Enabling multi-provider cloud network service bundling

Imen Jerbi, Nour Assy, Mohamed Sellami, Sami Bhiri, Olivier Tirat, Hayet Brabra, Walid Gaaloul, Djamal Zeghlache

Enabling Multi-Provider Cloud Network Service Bundling

Imen Jerbi*[x‡‡], Nour Assy*, Mohamed Sellami*, Hayet Brabra*, Walid Gaaloul*, Sami Bhiri¶[x], Olivier Tirat[†] and Djamal Zeghlache*

* SAMOVAR, Telecom SudParis, Institut Polytechnique de Paris, France
[†]BYO NETWORKS, France, ¶University of Monastir, Tunisia,
‡‡ISITCom Hammam Sousse, University of Sousse, Tunisia,
[x]OASIS, National Engineering School of Tunis, University of Tunis El Manar, Tunisia

***Abstract*—Network as a Service (NaaS) enables tenants and cloud users to connect their distributed services across multiple clouds without relying on their own networking services and resources. Cloud service providers (CSPs) offer networking services as bundles delivered on a pay-per-use basis. Although service bundling is a common practice in the cloud providing multiple components as a single service, forcing the packaging of network resources in a pure bundle can severely limit visibility and control over network services. We argue that current descriptions of NaaS services lack transparency making cloud users unable to efficiently discover and consume these services from different providers. To address this issue, we propose to describe network services based on their functionality and according to the OSI model. Our approach explicitly describes bundling between network services that promotes discovery and enables users to create their own network services bundles from different CSPs.**

***Index Terms*—Network-as-a-Service, pure bundling, capability model, service discovery, service composition**

I. Introduction

Network as a Service (NaaS) [1], [2] is becoming a more attractive cloud model to enable flexible use and consumption of network resources. In NaaS paradigm, a network service may represent any type of network component at different layers. Many Cloud Service Providers (CSPs) including Amazon Web Services (AWS), Microsoft Azure, etc., adopt the NaaS model to deliver their network components as services on a pay-per-use basis.

While reviewing Network Services (NS) offered by NaaS providers, we first noticed that each of them represents a logical network component, that is however always linked-to/delivered-with other network resources called "associated resources" (in AWS vocabulary) or "attached resources" (in other cloud providers vocabulary such Microsoft Azure). In other words, there is a *pure bundling* between the network components represented by the NS and their associated network resources. Pure bundling is a strategy in which only a bundle of items or components is available for purchase [3]. As a consequence, cloud users must purchase the bundle as is, even if they are only interested in one of the components.

Disadvantages of pure bundling can be observed from several perspectives. From an economic perspective [4], a cloud user will be indirectly forced to pay for a service/component that was not requested [5]. The user pays per-use, but is unaware of using a not requested associated component that is implicitly bundled with the user requested and desired component.

From a service discovery perspective, pure bundling complicates the analysis of the offering [6]. A *transparent description* of the offer must be provided to cloud users so as to be able to observe all the resources that are represented/accessible by each offered service, to identify additional resources bundled with the requested ones and to predict the consequences of this bundling at the design phase. In fact, pure bundling details are currently embedded or concealed in textual descriptions published by existing CSPs. Therefore, a cloud user will not be aware of the consequences of existing bundling until after the deployment of a networking infrastructure. Moreover, a user who already consumes a cloud NS, will be unable to replace it by another giving access to similar resources. The reason is simply because the user does not know which resources are accessible by the service to be replaced and how these are coupled.

From a service composition perspective, pure bundling of network resources prevents taking full advantage of multi-clouds. Indeed, cloud users are not in a position to identify all the resources that can be accessible via each NS and unaware of their bundling. The consequence is combining services from different CSPs is restricted to great extent by the underlying NS bundling structure.

In addition, we further noticed that associated resources are often components related to the data link layer, i.e. layer two (L2) of the OSI model. From a network architecture perspective [7], coupling network layer-specific resources with ones related to the data link layer so that they are accessible/represented by a same network service, (a) reduces the use of each layer's resources, (b) creates a functional dependency between the second and the third OSI layers and (b) breaks the main advantage of the OSI model in separating layers.

For a better explanation of our findings, let us consider the virtual Network Interface Card (NIC)-one of the network components provided by AWS. In on-premise solutions, a network card is provided with a Media Access Control (MAC) address. Once this card is installed on the motherboard of a

machine, this MAC address becomes its physical identifier allowing it to communicate with other machines within the same Local Area Network (LAN). Compared to on-premise solutions, AWS NIC comes not only with a MAC address, but also with two other *"associated resources"*: an *IP address* and a *subnet ID*. Thus, by consuming such AWS NIC , a cloud user implicitly consumes a *pure bundle* of MAC address, IP address and Subnet ID. The user does not only pay for the requested NIC network component, but also for its associated components: ie., IP address and Subnet ID (an economic loss). Moreover, associating a MAC address (L2 resource) with an IP address and a Subnet ID (which are two L3 resources) reduces the use of the resources of each layer and creates a functional dependency between L2 and L3 layers (side effects on the network system architecture).

It is worth mentioning that it is more beneficial for NaaS providers to keep pure bundling their network services. Indeed, this allows them to raise the consumption of their products [8]. However, offering bundling details concealed in textual descriptions, large documentations or unstructured formats and hiding the consequences of existing bundling can significantly affect not only cloud users but also competing providers. As for users, pure bundling of network components locks them at L2 and L3 layers, making them dependent on provider choices [9]. Locking users at lower layers (L2 and L3) leads indirectly and necessarily to locking them in the upper layers: Transport Layer (L4), Session Layer (L5), Presentation Layer (L6), and Application Layer (L7). As for providers, while top cloud providers (such as AWS, Microsoft Azure and Google Cloud Platform) keep bundling their network components with associated resources, there exists other CSPs (such as VMware [10]) that try to conceive their network services without creating a functional dependency between layers. However, those are ranked in a lower position in the market compared to the mentioned top ones. This phenomenon is caused by the lack of transparent and well structured descriptions that reveal fairly differences between multiple CSPs.

