On Semantic Detection of Cloud API (Anti)Patterns

Hayet Brabra*, Achraf Mtibaa, Fabio Petrillo, Philippe Merle, Layth Sliman, Naouel Moha, Walid Gaaloul, Yann-Gaël Guéhéneuc, Boualem Benatallah, Faïez Gargouri

Abstract—*Context:* Open standards are urgently needed for enabling software interoperability in Cloud Computing. Open Cloud Computing Interface (OCCI) provides a set of best design principles to create interoperable REST management APIs. Although OCCI is the only standard addressing the management of any kind of cloud resources, it does not support a range of best principles related to REST design. This often worsens REST API quality by decreasing their understandability and reusability.

Objective: We aim at assisting cloud developers to enhance their REST management APIs by providing a compliance evaluation of OCCI and REST best principles and a recommendation support to comply with these principles.

Method: First, we leverage patterns and anti-patterns to drive respectively the good and poor practices of OCCI and REST best principles. Then, we propose a semantic-based approach for defining and detecting REST and OCCI (anti)patterns and providing a set of correction recommendations to comply with both REST and OCCI best principles. We validated this approach by applying it on cloud REST APIs and evaluating its accuracy, usefulness and extensibility.

Results: We found that our approach accurately detects OCCI and REST(anti)patterns and provides useful recommendations. According to the compliance results, we reveal that there is no widespread adoption of OCCI principles in existing APIs. In contrast, these APIs have reached an acceptable level of maturity regarding REST principles.

Conclusion: Our approach provides an effective and extensible technique for defining and detecting OCCI and REST (anti)patterns in Cloud REST APIs. Cloud software developers can benefit from our approach and defined principles to accurately evaluate their APIs from OCCI and REST perspectives. This contributes in designing interoperable, understandable, and reusable Cloud management APIs. Thank to the compliance analysis and the recommendation support, we also contribute to improving these APIs, which make them more straightforward.

Index Terms—Cloud Computing, REST, OCCI, Pattern, Anti-pattern, Analysis, Specification, Detection, Ontology.

1 Introduction

Cloud Computing is becoming more and more attractive. Its economic pay-as-you-go model and elasticity nature are among the main assets characterizing this paradigm. Suitably, many projects have been carried out as a joint effort between industry and academia to develop open standards for the cloud with the aim of increasing its adoption. Nowadays, open standards are inevitably needed to enable interoperability among cloud services. Open Cloud Computing Interface (OCCI) is the only open standard that addresses the basic management tasks over any kinds of cloud resources, e.g., Infrastructure as a Service, Platform as a Service, and Software as a Service [1], [2]. OCCI defines a meta-model for cloud resources and a RESTful API[1] for managing these resources. Among the OCCI specifications, the OCCI HTTP Protocol [3] describes a set of recommended best principles to create unified REST APIs for managing cloud resources. These best principles form a minimal set of practices to achieve interoperability and provide a uniform way to discover and manage cloud resources across various providers [3]. The poor adoption of such principles in current cloud resource management APIs negatively impacts the interoperability of cloud services.

Currently, OCCI members provide a textual description of suggested principles [3] along with a compliance test tool [4] that does not provide a detailed description about the detected principles. However, this tool can be used to show the presence of best principles but not to show the absence

* *Corresponding author*

- *H. Brabra is with Faculty of Economics and Management of Sfax, Miracl Laboratory, University of Sfax, Tunisia.
E-mail: hayet.brabra@telecom-sudparis.eu*
- *H. Brabra and W. Gaaloul are with the Computer Science Department, SAMOVAR, Telecom SudParis, CNRS, Paris-Saclay University, Evry, France.
E-mail: Firstname.Lastname@telecom-sudparis.eu*
- *A. Mtibaa is with École Nationale d'Electronique et des Télécommunications de Sfax, Miracl Laboratory, University of Sfax, Tunisia.
E-mail:achraf.mtibaa@enetcom.usf.tn*
- *F. Petrillo is with École Polytechnique Montréal, Montreal, Canada.
E-mail: fabio@petrillo.com*
- *P. Merle is with Inria Lille - Nord Europe, Villeneuve d'Ascq, France.
E-mail: philippe.merle@inria.fr*
- *L. Sliman is with French Engineering School Efrei, Paris, France.
E-mail: layth.sliman@efrei.fr*
- *N. Moha is with Département d'Informatique, Université du Québec à Montréal, Montréal, Canada.
E-mail: moha.naouel@uqam.ca*
- *Y. Guéhéneuc is with DGIGL, École Polytechnique à Montréal, Montréal, Canada.
E-mail: yann-gael.gueheneuc@polymtl.ca*
- *B. Benatallah is with the University of New South Wales, Sydney, Australia.
E-mail: boualem@cse.unsw.edu.au*
- *F. Gargouri is with Higher Institute of Computer Science and Multimedia of Sfax, Miracl Laboratory, University of Sfax, Tunisia.
E-mail: faiez.gargouri@isims.usf.tn*

[1] Application Programming Interface

of one of them. Previous researches [5], [6] on best principles design for REST APIs mainly dealt with general REST APIs like Facebook and Twitter or made from the perspective of mobile applications as in [7] and networking domain like [8]. Hence, they do not perfectly fit REST APIs developed for managing cloud services or resources. Actually, in addition to REST aspects, OCCI provides principles that relate to the structure and definition of cloud resources, that is how cloud resources could be defined, created or linked to other resources. For example, to create a link between two cloud resources, the creation request should contain HTTP POST as a method along with the kind category defining the type of link (e.g., Storage Link, Network Link) as well as source and target attributes. To this end, such aspects should be considered in designing Cloud APIs to ensure sustainable interoperability and an easy discovery of cloud resources.

To address this need, in our previous work [9], we leveraged patterns and anti-patterns to drive respectively the good and poor practices of OCCI best principles. In particular, we defined compliance to OCCI best principles as OCCI patterns and non-compliance to OCCI best principles as OCCI anti-patterns. We then provided a semantic-based approach to specify these patterns and anti-patterns and to detect them automatically. Moreover, in another previous work [2], we conducted a systematic study of REST best principles on three cloud APIs including OCCI. In consequence, we showed that OCCI fails to support some of the best principles related to the REST aspects in the design of REST APIs. More specifically, OCCI only follows 56% of the best REST principles. This lack of support makes the design of Cloud REST APIs by cloud providers or developers difficult and decreases the understandability and reusability of these APIs. To alleviate this, we believe that both OCCI and REST best principles should be supported together in the design of Cloud REST APIs, which can enhance their understandability and reusability.

In the extension to our previous work [9], our ultimate objective is threefold: (i) Specifying along with OCCI (anti)patterns the REST (anti)patterns and providing their formal definitions; (ii) Assisting Cloud providers or API developers in revising their APIs by providing a set of correction recommendations to comply with both REST and OCCI best principles; And finally (iii) exploring the current application of OCCI and REST best principles on real Cloud APIs. To this end, we extend our previous work [9] (which only supports the definition of OCCI (anti)patterns and provides a set of SWRL [2] rules to automate their detection), with the following additional contributions:

- Reviewing literature with the aim of identifying the set of patterns that must be respected and anti-patterns that should be averted to conform to REST best principles;
- Proposing semantic definitions of 21 common REST (anti) patterns for Cloud REST APIs by specifying their detection rules in terms of SWRL rules in combination with SQWRL [3] queries.
- Proposing patterns and anti-patterns detection algorithms based on SPARQL[4] queries which provides an automated detection of both OCCI and REST (Anti) patterns along with a set of correction recommendations in case of any anti-pattern detection.

To validate our work, we developed a proof of concept implementation[5] to support the detection of REST and OCCI (anti)patterns while providing a set of correction recommendations in case of any anti-pattern detection. Thus, we contribute in assisting developers to revise their Cloud REST APIs to be compliant with both OCCI and REST best principles. To conduct this evaluation, we rely on a validation dataset that includes five real-world Cloud RESTful APIs: OOi, COAPS, OpenNebula OCCI, Amazon S3, and Rackspace.

The remainder of this article is organized as follows. In Section 2 we give an overview on REST architectural style and OCCI. Section 3 presents the OCCI and REST patterns and anti-patterns. In Section 4, we introduce our proposed approach. Section 5 presents a validation of our approach and an interpretation of the experiment results. In Section 6, we examine related work. Finally, we conclude the article and provide insights for future work in Section 7.

2 Background

2.1 Representational State Transfer (REST)

REpresentation State Transfer (REST) was defined in the PhD thesis of Roy Thomas Fielding [10]. REST is an architectural style defining a set of rules for the design of distributed systems that assists the design and development of web applications. It is commonly used in the design of APIs for modern web services. Web services supporting properly the REST architectural style are called RESTful Web services and the application programmatic interfaces of these services are called REST APIs. Indeed, the basic concepts driving the REST APIs design are originally the result of architectural choices of the Web to reinforce the scalability and robustness of networked and resource-oriented systems that are based on HTTP. These concepts specifically include the following [7], [10]:

- *Resource addressability*. APIs manage and manipulate resources, which represent any information that can be named. Each resource is uniquely recognized through an appropriate Uniform Resource Identifier (URI).
- *Resource representations*. REST components apply actions on a resource through a representation that defines the intended or current state of a resource. Generally, the representations are associated with metadata such as content-types in the headers of HTTP messages. This is done in order to allow clients and servers correctly handling these representations.
- *Representation caching*. Caching is the capability to hide the resource representations with the aim of reducing network traffic between servers and clients, which may enhance therefore the performance. Caching can be achieved at the client side or at any intermediate between clients and servers.
- *Uniform interface*. Resources are accessed and manipulated using the set of standard methods provided by the

[2] Semantic Web Rule Language

[3] Semantic Query-Enhanced Web Rule Language

[4] Simple Protocol and RDF Query Language

[5] http://www-inf.it-sudparis.eu/SIMBAD/tools/ORAP-Detection/

HTTP protocol, e.g., Get, Post, Put, Delete, Head, etc. Each method is conceived with own expected, standard behaviour and standard status codes.

- *Statelessness*. The interactions between a client and a server should be stateless, that is each request must have all required information for being understandable without using of any stored information from the server.
- *Hypermedia as the engine of state*. Resources can be interrelated together using links. Links between resources are embodied in their representations. This enables clients to discover and navigate relationships and to maintain a consistent interaction state.

2.2 Open Cloud Computing Interface (OCCI)

Open Cloud Computing Interface (OCCI) is the open cloud standard [11] addressing heterogeneity, interoperability, integration and portability in Cloud Computing. OCCI comprises a set of open community-lead specifications provided by the Open Grid Forum (OGF). In a nutshell, OCCI offers a RESTful Protocol and API for all kinds of management tasks related to any type of cloud resources, which include Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), and potentially Everything as a Service (XaaS) from hardware resources to business applications. With the aim of being modular and extensible, OCCI provides a set of specification documents[6] describing four main layers: Protocols, Renderings, Core, and Extensions as illustrated in Fig. 1.

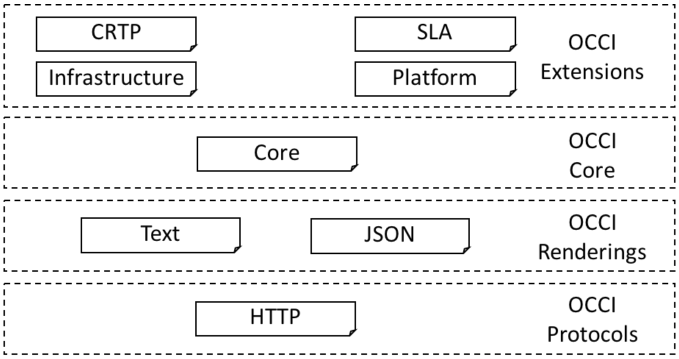

Fig. 1: OCCI Specifications (Source: [12])

The OCCI Core specification [13] defines a general-purpose resource-oriented model including a dedicated resource classification type system. For more details, readers can refer to [14]. Protocols and renderings represent together the way to interact with OCCI core. The OCCI Protocol specifies how a specific network protocol can be exploited with the aim of interacting with the OCCI Core Model. Currently, OCCI members only defined the HTTP Protocol [3], while the others protocols have been put in the future. Moreover, each OCCI Rendering specification provides a specific rendering of the OCCI Core Model. Only Text [15] and JSON[7] rendering have been provided currently. Furthermore, OCCI Core can be easily expanded using extensions, where each one provides a particular extension of the OCCI Core model describing a specific application domain. For instance, OCCI Infrastructure [16] aims at abstracting IaaS network, storage and compute resources. OCCI Compute Resource Templates Profile (CRTP) [17] specifies a set of well-known instances of compute resources, such as `large`, `small` and `medium` computes. OCCI Platform [18] defines PaaS application and component resources. Finally, OCCI Service Level Agreements (SLA) [19] defines how SLA can be applied to OCCI resources.

3 REST/ OCCI (ANTI)PATTERNS

In this section, we present the REST and OCCI patterns and anti-patterns that we consider in this article. Therefore, we analyzed both the literature and the OCCI standard with the aim of identifying the set of REST and OCCI (anti)patterns. This analysis is done in context of OCCIware project [8]. OCCIware is a scientific research project that aimed at providing a new precise metamodel for OCCI, along with an enhanced tooling environment called OCCIware Studio. Both OCCIware metamodel and Studio are developed for designing, managing and analyzing any kind of cloud resources. In the following, we provide and summarize both REST and OCCI (anti)patterns definitions.

