Anti-Pattern Specification and Correction Recommendations for Semantic Cloud Services

Molka Rekik[1], Khoulou Boukadi[1], Walid Gaaloul[2], Hanêne Ben-Abdallah[3]

[1]Mir@cl Laboratory, Sfax University, Tunisia
[2]Telecom SudParis, Samovar, France
[3]Mir@cl Laboratory, King Abdulaziz University, KSA
[1]{molka.rekik, khouloud.boukadi}@gmail.com
[2]walid.gaaloul@mines-telecom.fr, [3]hbenabdallah@kau.edu.sa

Abstract

The lack of standardized descriptions of cloud services hinders their discovery. In an effort to standardize cloud service descriptions, several works propose to use ontologies. Nevertheless, the adoption of any of the proposed ontologies calls for an evaluation to show its efficiency in cloud service discovery. Indeed, the existing cloud providers describe, their similar offered services in different ways. Thus, various existing works aim at standardizing the representation of cloud computing services by proposing ontologies. Since the existing proposals were not evaluated, they might be less adopted and considered. Indeed, the ontology evaluation has a direct impact on its understandability and reusability. In this paper, we propose an evaluation approach to validate our proposed Cloud Service Ontology (CSO), to guarantee an adequate cloud service discovery. To this end, this paper has a three-fold contribution. First, we specify a set of patterns and anti-patterns in order to evaluate our CSO. Second, we define an anti-pattern detection algorithm based on SPARQL queries which provides a set of correction recommendations to help ontologists revise their ontology. Finally, tests were conducted in relation to: (i) the algorithm efficiency and (ii) anti-pattern detection of design anomalies as well as taxonomic and domain errors within CSO.

1. Introduction

Over the last years, cloud computing has become an attractive strategy for the users thanks to its on-demand computing services provisioned and released with minimal management effort. Cloud computing offers its benefits through three types of services, namely, Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). However, the wide range adoption of this computing paradigm is hindered by the lack of a standard language for describing providers' offers making it difficult for users to discover appropriate services. Indeed, cloud providers describe similar services in different manners (different pricing policies, different quality of services, etc.). This motivated the proposition to standardize the description of cloud services by introducing ontologies [3][11][7]. However, the absence of an evaluation of the so-far proposed ontologies did not encourage their adoption. Indeed, while developing the ontology, ontologists may commit errors, such as redundancy, missing information, inconsistency, etc. (for example, error regarding CPU missing part_of virtual machine). Moreover, while populating the ontology, by automatically extracting information from the providers' catalogs and the cloud registries, some missing descriptions, hierarchical errors and redundant instances can be inserted (for example, error regarding missing to indicate the operating system of the virtual machine). Consequently, it is necessary to evaluate the ontology concept to ensure the consistency maintenance and the redundancy elimination. Usually, the ontology evaluation is an integral part of its development lifecycle. A set of tools known as reasoners have been proposed in the literature to get rid of errors in ontologies. According to [4], "*a reasoner is a program that provides automated support for reasoning tasks, such as*

URI: http://hdl.handle.net/10125/41672
ISBN: 978-0-9981331-0-2

classification, debugging and querying". Researchers in [2][5] identified new error types not covered by the traditional evaluation techniques. In fact, reasoners, such as FaCT++ [12], HermiT [10], Pellet [?]. can not deal with the inconsistency, the incompleteness and the redundancy errors. Evaluating such errors is a very difficult task [6].

In this paper, we define a novel ontology evaluation approach which considers the cloud domain to enhance the previously proposed CSO quality [7]. We denote that our approach is the building block for an adequate service discovery. To do so, we take profit from the existing reasoners and overcome their limitations by introducing a set of patterns and anti-patterns. In fact, the introduction of patterns represents a good solution to the recurrent design problems as they help to create and maintain correct ontologies [1]. Moreover, patterns help to create rigorous ontologies with the least effort. In opposition to the patterns, anti-patterns check the errors in order to correct them and consequently achieve our CSO correctness.

The rest of the paper is organized as follows. Section 2 overviews ontology evaluation techniques. Section 3 briefly describes our proposed CSO. Section 4 defines the ontology evaluation approach by demonstrating the limitations of the existing reasoners and presenting our anti-pattern detection algorithm. Section 5 describes the evaluation of both the anti-pattern detection algorithm and CSO and discusses the experimental results. Finally, we summarize the presented work the paper and present some future endeavors.

2. Related Work

Several techniques for ontology evaluation exist in the literature among which we can cite the use of Description Logics reasoners and anti-pattern techniques.

The authors in [2] proposed an evaluation framework of ontolingua ontologies. They suppose that the presence of partition problems lead to an inconsistent ontology. Thus, they identified a set of taxonomic errors that can occur within an ontology. This set was revised and new errors were added like disjointedness error, functional property omission for a single valued property and redundancy of disjoint relation error.

The authors in [6], criticized the DL reasoner performance by evaluating their consistency in a case study of automobile ontology. They proved that the reasoners such as Racer, Pellet and FaCT++ could not detect various types of errors, such as circulatory errors, semantic inconsistency errors and some types of redundancy errors like disjoint relation and identical formal definition.