To address this problem, we propose an approach for describing network services based on their functionality and according to the OSI model. The proposed approach takes into account the bundling between network services, and makes it transparent for users to promote their discovery and enable cloud users to create their own network services bundles from different cloud providers. The remainder of this paper is structured as follows. Section II introduces a model that we proposed in a previous work [11] for describing web services based on their functionalities (i.e. capabilities). Section III introduces a motivating example and then gives a brief overview of our approach towards a multi-provider cloud network service bundling and discovery. Section IV details how we extended our model [11] to describe NaaS services capabilities. In section V, we propose an algorithm to generate the proposed capability-based NaaS services description. Section VIII looks into efforts related to our work. Prior to concluding in Section IX, implementation and evaluation details are given in Section VI and Section VII, respectively.

II. Capability-based Service Description

A capability describes what a service, process, computer application, etc. achieve/does from a functional perspective [12]. The concept of capability plays a key role in Service-Oriented Computing (SOC) and enterprise information systems [13], [14]. In our previous work [11], we proposed a capability model that defines the high-level concepts required for defining domain-specific capabilities. This section introduces this capability model on which we will rely later to define the capabilities of network services. Figure 1 gives an object-oriented conceptual view of the capability model. Its main concepts are:

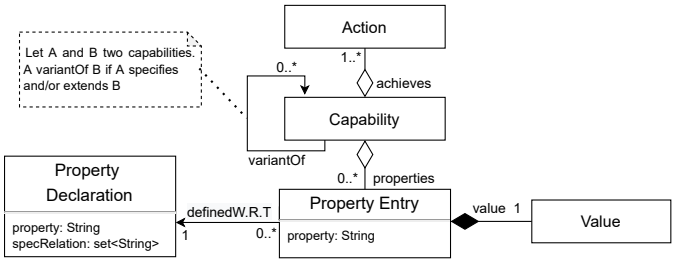

Fig. 1. Capability model [12]

- `Capability`, described by an `Action` category and zero or many functional or non-functional properties called `Property Entry`.
- `Action` is a property whose value indicates the action that the capability describes.
- The `Value` of a `Property Entry` may be of different types as illustrated in Figure 2.

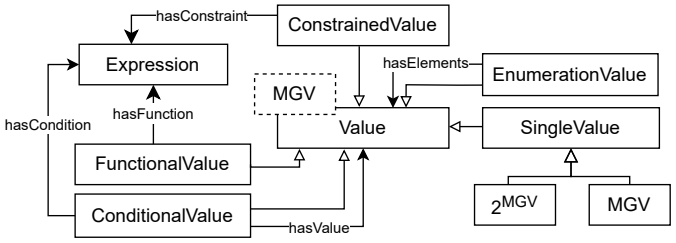

Fig. 2. Overview of the Property Entries Values Type [11].

  - `MGV` (Most General Value) refers to the instances of the `MGV` class.
  - $2^{\texttt{MGV}}$ refers to `MGV` sub-classes.
  - `ConstrainedValueMGV` allows to define a constraint as a value of a property.
  - `FunctionalValueMGV` allows to define a function as a value of a property.
  - `ConditionalValueMGV` enables assigning a value to a given property if a certain condition holds.
  - `EnumerationValueMGV` defines a property value as a finite set of values.

A more detailed description of theses types is given in [11].

- `Property Entry` (PE) gives more details about the carried action. It has a *property* and a `Value`. It is defined w.r.t a `Property Declaration` (having the same *property*).
- `Property Declaration` (PD) represents the declaration of the Property Entry. It has a *property*, a Most General Value (*MGV*) (Figure 2) and a set of specification relations (*specRelation*).
  - *property* is the identifier of a relevant functional or non-functional domain-specific feature,
  - *MGV* is the most general value/class that the PEs, defined according to PD, may have
  - *specRelation* is a specification relation related to *MGV* and reflecting the semantic meaning of the relevant feature. If defined, this relation identifies when an instance of the *MGV* class specifies another w.r.t the semantics of the relevant property.

`Capability` may be `variantOf` one or many other capabilities. Given two capabilities A and B, B `variantOf` A in three cases as illustrated in Figure 3:

1) B inherits all the PEs of A with at least one additional PE. In such case, B `variantOf` A because B *extends* A. For instance, in Figure 3, B inherits the PEs $< p1 = v1 >$, $< p2 = v2 >$ and $< p3 = v3 >$ of A and defines $< p4 = v4 >$ as an additional PE.
2) B shares the properties of A and for each shared property p, the value of p in B is equal or is more specific than its value in A. In such case, B `variantOf` A because B *specifies* A. For instance, in Figure 3, B inherits properties $p1$, $p2$ and $p3$ of A. $p1$ and $p2$ have the same values in A and B, while the value of $p3$ in B is more specific than its value in A.
3) B *specifies* and *extends* A at the same time.

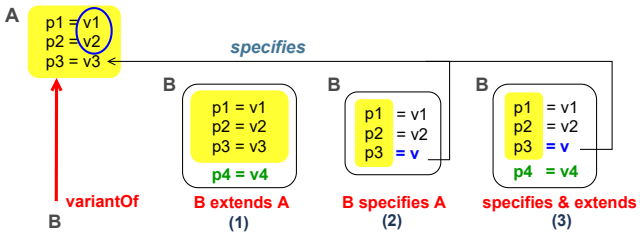

Fig. 3. Relation `variantOf` between two Capabilities A and B [11]

For the rest of the paper, given a capability C, we denote by $properties(C)$ the set of the properties of the PEs of C and by v_C^p the value of a property p in a property entry PE in C.

III. Motivating Example and Approach Overview

A. Motivating Example

Let us consider the following example: John, a cloud user, wants to connect his on-premise machine, which is already integrated by a network interface card (and therefore identified by a MAC address), through a Virtual Private Network (VPN) on the L2 cloud layer. Doing so, John seeks to attach his on-premise machine to a sub-network offered by a NaaS provider in order to assign, to this provider, the task of bridging packets coming from his machine.