3.1 REST (Anti)patterns

REST (anti) patterns represent the good and bad practices in the REST APIs regardless of any cloud standard. To identify them, we conduct a literature review both in research and industry. We performed our research using Google Scholar, Elsevier Scopus, ACM Digital Library, Web of Science, IEEE Xplore and arXiv.org with special focus on the critically-reviewed research conferences, journals and magazines that were relevant to our research context from the year of 2004. Initially, 25 approaches, catalogs and technical reports on REST (anti) patterns were chosen. However, some of these works contain redundant contents. After filtering them, we ended up with 7 studies, including [20], Rodrigues et al. [7], Palma et al. [5], [6], Vinoski [21], Stowe [22], Richardson and Ruby [23]. We analyzed all the above studies to define the REST (anti)patterns and organize them into categories, while inspiring from the work of Masse [20]. Tables 1, 2, 3, 4 and 5 define (anti)patterns we specified for each category.

- URI (anti)patterns: They represent the poor and good practices in URIs and how they are exposed by services (see Table 1).
- HTTP methods (anti)patterns: They represent the poor and good practices in HTTP methods and how they must be used by REST APIs (see Table 2).
- Error Handling (anti)patterns: They represent the poor and good practices in HTTP messages and how they must be used as a response of a HTTP request method (see Table 3).
- HTTP Header (anti)patterns: They represent the poor and good practices in HTTP headers and how they must be used to complete requests with metadata or complementary data (see Table 4).
- Hypermedia (anti)patterns: They represent the poor and good practices in hypermedia representation and how it

[6] Available at http://occi-wg.org/about/specification/
[7] JavaScript Object Notation

[8] Available at www.occiware.org

should be supported to link between resources (see Table 5).

TABLE 1: URI Design (Anti)Patterns

1. Tidy URIs vs. Amorphous URIs

Description: URIs in the REST resources should be simple to read and tidy. The Tidy URIs pattern appears when URIs use suitable lower resource naming and do not contain trailing slashes, underscores and extensions. While, the Amorphous URI anti-pattern appears when URIs contain symbols, capital letters, underscores, etc. This results in decreased readability and understandability of these URLs [2], [6].

2. Verbless URIs vs. CRUDy URIs

Description: Verbless URIs pattern appears when URIs use one of the Standard HTTP methods, namely POST, GET, DELETE or PUT. While, CRUDy URIs anti-pattern is appeared as consequence of using CRUDy terms such as read, create, delete, update or their equivalent in URIs. Using these terms in actions or resource URIs can be overloading the HTTP methods and prevent API users to employ the appropriate and common HTTP methods [2], [6].

3. Singularized nodes vs. pluralized nodes

Description: URIs should correctly employ singular/plural nouns for resources naming within an API [2], [6]. Singularized nodes pattern occurs when the last node is provided as a singular noun in the URI of Delete/Put requests and as a plural noun in POST requests. Contrariwise, the Pluralized Nodes anti-pattern can occur when singular nouns used in POST requests or plural names used in DELETTE/PUT requests. The occurrence of such anti-pattern may have negative impacts in certain cases. For instance, if the last node in Delete (or PUT) request URL is provided as plural, the API clients are not able to create or delete a collection of resources, which leads to 403 Forbidden as a server response.

TABLE 2: HTTP methods (anti)patterns

1. Correct use of POST, GET, PUT, DELETE, HEAD vs. Tunneling every things through GET and POST

Description: The correct use of POST, GET, PUT, DELETE, or HEAD pattern is occurred, whether the following principles are correctly considered by the API developer: [2]:

- GET must be used to retrieve a representation of a resource
- POST must be used to create a new resource in a collection or to execute controllers
- HEAD should be used to retrieve response headers
- PUT must be used to both insert and update a stored resource
- DELETE must be used to remove a resource

In contrast, the Tunneling every things through GET and POST anti-pattern can occur if the API developer relies only on GET or POST methods to execute any kind of actions or operations including deleting, updating or creating a resource. In general, the occurrence of this anti-pattern may lead to several problems: violation of the semantic purpose of each HTTP method, the crawlers from search engines can cause inappropriate side effects [24].

TABLE 3: Error handling (Anti)Patterns

1. Supporting Status Code vs. Ignoring Status Code

Description: The status codes in REST APIs from the classes 2xx, 3xx, 4xx, and 5xx allow servers and clients to communicate in a semantic way. Supporting status code pattern can occur when the provided status code in the response is correct. In general, the correct use of status codes should be as follows [2]:
200 (“OK”) should be used to indicate non-specific success
200 (“OK”) must not be used to communicate errors in the response body
201 (“Created”) must be used to indicate successful resource creation
202 (“Accepted”) must be used to indicate successful start of an asynchronous action
204 (“No Content”) should be used when the response body is intentionally empty
302 (“Found”) should not be used
304 (“Not Modified”) should be used to preserve bandwidth
400 (“Bad Request”) may be used to indicate non specific failure
401 (“Unauthorized”) must be used when there is a problem with the client’s credentials
403 (“Forbidden”) should be used to forbid access regardless of authorization state
404 (“Not Found”) must be used when a client’s URI cannot be mapped to a resource
405 (“Method Not Allowed”) must be used when the HTTP method is not supported
406 (“Not Acceptable”) must be used when the requested media type cannot be served
409 (“Conflict”) should be used to indicate a violation of resource state
500 (“Internal Server Error”) should be used to indicate API malfunction
In contrast, the wrong or unsupported status codes in REST APIs lead to Ignoring Status Code anti-pattern. Consequently, this would decrease the reusability, and hinder the loose coupling and good interoperability of these APIs.

TABLE 4: HTTP Header (Anti)Patterns

1. Supporting Caching vs. Ignoring Caching

Description: REST developers and clients often prefer to not use the caching capability as its implementation is complex. Nevertheless, caching capability is considered as one of the fundamental REST constraints [2], [5]. Supporting Caching pattern appears when the API developer does not indicate no-cache or no-store for Cache-Control parameter or specifies an ETag in the response header. Otherwise, the Ignoring Caching anti-pattern can take place. As a result of this anti-pattern, throughput and scalability in requests-per-second would be decreased, which degrades the overall performance.

2. Supporting MIME Types vs. Ignoring MIME Types
Description: The server should allow defining resources in different format, including xml, json, pdf, etc., which in turn may enable clients to develop a more adaptable service consumption using diverse languages. Supporting MIME Type pattern appears when the server supports multiple resource representation formats. In contrast, the Ignoring MIME Types anti-pattern can occur when the server relies on a unique representation or uses personalized formats. Consequently, this restricts the accessibility, reusability as well as the readability of the resources [2], [5].

TABLE 5: Hypermedia (Anti)Patterns

1. Supporting Hypermedia vs. Forgetting Hypermedia
Description: Hypermedia provides the way of linking resources together. Supporting Hypermedia pattern occurs when the API developer includes consistent links within the resource representations. In contrast, the absence of links within these representations leads to Forgetting Hypermedia anti-pattern. Consequently, the dynamic communication between clients and servers would be decreased because the servers do not provide for clients any link to follow.

TABLE 6: OCCI REST Related (Anti)Patterns

1. Compliant URL vs. Non-Compliant URL
Description: A URL path should be compliant, i.e. whenever the URL path is rendered it must be either a string or as defined in RFC6570 [3]. The non-Compliant URL anti-pattern occurs when one of these guidelines is ignored.
2. Compliant Request Header vs. Non-Compliant Request Header
Description: A Request Header can be considered compliant, i.e. client (e.g. OCCI client) : - should specify the media types its implementation data formats (e.g. OCCI Data formats) support in the Accept header, - must specify the implementation (e.g. OCCI version) version number in the User-Agent header, - must specify the media type its implementation data format (e.g. OCCI data format) support in the Content-type header [3]. The *Non-Compliant Request Header anti-pattern* occurs when one of these guidelines is ignored.
3. Compliant Response Header vs. Non- Compliant Response Header
Description: A Response Header can be considered compliant, i.e. a server (e.g. OCCI server): - should specify the media types its implementation data formats (e.g. OCCI Data formats) support in the Accept header, - must specify the media type its implementation data format (e.g. OCCI data format) used in an HTTP response in Content-type header, - must specify the implementation (e.g. OCCI version) version number in the Server header [3]. The Non-Compliant Response Header anti-pattern occurs when one of these guidelines is ignored.

TABLE 7: Cloud Structure (Anti)Patterns

1. Compliant Link between Resources vs. Non-Compliant Link between Resources
Description: To create a Link between two resources, HTTP POST must be used and its kind as well as a "source" and "target" attributes must be provided. The Non-Compliant Link Anti-pattern may occur when one of these attributes is omitted.
2. Compliant Association of Resource(s) with Mixin vs. Non-compliant Association of Resource(s) with Mixin
Description: To associate a Resource with a Mixin in accordance with OCCI, the HTTP POST must be used and the URIs that uniquely identify the resources must be introduced within the request. The Non-compliant Association anti-pattern may occur when one of these guidelines is ignored.
3. Compliant Dissociation of Resource(s) From Mixin vs. Non-compliant dissociation of resource(s) From Mixin
Description: To dissociate a resource from a Mixin in accordance with OCCI, the HTTP DELETE must be used and the URIs that uniquely identify the resources must be introduced within the request. The Non-compliant Dissociation of Resource(s) anti-pattern may occur when one of these guidelines is ignored.

3.2 OCCI (Anti)patterns

OCCI (anti)patterns represent the good and bad practices of the presented guidelines in the OCCI RESTful Protocol [3]. To identify them, we conducted an analysis study with 6 participants (1 computer science Professor, 2 computer science PHD students, 1 master student, 1 computer science post-doc, 1 engineer) that are familiar with OCCI standard. We devised the 6 participants into two groups of 3 participants each. We asked each group to manually analyse each textual description of each guideline provided in the OCCI specification and identify the appropriate patterns and anti-patterns. After this task, we have organized a meeting between two groups to share and to discuss the finding results. Finally, as a result of this study, we have considered all OCCI patterns and anti-patterns that have been commonly identified by both groups and extracted from each OCCI guideline that must (or should) be followed.
Ultimately, three categories of OCCI (anti)patterns have distinguished: Cloud OCCI REST Related (Anti)Patterns, Structure Related (Anti)Patterns, Management Related (Anti)Patterns.

- OCCI REST Related (Anti)Patterns: They represent the poor and good practices related to REST API components. In contrast to general REST (Anti) patterns defined previously (section 3.1), OCCI REST (Anti) Patterns are defined according to OCCI standard. We identify 3 OCCI REST (anti)patterns that relate to the URL, response header and request header (see Table 6).
- Cloud Structure Related (Anti)Patterns: They represent the poor and good practices to link cloud resources between each other as well as to create a collection of resources using a Mixin, with respect to OCCI perspective (see Table 7).
- Management Related (Anti)Patterns: They represent the poor and good practices in the main management operations applied on cloud resources and services, with respect to OCCI perspective (see Table 8). We define 6 patterns and their corresponding anti-patterns respectively in Query interface, Create, Retrieve, Update, Delete operations and in Trigger actions.

Here, it should be noted that, in contrast to REST anti-patterns, where the occurrence of each one may lead to specific impact, the occurrence of each OCCI anti-pattern would restrict the discovery and the interoperability of cloud resources [3].

TABLE 8: Management Related (Anti)Patterns

1.Query Interface Support vs Missing Query Interface
Description: To be compliant with OCCI, Query interface must be implemented, which enables the client to discover all provider capabilities [3]. It defines three operations applied on Mixins, Actions and Kind, including requesting of all available Kinds, Actions and Mixins, and adding or removing a Mixin. Query interface must be found at the path /-/ on the implementation root and carried out through the HTTP method GET, POST and DELETE. The no support of query interface on all requests engenders the Missing Query Interface anti-pattern.
2. Compliant Create vs. Non-Compliant Create
Description: The creation of any OCCI entity (i.e., Resource, Mixin) should be compliant with OCCI guidelines defined in OCCI RESTful Protocol specification [3], which include the following constraints: - To create a Mixin, the HTTP POST must be used and HTTP Category term, scheme and location must be introduced in the request. - To create a Resource instance within Mixin or collections, the HTTP POST must be used, otherwise HTTP PUT must be used. Additionally, the HTTP Category rendering that uniquely defines a particular Kind instance must be provided to define the kind of a resource instance. The Non-Compliant Create anti-pattern may occur when one of these guidelines is not supported.
3. Compliant Update vs. Non-Compliant Update
Description: The update of any OCCI entity should be compliant with OCCI guidelines defined in OCCI RESTful Protocol specification [3], which includes specifically the following : - To fully update a Mixin, the HTTP PUT must be used and all URIs defined in the collection of Mixin must be specified within the update request. - To partially update a Resource or Link, the HTTP POST must be used and the new information that will be updated must be specified within the update request. - To fully update a Resource or Link, the HTTP PUT must be used in the request. The Non-Compliant Update anti-pattern may occur when one of these guidelines is poorly adopted.
4. Compliant Delete vs. Non-Compliant Delete
Description: The Delete of a Resource, Mixin or Link should be compliant with OCCI guidelines defined in OCCI RESTful Protocol specification [3]. These guidelines specifically include the following: - To remove a Mixin, HTTP DELETE must be used and the Mixin URI defining the Mixin that will be deleted, must be defined in the request. - To delete a Resource or Link below a given path or only one, the HTTP DELETE must be used and only the URI identifying the entity that will be removed must be provided. The Non-Compliant Delete anti-pattern may occur when one of these guidelines is poorly adopted.
5. Compliant Retrieve vs. Non-Compliant Retrieve
Description: The retrieve of a Resource or Link should be compliant with the OCCI guidelines defined in the OCCI RESTful Protocol specification [3]. These guidelines specifically include the following: - To retrieve a Resource or Link instance, the HTTP GET must be used and the server must provide as a response the HTTP Category associated with a set of attributes that identify the resource or link kind. - To retrieve all Resources belonging to kind or mixin, the HTTP GET must be used and a list comprising all instances of a resource that belonging to the requested mixin (or kind) must be returned in the response. The Non-Compliant Retrieve anti-pattern may occur when one of these guidelines is poorly adopted.
6. Compliant Trigger Action vs. Non-Compliant Trigger Action
Description: To trigger an action on a Resource or Link while following the OCCI guidelines, the HTTP POST must be provided and the URI must contain a query with the action term. Additionally, the specific HTTP Category that identifies the applied action, must be also introduced in the request. The *Non-Compliant Trigger Action anti-pattern* may occur when one of these guidelines is poorly adopted.