To overcome this limit of reasoner-based evaluation, anti-patterns were introduced [1]. In [8], the authors identified an anti-pattern catalogue which is defined with DL expressions. In [9], the authors proposed an anti-pattern detection approach based on SPARQL queries for the OIL anti-pattern. They showed how this anti-pattern could not be detected by the reasoners.
In [9], the authors defined a detection method based on ontology transformations to simulate the reasoner's work.

In summary, reasoners can not detect the taxonomic errors as well as the domain errors. In addition, starting from scratch is not a good solution. To optimize the effort, it is better to reuse and benefit from the existing reasoners and cover their limitations by a set of anti-patterns. In order to evaluate our CSO [7], we combine, in this work, the two evaluation techniques, the reasoner and anti-patterns, to identify and correct the redundancy, the incompleteness and the domain knowledge missing of the ontology.

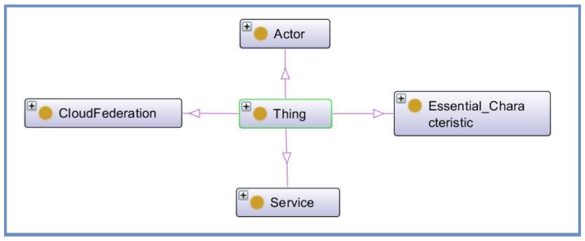


Figure 1. The Top Level Concepts

3. Cloud Service Ontology: CSO

Our CSO building process was based on cloud standards, such as NIST [1], OCCI [2], CIMI [3]. For more details about CSO, we refer the reader to [7]. In this section, we present the important CSO's concepts and axioms. Among the main concepts in CSO, we cite: Actor, Service, Essential_Characteristic and CloudFederation, as shown in Figure 1. The Actor class presents the different actors which participate in the cloud. In Figure 2, we distinguish between: the user, the provider and the broker. The Provider class includes three subclasses, Provider_IaaS, Provider_PaaS and Provider_SaaS. Each cloud provider offers three different service types which are: *(i)* IaaS including the Storage and the Virtual_Machine resources, *(ii)* PaaS covering the Platform resource and *(iii)* SaaS encompassing the Software resource as CRM, mailing list, video gaming, etc.

[1]http://www.nist.gov/itl/cloud
[2]http://occi-wg.org
[3]https://www.dmtf.org/standards/cloud

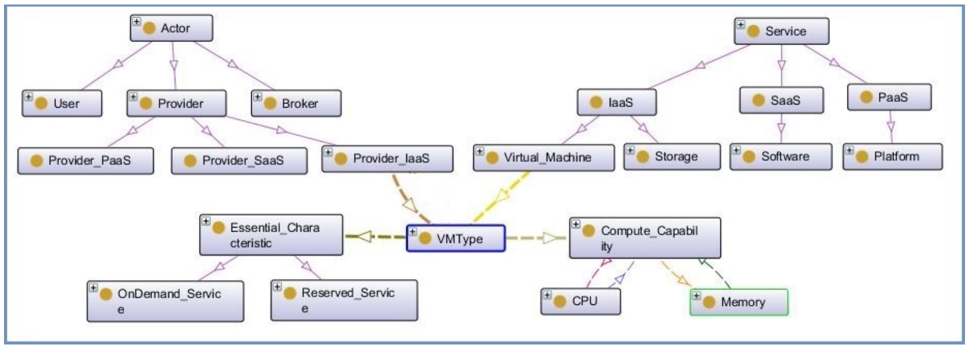


Figure 2. An extract of CSO

Table 1. CSO important axioms

Axiom
$Broker \sqsubseteq Actor; User \sqsubseteq Actor; Provider \sqsubseteq Actor;$
$Broker \sqcap Provider \sqcap User \sqsubseteq \bot;$
$User\_IaaS \sqcap User\_PaaS \sqcap User\_SaaS \sqsubseteq \bot;$
$Provider \equiv Provider\_IaaS \sqcup Provider\_PaaS \sqcup Provider\_SaaS;$
$IaaS \sqsubseteq Service; PaaS \sqsubseteq Service; SaaS \sqsubseteq Service;$
$IaaS \sqcap PaaS \sqcap SaaS \sqsubseteq \bot;$
$Storage \sqsubseteq IaaS; Virtual\_Machine \sqsubseteq IaaS;$
$Platform \sqsubseteq PaaS;$
$Software \sqsubseteq SaaS;$
$OnDemand\_Service \sqsubseteq Essential\_Characteristic;$
$Reserved\_Service \sqsubseteq Essential\_Characteristic;$
$OnDemand\_Service \sqcap Reserved\_Service \sqsubseteq \bot;$
$ComputeCapability\_ContainsCpu \equiv cpuIsPartOf-;$
$ComputeCapability\_ContainsMemory \equiv memoryIsPartOf-;$

Table 1 identifies the set of axioms which cover the specialization, the disjoint and the union relations between the important concepts within CSO. Each Virtual_Machine (VM) has specific characteristics depending on its type. The object property *hasVMType* links between the two concepts Virtual_Machine and VMType, as shown in Table 2.