This simple running example raises the following research questions:

- Are cloud networks, in their current design, able to meet John's needs and therefore bridge packets coming from an external MAC address?
- If not, what makes them unable to satisfy such a need? How to make users, like John, aware of that at the design phase of an on-premise/cloud connection project? and how to make these causes explicitly transparent to them?
- If yes, how can John locate the network services that satisfy his need? How to make the discovery of the requested network service efficient?
- Suppose that after locating the required service and realizing his need, John wants to replace the service in-use by another service that offer the same functionality but located within another provider (region or even availability zone). How could he efficiently locate this service without repeating the discovery process from scratch?

B. Approach Overview

Figure 4 illustrates our approach for enabling multi-provider cloud network service bundling and discovery. To this end we extend the capability model introduced in Section II to allow a **capability-based description of NaaS services** and we consider two stopovers: **NaaS services description** and **NaaS services discovery**. The extended model, NaaS capability model, defines high-level concepts required for defining capabilities of network services, and is presented in Section IV. During the first stopover of our approach, NaaS services description, we rely on a catalog of network services defined by a network expert[1] and unstructured textual NaaS services descriptions extracted from CSP documentations. During the NaaS services description stopover, we create a catalog of capability-based NaaS services descriptions according to our NaaS capability model. Specifically, the Capability extraction module extracts the NaaS services' capabilities: based on the Network services catalog using the OSI specific module and based on the Unstructured NaaS services descriptions using the provider specific module. The so extracted non-hierarchical capabilities are then passed to the Description generation module. This module, first generates "missing" capabilities linking provider specific capabilities to OSI specific ones. Then, the Relations generation module uses a rule-based inference method for determining the relations linking the generated and the extracted capabilities together. The so created hierarchical descriptions are then stored in the Capability-based NaaS services catalog. Based on this catalog and a cloud customer provider-independent request, the NaaS services discovery module returns a custom-made NaaS service bundle. In the following we detail the proposed **capability-based model for NaaS services** and the **NaaS**

[1] In this work, this catalog was defined by a network expert from the ISChyO partnership project considering L2 and L3 network services.

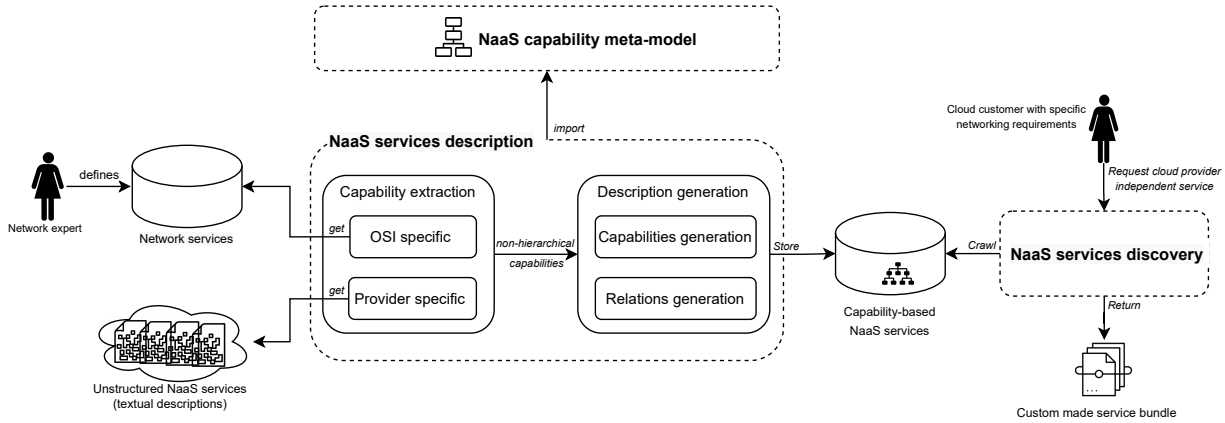

Fig. 4. Approach for multi-provider cloud network service bundling and discovery

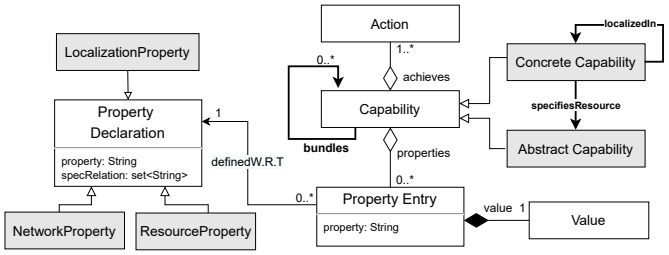

Fig. 5. Capability model for NaaS services description

services description stopover. The **NaaS services discovery** stopover is not presented in this paper.

IV. NaaS Capability Model

In this section, we present the extended capability model for NaaS services description. This model allows NaaS services description based on their capabilities, and hence offers the means for their capability-based discovery.

Figure 5 shows the structure of the extended model. Extended concepts are in grey and new relations are in bold. Compared to the capability model presented in Section II, we specialize a `Capability` as `AbstractCapability` and `ConcreteCapability`. Abstract capabilities allow to describe NaaS services independently of the technical details of specific CSPs while concrete capabilities describe NaaS services offers. We also specialize capabilities' properties, i.e., `PropertyDeclaration`, as `ResourceProperty`, `NetworkProperty`, and `LocalizationProperty`. In addition, we introduce three types of relations between capabilities: `bundles`, `localizedIn`, and `specifiesResource`. These relations create a hierarchical description of NaaS services capabilities given then means for an efficient discovery. In the following, we introduce these capability types (Section IV-A) and relations (Section IV-B).

A. NaaS Service Capability

A NaaS service can be seen as an access mechanism to network resources that can be provided by an entity other than the requester [15]. In fact, a cloud network service makes a network resource accessible, i.e. exploitable. Therefore, we define the `Action` of a NaaS capability as `ExposingResource`. In order to discover and invoke a NaaS service, the user needs to *localize* the resources by specifying, in addition to the resource types, the provider, area, region and availability zones in which the resources are offered. Hence, NaaS capabilities need to include details related to resource localization. For a better description of service capabilities, and for more efficient discoverability, we propose in this work to *decouple* the description of the service functionality from the technical access details related to resource localization. To this end, we distinguish between two types of NaaS services: *non-localized* and *localized*. Non-localized services are defined independently of any CSP provider. They describe high level functionalities of network services. We refer to their capabilities as *abstract capabilities*. Localized NaaS services on the other hand are concrete services offered by CSPs and can be consumed by end users. We refer to their capabilities as *concrete capabilities*.