4 Approach Overview

After defining OCCI and REST patterns and anti-patterns, in this section, we describe our approach to detect their occurrences. The proposed approach relies on ontologies with the aim of formally specifying OCCI and REST (anti)patterns, ensuring their automatic detection and providing a set of correction recommendations in case of any anti-pattern detection. As shown in Fig. 2, the proposed approach proceeds in four steps as follows:

Step 1. Definition of OCCI/REST (Anti)Patterns: This step consists in defining the basic ontology *(Anti)Patterns Ontology* allowing a semantic specification of OCCI and REST patterns and anti-patterns. The proposed ontology embodies the most important and relevant concepts needed for the detection and the recommendation purposes.

Step 2. Analysis and Definition of Detection Rules: This step aims to analyze the textual definitions of OCCI and REST (anti) patterns detailed in Section 3 for eliciting their pertinent properties. These pertinent properties will be then exploited to specify the semantic rules to detect patterns and anti-patterns as well as the explanations of suggested recommendations. Both (Anti)Patterns Ontology and detection rules form the knowledge base (KB), which will be later interrogated through SPARQL queries for the detection and recommendation purposes.

Step 3. Detection of (Anti) Patterns: This step aims at checking the compliance of the selected Cloud REST API with both REST and OCCI best principles while suggesting appropriate recommendations in case of the anti-pattern detection. As shown in Fig. 2, this step includes the following two phases:

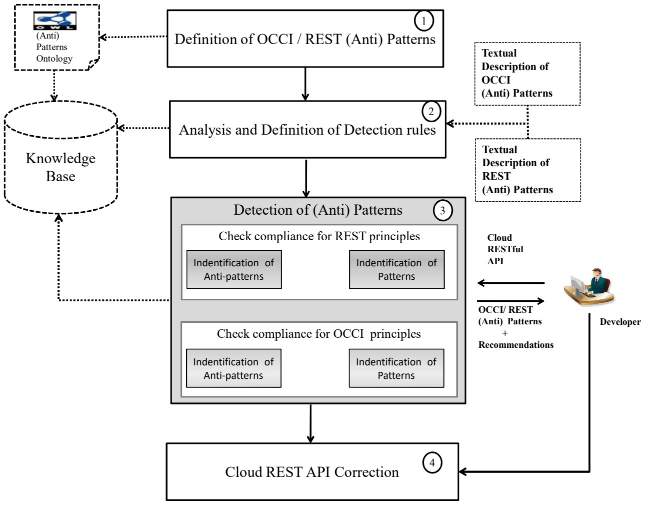
(Anti) Patterns Ontology
Definition of OCCI / REST (Anti) Patterns
1
Textual Description of OCCI (Anti) Patterns
Knowledge Base
Analysis and Definition of Detection rules
2
Textual Description of REST (Anti) Patterns
Detection of (Anti) Patterns
3
Check compliance for REST principles
Indentification of Anti-patterns
Indentification of Patterns
Cloud RESTful API
OCCI/ REST (Anti) Patterns + Recommendations
Developer
Check compliance for OCCI principles
Indentification of Anti-patterns
Indentification of Patterns
Cloud REST API Correction
4

Fig. 2: Approach overview

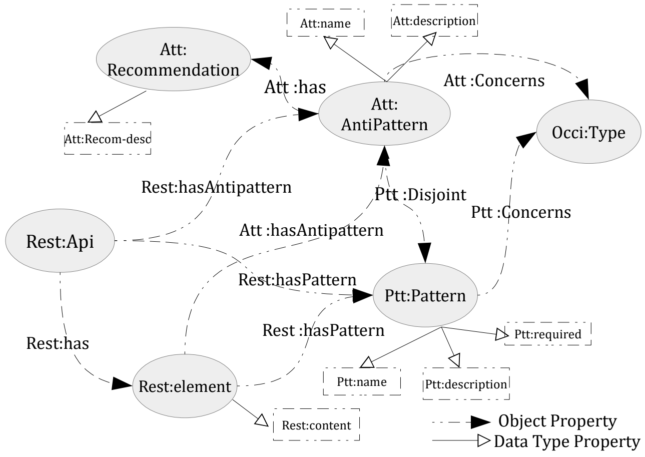

Fig. 3: (Anti) Patterns Ontology

- *Check compliance for REST principles:* This step aims at applying on Cloud REST API the REST detection rules specified in Step 2 to detect REST (anti)patterns. If no REST anti-pattern is detected, the evaluated REST API is considered as REST-compliant and details related to REST patterns will be provided to the Cloud API developer for analysis or understanding purposes. Otherwise, details related to both REST patterns and anti-patterns will be displayed. While regarding REST anti-patterns, a set of recommendations is also returned to avoid their occurrence.
- *Check compliance for OCCI principles:* This step aims at applying, on cloud REST API, the OCCI detection rules specified in Step 2 to detect OCCI (anti)patterns. If no OCCI anti-pattern is detected, the evaluated REST API is considered as OCCI-compliant and details related to OCCI patterns will be given to the Cloud API developer for analysis or understanding purposes. Otherwise, details related to both OCCI patterns and OCCI anti-patterns along with a set of suggested recommendations to avoid the OCCI anti-patterns occurrence, will be displayed. It is worth mentioning that the assessed REST API can be considered as OCCI and REST compliant when it does not contain any REST and OCCI anti-pattern.

Step 4. Cloud REST API Correction: This step allows developers to manually revise their API by following the obtained recommendations on REST and OCCI anti-patterns in order to avoid their occurrences observed in Step 3. Here, it should be noted that the developer can take into consideration or ignore the suggested recommendations basing on the relevance of the detected anti-patterns.

In the following sections, we detail the first three steps of our approach namely, definition of OCCI/REST (Anti)Patterns, analysis and definition of detection rules, and detection of (Anti)Patterns.

4.1 Definition of OCCI/REST (Anti)Patterns

In this step, a domain analysis on RESTful API documentations and the OCCI specification for cloud resources is performed with the aim of building the *(Anti) Patterns Ontology*. The *(Anti) Patterns Ontology* aims at providing a semantic specification of OCCI and REST (anti) patterns using the common Web Ontology Language (OWL) [25]. It is defined as a collection of four ontologies, which are mainly: Anti-Pattern Ontology, Pattern Ontology, and OCCI Ontology and REST API Ontology.

Pattern Ontology: The Pattern Ontology, as depicted in Fig. 3, defines the relevant information describing OCCI and REST pattern through the attributes that are associated to its core concept *Ptt:Pattern*. Those attributes (i.e. correspond to data type properties in OWL language) are *Ptt:name*, *Ptt:description* and *Ptt:required*, which represents a boolean value indicating that the given pattern is required or no. In addition, the *Ptt:Pattern* concept holds two relationships: *Ptt:Disjoint* and *Ptt:Concerns*. The *Ptt:Disjoint* relationship defines the opposite anti-pattern for a given pattern. The *Ptt:Concerns* relationship depicts that a pattern relate to a given OCCI Type (i.e. Resource, Link, etc.). Finally, the *Ptt:Pattern* concept also represents the range of *Rest:hasPattern*, indicating that a given API or its elements can have this pattern.

Anti-Pattern Ontology: As depicted in Fig. 3, the definition of Anti-Pattern Ontology is similar to the Pattern Ontology. However, as opposed to OCCI or REST patterns, we use OCCI or REST anti-patterns to capture a bad practice of such OCCI or REST principles. Each anti-pattern denoted by Att: AntiPattern , is defined by Att:name and Att:description, and associated with the concept Att:Recommendation through the Att:has relationship. Att:Recommendation defines the suggested recommendation to avoid the anti-pattern that is associated with once it occurs. More precisely, it contains Att:Recomm-desc that provides a textual explanation explaining which best principle is not respected, that leads to the occurrence of the associated anti-pattern and suggesting as a result an advice to avoid it.

REST API Ontology: The REST API ontology allows providing a semantic definition of the functional and structural features of the REST APIs. To build this ontology, we examine several documentation on REST APIs while considering OCCI RESTful API into account. The principal concept is *Rest:API* representing a REST API, which is linked, as shown in Fig. 4(b), to the following concepts:

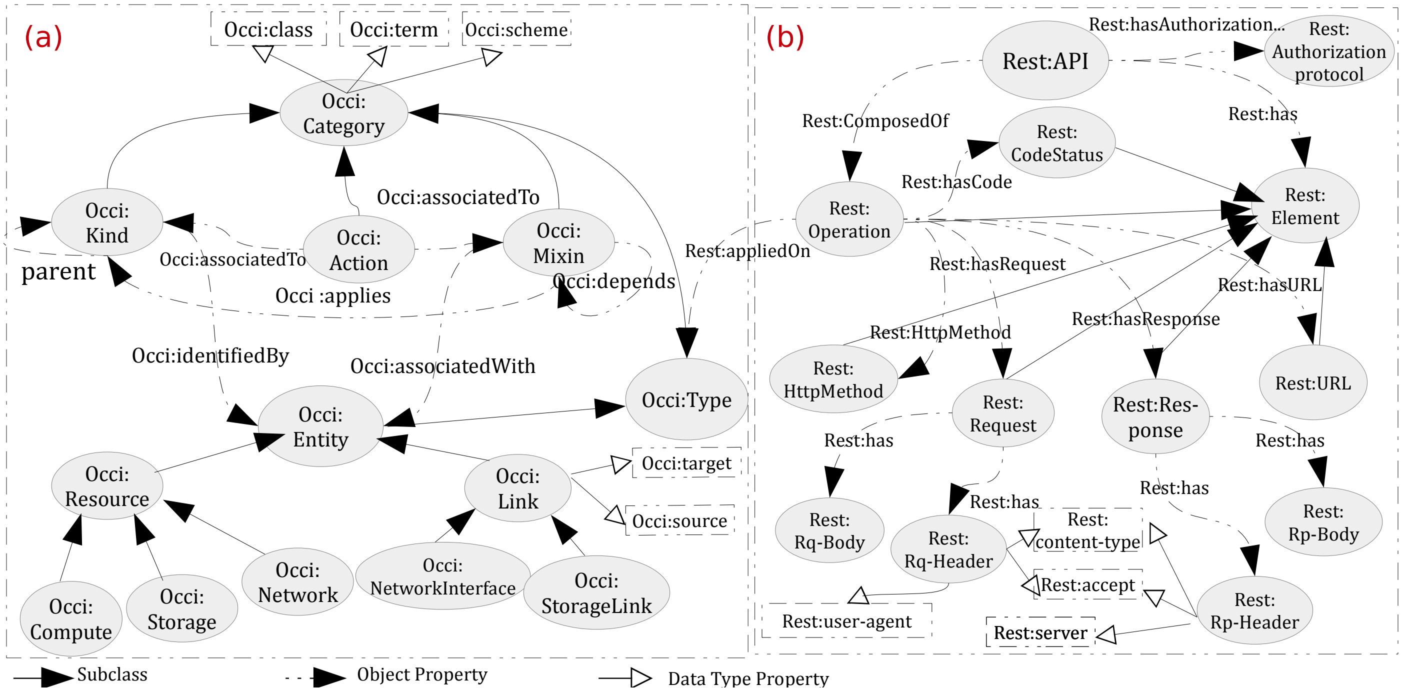

Fig. 4: (a) OCCI Ontology; (b) REST API Ontology

Rest:AuthorizationProtocol, *Rest:Element*, *Rest:Operation*. *Rest:AuthorizationProtocol* concept defines the authorization protocol used to access the REST API. The *Rest:Element* describes through its subclasses the most important components that we can find in a REST API, including, response header, request header, code status, URL, operation, response, request, response body, request body etc. Finally, we use the *Rest:Operation* concept to define the possible operations that can be applied on cloud resources e.g., *Create a Server*.

OCCI Ontology: All RESTful API operations are applied on OCCI types (i.e. Resource, Link, etc.) that are already specified both in OCCI Infrastructure [16] and OCCI Core [13]. To allow such capability, we specify the OCCI ontology that presents a semantic specification of the cloud resources abstraction provided in these two specifications while following the OCCI Rendering syntax of these resources [15], [26]. As shown in Fig. 4(a), the heart of this ontology is the *Occi:Resource* concept, which in turn associated with three concepts: *Occi: Compute*, *Occi: Network* and *Occi: Storage*. Accordingly, a cloud resource may be a virtual switch, a virtual storage, a virtual sever, etc. Additionally, *Occi:Resource* is associated to the *Occi:Link* concept, which used to link one resource instance with another. We distinguish two type of link *Storage Link* and *Network Interface*. Each type is described through two attributes (e.g. *Occi:source*, *Occi:target*). Both *Occi:Resource* and *Occi:Link* inherit the *Occi:Entity* concept. The *Occi:Kind* is the core of the classification type system built into the OCCI Core Model. *Occi:Kind* is a specialization of *Occi:Category* and introduces additional capabilities in terms of actions. *Occi:Action* describes a set of operations applicable to an entity instance. The last type specified by the OCCI Core Model is the Occi:Mixin, which allows extending the OCCI entity by plugging in/out a set of attributes and actions. An instance of Mixin can be attached to an entity instance, which may provide additional capabilities at run-time [13].