The Provider_IaaS offers a wide range of virtual machine types, such as *small*, *medium* and *large*. Each VMType has an essential characteristic which is either on demand self-service (OnDemand_Service), which is a prime feature of most cloud offerings where the user pays for use, or on reserved service (Reserved_Service) where the user will get a discount after a service is used for a specific amount of time. Moreover, each VMType should have a compute capability which includes two characteristics: CPU and memory. We admit that the two characteristics could not exist without a compute capability. The object properties *cpuIsPartOf* and *memoryIsPartOf* are the inverse of the object properties *ComputeCapability_ContainsCpu* and *ComputeCapability_ContainsMemory* respectively, as shown in the two last lines in Table 1.

Figure 3 shows the CloudFederation concept. In a cloud federation, the services are provisioned by a group of cloud providers that collaborate together to share resources. The property *interconnected* represents the different members of a cloud federation. Each federation has an architecture *hasFederationArchitecture*, a type *hasFederationType* and a network *hasNetwork*, as shown in Table 2. The federation architecture can be installed with broker, FederationArchitecture_Centralized, where there is a central broker that performs and facilitates the resource allocation or without broker, FederationArchitecture_Decentralized. The federation type can be vertical which spans multiple levels, horizontal which takes place on one level of the cloud stack, hybrid which covers both vertical and horizontal or delegation. It is worth noting that tables 1 and 2 present the axioms which are useful in the evaluation process, particularly, in the anti-pattern detection phase.

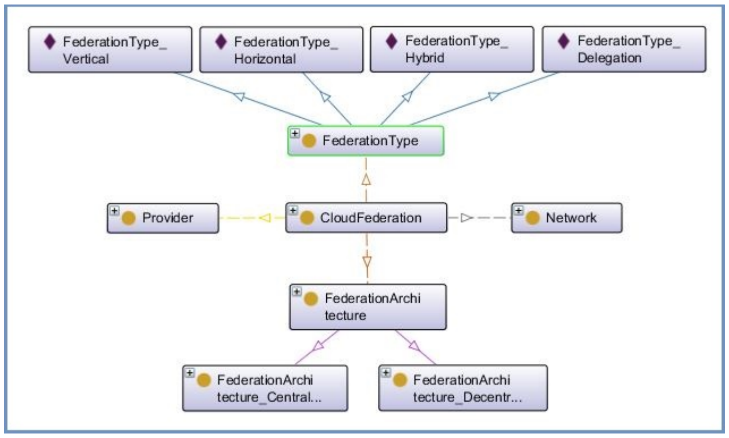

FederationType_ Vertical
FederationType_ Horizontal
FederationType_ Hybrid
FederationType_ Delegation
FederationType
Provider
CloudFederation
Network
FederationArchi tecture
FederationArchi tecture_Central...
FederationArchi tecture_Decentr...

Figure 3. Cloud Federation Description

Table 2. Object properties in CSO

Object property	domain	range
hasVMType	Virtual_Machine	VMType
hasEssentialCharacteristic	VMType	Essential_Characteristic
ComputeCapability_ContainsCpu	Compute_Capability	CPU
cpuIsPartOf	CPU	Compute_Capability
ComputeCapability_ContainsMemory	Compute_Capability	Memory
memoryIsPartOf	Memory	Compute_Capability
hasFederationArchitecture	CloudFederation	FederationArchitecture
hasFederationType	CloudFederation	FederationType
hasNetwork	CloudFederation	Network
interconnected	CloudFederation	Provider

4. Ontology Evaluation Approach

The evaluation is an important phase in the ontology lifecycle. According to Gómez-Pérez, ontology evaluation ”*is a technical judgment of the content of the ontology*” [2]. The main goal is to detect the errors within an ontology and then correct them in order to increase the ontology quality and consequently to guarantee its utilization. In this section, we present an overview of our CSO evaluation approach (see Figure 4). Our approach is composed of two phases:

- **Evaluation process**
  In order to validate CSO's consistency, we choose the DL reasoner FaCT++, developed at the University of Manchester and designed as a platform for experimenting through tableaux algorithms and optimization techniques. It is an open source, free and available with Protégé. Besides, in order to validate CSO cloud standard conformity, we define a set of patterns and anti-patterns. The detection of ontology anti-patterns contributes to ontology quality assessment [9].

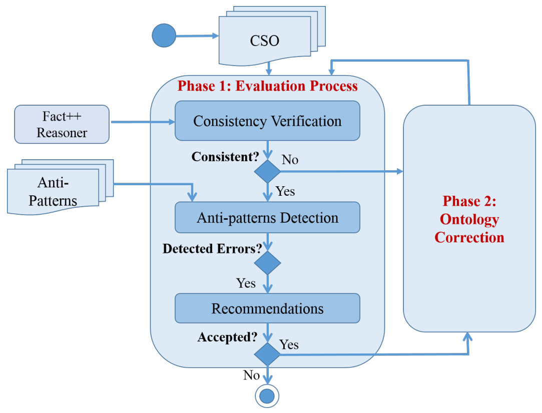


Figure 4. Ontology evaluation approach

As shown in Figure 4, this phase includes the following three steps:

– **Consistency Verification:** we use the FaCT++ reasoner in order to validate the consistency of CSO. According to [2], ”*a given* *definition is consistent if and only if the individual definition is consistent and no contradictory knowledge can be inferred from other definitions and axioms*". If ontology is inconsistent, then the correction phase should be applied and the reasoner's application should be repeated. Otherwise, the next step, namely anti-pattern detection, will be triggered.