1) Abstract Capabilities: As already mentioned, abstract capabilities are defined for non localized NaaS services. They describe high level functionalities of network services independently of technical details and specific implementations of CSPs. These capabilities are usually defined by domain experts. They have two types of properties: `ResourceProperty` and `NetworkProperty`. `ResourceProperty` declaration indicates the type of exposed resources. Its `property name` is `resourceType` and its value is an element of $2^{\text{NetworkResource}}$ [2], where `NetworkResource` is an MGV and is defined as follows (Figure 6):

- An L2 address is the physical identifier of a machine. It is equivalent to the MAC address if the protocol used at the linked data layer is Ethernet;
- An L3 address is the network identifier of a machine. It is equivalent to the IP (Internet Protocol) address if the

[2] $2^{\text{NetworkResource}}$ is the powerset of *NetworkResource*

protocol used at the network layer is the Transmission Control Protocol (TCP);
- A virtual L3 address is a virtual L3 address associated with a virtual router;
- A subnet is a range of L3 addresses;
- A route refers to the path a packet travels on a network. It refers to each device that handles the packet between its source and destination (as routers, switches, firewalls).

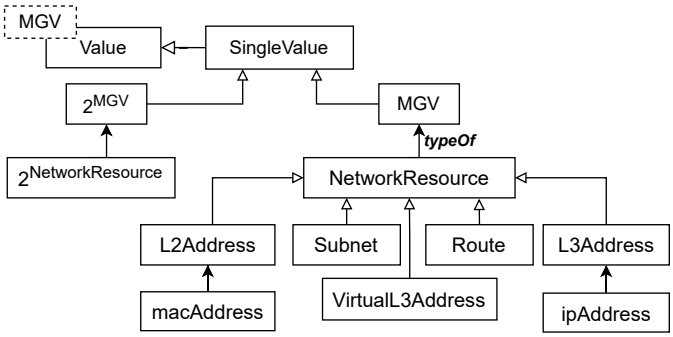

Fig. 6. *NetworkResource*: most general value of the property *resourceType*

Besides `ResourceProperty`, additional functional or non functional properties can be described using the property declaration `NetworkProperty`. Table I shows some of the network properties we identified. These properties were defined based on a domain expertise in the area of networking systems. For instance, the `requiresCapability` property indicates whether the capability is dependent on another capability. Its MGV is of type `Capability`. The `requiresResource` property indicates which specific resource types are needed from the required capability. Its MGV is of type `NetworkResource` [3].

TABLE I
NETWORKPROPERTY DECLARATION

NetworkProperty		
PropertyName	***MGV***	***SR***
requiresCapability	Capability	none
requiresResource	NetworkResource	none
attachedTo	Network	none
hostedOn	ServiceHost	none
protocol	String	none
subnetId	String	none

Figure 7 shows an excerpt of a NaaS capability-based description of the Network Interface services offered by AWS and Microsoft Azure. Examples of abstract capabilities are shown in the top layer of this description sample. For better readability, we include only `Action` and `ResourceProperty`, properties related to `NetworkProperty` are not shown. These capabilities are defined based on the OSI model. For instance, the capability $C_{\text{L2endPoint}}$ describes the functionality of a networking service

[3]Due to space constraints, additional details about the network properties and their MGVs can be found at Supplementary-material-1

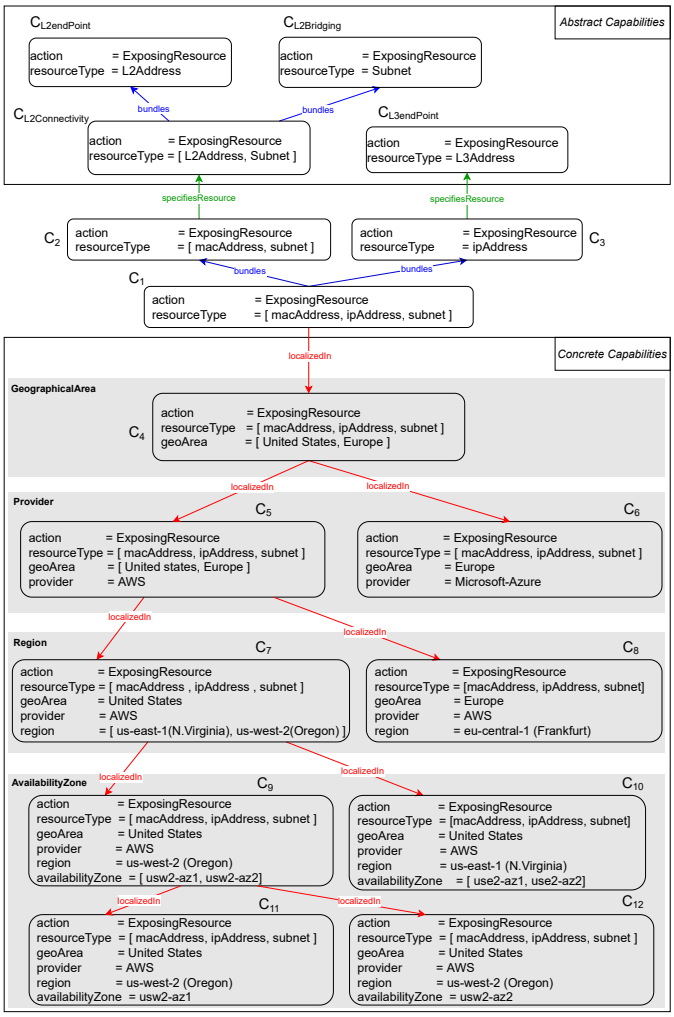

Fig. 7. An excerpt of NaaS services description using the NaaS capability model

that gives access to the L2 address of a machine; $C_{\text{L2Bridging}}$ describes the functionality of a service that makes the subnet of a local network accessible; $C_{\text{L2Connectivity}}$ describes the functionality of a service that allows to create an L2 connectivity. $C_{\text{L2Connectivity}}$ exposes two types of resources: the machine's L2 address and the local network subnet.