4.2 Analysis and Definition of Detection rules

In this step, the textual definitions of the (anti)patterns listed in Tables 1 to 8 were analyzed with the aim of eliciting their pertinent properties. We then exploit these properties to specify the semantic rules needed to detect (anti)patterns. To do so, we adopt SWRL language to express these rules.

Listing 1: Syntax of SWRL Rules

```
$a_1$ ^ $a_2$ ^...$a_n$ → b
$a_1$ , $a_2$ , $a_n$ : Antecedent; b : Consequent ; where a
    and b are atoms
Atom ← C(i)|D(v)| R(i,j)|U(i,v)|built-In(p,$v_1$,...,
    $v_n$)|i=j|i # j
C = Class; D = Data Type; R = Object Property; U =
    Data Type Property
i,j = Object variable names or Object individual
    names
$v_1$,..., $v_n$ = Data type variable names or Data type
    value names
p = Built-In names
```

Listing 1 shows the syntax used to express SWRL rules, which is based on two parties: Antecedent and Consequent. The Antecedent consists of one or more atoms with the aim of specifying the conditions that should be fulfilled. On the other hand, the Consequent allows defining the impacts that can be produced once the conditions defined in Antecedent are fulfilled. It often results in one atom. An atom can be a class (e.g., Rest:Operation concept), a data type (e.g., POST:xs:string), an Object Property (e,g.,

Rest:has), a Data type Property (e.g. Rest:verb) or a Built-in. The latter can be SWRL built-ins (e.g., swrlb:matches) or SQWRL built-ins (e.g., sqwrl:select). SWRL built-ins are user-defined predicates that can be used in SWRL rules. They include basic mathematical operators and functions for string and date manipulations. Among the strong points in this language is its extensibility. In fact, SWRL language allows users to create their personalized built-ins functions that will be integrated straightaway in SWRL rules. We exploit such features in order to create our custom built-ins that will be exploited in detection rules. Furthermore, SQWRL built-ins define a set of operators that can be seen as SQL-like operations used in conjunction with SWRL rules to make use of the knowledge inferred by SWRL rules. In some cases, the detection of anti-pattern depends on the value of inferred knowledge itself. SQWRL built-in like sqwrl:select allows us to access such value.
Listings 2 and 3 illustrate, respectively, the SWRL rules for Verbless URI pattern and their anti-pattern CRUDy URIs.

Listing 2: SWRL rule for Verbless URIs Pattern

```
1.Rest:Operation(?operation)^ Rest:hasHttpMethod(?
    operation, ?httpmethod) ^ Rest:verb(?httpmethod,
     ?verb) ^detection: matchesOne(?verb, "POST", "
    PUT", "GET", "DELETE") ^ Rest:hasURL(?operation,
     ?url)^ Rest:value(?url,?urlval)^ detection:
    contain(?return,?urlval, "create", "update", "
    read", "delete") ^ swrlb:matches(?return, "False
    ") → Rest:hasPattern(?operation, Ptt:
    Verbless_URIs)
```

As shown in Listing 2, the SWRL rule for the Verbless URIs pattern aims at evaluating an operation of each request in the API (i.e., Rest:Operation(?operation)). This evaluation consists of checking whether the verb of HTTP method (i.e., Rest:verb(?httpmethod, ?verb)) included in the operation contains one of the HTTP common verb i.e., POST, PUT, GET, DELETE. This is accomplished through our custom built-in “detection: matchesOne(?verb, ”POST”, ”PUT”, ”GET”, ”DELETE”)”. Additionally, the occurrence of this pattern also requires that the URL value (i.e., Rest:value(?url, ?urlval) does not contain one of the common CRUDy terms i.e., create, update, read, delete. The detection:contain built-in consists of checking whether the URL value contains one of the CRUDy terms and returns a boolean value as a result. This value will then be checked through swrlb:matches to make sure that its value corresponds to the “False” term.

Listing 3: SWRL rules for CRUDy URIs Anti-pattern

```
1.Rest:Operation(?operation) ^ Rest:hasHttpMethod(?
    operation, ?httpmethod) ^ Rest:verb(?httpmethod,
     ?verb) ^ detection:matchesOne (?verb, "create",
     "update", "read", "delete") → Rest:
    hasAntipattern (?operation, Att:CRUDy_URIs)
2.Rest:Operation(?operation) ^ Rest:hasURL(?
    operation, ?url) ^ Rest:value(?url,?urlval) ^
    detection:contain(?return,?urlval, "create", "
    update", "read", "delete") ^ swrlb:matches(?
    return, "True") → Rest:hasAntipattern (?
    operation, Att:CRUDy_URIs)
```

In contrast, as shown in Listing 3, the CRUDy URIs anti-pattern can be detected through two SWRL rules. The first rule consists of checking for each operation, through the built-in detection:matchesOne (?verb, “create”, “update”, “read”, “delete”), whether the used verb of HTTP method contains one of the common CRUDy terms. The positive satisfaction of this condition indicates the occurrence of CRUDy URIs anti-pattern. The second rule consist of evaluating the URL of each operation defined in an API in order to detect whether its value contains one of the common CRUDy terms. Both detection:contain and swrlb:matches are exploited to carry out this detection. The existence of one of the CRUDy terms in the URL value results in the occurrence of CRUDy URIs anti-pattern in the analyzed operation.

Listing 4: SWRL rule for Compliant Link between Resources Pattern

```
1. Rest:Operation(?op) ˆ Rest:hasHttpMethod(?op, ?
    httpmd) ˆ Rest:verb(?httpmd, ?verb) ˆ swrlb:
    matches(?verb, "POST") ˆ Rest:hasRequest(?op,?
    req) ˆ Rest:has(?req, reqbody) ˆ Rest:
    hasParameterDefinition(?reqbody, ?pradef) ˆ Occi
    :Link(?link) ˆ Rest:isComposedOf(?pradef, ?link)
     ˆ Occi:identifiedBy(?link, ?kind) ˆ Occi:term(?
    kind, ?term) ˆ detection:matches(?term, "
    Storagelink", "Network Interface") ˆ Occi:scheme
    (?kind, ?schee) ˆ Occi:class(?kind, "kind") ˆ
    Occi:source(?link, ?source) ˆ Occi:target(?link,
     ?target) -> Rest:hasPattern(?op, Ptt:
    Compliant_Link)
```

In the same way, as shown in Listing 4, we define a SWRL rule to detect Compliant Link between Resources Pattern, which aims at checking for each link operation the verb of HTTP Method , the Kind of link and whether its source and target attributes do exist or not. A given link operation is reported as it has Compliant Link between Resources Pattern if we ensure that the used HTTP verb is “POST”, the Kind term *?term* has either “Network Interface” or “Storage Link”, the Kind scheme (?kind, ?scheme) is not empty, the Kind class ?class has as value “kind”. Finally, the link source and target attributes of the concerned link should not contain empty values. Fig. 5 shows a partial instantiation of the (Anti) Pattern Ontology with information that we have extracted from a REST operation in the OOi RESTful API[9] in order to add a storage link between a volume and an instance of a VM. Once the detection rule for Compliant Link pattern was executed, the relationship ”Rest:hasPattern” illustrated with red color, was instantiated between the Ptt:Compliant_ Link (Ptt:Pattern instance) and the Rest:Link Volume to Instance (REST:Operation instance).

Listing 5: SWRL rules for Non-Compliant Link between Resources Anti-pattern

```
1.Rest:Operation(?op) ^Rest:type (?op, "linkResource
    ")^Rest:hasHttpMethod(?op, ?httpmd) ^Rest:verb(?
```

[9] Available at http://ooi.readthedocs.io/en/stable/user/usage.html

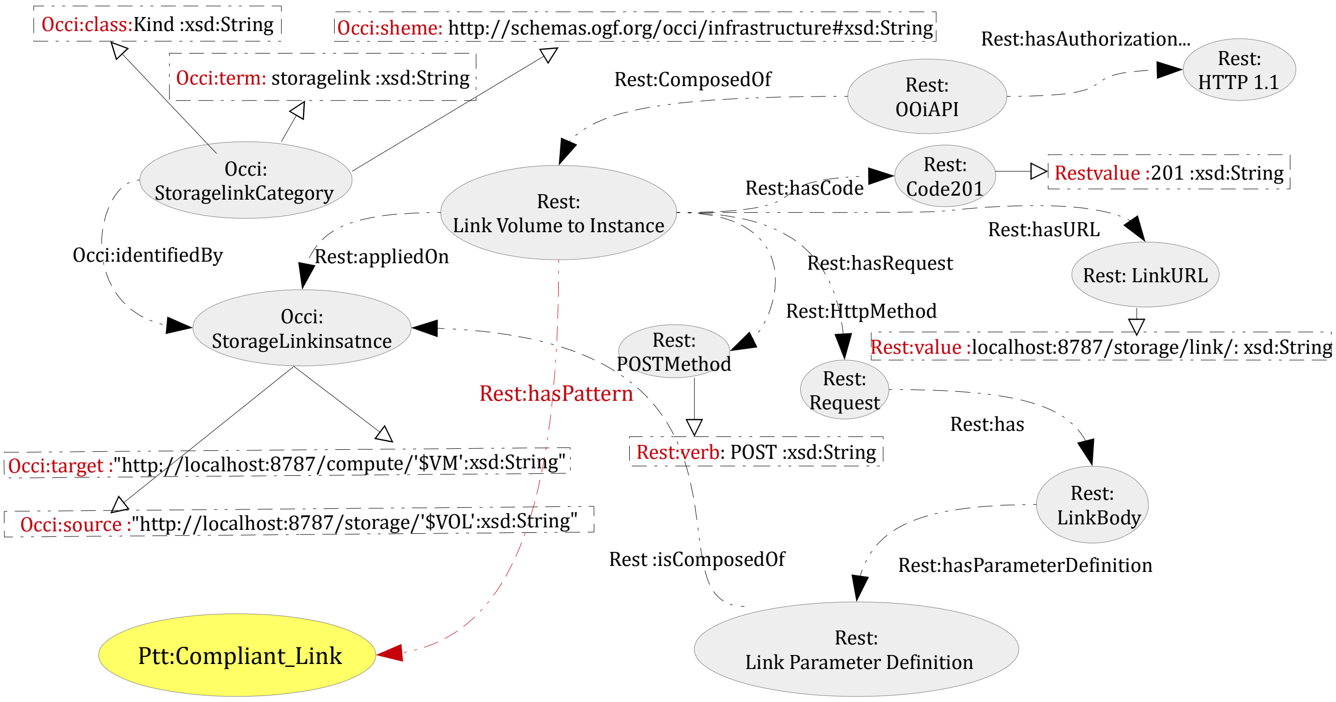
Occi:class:Kind :xsd:String
Occi:sheme: http://schemas.ogf.org/occi/infrastructure#xsd:String
Occi:term: storagelink :xsd:String
Rest:hasAuthorization...
Rest: HTTP 1.1
Rest:ComposedOf
Rest: OOiAPI
Occi: StoragelinkCategory
Rest: Link Volume to Instance
Rest:hasCode
Rest: Code201
Restvalue :201 :xsd:String
Rest:hasURL
Rest: LinkURL
Occi:identifiedBy
Rest:appliedOn
Rest:hasRequest
Rest:HttpMethod
Rest:value :localhost:8787/storage/link/: xsd:String
Occi: StorageLinkinsatnce
Rest: POSTMethod
Rest: Request
Rest:hasPattern
Rest:has
Occi:target :"http://localhost:8787/compute/'$VM':xsd:String"
Rest:verb: POST :xsd:String
Rest: LinkBody
Occi:source :"http://localhost:8787/storage/'$VOL':xsd:String"
Rest :isComposedOf
Rest:hasParameterDefinition
Ptt:Compliant_Link
Rest: Link Parameter Definition

Fig. 5: A partial instantiation of the (Anti) Pattern Ontology with information extracted from a REST operation in the OOi API, shows the impact after executing the SWRL rule for the *Compliant Link between Resources Pattern*

```
    httpmd, ?verb)^swrlb:notEqual(?verb, "POST") →
    Rest:hasAntiPattern(?op, Att:Non-Compliant_Link)

2. Rest:Operation(?op) ^Rest:type (?op, "
    linkResource")^ Rest:hasRequest(?op,?req)^Rest:
    has(?req,?reqbody) ^Rest:hasParameterDefinition
    (?reqbody, ?pradef) ^Occi:Link(?link) ^Rest:
    isComposedOf(?pradef, ?link) ^ Occi:identifiedBy
    (?link, ?kind) ^ Occi:term(?kind, "") → Rest:
    hasAntiPattern(?op, Att:Non-Compliant_ Link)

3.Rest:Operation(?op) ^ Rest:type (?op, "
    linkResource")^ Rest:hasRequest(?op,?req)^ Rest:
    has(?req,?reqbody) ^ Rest:hasParameterDefinition
    (?reqbody, ?pradef) ^ Occi:Link(?link) ^Rest:
    isComposedOf(?pradef, ?link) ^ Occi:source(?link
    , "") → Rest:hasAntiPattern(?op, Att:Non-
    Compliant_ Link)

4.Rest:Operation(?op) ^ Rest:type (?op, "
    linkResource") ^ Rest:hasRequest(?op,?req) ^
    Rest:has(?req, ?reqbody) ^ Rest:
    hasParameterDefinition(?reqbody, ?pradef) ^ Occi
    :Link(?link) ^ Rest:isComposedOf(?pradef, ?link)
     ^ Occi:target(?link, "") → Rest:hasAntiPattern
    (?op, Att:Non-Compliant_ Link)
```

Contrariwise, Non-Compliant Link between Resources Anti-pattern is reported if at least one of the above listed practices was not respected. As shown in Listing 5, we specify four SWRL rules defining the different symptoms needed to detect this anti-pattern.