  – **Anti-pattern detection:** this step consists in detecting the errors undiscovered by the FaCT++ by using a set of proposed anti-patterns (see Sect. 4.2). If no error is detected, ontology is considered as evaluated. Otherwise, the recommendations for the detected errors and anomalies will be triggered.

  – **Recommendations:** in this step, we use the anti-pattern catalogue which associates correction recommendations for each anti-pattern.
  The proposed recommendations can be taken into consideration by ontologists, and in this case, the correction phase will be triggered. After that, we restart the evaluation process. If the recommendations are ignored by the ontologists, who consider that the errors are not relevant, the evaluation process is terminated.

- **Ontology correction**
  This manual phase aims at changing the ontology in order to correct the errors detected by either the reasoner or by following the recommendations proposed by the anti-patterns.

4.1. Pattern Definition

As defined in [1], the Ontology Design Pattern (ODP) is a modeling solution to a recurrent ontology design problem. In our work, the pattern definition is mainly based on taxonomy.

In general, taxonomy hierarchically organizes classes and instances within an ontology [2]. In the cloud domain, taxonomy can be identified based on standards, such as NIST [1], OCCI [2], CIMI [3], which hierarchically present cloud service characteristics. We define the following generic decomposition pattern, which a common pattern, in order to guarantee cloud computing taxonomy. It is worth mentioning that the following pattern is generic that can be applied to any ontology treating any domain.

The decomposition pattern: to respect the cloud standards, the hierarchy definition should meet one of the following decomposition manners.

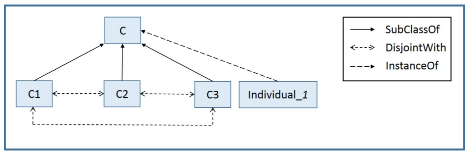


Figure 5. Disjoint Decomposition Pattern

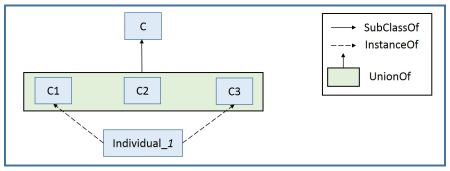


Figure 6. Exhaustive Decomposition Pattern

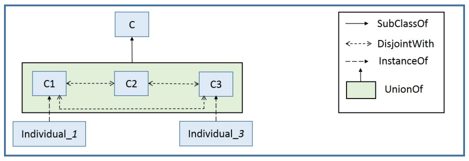


Figure 7. Partition Pattern: Disjoint Exhaustive Decomposition

- **Disjoint decomposition:** this pattern defines a set of disjoint subclasses of class C. This classification implies that each instance can belong directly to class C or to one of the subclasses of C, as shown in Figure 5. This pattern is expressed through DL as follows:

$$C1 \sqsubseteq C; C2 \sqsubseteq C; C3 \sqsubseteq C; C1 \sqcap C2 \sqcap C3 \sqsubseteq \bot; \tag{1}$$

- **Exhaustive decomposition:** this pattern defines a complete classification of subclasses of class C. These subclasses are not necessarily disjoint. Indeed, each instance of class C should be an instance of one or more of the subclasses as shown in Figure 6. This pattern is expressed through DL as follows:

$$C1 \sqsubseteq C; C2 \sqsubseteq C; C3 \sqsubseteq C; C1 \sqcup C2 \sqcup C3 \equiv C; \tag{2}$$

- **Partition:** this pattern defines a set of disjoint subclasses of class C. This classification is also complete and class C is the union of all the subclasses. Each instance of class C should be an instance of only one subclass, hence no shared instances are allowed. Figure 7 shows the partition pattern. This pattern is expressed through DL as follows:

$$C1 \sqsubseteq C; C2 \sqsubseteq C; C3 \sqsubseteq C;$$
$$C1 \sqcap C2 \sqcap C3 \sqsubseteq \bot; C1 \sqcup C2 \sqcup C3 \equiv C; \quad (3)$$

4.2. Anti-Pattern Definition

The anti-pattern is similar to a pattern, "*except that instead of a solution, it gives something that looks superficially like a solution, but it is not*" [13].

In our work, we propose three anti-pattern categories: taxonomic errors, design anomalies and domain errors. Each category includes a set of anti-patterns. It is worth noting that taxonomic errors and design anomalies are generic and can be used with others ontologies in different domains. However, domain errors are specific to our CSO. The reason behind the proposal of taxonomic and domain errors is to cover, respectively, the errors related to the ontology concepts as well as to the ontology instances. Besides the errors, we propose the design anomaly category. In fact, anomalies removal is necessary to improve ontology usability [6]. We present in what follows, the anti-pattern definitions using the DL expressions.