2) Concrete Capabilities: Concrete capabilities describe concrete NaaS services offered by CSPs. Similarly to abstract capabilities, concrete ones are described in terms of the action `ExposingResource` and the property declarations `ResourceProperty` and `NetworkProperty`. In addition, they include localization related properties under the property declaration `LocalizationProperty`. To localize resources, we define four properties: `GeographicalArea`, `Provider`, `Region` and `AvailabilityZone`. The MGVs of these properties are defined as shown in Fig. 8. Examples of concrete capabilities are shown in the gray layers of the sample description of Figure 7. These capabilities describe the NetworkInterface service offered by Amazon AWS and Microsoft Azure. As shown in the example, each

localization property creates a localization layer making the model hierarchical. This hierarchy helps in localizing NaaS services at different levels of abstraction during discovery, e.g. discovering all NaaS services that allow to create a local network connectivity (i.e. L2 connectivity) and that are offered in a specific geographical area corresponds to the subtree rooted at C_4.

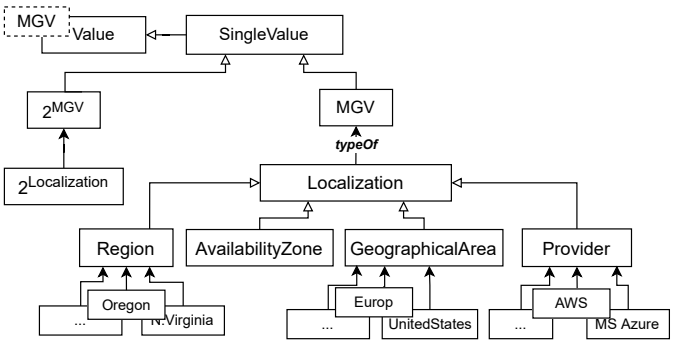

Fig. 8. *Localization*: most general value that may be defined in a *LocalizationProperty*

B. NaaS Capability Relations

NaaS capabilities are connected through three types of relations: `bundles`, `localizedIn` and `specifiesResource`. All the three relations are derived from the `variantOf` relation of the capability model introduced in Section II. The relation `bundles` allows to explicitly model the packaging of NaaS services. It is defined between a parent capability and at least two children capabilities where the parent capability is a bundle of the children capabilities.

*Definition 4.1 (**bundles**):* Let $C_1, \ldots, C_n$, $n \geq 3$ be a set of capabilities achieving the same action. C_1 *bundles* $\{C_2, \ldots, C_n\}$ iff $\forall i \geq 2$, C_i *variantOf* C_1 and $v_{C_1}^{resourceType} = \cup_{i=2}^{n} v_{C_i}^{resourceType}$.

The relation `localizedIn` is defined between an abstract or concrete capability and a concrete one. It allows to localize NaaS services by describing their capabilities at different levels of localization, i.e. in terms of geographical area, provider, region, and/or availability zones.

*Definition 4.2 (**localizedIn**):* Let C_1, C_2 be two capabilities achieving the same action. C_1 *localizedIn* C_2 iff C_2 $variantOf$ C_1 and $\forall p \in properties(C_2) \setminus properties(C_1)$, $< p, MGV, specRelation > is\_a$ *LocalizationProperty*

Finally, the `specifiesResource` relation describes a relation of specification between the exposed resources of two capabilities. The specification of network resources are defined based on the model in Figure 6.

*Definition 4.3 (**specifiesResource**):* Let C_1, C_2 be two capabilities achieving the same action. C_1 *specifiesResource* C_2 iff $properties(C_1) = properties(C_2)$ and $v_{C_1}^{resourceType}$ *specifies* $v_{C_2}^{resourceType}$ and $\forall p \in properties(C_1)$ such that $p \neq resourceType$), $v_{C_1}^{p} = v_{C_2}^{p}$

Examples of these relations are shown in the sample NaaS capability-based description of Figure 7. For instance, $C_{\text{L2Connectivity}}$ bundles the two capabilities $C_{\text{L2Endpoint}}$ and $C_{\text{L2Bridging}}$. It exposes the set of resources exposed by its children. The concrete capabilities are interconnected through the `localizedIn` relation, e.g. capabilities C_4 to C_{12}. The `specifiesResource` relation is between C_2 and $C_{\text{L2Connectivity}}$, and C_3 and $C_{\text{CL3Endpoint}}$.

V. NaaS Services Description

In this section, we detail our approach for describing NaaS services based on the NaaS capability model presented in Section IV. NaaS services description consists of two steps as shown in Figure 4: capability extraction (detailed in Section V-A) and description generation (detailed in Section V-B).

A. Capability Extraction

The capability extraction component takes two inputs. The first is a catalog of network services defined by network experts. These services are defined independently of the specific vocabulary and implementation of CSPs. Therefore, their corresponding capabilities, modeled with reference to the OSI model, are abstract. In this work, network services were defined by our industrial partner. The corresponding capabilities were modeled manually and validated by network experts. The result of this step is a set of abstract capabilities, as shown in the first layer of Figure 7 (e.g., $C_{L2endPoint}$ and $C_{L2Connectivity}$).

The second input is the set of NaaS services descriptions offered by different CSPs. Different sources, such as textual descriptions and APIs documentations, can be used to extract the information required to model provider specific capabilities. For each NaaS service, we assume that the following minimal information can be extracted from the available descriptions: type of exposed resources, geographical area, provider, regions, and availability zones in which the service is available. For each NaaS service, a capability is created per each georgraphical area, region, provider, and availability zone. A sample description of these concrete capabilities is shown in the bottom layer of Figure 7 (e.g., C_{11} and C_{12}).

B. Description Generation

This component takes as input the output of the previous component: the abstract capabilities and the concrete provider specific capabilities. It then infers the set of capabilities and relations allowing to construct a hierarchical model through the following steps.