4.3 Detection of OCCI/REST (Anti)Patterns

In this section, we aim at evaluating the compliance of the selected Cloud REST API with both REST and OCCI best principles by detecting both patterns and anti-patterns. Also, in case of any anti pattern detection, we intend to provide developers with a set of correction recommendations to help them revise and correct their APIs. As mentioned above, this step involves two phases: Check compliance for REST principles and Check compliance for OCCI principles. The first phase allows applying, on the Cloud REST API, the SWRL detection rules defined above to detect both REST patterns and anti-patterns, along with a set of SPARQL queries. These SPARQL queries are used to obtain the related details on each detected REST pattern and anti-pattern, as well as the suggested recommendation to avoid the detected anti-pattern. To do that, we firstly propose a REST pattern detection algorithm with the aim of performing the REST pattern detection.

Listing 6: The REST Pattern Detection Algorithm

```
Input:(Anti)patternOntology.owl,
    SWRL_Detection_Rules_List_For_RESTPatterns,
    SPARQL_queries_RESTpatterns_list
Output:REST Patterns_List with relevant Details
begin
  For (each SWRL_rule in
      SWRL_Detection_Rules_List_For_RESTPatterns) do
  {
    run(SWRL_rule)
  }
  For (each query in
      SPARQL_queries_RESTpatterns_list) do
  {
    result=execute(query)
    if( Exist(result) ) then
      Display("the pattern detected and related
        details")
    else                                          13
      Display ("no pattern is detected")          14
  }                                               15
end                                               16
```

As shown in Listing 6, the provided algorithm takes as input (Anti) Pattern Ontology, the list of SWRL rules to detect REST patterns and the list of SPARQL queries to return relevant details related to each REST pattern. As output, it returns the list of patterns with relevant details, which may help developers in the analysis or understanding. During the execution, the proposed algorithm proceeds as follows: apply the inference process that allows executing all SWRL rules using the Drools engine[10] to detect REST patterns (lines 4-7), execute all SPARQL queries defined in the list using the Pellet reasoner[11], which return for each pattern the relevant details once it is detected or ”no pattern detected” text otherwise (lines 8-15). All the SPARQL queries were defined in the same way, but they differ only in respect to the name of the pattern that would be interrogated. In Listing 7, we present a SPARQL query that returns the Verbless URIs pattern, each REST element that contains this pattern and the related contents.

Listing 7: SPARQL Query to retrieve the relevant details for Verbless URIs pattern

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax
    -ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX base: <http://www.semanticweb.org/asus/
    ontologies/2017/5/APatterns-ontology#>

SELECT distinct ?pattern ?restelement ?content
WHERE {?restelement base:Rest:content ?content. ?
    restelemnt base:Rest:hasPattern ?pattern. ?
    pattern base:Ptt:name ?name.
FILTER(?name="Ptt:Verbless_URIs"^^xsd:string)
}
```

Like the REST pattern detection, we propose a REST anti-pattern detection algorithm based on a set of SWRL rules and SPARQL queries. As shown in Listing 8, our algorithm takes as input (Anti)Pattern Ontology, the list of SWRL rules to detect REST anti-patterns and the list of SPARQL queries to return relevant details related to each REST anti-pattern along with suggested recommendations to avoid its occurrence. As output, it provides the detected anti-patterns associated with relevant details and suggested recommendations which can be further exploited by cloud API developers to correct these anti-patterns.

Similar to the execution process within the above algorithm, SWRL rules would be firstly executed to infer the existence of anti-patterns on the selected API (lines 4-7). Then, the SPARQL queries would be executed with the aim of providing the detected anti-patterns associated with relevant details and suggested recommendations that are useful for the correction purpose (lines 8-15). For example, Listing 9 shows the SPARQL query that we defined for the CRUDy URI anti-pattern. It returns each REST element that has this anti-pattern and its related contents as well as the following textual recommendation: ”*Method name or resource URL contains one of the following terms “read”, “create”, “delete”, “update”. Please change the given value into one from the following terms: “Post”, “Get”, “Delete” or “Put”*”. Thus, the cloud API developer may use these details and the suggested recommendations to revise her API. It is worth mentioning that the developer can take into consideration the detected anti-patterns and therefore the correction step can take place. Otherwise, the developer can ignore the detected anti-patterns because she thought that they are not relevant.

Listing 8: The REST Anti-pattern Detection Algorithm

```
Input:(Anti)patternOntology.owl,                         1
    SWRL_Detection_Rules_List_For_RESTAntiPatterns,
    SPARQL_queries_REST_Antipatterns_list
Output:REST AntiPatterns_List with relevant Details     2
begin                                                   3
  For (each SWRL_rule in SWRL_Detection-Rules-list      4
      for antipattern) do
  {                                                     5
    run(SWRL_rule)                                      6
  }                                                     7
  For (each query in SPARQL_query_antipattern_list)     8
      do
  {                                                     9
    result=execute(query)                               10
    if( Exist(result) ) then                            11
      Display("Detected anti-pattern with Relevant      12
          Details and Recommendation Text)
    else                                                13
      Display("no Anti-pattern is detected")            14
  }                                                     15
end                                                     16
```

Listing 9: SPARQL Query to retrieve the relevant details and suggested recommendations for CRYDy URI Anti-pattern

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax
    -ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX base: <http://www.semanticweb.org/asus/
    ontologies/2017/5/APatterns-ontology#>

SELECT distinct ?antipattern ?restelement ?content ?
    recommendation
WHERE {?restelement base:Rest:content ?content. ?
    restelemnt base:Rest:hasAntipattern ?antipattern
    . ?antipattern base:Att:
    hascorrectionrecommendation ?recommendation. ?
    antipattern base:Att:name ?name.
FILTER(?name="Att:CRYDy_URI"^^xsd:string)
}
```

After checking the compliance with REST principles via detecting both REST patterns and anti-patterns, checking the compliance with OCCI principles can take place. Like REST (anti) patterns, we conduct the detection of OCCI

[10] Drool: is a business rule management system that is based on forward and backward chaining inference to produce and execute rules. More details can be found at http://www.drools.org/

[11] https://www.w3.org/2001/sw/wiki/Pellet

(anti) patterns using two algorithms while relying on both SWRL rules and SPARQL queries.

5 Experiments and Validation

In this section, we discuss the evaluation of our proposed approach. We conduct this evaluation through a proof of concept implementation and a validation study on a dataset, in which we evaluated the accuracy (i.e. recall, precision, F-measure) of the proposed approach in detecting of OCCI (anti)patterns. In this article, we intend to provide an extensive validation of the proposed approach. Firstly, we assess the compliance of Cloud REST APIs with both OCCI and REST principles. Then, we show the effectiveness of our approach by analyzing its extensibility, the accuracy of the detection rules and the usefulness of the provided detection and recommendation support. In the following, we firstly present our proof of concept. Secondly, we describe the hypotheses and the experimental setup followed to conduct the validation of our approach. Finally, we analyze and interpret the experiment results.

5.1 Proof of Concept

To evaluate our approach, we built a proof-of-concept (POC) prototype[5] to support the detection of both OCCI and REST anti-patterns. Our POC implementation is provided as a web application based on J2EE solutions concretizing the (anti) patterns detection step described in Section 4.3. It provides to API developers two main functionalities, which are mainly: Check compliance for REST principles and Check compliance for OCCI principles. Moreover, we relied on three semantic Web APIs namely Pellet API[12], Apache Jana API[13] and SWRL API[14] to implement the different algorithms described above for the detection of (anti) patterns. These APIs are exploited to handle SPARQL/SQWRL queries and SWRL rules.

5.2 Hypotheses and Experimental Setup

Hypotheses. To evaluate the effectiveness of the proposed approach, we formulate the following hypotheses:

- *H1 (Accuracy).* The set of all defined rules have an average precision, recall and F-measure of more than 85%.
- *H2 (Usefulness).* The key detection rules and provided recommendations are useful and relevant.
- *H3 (Extensibility).* Our approach is extensible and allows adding new patterns and anti-patterns.

Experimental Setup. We perform an analysis in the Cloud RESTful APIs of cloud services to build the experimental dataset. As shown in Table 9, 5 candidates including OOi, COAPS, OpenNebula, Amazon S3 and Rackspace were selected. This selection has been done because their associated REST operations are well illustrated. Using the operations details existing in each API, we have collected all the requests and responses that build the required knowledge for semantically describing each API. Regarding the accuracy evaluation, we have to build our truth knowledge. Suitably, we manually evaluated the REST operations in order to identify the true positives and false negatives required to compute precision, recall and F1-measure values. Precision is the ratio between the true detected (anti) patterns and detected (anti) patterns reported as positive. Recall is the ratio between the true detected (anti) patterns and all existing true (anti) patterns. Finally, the F1-measure defines the weighted harmonic mean of the precision and recall values.

TABLE 9: List of the 5 analyzed Cloud APIs and their on-line documentations

Cloud RESTful APIs	On-line documentations
OOi RESTful API	http://ooi.readthedocs.io/en/stable/user/usage.html
COAPS RESTful API	http://www-inf.it-sudparis.eu/SIMBAD/tools/COAPS/
OpenNebula OCCI RESTful API	http://archives.opennebula.org/documentation:archives:rel4.0:occidd#overview
Amazon S3 RESTful API	http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
Rackspace RESTful API	https://developer.rackspace.com/docs/cloud-servers/v2/api-reference/

5.3 Results Analysis and Findings

In this section, we present respectively the detection results of REST (anti) patterns and OCCI (anti) patterns in all selected Cloud REST APIs. Then, a compliance evaluation of the selected API with OCCI and REST best principles is described. Finally, we discuss the validation of our approach in terms of accuracy, usefulness and extensibility.

5.3.1 Detection of REST of (anti) patterns

We present in Table 10 the detection results of 21 REST patterns and anti-patterns for the five Cloud RESTful APIs. The first column reports the patterns and anti-patterns, while the analyzed Cloud RESTful APIs are presented in the following columns. For each Cloud RESTful API, we show the occurrence number of each REST pattern and anti-pattern. The last three columns report respectively: the occurrence percentage (OP) of each (anti) pattern compared to the total number of operations that may contain such (anti) pattern (i.e. the percentage of Correct use of POST is computed compared only with the existing POST operations) and finally the total number of occurrences of each (anti) pattern compared to the total number of all existing operations.

As specified in Table 10, the most commonly followed REST pattern is Verbless URIs. It's quite evident that Cloud API designers are aware of avoiding the use of any of common CRUDy terms or their equivalent. This represents a good practice to obviate confusing API client developers. Additionally, more than 91% of the analyzed operations (191 out of 209) support compliant status codes. This enhances the understandability for clients APIs. Also, it is very reassuring to observe that more than 89% (188 out of 209) of the analyzed operations follows MIME type pattern, which increases the resource or service accessibility and reusability. In contrast, it is frustrating to observe that the most selected Cloud APIs except Amazon S3 failed to support the caching capability, although it is one of the principle REST constraints.

