] is a draft specification of OCCI that defines sensor and collector types for monitoring cloud systems.
12) CLOUD SIMULATION [23] 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 [17] 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.

B. *Verification of* FCLOUDS *properties*

Being rigorously encoded using the same formal language, i.e., FCLOUDS, our 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 Alloy analyzer, we verify that our thirteen

formal specifications satisfy all the assertions, i.e., properties, that we formulated in our FCLOUDS language (cf. Table II). For instance, we verify that *Create Compute* operation of OCCI INFRASTRUCTURE is idempotent and that *Update Instance* operation of GCP is not safe.

C. Definition & validation of domain-specific properties

Our framework allows us to verify some domain-specific properties. For instance, in the following listing, we 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 our formal specification respects and implements the following requirement of OCCI INFRASTRUCTURE specification: *"NetworkInterface connects a Compute instance to a Network instance"* [6].

```
assert
    NetworkInterfaceBetweenComputeAndNetwork {
 all config : Configuration, linkId : URI,
     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)
} } } }
```

D. Formal transformation rules & equivalence properties

The last step for achieving semantic interoperability between heterogeneous domains is to define predicates that implement bidirectional formal transformation rules between resources with similar semantics. In the following example, we show how we can migrate from an OCCI INFRASTRUCTURE virtual machine at to a GCP virtual machine. We map the attributes of a given COMPUTE resource to those of an INSTANCE resource. Also, since we learned from the OCCI INFRASTRUCTURE and GCP documentations that the memory in OCCI is expressed in gigabytes, whereas it is in megabytes in GCP, we 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 our transformation rules and to Model-Driven Engineering (MDE) principles, we can later imagine a model that factorizes common attributes for re-usability between OCCI INFRASTRUCTURE and GCP.

V. RELATED WORK

Only few works from the literature applied *formal methods for the cloud*, which proves the novelty of this domain. Di Cosmo et al. [24] adopted automata to define the Aeolus formal component model that captures cloud resources. Benzadri et al. [25] proposed a formal model for cloud computing using Bigraphical Reactive Systems (BRS). Amazon Web Services (AWS) [26] used TLA+ specification language in their complex systems such as S3 and DynamoDB. Brogi et al. [27] used Petri nets to formally model the behaviour of TOSCA operations, requirements and capabilities. Bobba et al. [28] specified and validated Google' Megastore, Apache Cassandra, Apache ZooKeeper, and RAMP using Maude language and model checker. Besides using different techniques to reason over the cloud, these five works differ in their objectives too. [24] studies the complexity of finding a deployment plan in the cloud. [25] also reasons over cloud concepts for deployment and adaptation purposes. [26] aims at finding subtle bugs in their internal distributed algorithms, which helps optimizing their systems. [27] analyzes the validity of the deployment plan. [28] verifies the performance and correctness of cloud storage systems. Our work focuses on the interoperability concern by formalizing the static and operational semantics of the cloud domain.

VI. CONCLUSION

To promote multi-cloud systems, we advocate exploring the semantics of each cloud API and reason about them in order to understand their similarities and differences, hence to achieve semantic interoperability. This work occurs in the context of the FCLOUDS framework, where we advocate the use of formal specification techniques, specifically the Alloy language, to rigorously and clearly describe the requirements of cloud APIs. We formalize the OCCI open standard in order to implement our formal language for the clouds. Our 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 our approach, we conducted thirteen case studies to show how our 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 us 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, we define formal transformation rules between their formal specifications, thus ensure semantic interoperability between them.

For future work, we will extend our catalog of formal cloud APIs in order to achieve our vision of building FCLOUDS, the first framework for semantic interoperability in multi-clouds. Using the FCLOUDS language, we will formally specify Amazon Web Services, OpenStack, etc. We will also enrich the FCLOUDS language, which is the backbone of all the FCLOUDS precise models, 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. To do so, we might require to use widespread formal techniques other than Alloy and which are more suitable for expressing such property, like the TLA+ specification language and TLC, its model checker [29]. Furthermore, although model checkers verify that the properties are valid within a big scope of research, we need to prove it in the absolute through a convincing argument. Hence, we will use automated proof assistants [30], namely Coq [31], which implements algorithms and heuristics to build a proof describing the sequence of needed moves in order to solve a property. Finally, we will allow the FCLOUDS framework to be executable inside the first formal-based real-world interoperability bridge.

AVAILABILITY

Readers can find the FCLOUDS formal language and all the thirteen formal specifications at https://github.com/occiware/fclouds-Framework.

ACKNOWLEDGMENT

This work is supported by the OCCIware research and development project funded by French Programme d'Investissements d'Avenir (PIA) and the Hauts-de-France Regional Council.

REFERENCES

[1] D. Petcu, "Multi-Cloud: Expectations and Current Approaches," in *International workshop on Multi-cloud applications and federated clouds*. ACM, 2013, pp. 1–6.

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

[3] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "CloudMF: Model-Driven Management of Multi-Cloud Applications," *ACM Transactions on Internet Technology (TOIT)*, vol. 18, no. 2, p. 16, 2018.