1) Capabilities Generation: The capabilities that need to be generated are concrete capabilities belonging to different localization layers. Indeed, these capabilities create the hierarchical structure of the NaaS capability model and therefore, during service discovery, they allow to navigate from the top abstract capabilities to the bottom concrete provider specific capabilities. For instance, given the capabilities C_{11} and C_{12} in Figure 7, the capability generation sub-component generates the capabilities C_4 to C_{10}. The pseudo-code of the capability

generation sub-component is shown in Algorithm 1. The input is the set of the provider specific capabilities and the output is the set of the generated capabilities. Two main operations are performed: *split* and *merge*.

Algorithm 1 Generating capabilities from provider specific concrete capabilities

Input $\mathcal{C} = \{C_1, \ldots, C_n\}$: *provider specific capabilities*
Output $\mathcal{C}_g$: *set of generated capabilities*

1: **for each** C *in* $\mathcal{C}$ **do**
2: $recursive\_split(C, \mathcal{C}_g)$
3: **end for**
4: $\mathcal{C}_g.remove\_duplicates()$
5: **for each** *localization property* p_l **do**
6: $\mathcal{C}_f \leftarrow get\_factorizable(\mathcal{C} \cup \mathcal{C}_g, p_l)$
7: $C_m \leftarrow merge(\mathcal{C}_f)$
8: $\mathcal{C}_g \leftarrow \mathcal{C}_g \cup \{C_m\}$
9: **end for**
10: **return** $\mathcal{C}_g$ =0

The *split* operation decomposes the concrete capabilities outputted by the capability extraction component into different capabilities belonging to different localization layers. This is done recursively by removing each time one localization property (Lines 1-3). The split operation is defined as follows.

*Definition 5.1 (**Split operation**):* Given a concrete capability C and a localization property p_l, the split operation generates a capability C_s such that $C_s = C.remove(p_l)$ where *remove*(p_l) removes the property p_l and its value from C.

For instance, given the capability C_{11} in Figure 7, the split operation generates recursively three capabilities by removing each time one localization property (and its corresponding value), i.e. it generates a capability without the `region` property, a capability without the `region` and `provider` properties, and a capability without the `region`, `provider`, and `geographical area` properties. This step may generate duplicate capabilities that are removed in a postprocessing step (Line 4).

The *merge* operation aggregates capabilities belonging to the same localization layer that are factorizable (Lines (5-9). Factorizable capabilities are those that, in a given localization layer, are identical, excluding the localization property of that layer. For instance, in the description of Figure 7, the capabilities C_{11} and C_{12} can be factorized since they belong to the same localization layer (i.e. availability zone layer) and they are identical except for the availabilityZone property.

*Definition 5.2 (**Factorizable capabilities**):* Given a set of capabilities $C_1, \ldots, C_n, n \geq 2$ that have the same localization property p_l. $C_1, \ldots, C_n$ are factorizable iff $C_1.remove(p_l) = \cdots = C_n.remove(p_l)$

Factorizable capabilities are merge into one capability whose variable property has as value the union of the properties values of its variants. The formal definition of the merge operation is given below.

*Definition 5.3 (**Merge operation**):* Given a set of factorizable capabilities $C_1, \ldots, C_n, n \geq 2$. The merge operation generates a capability C_m such that $C_m = \cup_{i=1}^{i=n} C_i$

In our example, C_{11} and C_{12} are aggregated into C_9. The `availabiltyZone` property of C_9 is equal to the union of the `availabilityZone` values of C_{11} and C_{12}, i.e., `usw2-az1` and `usw2-az2`. It is worth noting that factorizable capabilities are not removed from the set of capabilities and the resulting aggregated capability is simply added.

2) Relations Generation: Once the capabilities are generated, the last step consists of inferring the relations `bundles`, `localizedIn`, and `specifiesResource` between capabilities. As shown in *Definition 4.1* and *Definition 4.2*, the relations `bundles` and `localizedIn` are defined based on the relation `variantOf` which is itself defined based on two sub-relations *specifies* and *extends* (as mentioned in section II). In [11], we explained that the relation *extends* is a special case of the relation *specifies*. The relation `specifiesResource`, introduced in Definition 4.3, could also be seen as a special case of the relation *specifies*. In fact, a capability C_1 `specifiesResource` C_2 if (1) C_1 inherits all the properties of C_2, (2) for each shared property p different from the $resourceType$ property, the value of p in C_1 is equal to its value in C_2, and (3) the value of the property $resourceType$ in C_1 specifies the one in C_2. Thus, inferring the relations `bundles`, `localizedIn` and `specifiesResource` between capabilities consists mainly in inferring the *specifies* relations as described in Algorithm 2.

Algorithm 2 Inferring *specifies* between capabilities

Require: C_i , C_j : *two capabilities*
Ensure: $true \;\; if \;\; C_i$ *specifies* C_j ; $false \;\; else.$

1: $countS \leftarrow 0$ {Number of inferred specifies relations}
2: **if** ! $Properties(C_i).contains(Properties(C_j))$ **then**
3: $return \;\; false$
4: **else**
5: **for each** p *in* $Properties(C_i)$ **do**
6: $v_i \leftarrow Properties(C_i).get(p)$
7: $v_j \leftarrow Properties(C_j).get(p)$
8: **if** ! $v_i.equals(v_j)$ **then**
9: **if** $specifies(v_i, v_j)$ **then**
10: $countS \leftarrow countS + 1$
11: **else**
12: $return \;\; false$
13: **end if**
14: **end if**
15: **end for**
16: **end if**
17: $return \;\; countS$! =0

Algorithm2 takes as input two capabilities C_i and C_j and returns true if C_i specifies C_j or false if not. First, the algorithm checks if C_i inherits all the properties of C_j (line 2-3). If so, for each property p in C_i, the algorithm gets $v^p_{C_i}$ (resp. $v^p_{C_j}$) and assigns it to a variable v_i (resp. v_j) (line 6-7). If $v^p_{C_i}$ is equal to $v^p_{C_j}$ (line 8), the algorithm selects the next p in

$properties(C_i)$ and repeats the same steps as for the previous p (i.e. go back to line 5). If not, it checks if $v^p_{C_i}$ specifies $v^p_{C_j}$ (line 9). If so, it increments the number of the inferred specifies relations (line 10) then go back to line 5. The algorithm returns false if it finds a property p in $properties(C_i)$ such that $v^p_{C_j}$ is neither equal nor more specific than $v^p_{C_j}$.