[12] https://github.com/stardog-union/pellet
[13] https://jena.apache.org/
[14] https://github.com/protegeproject/swrlapi

TABLE 10: Detection results of the REST (Anti) Patterns

Cloud REST API	**Open-Stack OCCI (28)**	**COAPS (18)**	**Open-Nebula OCCI (20)**	**Amazon S3 (71)**	**Rack-space (72)**	**O.P.**	**Total (209)**
URI Design (Anti) Patterns							
Tidy URIs (156/209)	6	13	17	70	50	74%	156 (74%)
Amorphous URIs (52/209)	21	5	3	1	22	52%	52 (24%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Verbless URIs (205/209)	28	16	20	69	72	98%	205 (98%)
CRUDy URIs (4/209)	0	2	0	2	0	1%	4 (1%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Singularized nodes (73/96)	6	3	6	33	25	76%	73 (34%)
Pluralized nodes (23/96)	8	3	3	6	4	23%	23 (11%)
No Detection (0/96)	0	0	0	0	0	0%	0 (0%)
Request methods (Anti) Patterns							
Correct use of POST (49/53)	8	7	3	4	27	92%	49 (23%)
Correct use of GET (74/74)	0	6	11	25	32	100%	74 (35%)
Correct use of PUT (29/29)	0	0	3	22	4	100%	29 (14%)
Correct use of DELETE (34/34)	6	3	3	13	9	34%	34 (16%)
Correct use of HEAD (5/5)	0	0	0	5	0	100%	5 (2%)
Tunneling every things through POST (4/53)	0	2	0	2	0	7%	4 (2%)
Tunneling every things through GET (0/47)	0	0	0	0	0	0%	0 (0%)
No Detection (0/199)	0	0	0	0	0	0%	0 (0%)
Error handling (Anti) Patterns							
Supporting Status Code (191/209)	28	18	20	70	55	91%	191 (91%)
Ignoring Status Code (18/209)	0	0	0	1	17	8%	18 (8%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
HTTP Header (Anti) Patterns							
Supporting Caching (71/209)	0	0	0	71	0	33%	71 (33%)
Ignoring Caching (138/209)	28	18	20	0	72	66%	138 (66%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Supporting MIME Types (188/209)	8	18	20	71	71	89%	188 (89%)
Ignoring MIME Types (21/209)	20	0	0	0	1	10%	21 (10%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Hypermedia (Anti) Patterns							
Supporting hypermedia (36/133)	8	13	0	0	15	27%	40 (17%)
Forgetting hypermedia (87/133)	0	0	14	31	42	63%	87 (41%)
No Detection (10/133)	10	0	0	0	0	7%	10 (4%)

5.3.2 Detection of OCCI (anti) patterns

Table 11 summarizes the detection results of 24 OCCI patterns and anti-patterns on the five Cloud RESTful APIs. As shown in Table 11, we observe that the most frequent patterns (*Compliant Delete* and *Compliant Update* patterns) dominate the management related (anti) patterns category. It seems that Cloud API designers follow either explicitly or implicitly the OCCI guidelines to delete and update a cloud resource. On the contrary, *Non-Compliant Trigger Action* and *Non-Compliant Create* represent the most recurrent anti-patterns in this category. In fact, most of the Cloud RESTful APIs do not specify the resource category needed to define a specific type of the resource to be created. In addition, it seems that the cloud API designers ignore both the query exposing the term of the action and its associated HTTP category defining its functionality in the REST operation to trigger an action on a resource. Regarding the Cloud Structure (Anti)Patterns category, *Compliant Link between Resources pattern* is the most commonly followed pattern. It should be noted that there is not a large number of operations we find to test this pattern as the link resource is rarely considered in the selected APIs. Moreover, we find no occurrence of (anti) patterns related to the association and dissociation of resources from Mixin. Regarding the REST anti-patterns and patterns according to OCCI perspective, 100% of the analyzed URIs are compliant with OCCI principles that are related to URI format. In addition, all analyzed request and response headers both in Amazon S3 and Rackspace are compliant with OCCI. In contrast, even though OOi, OpenNebula OCCI and COAPS are OCCI-based APIs, they failed to support compliant headers in any of their operations.

5.3.3 Compliance Evaluation

Herein, we aim at assessing whether the selected cloud APIs are compliant with REST and OCCI best principles by computing for each one its compliance degree. The compliance degree shows the percentage of patterns (OCCI or REST) that each API has over all its operations, which is defined as follows:

$$\textbf{Compliance degree}= \frac{1}{N} * \sum_{i=1}^{N} (\frac{\sum P_i}{\sum OP_P i})$$

where N is the number of patterns (e.g. 12 patterns for REST), P_i is a pattern (for instance P_1 denotes the Tidy URIs pattern), $\sum Pi$ the number of operations that really contain the pattern P_i (e.g. only 6 operations contain the Tidy URIs pattern in OOi RESTful API), $\sum OP_P i$ is the total number of operations that may contain the pattern P_i (e.g. 28 operations that may contain the Tidy URIs pattern in OOi RESTful API).

Now, we discuss the compliance of the selected Cloud REST APIs with REST good principles before assessing their compliances with the OCCI ones. As described in Fig. 6, we observe that all selected API have reached acceptable REST compliance degrees as mean value for all selected APIs is greater than 50% with reasonable difference. This shows the maturity of these APIs regarding the REST best principles. Indeed, Amazon S3 API represents the most compliant API with REST best principles with 78% of compliance degree, which means that 78% of its operations properly follow the REST best principles. This is not surprising as Amazon

TABLE 11: Detection results of the OCCI (Anti)patterns

Cloud REST API	OOi (28)	COAPS (18)	Open-Nebula OCCI (20)	Amazon S3 (71)	Rack-space (72)	O.P.	Total (209)
Management Related (Anti)patterns							
Query Interface Support (0/161)	0	0	0	0	0	0%	0 (0%)
Missing Query Interface (5/161)	1	1	1	1	1	3%	5 (2%)
No Detection (0/161)	0	0	0	0	0	0%	0 (0%)
Compliant Create (6/31)	4	1	0	1	0	19%	6 (2%)
Non-Compliant Create (24/31)	0	1	3	11	9	77%	24 (11%)
No Detection (1/31)	1	0	0	0	0	3%	1 (0.4%)
Compliant Update (15/24)	3	2	3	0	7	62%	15 (7%)
Non-Compliant Update (8/24)	0	0	0	8	0	8%	8 (3%)
No Detection (1/24)	0	0	0	1	0	4%	1 (0%)
Compliant Delete (32/32)	3	2	3	14	10	100%	32 (15%)
Non-Compliant Delete (0/32)	0	0	0	0	0	0%	0 (0%)
No Detection (0/32)	0	0	0	0	0	0%	0 (0%)
Compliant Retrieve (26/90)	10	5	11	0	0	28%	26 (12%)
Non-Compliant Retrieve (64/90)	0	0	0	31	33	71%	70 (30%)
No Detection (0/90)	0	0	0	0	0	0%	0 (0%)
Compliant Trigger Action (0/21)	0	0	0	0	0	0%	0 (0%)
Non-Compliant Trigger Action (21/21)	0	7	0	3	11	100%	21 (10%)
No Detection (0/21)	0	0	0	0	0	0%	0 (0%)
Cloud Structure (Anti)patterns							
Compliant Link between Resources (4/5)	2	0	2	0	0	80%	4 (1%)
Non-Compliant Link between Resources (1/5)	0	0	0	0	1	20%	1 (0.4%)
No Detection (0/5)	0	0	0	0	0	0%	0 (0%)
Compliant Association of Resource with Mixin	-	-	-	-	-	-	-
Non-Compliant Association of Resource with Mixin	-	-	-	-	-	-	-
No Detection	-	-	-	-	-	-	-
Compliant Dissociation of Resource from Mixin	-	-	-	-	-	-	-
Non-Compliant Dissociation of Resource from Mixin	-	-	-	-	-	-	-
No Detection	-	-	-	-	-	-	-
REST Related (Anti) Patterns							
Compliant URL (209/209)	28	18	20	71	72	100%	209 (100%)
Non-Compliant URL (0/209)	0	0	0	0	0	0%	0 (0%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Compliant Request Header (0/209)	0	0	0	71	72	68%	143 (68%)
Non-Compliant Request Header (209/209)	28	18	20	0	0	66%	66 (31%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)
Compliant Response Header (143/209)	0	0	0	71	72	68%	143 (68%)
Non-Compliant Response Header (66/209)	28	18	20	0	0	100%	66 (31%)
No Detection (0/209)	0	0	0	0	0	0%	0 (0%)

S3 is one of important cloud leaders aiming to attract API developers to increase the use of their API. Moreover, we report that OCCI OpenNebula API reaches 70%, Rackspace reaches 69%, OOi reaches 66% and finally COAPS reach 63% as compliance degree, which are also good values.

Regarding the OCCI best principles, as illustrated in Fig. 7, OOi and OpenNebula OCCI APIs represent the most compliant APIs with OCCI best principles. This is not surprising because both APIs are already based on OCCI standard. However, the reached compliance degree is still not convenient enough. It would be more reassuring if future releases of OCCI plan to improve this as the OCCI REST API will be automatically generated from a meta-model instead of designed by hand. Additionally, Rackspace as well as Amazon S3 have reached 48% and 42% of compliance degree respectively. This means that over 40% of operations in those APIs implicitly follow the OCCI standard, although they have based on own model to describe cloud resources. In contrast, even though COAPS API is an OCCI-based API, it reaches only 45%, which shows that its developers did not carefully follow all OCCI best principles.

5.3.4 Discussion of validation results

In this section, we aim at discussing the validation hypotheses mentioned in Section 5.2.

Evaluation of H1 (Accuracy). The hypotheses H1 is evaluated according to three parameters: Precision, Recall and F-measure. Two validations were conducted to evaluate this hypothesis. Table 12 illustrates the results of the first validation that aim at evaluating our approach validation in detecting REST patterns and anti-patterns on OOi and Rackspace RESTful APIs. The first column presents the identified (anti)patterns. The remaining columns list the two selected APIs for the validation, including Validated (i.e., the number of validated pattern (or anti-pattern) considered as true which is ensured manually), P (i.e., the number of pattern occurrences reported as positives by our detection algorithms), TP (i.e., the number of pattern occurrences reported as true positives), Precision, Average Precision, Recall and Average Recall. Finally, we report the total average of Precision, Recall and F-measure on the last two rows respectively. Regarding OOi RESTful API, it was pleasantly surprising that our REST detection algorithms allow detecting of REST patterns and anti-patterns on average with a precision of 100% and Recall of 95,1 %, signifying that all the detected (anti) patterns are in the list that we determined

TABLE 12: Complete validation results of REST patterns and anti-patterns on OOi and Rackspace REST APIs

	OOi							Rackspace						
(anti)Patterns	Validated	P	TP	Precision	Avg.Precision	Recall	Avg.Recall	Validated	P	TP	Precision	Avg.Precision	Recall	Avg.Recall
Tidy URIs	6	5	5	100%	100%	83%	91.7%	20	19	19	100%	100%	95%	97.5%
Amorphous URIs	8	8	8	100%		100%		20	20	20	100%		100%	
No Detection	0	0	0	-		-		0	0	0	-		-	
Verbless URIs	10	10	10	100%	100%	100%	100%	20	20	20	100%	100%	100%	100%
CRUDy URIs	0	0	0	-		-		0	0	0	-		-	
No Detection	0	0	0	-		-		0	0	0	-			
Singularized nodes	6	5	5	100%	100%	83%	91.7%	20	18	18	100%	100%	90%	70%
Pluralized nodes	8	8	8	100%		100%		4	2	2	100%		50%	
No Detection	0	0	0	-		-		0	0	0	-		-	
Correct use of POST	8	6	6	100%	100%	75%	87.5%	10	10	10	100%	100%	100%	97.2%
Correct use of GET	0	0	0	-		-		10	10	10	100%		100%	
Correct use of PUT	0	0	-	-		-		4	4	4	100%		100%	
Correct use of DELETE	6	6	6	100%		100%		9	9	8	100%		0.88%	
Correct use of HEAD	0	0	0	-		-		0	0	0	-		-	
Tunneling every things through POST	0	0	0	-		-		0	0	0	-		-	
Tunneling every things through GET	0	0	0	-		-		0	0	0	-		-	
No Detection	0	0	0	-		-		0	0	0	-			
Supporting Status Code	10	10	10	100%	100%	100%	100%	20	20	20	100%	100%	100%	100%
Ignoring Status Code	0	0	0	-		-		10	10	10	100%		100%	
No Detection	0	0	0	-		-		0	0	0	-			
Supporting Caching Code	0	0	0	-	100%	-	100%	0	0	0	-	100%	-	100%
Ignoring Caching	10	10	10	100%		100%		10	10	100%				
No Detection	0	0	0	-		-		0	0	0	-			
Supporting MIME Types	8	8	8	100%	100%	100%	100%	10	9	9	100%	100%	90%	90%
Ignoring MIME Types	10	10	10	100%		100%		0	0	0	-		-	
No Detection	0	0	0	-		-		0	0	0	-			
Supporting hypermedia	5	4	4	100%	100%	80%	90%	10	7	7	100%	100%	70%	75%
Forgetting hypermedia	0	0	0	-		-		10	8	8	100%		80%	
No Detection	4	4	4	100%		100%		0	0	0	-		-	
Average	**Precision**				100%	**Recall**	95,1%	**Precision**				100%	**Recall**	91.2%
	F-measure				97.4%			**F-measure**				95.3%		

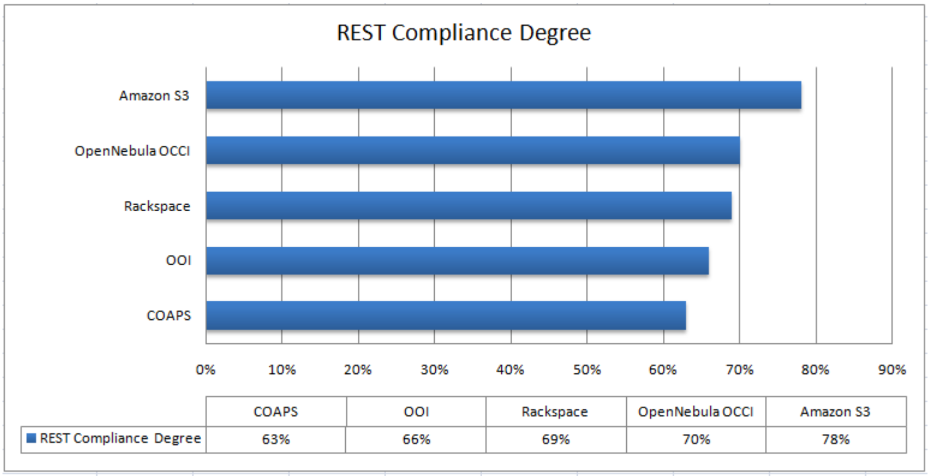

Fig. 6: REST Compliance Degrees of Cloud RESTful APIs

manually. Also, we obtain on average almost similar values for Rackspace RESTful API viz. a precision of 100 % and a recall of 91,2. On the whole, we obtain on average F-measure of 97,4 % for OOi and 95,3% for Rackspace.