[4] C. Quinton, D. Romero, and L. Duchien, "SALOON: A Platform for Selecting and Configuring Cloud Environments," *Software: Practice and Experience*, vol. 46, no. 1, pp. 55–78, 2016.

[5] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[6] R. Nyrén, A. Edmonds, A. Papaspyrou, T. Metsch and B. Parák, "Open Cloud Computing Interface - Infrastructure," Open Grid Forum, Specification Document GFD.224, Feb. 2016.

[7] T. Metsch and M. Mohamed, "Open Cloud Computing Interface - Platform," Open Grid Forum, Specification Document GFD.227, Feb. 2016.

[8] R. Nyrén, A. Edmonds, A. Papaspyrou, T. Metsch and B. Parák, "Open Cloud Computing Interface - Core," Open Grid Forum, Specification Document GFD.221, Feb. 2016.

[9] R. Nyrén, A. Edmonds, T. Metsch and B. Parák, "Open Cloud Computing Interface - HTTP Protocol," Open Grid Forum, Specification Document GFD.223, Feb. 2016.

[10] M. Ahmed-Nacer, S. Tata, W. Gaaloul, P. Merle, J. Parpaillon, N. Plouzeau, S. Challita, "OCCI Behavioural Model," OCCIware Deliverable 2.2.2, Dec. 2016.

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

[12] A. Garis, A. C. Paiva, A. Cunha, and D. Riesco, "Specifying UML Protocol State Machines in Alloy," in *International Conference on Integrated Formal Methods*. Springer, 2012, pp. 312–326.

[13] M. Belshe, M. Thomson, and R. Peon, "Hypertext Transfer Protocol Version 2 (HTTP/2)," 2015.

[14] M. Drescher, B. Parák and D. Wallom, "Open Cloud Computing Interface - Compute Resource Template Profile," Open Grid Forum, Specification Document GFD.222, Feb. 2016.

[15] F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-Driven Management of Docker Containers," in *9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 718–725.

[16] S. Challita, F. Paraiso, and P. Merle, "A Precise Model for Google Cloud Platform," in *6th International Conference on Cloud Engineering (IC2E)*. IEEE, 2018.

[17] F. Zalila, P. Merle, J. Parpaillon, S. Kallel, M. Ahmed-Nacer, W. Gaaloul, C. Gourdin, "OCCI Extension Models," OCCIware Deliverable 2.4.1, Sep. 2017.

[18] F. Korte, S. Challita, F. Zalila, P. Merle, and J. Grabowski, "Model-Driven Configuration Management of Cloud Applications with OCCI," in *8th International Conference on Cloud Computing and Services Science (CLOSER)*, 2018, pp. 100–111.

[19] P. Merle, C. Gourdin, and N. Mitton, "Mobile Cloud Robotics as a Service with OCCIware," in *2nd International Congress on Internet of Things (ICIOT)*. IEEE, 2017, pp. 710–713.

[20] E. Rachkidi, D. Belaïd, N. Agoulmine, and N. Chendeb, "Cloud of Things Modeling for Efficient and Coordinated Resources Provisioning," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2017, pp. 175–193.

[21] G. Katsaros, "Open Cloud Computing Interface - Service Level Agreements," Open Grid Forum, Specification Document GFD.228, Oct. 2016.

[22] A. Ciuffoletti, "Open Cloud Computing Interface - Monitoring Extension," Open Grid Forum, Specification Document 1.2, Jan. 2016.

[23] M. Ahmed-Nacer, W. Gaaloul, and S. Tata, "OCCI-Compliant Cloud Configuration Simulation," in *International Conference on Edge Computing (EDGE)*. IEEE, 2017, pp. 73–81.

[24] R. Di Cosmo, S. Zacchiroli, and G. Zavattaro, "Towards a Formal Component Model for the Cloud," in *International Conference on Software Engineering and Formal Methods*. Springer, 2012, pp. 156–171.

[25] Z. Benzadri, F. Belala, and C. Bouanaka, "Towards a Formal Model for Cloud Computing," in *International Conference on Service-Oriented Computing*. Springer, 2013, pp. 381–393.

[26] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services Uses Formal Methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015.

[27] A. Brogi, A. Canciani, J. Soldani, and P. Wang, "A Petri Net-based Approach to Model and Analyze the Management of Cloud Applications," in *Transactions on Petri Nets and Other Models of Concurrency XI*, 2016, pp. 28–48.

[28] R. Bobba, J. Grov, I. Gupta, S. Liu, J. Meseguer, P. C. Olveczky, and S. Skeirik, "Design, Formal Modeling, and Validation of Cloud Storage Systems Using Maude," Tech. Rep., 2017.

[29] L. Lamport, *Specifying Systems: the TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[30] D. W. Loveland, *Automated Theorem Proving: a logical basis*. Elsevier, 2016.

[31] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy *et al.*, *The Coq Proof Assistant Reference Manual: Version 6.1*, 1997.