Line 9 in Algorithm2 shows that the *specifies* relation between the capabilities C_i and C_j is based on a specifies relation between the values v_i and v_j of their common properties p. In our previous work [11], we defined a set of 21 rules that allows to infer the specifies relation between values of PEs. Function $specifies$ called at line 6 infer the specifies relation between v_i, the value of p in C_i, and v_j, the value of p in C_j using the proposed inference rules [4].

VI. Proof-of-Concept

In what follows, we first detail the capability extraction and modeling process that was carried out by domain experts, followed by the NaaS services description generation steps.

A. Capability Extraction and Modeling

To define abstract capabilities, we collaborated with two experts in networking system management and architecture (P1, P2). Both are currently active in their field. One of them is a researcher having more than 25 years of expertise, while the other is a manager of a start-up offering networking services. The role of these experts was to ensure the definition of abstract capabilities according to the OSI model. Five other experts that are currently active in the field of service computing and software engineering helped us to model provider specific capabilities. Each of them has at least 5 years of professional experience. The following are their profiles:

- One expert in the area of cloud computing (P3). P3 is an AWS accredited instructor.
- One expert in the area of cloud computing (P4). P4 works on designing and developing of cloud and cognitive services/APIs.
- One senior software engineer and two professors (P5, P6, P7) leading collaborative projects and having expertise in automating (cloud-)service discovery and orchestration. P7 is the designer of the capability model presented in section II.

P3 and P4 helped us in analyzing the textual descriptions and API documentations of some NaaS providers (such as AWS and Microsoft Azure), as well as in extracting the capabilities of network services localized in each provider's availability zones. Collaborating with these cloud computing experts ensures the correct analysis of the offering. P5, P6, and P7 supervised us during the modeling steps to ensure the correct application of the capability model.

In term of realization, we model the capabilities, their Property Declarations as well as their values in the form of Resource Description Framework (RDF) triplets. We also model the NaaS services capability model as an RDF ontology.

[4]details and implementation of these rules could be found at supplementary-material-2

B. NaaS Services Description Generation

1) Capabilities Generation: We implemented the `split` and `merge` operations as Java functions. Then, we implemented an algorithm that takes as input RDF descriptions of the abstract and provider-specific capabilities (modeled in section VI-A), uses the `split` and `merge` operations to generate new capabilities and returns as output an RDF knowledge base (noted *RDF-KB*). *RDF-KB* contains the RDF descriptions of the capabilities given as input and those generated. We used JENA[5] framework to get object representations of capabilities.

2) Relations Generation: We implemented an algorithm that takes as input the generated knowledge base in section VI-B1, infers the `bundles`, `specifiesResource` and `localizedIn` relations between the capabilities that are modeled in this base, add the inferred relations to this last and return it as output. To infer existing relations, the algorithm uses the inference rules that were proposed in our previous work in [11] and implemented as SPARQL queries.

VII. Preliminary Evaluation

In this section, we evaluate our approach based on a use case in which we consider an example of a cloud user who looks for locating a certain network service. Our objective is twofold. First, we aim to evaluate the usefulness of the proposed NaaS services description in the discovery of networking services based on their capabilities. We show that our description allows to answer all needs of the cloud user in the use case example. Second, we aim to measure the effectiveness of our approach in terms of accuracy (i.e., precision and recall). In the following subsections, we describe the use case example, then we discuss and analyze the experiment results.

A. Use Case

Let us consider John, a cloud user who wants to connect his on-premise machine, which has an integrated network interface card (and therefore identified by a MAC address), to his cloud environment through an L2 Virtual Private Network (VPN). Doing so, John seeks to attach his on-premise machine to a sub-network offered by a NaaS provider in order to assign, to this provider, the task of bridging packets coming from his machine. As a first step toward satisfying his need, John decided to search the World Wide Web in order to *(Q1) index all cloud networking services providing a sub-network*.

While analyzing existing API documentations, John realized that NaaS providers categorize their services by regions and availability zones. Thus, he expressed his request differently looking rather for *(Q2) indexing all cloud networking services providing a sub-network in the United States (US) region*, i.e. where his on-premise machine is located.

By searching the Web, John is indirectly trying to manually create an index of available networking services which is a time-consuming and an error-prone task especially if he is not familiar with cloud computing and network systems concepts. In fact, John will not recognize for example that

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

his need could not be satisfied by AWS. AWS bundles the sub-network resource with IP and MAC addresses which, by default, disables the bridging of packets coming from an external machine using an AWS sub-network[6].

Suppose that after recognizing that AWS could not satisfy his need, John accepted to purchase the AWS service bundle and abandon using his on-premise machine. He changed his request again looking this time for *(Q3) indexing all AWS bundles of IP address, MAC address and sub-network which are available in Oregon*.

Motivated by its offered cost, the user could decide to *(Q4) replace the already selected and in-use service by another one from Microsoft Azure realizing the same capability with no localization requirement*.

Figure 9 illustrates the description of John's queries (Q1, Q2, Q3 and Q4) according to the NaaS capability model.

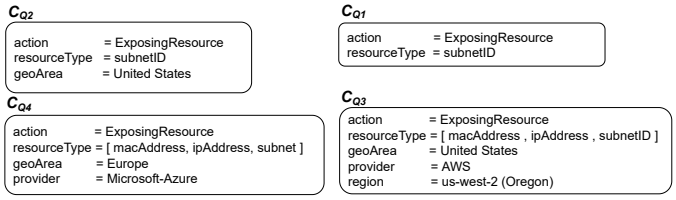

Fig. 9. John's queries described as requested capabilities according to the NaaS capability model

B. Experiments

In this section, we use our NaaS services description as a hierarchical indexing structure for answering John's requests. To browse our hierarchical description, we use EASY-M, an algorithm proposed in [16] for matching a requested capability with a number of advertised capabilities structured in a Direct Acyclic Graph (DAG) based on their semantic similarity. We tested the EASY-M algorithm to match the capabilities that are requested by John (illustrated in Figure 9) with the capabilities in our NaaS services description (illustrated in Figure 7).