Table 13 illustrates the results of the second validation of our approach in detecting OCCI (anti) patterns on OOi and Rackspace RESTful APIs. Similarly, as for the REST (anti) pattern detection, our approach has reached good results. More precisely, we obtain, on average, precision of 100%, recall of 97,9%, F-measure of 98,9% for OOi API and, precision of 100%, recall of 97%, F-measure of 98% for Rackspace. Accordingly, giving this observation and the above one, we confirm the truth of the first hypothesis H1 with significant difference.

Evaluation of H2 (Usefulness). The usefulness is determined via a questionnaire[15] that provides the participants feedbacks about our detection rules and suggested recommendations in case of any anti-pattern detection. Participants were recruited from software engineering experts who have sophisticated understanding of REST and Cloud

[15] https://docs.google.com/forms/d/e/1FAIpQLScTPsXTy2PrFpgrzgn8Iq2WY1zXUk5UEkDNngkB95Q4Jjq2tw/viewform

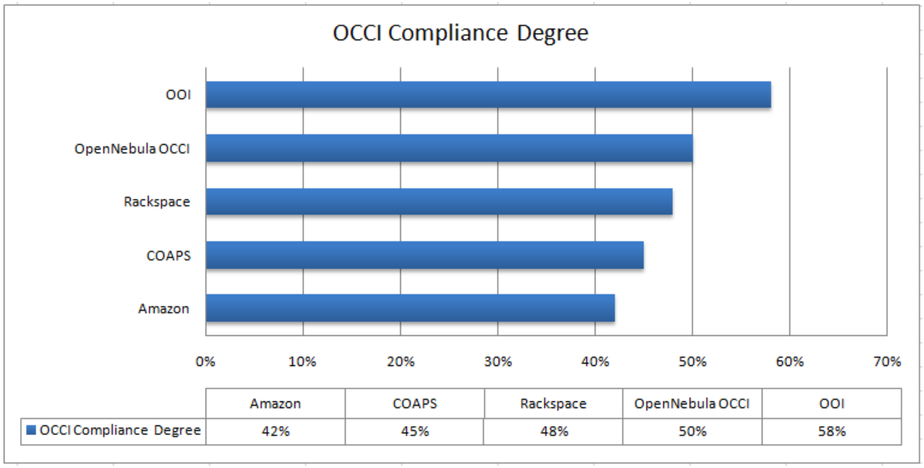

Fig. 7: OCCI Compliance Degrees of Cloud RESTful APIs

APIs. The questionnaire consists of two main parts: Usefulness evaluation and Insights/Improvements. The usefulness evaluation questions aim at evaluating whether the detection rules and suggested recommendations are useful and relevant. Whereas, the Insights/Improvements provide some suggestions for improving our detection and recommendation support. Furthermore, to perform this experiment, we ask participants to rate the usefulness of the detection of a set of OCCI/REST patterns and anti-patterns using a 0-5 scale. We examined 2 REST patterns (Correct use of POST, Tidy URLs), 2 REST anti-patterns (Amorphous URIs, Forgetting Hypermedia), 2 OCCI patterns (Compliant Delete, Compliant URL), and 2 OCCI anti-pattern (Non-Compliant Create, Non-Compliant Trigger Action). Moreover, to evaluate the detection of these (anti) patterns, we employ a set of REST operations from three different management APIs mainly, Rackspace, COAPS and OOI.

As shown in Fig. 8 (a) (b), regarding the detection of selected REST patterns/anti-patterns (i.e., Correct use of POST and Tidy URLs patterns, Amorphous URIs and Forgetting Hypermedia anti-patterns), we observed that the mean usefulness rate is greater than 4, which represents a good and acceptable rate. Similarly, the most of participants have positively rated the detection usefulness of the selected OCCI patterns and anti-patterns (i.e., Compliant Delete and Compliant URL patterns, Non-Compliant Create and Non-Compliant Trigger Action anti-patterns) as the observed score is greater than 3.9 (refer to Fig. 8 ((c) (d)). Overall, participants reported that the provided detection and recommendation support are considerably useful. Given this observation, we confirm that the key detection rules and the provided recommendations are useful and relevant. Moreover, as feedbacks for improvement, most participants highlight the need of sophisticated way to visualize both the detected patterns and anti-patterns. Regarding the detected patterns, it has been suggested to provide a short string explaining what the normal patterns. While regarding anti-patterns, it is better to provide instead of a just an explanation a link that present this explanation in more details and to represent recommendations using markers (i.e., colors, formatting, etc.). Finally, one of the participants suggests to add more practices and a guide/description with examples of each pattern/anti-pattern. We will take carefully those points in the future, which would increase the understandability and applicability of our patterns and anti-patterns.

Evaluation of H3 (Extensibility). In our previous work [9], our approach allows defining 24 OCCI patterns and anti-patterns. Currently, we added to this approach 21 new REST patterns and anti-patterns. As illustrated in Table 12, our approach shows its potentiality, in detecting these new (anti) patterns with good results. Indeed, adding new patterns and anti-patterns is also possible and even in a simple way. It only needs to define the required semantic rules and integrate them within the already established knowledge base. However, dealing with semantic rules can seem not a straightforward and easy task to be done by non-expert users. Therefore, providing them with graphic interface allowing them to describe constraints for their (anti) patterns and our system generates accordingly the semantic rules for detecting them, need to be considered in the future. Furthermore, except COAPS and OpenNebula OCCI which they are still as in previous work [9], we validated our approach with new updated APIs including new REST operations. Giving these observations, it is possible to add new cloud REST APIs and patterns and antipatterns. Therefore, we confirm the validity of H3.

6 Related Work

Over the last years, patterns and anti-patterns have been widely adopted by various researches with the aim of expressing architectural solutions and concerns in Service Oriented Architectures (SOAs), Object Oriented Systems and lately in RESTful APIs. Suitably, various techniques have followed to specify and detect these patterns, including genetic programming [27] ; [28], domain specific language (DSL) [29] ; [30] ; [24], temporal logic [31], Bayesian networks [32] , Natural language processing (NLP) [33] , semantic web ontologies , DL logic and SPARQL queries [34]; [35] , to name a few.

In the context of Object Oriented Systems, Kessentini et al. [27] presented a detection approach with the aim of detecting automatically different kind of design defects that can

TABLE 13: Complete validation results of OCCI patterns and anti-patterns on OOi and Rackspace REST APIs

	OOi							Rackspace						
(anti)Patterns	Validated	P	TP	Precision	Avg.Precision	Recall	Avg.Recall	Validated	P	TP	Precision	Avg.Precision	Recall	Avg.Recall
Query interface support	0	0	0	-	100%	-	100%	0	0	0	-	100%	-	100%
Missing query interface	1	1	1	100%		100%		1	1	1	100%		100%	
No Detection	0	0	0	-		-		0	0	0	-		-	
Compliant Create	4	3	3	100%	100%	75%	75%	0	0	0	-	100%	-	100%
Non-Compliant Create	0	0	0	-		-		9	9	9	100%		100%	
No Detection	0	0	0	-		-		10	10	10	100%		100%	
Compliant Update	3	3	3	100%	100%	100%	100%	7	7	7	100%	100%	100%	100%
Non-Compliant Update	0	0	0	-		-		0	0	0	-		-	
No Detection	0	0	0	-		-		6	6	6	100%		100%	
Compliant Delete	3	3	3	100%	100%	100%	100%	10	10	10	100%	100%	100%	100%
Non-Compliant Delete	0	0	0	-		-		0	0	0	-		-	
No Detection	0	0	0	-		-		0	0	0	-			
Compliant Retrieve	10	10	10	100%	100%	100%	100%	0	0	0	-	100%	-	90%
Non-Compliant Retrieve	10	0	0	-		-		10	9	9	100%		90%	
No Detection	0	0	0	-		-		0	0	0	-			
Compliant Trigger Action	0	0	0	-	100%	-	100%	0	0	0	-	100%	-	80%
Non-Compliant Trigger Action	0	0	0	-		-		10	8	8	100%		80%	
No Detection	8	8	8	100%		100%		0	0	0	-			
Compliant Link between Resources	2	2	2	100%	100%	100%	100%	0	0	0	-%	100%	-	100%
Non-Compliant Link between Resources	0	0	0	-		-		1	1	1	100%		100%	
No Detection	0	0	0	-		-		5	5	5	100%		100%	
Compliant Association of resource with Mixin	0	0	0	-	100%	-%	100%	-	-	-	-	-	-	-
Non-Compliant Association of resource with Mixin	0	0	0	-		-		-	-	-	-		-	
No Detection	5	5	5	100%		100%		-	-	-	-			
Compliant Dissociation of resource from Mixin	0	0	0	-	100%	-	100%	-	-	-	-	-	-	-
Non-Compliant Dissociation of resource from Mixin	0	0	0	-		-		-	-	-	-		-	
No Detection	6	6	6	100%		100%		-	-	-	-			
Compliant URL	10	10	10	100%	100%	100%	100%	10	10	10	100%	100%	100%	100%
Non-Compliant URL	0	0	0	-		-		0	0	0	-		-	
No Detection	0	0	0	-		-		0	0	0	-			
Compliant Request Header	0	0	0	-	100%	-	100%	10	10	10	100%	100%	100%	100%
Non-Compliant Compliant Request Header	10	10	10	100%		100%		0	0	0	-		-	
No Detection	0	0	0	-		-		0	0	0	-			
Compliant Response Header	0	0	0	-	100%	-	100%	10	10	10	100%	100%	100%	100%
Non-Compliant Compliant Response Header	10	10	10	100%		100%		0	0	0	-		-	
No Detection	0	0	0	-		-		0	0	0	-			
Average	**Precision**				100%	**Recall**	97,9%	**Precision**				100%	**Recall**	97%
	F-measure				98.9%			**F-measure**				98%		

occur in the source code. Genetic programming is adopted with the aim of optimally identifying a set of rules that maximize the smell detection. Another approach proposed by Fourati et al. [36], aims at identifying anti-patterns in UML design while using existing and new defined quality metrics. It adopted rule-based approach over these metric to detect UML design anti-patterns. The authors, in this work, focus on the structural and behavioral information under sequence and class diagrams. Another important effort has been made by Settas [34]. It proposed a knowledge system based on OWL ontology called SPARSE which provides a detection support for software project managers to detect anti-patterns. In our work, we followed a similar approach but in the context of the cloud computing for detecting REST and OCCI (anti) patterns.

Others related work accentuate the need for detecting patterns and anti-patterns related to SOA. In [37], Dudney et al. identified a catalog of 53 anti-patterns that include design, architecture and implementation of J2EE-based systems. In [30], Moha et al. have illustrated the absence of techniques and methods in order to detect SOA anti-patterns of service-based systems (SBSs). They based on DSL specified using BNF grammars. The proposed approach allows the detection of SOA anti-patterns, which may assist the software engineer in evaluating the QoS and design. This approach is also extended to support the detection of SOA anti-patterns in Web Services [24]. However, Both OO and SOA detection methods are not tailored to deal with Cloud RESTful APIs as OO focuses only on classes and SOA focuses on services and WSDL descriptions.

To the best of our knowledge, cloud REST API design and evaluation are studied by few works. In [7], Rodríguez et al. studied the good and bad practices on mobile application and evaluated their conformity. A huge number of data logs were collected from Internet traffic of mobile applications with the aim of analyzing these logs and identifying the involved design patterns. A comparison between the identified patterns and good design practices was conducted to evaluate the conformance level of these patterns. Zhou et al. [8] proposed hypertext-driven framework in order to design a REST northbound API in a Software Defined Network using REST API design patterns. Instead of directly relying on

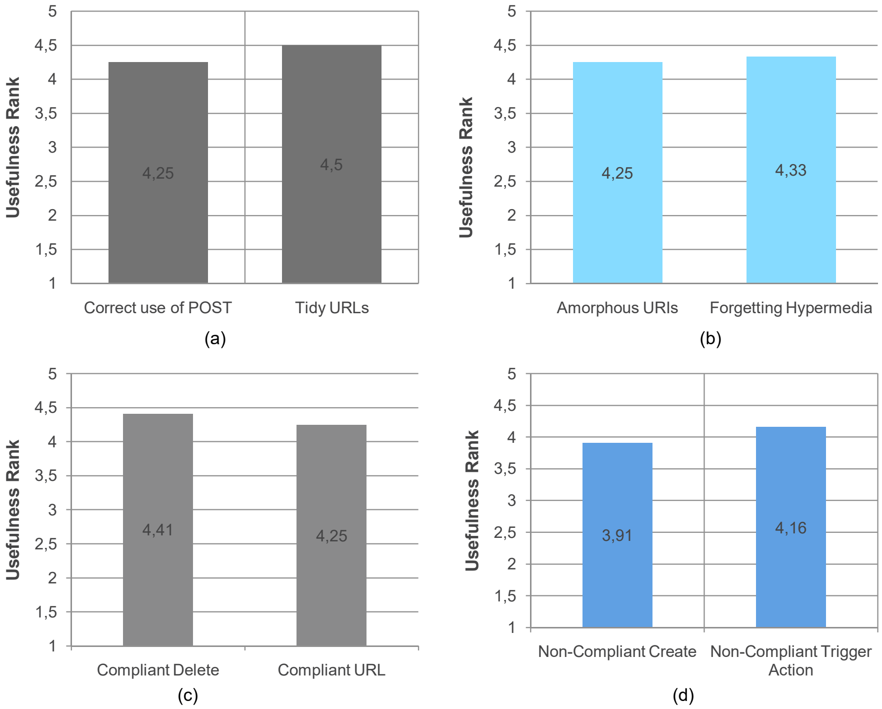

Fig. 8: Usefulness Detection Rate for (a) REST patterns: Correct use of POST, Tidy URLs; (b) REST anti-patterns: Amorphous URIs, Forgetting Hypermedia; (c) OCCI patterns: Compliant Delete, Compliant URL; (d) OCCI anti-patterns: Non-Compliant Create, Non-Compliant Trigger Action

fixed resource URIs, hypertext links are used between REST resources with the aim of assisting clients to determine the exact resources. However, these previous works, consider two specific domains which are mobile and networking in the REST APIs design evaluation. In contrast, in our work, we mainly focus on the cloud services domain.