4.2.1 Generic Taxonomic Errors

In this category, we aim at detecting possible violations of the set of patterns already defined in Sect. 4.1. Referring to the work of Gómez-Pérez [2], we identify different error types that occur when modeling taxonomic knowledge within an ontology. In this context, these errors specify :

- **Inconsistency**
  The inconsistency errors occur when there are external instances in a complete classification or when the disjoint relation is defined between classes with different hierarchies.
  - **External instances in exhaustive decomposition and partition:** these errors occur when there are one or more instances of the class that do not belong to any subclass. This anti-pattern is expressed through DL as follows:

$$C(Individual\_1); C1 \sqsubseteq C; C2 \sqsubseteq C;$$
$$C3 \sqsubseteq C; C1 \sqcup C2 \sqcup C3 \equiv C; \quad (4)$$

  - **Partition Error:** this error appears when a class is disjoint with the sibling of its super class instead of being disjoint with its sibling classes. This anti-pattern can be expressed through DL as follows:

$$C1 \sqsubseteq C; C2 \sqsubseteq C; C3 \sqsubseteq C1;$$
$$C2 \sqcap C3 \sqsubseteq \bot; \quad (5)$$

- **Incompleteness**
  This kind of error occurs when the ontologists omit the definition of disjoint relation or the union relation. In fact, the information about the super class or the subclasses is missing.
  - **Decomposition knowledge Omission:** this error covers two types of omission whenever identifying a decomposition of a concept: *(i)* the omission of which subclasses are disjoint or *(ii)* the omission, which it is the union of all its subclasses. This anti-pattern is expressed through DL as follows:

$$C1 \sqsubseteq C; C2 \sqsubseteq C; \quad (6)$$

- **Redundancy**
  A redundancy error appears when two object properties have the same formal definition or when the disjoint relation is duplicated. The redundancy error covers:
  - **Identical formal definition of object properties:** this error occurs when different object properties have the same formal definition, i.e. they have the same couple (domain, range), although they have different names. This anti-pattern is expressed through DL as follows:

$$\exists R.\top \sqsubseteq C1; \top \sqsubseteq \forall R.C2;$$
$$\exists R1.\top \sqsubseteq C1; \top \sqsubseteq \forall R1.C2; \quad (7)$$

  - **Redundancy of Disjoint Relation:** it is the fact of defining a concept as disjoint with other concepts more than once, i.e. classes that have more than one disjoint relation. The DL expression shows that the disjoint relation between subclasses C3 and C4 is repeated since they already inherit it from their base classes C1 and C2.

$$C3 \sqsubseteq C1; C4 \sqsubseteq C2; C1 \sqcap C2 \sqsubseteq \bot;$$
$$C3 \sqcap C4 \sqsubseteq \bot; \quad (8)$$

4.2.2 Generic Design Anomalies

Likewise, we identify the design anomalies within ontology. The correction of these anomalies guarantees the usability of an ontology [6]. Actually, these anomalies include:

- **Lazy Concept:** is an instantiated concept of which all the datatype properties are not defined. This anti-pattern is expressed through DL as follows:

$$\neg\exists T.d \equiv \forall T.\neg d; \tag{9}$$

- **Weak Lazy Concept:** is an instantiated concept of which the datatype properties are not defined. This anti-pattern can be expressed through DL as follows:

$$\neg\forall T.d \equiv \exists T.\neg d; \tag{10}$$

- **Weak Concept Definition:** this refers to the situation when an instantiated concept has all or some of its object properties not defined. This anti-pattern is expressed through DL as follows:

$$\neg\forall R.C; \neg\exists R.C; \tag{11}$$

4.2.3 Domain Errors

The previous cited anti-patterns are generic, which can be useful in different domains. However, our objective is to evaluate a cloud service description ontology. For this reason, the following set of anti-patterns focuses on cloud service description errors. It is worth noting that due to space limitations, we choose only the following set of anti-patterns and we can refer the reader to our website https://sites.google.com/site/csoevaluation/home in order to find the complete list illustrated with examples.

- **Invalid VMType Characteristic:** each VM type must have at least an essential characteristic and a compute capability. Such anti-pattern occurs, as shown in the DL expression, when these two characteristics are not defined.

$$\begin{aligned}\neg\exists hasEssentialCharacteristic.\\ Essential\_Characteristic;\\ \neg\exists Compute\_Capability\_Attachment.\\ Compute\_Capability;\end{aligned} \tag{12}$$

- **Invalid Provider Description:** each provider follows a deployment model, has a hosting type, a pricing model and a set of service capabilities. The omission of one or more of the information leads to an invalid provider description.

$$\begin{aligned}\neg\exists hasDeploymentModel.Deployment\_Model;\\ \neg\exists hasHostingType.HostingType;\\ \neg\exists hasPricingModel.PricingModel;\\ \neg\exists Provider\_ServiceCapabilities.\\ Provider\_Capability;\end{aligned} \tag{13}$$

- **Invalid Compute_Capability Definition:** each virtual machine type must have a compute capability. This capability includes two characteristics: CPU and memory. We admit that these two characteristics could not exist without a compute capability. The following DL expression shows the verification in both directions.

$$\begin{aligned}\neg\exists cpuIsPartOf.Compute\_Capability;\\ \neg\exists Compute\_Capability\_ContainsCPU.CPU;\\ \neg\exists memoryIsPartOf.Compute\_Capability;\\ \neg\exists Compute\_Capability\_ContainsMemory.\\ Memory;\end{aligned} \tag{14}$$

- **Invalid Service_Capability Description:** each service must have a set of capabilities. For example, the storage and network capacities must be provided by the IaaS service. The following DL expression shows the capabilities that should be offered by each service type.

$$\begin{aligned}\neg\exists Platform\_Capabilities.Functional\_Capability;\\ \neg\exists Software\_Capabilities.Functional\_Capability;\\ \neg\exists Storage\_Capabilities.Functional\_Capability;\\ \neg\exists Network\_Capabilities.Functional\_Capability;\end{aligned} \tag{15}$$

- **Faulty Value:** the previous set of domain errors considers object properties. In the following part, we focus on the datatype properties. The values used in the two following anti-patterns are extracted from the three giant cloud provider catalogues, namely AWS Cloud [4], Google Cloud [5] and Microsoft Azure [6].
  - **Faulty Value_CPU_RAM:** each cloud provider offers a specific capability of a core number and memory size. This anti-pattern occurs, as shown in the DL expression, when one or more datatype properties violate the restrictions of each provider.

$$\begin{aligned}Provider\_IaaS(Microsoft\_Azure);\\ \exists Cores. < 1; \exists Cores. > 32;\\ \exists MemorySize. < 0.75; \exists MemorySize. > 448;\\ Provider\_IaaS(Google\_Cloud);\\ \exists Cores. < 1; \exists Cores. > 32;\\ \exists MemorySize. < 0.6; \exists MemorySize. > 208;\end{aligned}$$

[4]https://aws.amazon.com/
[5]https://cloud.google.com/
[6]https://azure.microsoft.com/

$$Provider\_IaaS(AWS);$$
$$\exists Cores. < 1; \exists Cores. > 40;$$
$$\exists MemorySize. < 0.5; \exists MemorySize. > 244; \quad (16)$$

4.3. Consistency Verification

In order to investigate how FaCT++ handles various errors, we load CSO in Protégé and introduce some illustrating errors, such as:

- we define an instance ”IaaS_1” and link it directly to class IaaS instead of IaaS's subclasses. This introduces an external instance in exhaustive decomposition and partition error,
- we add a disjoint relation between the class Broker and the subclass User_IaaS in order to make a partition error,
- we omit the union relation of the class Provider_Capability. Due to the absence of disjoint relation between the subclass Functional_Capability and the subclass Non_Functional_Capability, a decomposition knowledge omission error will occur,
- in order to introduce an identical formal definition object properties error, we define a new object property named *hasArchitectureFederation* that has the same couple (domain, range) of *hasFederation-Architecture* which is (FederationCloud, ArchitectureFederation) and
- we add a disjoint relation between the two Platform and the Software subclasses, yet, we have already one between PaaS and SaaS. This new disjoint relation introduces redundancy of disjoint relation.

Thereafter, we execute the FaCT++ reasoner which considers our definition of the instance ”IaaS_1” as a normal one, and gives no error or warning. Furthermore, it does not detect any errors from these, previously, introduced ones.

In summary, the FaCT++ reasoner is not able to detect the problems related to the taxonomic errors defined in Sect. 4.2. To the best of our knowledge, there is no existing reasoner able to detect the design anomalies and the domain errors.

4.4. Anti-Pattern Detection and Recommendations

According to the results obtained through the FaCT++ reasoner, it seems necessary to apply the anti-patterns defined in Sect. 4.2. We propose an anti-pattern detection algorithm based on SPARQL queries. Due to space limitations, we present only the SPARQL query of the invalid VMType characteristic anti-pattern and its correction recommendation. A complete description of the queries and the recommendations is available in our website https://sites.google.com/site/csoevaluation/home.

Invalid VMType Characteristic: the query in listing 1 returns each type of virtual machine (?VMt) that is instantiated, but its definition is missing an essential characteristic (?E_C) and a compute capability (?compC).

Listing 1. SPARQL Query: Invalid VMType Characteristic

```
SELECT ?VMt ?E_C ?compC
WHERE
{?VMt rdf:type ns:VMType
 OPTIONAL
 {?VMt ns:hasEssentialCharacteristic ?E_C}
 OPTIONAL
 {?VMt ns:Compute_Capability_Attachment
       ?compC}
 FILTER ((!bound(?E_C))||(!bound(?compC)))
}
```

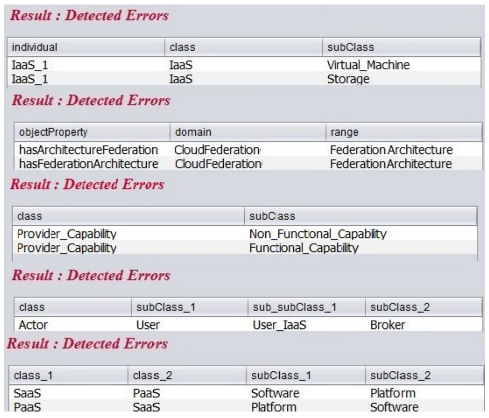
Once this anti-pattern is detected, the following correction recommendation is displayed in order to guide the ontologists to correct the error, ”*Please complete the description of the VM type class by defining these two object properties: hasEssentialCharacteristic and Compute_Capability_Attachment !*”.