In [38], Maleshkova et al. provided a deep and comprehensive analysis of the current state of Web APIs. More precisely, more than 220 available Web APIs that include REST, RPC and hybrid were analysed. In this analysis, the authors investigated six characteristics of Web APIs, including specifically, their types, output formats, input parameter, general information, invocation details and finally their complementary documentations. According to the authors, the analysed Web API are not naturally REST and sustained from under-specification as they omit most of the indispensable information like HTTP methods and datatype. Therefore, this illustrates the need to study REST APIs design in cloud services, which support the contribution of our article.

In our previous works [5], [6], we investigated the design of various REST APIs by leveraging the notion of REST patterns and anti-patterns to drive respectively the good and bad practices that may involve in the REST services design. However, the analysed APIs were chosen only from general domains, including Bestbuy, Dropbox, Twitter and Facebook and not from cloud services domain. Moreover, in [9], we assessed the compliance of Cloud REST APIs with OCCI standard by exploiting both patterns and anti-patterns. However, through a research study that we have done in [2], we demonstrated that OCCI fails to support some of the best principles related to the REST aspects in the designing of Cloud REST APIs. Therefore, this motivated us to propose this contribution, which is the first one that combines general REST principles with ones proposed by OCCI which is one of the most important standards in the cloud era. Drawing analogies from [34], we are inspired to likewise support the definition and detection of pattern and anti-patterns using semantic solutions, in particular ontologies. We adopted this choice as we need a formalism to deal with structure and semantic relations over cloud resources and their associated parameters [39]. In addition, an ontology-based model is enriched with a reasoning process that is able to draw new and hidden knowledge from the existing information. These knowledge could help us in the automatic evaluation of each element defined in REST API and the detection of patterns and anti-patterns that may occur in it.

7 Conclusion

The growth of the cloud computing has motivated many projects to develop open standards for facilitating and increasing its adoption. Open Cloud Computing Interface (OCCI) is the only open standard that provides a set of good design principles to develop and design interoperable Cloud Restful APIs for any kind of management tasks over cloud resources. However, it failed to support a range of good principles related to REST aspects decreasing the understandability and reusability of these APIs [2]. In this article, we argued that both OCCI and REST best principles should be supported together in the design of Cloud REST

APIs. Thereby, this can contribute in ensuring the interoperability and facilitate the discovery of cloud resources on one hand, and increasing their reusability and understandability on other hand.

Furthermore, we leveraged patterns and anti-patterns to drive respectively the good and poor practices of both OCCI and REST best principles that API developers or cloud providers should be taken carefully on when designing their APIs. Furthermore, semantic models, in particular ontologies, have been adopted as an appropriate means relying on SWRL, SQWRL and SPARQL languages to specify and to detect both patterns and anti-patterns and to provide correction recommendations in case of any anti-pattern detection. We proposed a semantic definition of 21 common REST (anti) patterns and 24 OCCI (anti) patterns for Cloud RESTful APIs and four detection algorithms acting on these specifications to detect OCCI (anti)patterns and REST (anti)patterns respectively.

We validated our approach by analyzing both OCCI and REST (anti) patterns on real world Cloud RESTful APIs that invoking over 200 operations, and assessing its feasibility in term of accuracy, usefulness and extensibility. The observed accuracy shows that our approach is an effective technique for detecting OCCI and REST (anti) patterns in Cloud RESTful APIs, which may assist cloud API developers and providers when assessing the quality of their APIs and correcting them to avoid the anti-patterns occurrences. In addition, through the usefulness evaluation, we proved that the key detection rules and provided recommendations are useful and relevant. Moreover, we showed that our approach is extensible enough to define other new (anti) patterns when they are required for the Cloud RESTful API. In addition, it enable the integration of new cloud RESTful APIs. Besides, through the compliance analysis, regarding REST best principles, we observed that the most of the analyzed Cloud RESTful APIs have reached an acceptable level of maturity by considering the most of good REST principles. In opposition, we observed also through the obtained OCCI compliance degrees, that there is no correct and adequate adoption of the OCCI best principles in the selected Cloud RESTful APIs. Hopefully, this inspires the developers of these APIs and other practitioners to include REST and OCCI good principles as much as possible. Thus, we will contribute to the improvement of OCCI specifications as supporting of REST best principles in OCCI-based APIs would make them more visible and easy to understand.

Future work includes studying whether these best principles apply universally to all cloud APIs. In particular, we intend to apply our approach on other Cloud RESTful APIs for better understanding the OCCI and REST principles and their applicability, and therefore revealing other possible characteristics. Also, we intend to extend our previous approach CloudLex [40] for extracting the required data describing the Cloud RESTful APIs to populate our semantic knowledge base needed for the detection of patterns and anti-patterns, hence avoiding the manual work that can be involved towards performing an automatic instantiation. Thus, it would be possible to make our detection tool available for to be used by cloud API developers and providers. Finally, the analysis of OCCI compliance across cloud providers inspired us to propose a semantic model (i.e. ontology) while taking OCCI as reference to build semantic joins between cloud providers' APIs. This model will act as proxy to resolve heterogeneity, interoperability and evolution problems of the manageable resources among the different cloud providers.

Acknowledgments

This work is partially funded by the French PIA OCCIware research and development project (www.occiware.org).

References

[1] Andy Edmonds, Thijs Metsch, Alexander Papaspyrou, and Alexis Richardson. Toward an open cloud standard. *IEEE Internet Computing*, 16(4):15–25, 2012.

[2] Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study. In *Proceedings of 14th International Conference on Service-Oriented Computing, ICSOC'16*, pages 171–187. Springer International Publishing, 2016.

[3] Ralf Nyrén, Andy Edmonds, Thijs Metsch, and Boris Parák. Open Cloud Computing Interface – HTTP Protocol. Recommandation GFD-R-P.223, Open Grid Forum, October 2016.

[4] *OCCI Compliance Testing Tool*, 2011. http://occi-wg.org/2011/01/18/occi-compliance-testing-tool/.

[5] Francis Palma, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc. Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. In *Proceedings of 12th International Conference on Service-Oriented Computing, ICSOC'14*, pages 230–244, 2014.

[6] Francis Palma, Javier Gonzalez-Huerta, Naouel Moha, Yann-Gaël Guéhéneuc, and Tremblay Guy. Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns. In *Proceedings of 13th International Conference on Service-Oriented Computing, ICSOC'15*, pages 171–187, 2015.

[7] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos, Luigi Canali, and Gianraffaele Percannella. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In *Proceedings of 16th International Conference on Web Engineering (ICWE2016)*, pages 21–39, Lugano, 2016.

[8] Wei Zhou, Li Li, Min Luo, and Wu Chou. REST API Design Patterns for SDN Northbound API. In *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, pages 358–365. IEEE, may 2014.

[9] Hayet Brabra, Achraf Mtibaa, Layth Sliman, Walid Gaaloul, Boualem Benatallah, and Faiez Gargouri. Detecting Cloud (Anti)Patterns: OCCI Perspective. In *Proceedings of 14th International Conference on Service-Oriented Computing, ICSOC'16*, pages 202–218. Springer International Publishing, 2016.

[10] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] Andy Edmonds, Thijs Metsch, Alexander Papaspyrou, and Alexis Richardson. Toward an Open Cloud Standard. *IEEE Internet Computing*, 16(4):15–25, 2012.

[12] Philippe Merle, Christophe Gourdin, and Nathalie Mitton. Mobile Cloud Robotics as a Service with OCCIware. In *Proceedings of the 2nd IEEE International Congress on Internet of Things, IEEE ICIOT 2017*, Honolulu, Hawaii, United States, June 2017. To appear. Best Paper Award.

[13] Ralf Nyrén, Andy Edmonds, Alexander Papaspyrou, Thijs Metsch, and Boris Parák. Open Cloud Computing Interface – Core. Recommendation GFD-R-P.221, Open Grid Forum, October 2016.

[14] Philippe Merle, Olivier Barais, Jean Parpaillon, Noël Plouzeau, and Samir Tata. A Precise Metamodel for Open Cloud Computing Interface. In *Proceedings of 2015 IEEE 8th International Conference on Cloud Computing, CLOUD'15*, pages 852–859, June 2015.

[15] Andy Edmonds and Thijs Metsch. Open Cloud Computing Interface – Text Rendering. Recommandation GFD-R-P.229, Open Grid Forum, October 2016.

[16] Thijs Metsch, Andy Edmonds, and Boris Parák. Open Cloud Computing Interface – Infrastructure. Recommandation GFD-R-P.224, Open Grid Forum, October 2016.

[17] Michel Drescher, Boris Parák, and David Wallom. OCCI Compute Resource Templates Profile. Recommandation GFD-R-P.222, Open Grid Forum, October 2016.

[18] Thijs Metsch and Mohamed Mohamed. Open Cloud Computing Interface – Platform. Recommandation GFD-R-P.227, Open Grid Forum, October 2016.

[19] Gregory Katsaros. Open Cloud Computing Interface – Service Level Agreements. Recommandation GFD-R-P.228, Open Grid Forum, October 2016.

[20] Mark Massé. REST API Design Rulebook. Technical report, O'Reilly Media, Sebastopol, 2011.

[21] S. Vinoski. RESTful Web Services Development Checklist. *IEEE Internet Computing*, 12(6):96–95, 2008.

[22] Michael Stowe. Undisturbed REST: A Guide to Designing the Perfect API. Technical report, MuleSoft, San Francisco, USA, 2015.

[23] Leonard Richardson and Sam Ruby. RESTful Web Services. Technical report, O'Reilly Media Inc., Sebastopol, 2007.

[24] Francis Palma, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc. Specification and Detection of SOA Antipatterns in Web Services. In *Proceedings of 8th European Conference on Software Architecture, ECSA'14*, pages 58–73, 2014.

[25] *OWL 2 Web Ontology Language Document Overview (Second Edition)*, 2012. https://www.w3.org/TR/owl2-overview/.

[26] Ralf Nyrén, Florian Feldhaus, Boris Parák, and Zdenek Sustr. Open Cloud Computing Interface – JSON Rendering. Recommandation GFD-R-P.226, Open Grid Forum, October 2016.

[27] Marouane Kessentini, Stephane Vaucher, and Houari Sahraoui. Deviance from Perfection is a Better Criterion Than Closeness to Evil when Identifying Risky Code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE'10, pages 113–122, 2010.

[28] Sunny Wong, Melissa Aaron, Jeffrey Segall, Kevin Lynch, and Spiros Mancoridis. Reverse engineering utility functions using genetic programming to detect anomalous behavior in software. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 141–149, 2010.

[29] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[30] N. Moha and al. *Service-Oriented Computing: 10th International Conference, ICSOC'12*, chapter Specification and Detection of SOA Antipatterns, pages 1–16. Springer Berlin Heidelberg, 2012.

[31] Nikola Trcka, Wil M. P. van der Aalst, and Natalia Sidorova. Data-flow anti-patterns: Discovering data-flow errors in workflows. In *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings*, pages 425–439, 2009.

[32] D. Settas, S. Bibi, P. Sfetsos, I. Stamelos, and V. Gerogiannis. Using bayesian belief networks to model software project management antipatterns. In *Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*, pages 117–124, Aug 2006.

[33] Francis Palma, Javier Gonzalez-Huerta, Mohamed Founi, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc. Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns. *Int. J. Cooperative Inf. Syst.*, 26(2):1–37, 2017.

[34] Dimitrios L. Settas, Georgios Meditskos, Ioannis G. Stamelos, and Nick Bassiliades. SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies. *Expert Systems with Applications*, 38(6):7633 – 7646, 2011.

[35] Molka Rekik, Khouloud Boukadi, Walid Gaaloul, and Hanêne Ben-Abdallah. Anti-pattern specification and correction recommendations for semantic cloud services. In *50th Hawaii International Conference on System Sciences, HICSS 2017, Hilton Waikoloa Village, Hawaii, USA, January 4-7, 2017*, 2017.

[36] Rahma Fourati, Nadia Bouassida, and Hanene Ben Abdallah. *A Metric-Based Approach for Anti-pattern Detection in UML Designs*, pages 17–33. Springer Berlin Heidelberg, 2011.

[37] Bill Dudney, Stephen Asbury, Joseph K. Krozak, and Kevin Wittkopf. *J2EE Anti-Patterns*. John Wiley Sons Inc, 2003.

[38] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. In *2010 Eighth IEEE European Conference on Web Services*, pages 107–114. IEEE, dec 2010.

[39] Hayet Brabra, Achraf Mtibaa, Layth Sliman, Walid Gaaloul, and Faïez Gargouri. Semantic Web Technologies in Cloud Computing: A Systematic Literature Review. In *Proceedings of IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 744–751, 2016.

[40] Fábio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. Towards a REST Cloud Computing Lexicon. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science, CLOSER 2017*, pages 348–355, Porto, Portugal, 2017.