4.4.1 The anti-pattern detection algorithm

The proposed Algorithm 2, which is responsible for performing the anti-pattern detection, takes as input CSO and the list of SPARQL queries. As an output, it shows the detected errors and guides the ontologists to correct these errors by providing a set of recommendations. During execution, the following cycle happens: *(i)* select a query from the list; *(ii)* execute the query and *(iii)* once the anti-pattern is detected, the error appears with a list of correction recommendations.

Listing 2. The anti-pattern detection algorithm

```
Input:CSO.owl,SPARQL_query_antipattern_list
Output:Detected Errors and Recommendations
      for each anti-pattern
Begin
      for (each query in
      SPARQL_query_antipattern_list) do
```



Result : Detected Errors

individual	class	subClass
IaaS_1	IaaS	Virtual_Machine
IaaS_1	IaaS	Storage

Result : Detected Errors

objectProperty	domain	range
hasArchitectureFederation	CloudFederation	Federation Architecture
hasFederationArchitecture	CloudFederation	FederationArchitecture

Result : Detected Errors

class	subClass
Provider_Capability	Non_Functional_Capability
Provider_Capability	Functional_Capability

Result : Detected Errors

class	subClass_1	sub_subClass_1	subClass_2
Actor	User	User_IaaS	Broker

Result : Detected Errors

class_1	class_2	subClass_1	subClass_2
SaaS	PaaS	Software	Platform
PaaS	SaaS	Platform	Software

Figure 8. Detected Taxonomic Errors

```
      {
            result=Execute(query)
            if( Exist(result) ) then
            Display("Recommendation␣Text!")
      }
End
```

To verify the anti-pattern detection algorithm efficiency, we implement it and we test its performance on a test data (i.e. CSO modified by the taxonomic error introduction). The algorithm is implemented using the apache jena [7].

5. Experimentation: Algorithm evaluation and Detecting anti-patterns in CSO

A modified CSO integrating several taxonomic errors, which are presented in Sect. 4.3, is taken as a test data. First, we run the algorithm on this ontology to verify its efficiency (Sect. 5.1). Then, we test the algorithm on the real CSO (Sect. 5.2).

5.1. Algorithm evaluation

We qualify the algorithm as efficient when it successfully detects the introduced errors. Figure 8 shows that the proposed algorithm detects the taxonomic errors as well as the design anomalies and the domain errors within the modified CSO.

Table 3 presents the three anti-pattern categories used for our evaluation. For each category, this table indicates the number of the detected errors by the anti-pattern detection algorithm. The precision measures the ratio of correctly found anti-patterns over the total number of detected anti-patterns. The last column indicates the recall which infers the ratio of correctly found anti-patterns over the total number of proposed anti-patterns. Results from Table 3 are encouraging. In fact, the anti-pattern detection algorithm covers the limitations of FaCT++ reasoner. Indeed, the set of SPARQL queries is sufficient for detecting the three anti-pattern categories, such as, taxonomic errors, design anomalies and domain errors.

[7]https://jena.apache.org/

5.2. CSO's evaluation

After the efficiency verification, the algorithm was applied on CSO to detect the existing errors. We notice that the taxonomic errors are not detected in CSO which is mainly based on the patterns (see Sect. 4.1). Besides the design anomalies, the anti-pattern detection algorithm found some domain errors, as shown in Table 4.

These results prove that the classification of CSO concepts is well-defined since the absence of taxonomic errors. However, the presence of the design anomalies and the domain errors can be explained by several reasons. The main reason is related to the ontology population process. In fact, CSO population is based on the providers' catalogues and the cloud registries. Due to standard unconformity, these data resources can include some errors on their textual content. Moreover, they do not provide a complete cloud service description. This is the reason why concepts, such as VMType and Service_Capability, have some missing information within CSO. Figure 9 shows the anti-pattern detected errors, namely Invalid VMType Characteristic, and its correction recommendations. A complete demonstration of the CSO evaluation can be found in [8].

6. Conclusion and Future Work

Ontology evaluation is an active research domain which is mainly necessary to improve the ontology quality. In fact, only a well defined ontology is utilized. In this paper, we proposed an ontology evaluation approach composed of two phases: evaluation process and ontology correction. We applied the FaCT++ reasoner to validate the consistency of the previously proposed CSO. After that, we applied the anti-pattern detection algorithm based on SPARQL queries in order to detect errors and anomalies not covered by this reasoner. To help ontologists revise CSO, the algorithm provided a set of correction recommendations. The experimental results showed that our algorithm is consistent and efficient as it can detect errors within CSO and present correction recommendations.

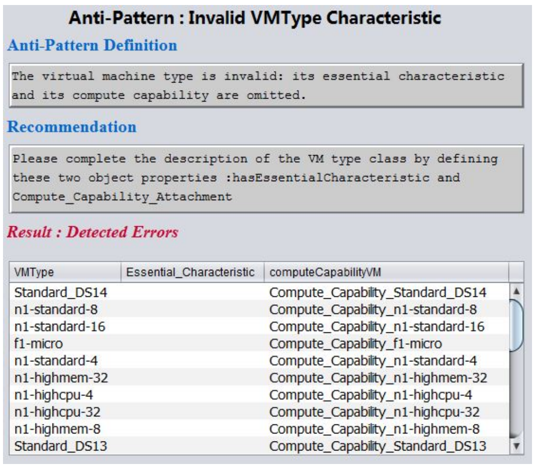
[8]https://sites.google.com/site/csoevaluation/cso-evaluation-s-demonstration

Table 3. Algorithm's evaluation results

Category	Number of Detected Errors	Precision	Recall
Taxonomic Errors	5	5/5	5/5
Design Anomalies	3	3/3	3/3
Domain Errors	5	8/8	8/8
Total	13	16/16	16/16

Table 4. Errors detected within CSO per category

Errors	Number of Detected Errors	Percentage of CSO's Errors
Taxonomic Errors	0/5	0%
Design Anomalies	3/3	100%
Domain Errors	2/8	25%
Total	5/16	31.25%


Anti-Pattern : Invalid VMType Characteristic

Anti-Pattern Definition

```
The virtual machine type is invalid: its essential characteristic
and its compute capability are omitted.
```

Recommendation

```
Please complete the description of the VM type class by defining
these two object properties :hasEssentialCharacteristic and
Compute_Capability_Attachment
```

Result : Detected Errors

VMType	Essential_Characteristic	computeCapabilityVM
Standard_DS14		Compute_Capability_Standard_DS14
n1-standard-8		Compute_Capability_n1-standard-8
n1-standard-16		Compute_Capability_n1-standard-16
f1-micro		Compute_Capability_f1-micro
n1-standard-4		Compute_Capability_n1-standard-4
n1-highmem-32		Compute_Capability_n1-highmem-32
n1-highcpu-4		Compute_Capability_n1-highcpu-4
n1-highcpu-32		Compute_Capability_n1-highcpu-32
n1-highmem-8		Compute_Capability_n1-highmem-8
Standard_DS13		Compute_Capability_Standard_DS13

Figure 9. Detected Invalid VMType Characteristic Error

As future work, we plan to extend our proposed patterns and anti-patterns by proposing ones dealing with semantic evaluation. In addition, we intend to integrate CSO in a cloud federation environment and then interrogate it with the users' requirements.

References

[1] V. P. Aldo Gangemi. *Handbook on Ontologies*, chapter Ontology Design Patterns, pages 221–243. International Handbooks on Information Systems. Springer Berlin Heidelberg, 2 edition, 2009.

[2] M. M. F.-L. P. M. O. C. M. a. Asuncin Gmez-Prez PhD, MSc. *Ontological Engineering: With Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web*. Advanced Information and Knowledge Processing. Springer-Verlag London, 1 edition, 2004.

[3] A. V. Dastjerd, S. K. Garg, F. R. Omer, and R. Buyya. Cloudpick: a framework for qos-aware and ontology-based service deployment across clouds. *Softw., Pract. Exper.*, 45(2):197–231, 2015.

[4] K. Dentler, R. Cornet, A. Ten Teije, and N. De Keizer. Comparison of reasoners for large ontologies in the owl 2 el profile. *Semantic Web*, 2(2):71–87, 2011.

[5] M. Fahad, M. A. Qadir, and M. W. Noshairwan. Ontological errors-inconsistency, incompleteness and redundancy. In *ICEIS (3-2)*, pages 253–285, 2008.

[6] M. Fahad, M. A. Qadir, and S. A. H. Shah. Evaluation of ontologies and DL reasoners. In *Intelligent Information Processing IV, 5th IFIP International Conference on Intelligent Information Processing, October 19-22, 2008, Beijing, China*, pages 17–27, 2008.

[7] M. Rekik, K. Boukadi, and H. Ben-abdallah. Cloud description ontology for service discovery and selection. In *Proceedings of the 10th International Conference on Software Engineering and Applications*, pages 26–36, 2015.

[8] C. Roussey, O. Corcho, and L. M. Vilches-Blázquez. A catalogue of owl ontology antipatterns. In *Proceedings of the fifth international conference on Knowledge capture*, pages 205–206. ACM, 2009.

[9] C. Roussey and O. Zamazal. Antipattern detection: how to debug an ontology without a reasoner. In *Proceedings of the Second International Workshop on Debugging Ontologies and Ontology Mappings, Montpellier, France, May 27, 2013*, pages 45–56, 2013.

[10] R. Shearer, B. Motik, and I. Horrocks. Hermit: A highly-efficient owl reasoner. In *OWLED*, volume 432, page 91, 2008.

[11] A. Souza, N. Cacho, T. Batista, and F. Lopes. Cloud query manager: Using semantic web concepts to avoid iaas cloud lock-in. In *IEEE 8th International Conference on Cloud Computing*, pages 702–709, June 2015.

[12] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Automated reasoning*, pages 292–297. Springer, 2006.

[13] D. Vrandečić. *Ontology evaluation*. Springer, 2009.