C. Results

- *Browsing the hierarchy using existing algorithms*
  The results show that our NaaS services description can be traversed using the EASY-M algorithm. However, not all localization layers could be reached during the description browsing, i.e. during the service discovery process. This is foreseeable since the EASY-M algorithm was proposed to browse DAG of capabilities linked through subsumption relations. Subsumption relation is equivalent to the extends relation in our work. Thus, the algorithm will not be able to reach all localization specific capabilities in the input DAG[7].
- *Effectiveness of the hierarchical NaaS services description*
  We evaluate the effectiveness of our NaaS services description for the discovery of NS in terms of accuracy (precision and recall). To do so, we invited P7, the designer of the capability model, to manually evaluate the hierarchy browsing in order to identify true positive and false negative capabilities to compute precision and recall values.
  - (Q1): EASY-M returns 2 capabilities, only 1 of them is relevant and no relevant capabilities were missing in the returned result. Its precision is 1/2 , which tells us how valid the results are, while its recall is 1, which tells us how complete the results are.
  - (Q2): EASY-M returns 2 relevant capabilities. No relevant capabilities were missing in the returned result.
  - (Q3): EASY-M returns 2 relevant capabilities, while failing to return 2 others relevant capabilities.
  - (Q4): EASY-M returns 3 capabilities whereby, only 1/3 is relevant and 2/3 aren't. No relevant capabilities were missing in the returned result. Accuracy results in terms of precision and recall are showed in Table II.

TABLE II
ACCURACY RESULTS

Query	***EASY-M Precision***	***EASY-M Recall***
(Q1)	50%	100%
(Q2)	100%	100%
(Q3)	100%	50%
(Q4)	30%	100%
Average	**70%**	**87.5** %

VIII. RELATED WORKS

NS descriptions determine the information about a network service that are required for enabling its discovery and therefore its consumption. This section reviews the relevant work that focus on NS description (i) within the standardization bodies, (ii) within cloud providers, and (iii) within academia.

Valuable standardization initiatives (e.g., ETSI NFV [17], TOSCA-NFV [18], IETF-SFC [19]) focus only on enabling virtual network function (VNF) provisioning capabilities such as description, publication, and discovery mechanisms for VNFs. They assume that the network service/resources are already discovered and made available by the underlying virtual infrastructure managers (VIMs) such as AWS, Openstack, etc. Thus our approach can be seen complementary to this efforts since it provides a transparent description of network services that can be required by any VNF or needed to build the connection between VNFs toward creating more advanced network topologies.

As for cloud providers' approaches, the majority of them (e.g., AWS, Azure, to name a few) rely on the natural language present their network services, whereby each depends on its specific vocabulary. For instance, services with the same functional capabilities are described with different naming conventions and specifications [20]. Although the main description related to cloud NSs does not allow discovering the underlying bundling structure within NS capabilities, the accompanied description within the NS-related APIs once analyzed by domain experts enables the discovery of such

[6]This information could be checked at the following link https://docs.aws.amazon.com/vpn/latest/s2svpn/VPC_VPN.html

[7]Exploring model navigation with new or other existing algorithms is the subject of future research work.

bundling structure. Our approach discovers and describes the NS bundling structure so that users with low knowledge are able to get transparent and detailed descriptions for their NSs of interest.

As for academia approaches, initial efforts (such as [21]) tried to apply WSDL (Web Service Description Language), to describe network services as Web services. However, WSDL focuses on the description of interaction interfaces and lacks the ability to describe NS capabilities. Other attempts [1], [22], [23] focus on describing network service capabilities. [22] introduced a high-level abstraction model for network service capabilities. However, the considered capabilities include only two aspects: The "connectivity" which refers to pairs of sources and destinations between which the network transports data, and "capacity" of data transportation between each pair. The author assume that the network services already discovered, and use the capability model to support network service orchestration. [1] adopted a component-based modeling to describe cloud networking services with a focus on their QoS aspects and management/control interfaces. The model considers a network service as black-box without detailing its underlying capabilities, i.e., associated resources they may expose. [23] introduced a domain-independent ontology called VIKING that provides a generic description of the VNF capabilities from functional and non-functional perspectives. Functional characteristics of a VNF capability refer to the business functionality that a given VNF supports (e.g. video compression, data mixer), while non-functional refers to what precisely the VNF requires for proper functioning, this includes among others QoS such as response time and operation cost. Although the proposed model allows to describe some network service capabilities, it does not allow compositions over VNFs, a key important aspect required to describe network service bundling. In addition, the model focuses only on the description of network services designed by VNFs, and does not consider the specificities of network services provided by cloud NaaS providers.

IX. Conclusion

In this paper, we proposed a model for describing network services based on their capabilities. We also defined a method to generate the proposed NaaS services description. To validate our approach, we implemented a proof-of-concept and we conducted a use case-based preliminary evaluation. We aimed to evaluate the usefulness as well as the effectiveness of our approach for the discovery of network services.

The benefits gained by our NS description are promising, albeit far from perfect. As future work, we plan to investigate more advanced algorithms that allow an efficient model browsing ensuring precision-complexity trade-off. Finally, given that the capability extraction step may require extensive manual efforts, it would be useful to apply crowdsourcing mechanisms in tandem with machine learning solutions, (particularly, natural language processing techniques), to derive the capability-related knowledge from textual service descriptions of CSPs.

X. Acknowledgment

This work was partially funded by a French national program via the public private partnership project ISChyO, n° 192906122-RAPID.

References

[1] Ines Ayadi, Noemie Simoni, and Gladys Diaz. Naas: Qos-aware cloud networking services. In *2013 IEEE 12th International Symposium on Network Computing and Applications*, 2013.

[2] Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. Naas: Network-as-a-service in the cloud. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, 2012.

[3] Sergio Guidon, Michael Wicki, Thomas Bernauer, and Kay Axhausen. Transportation service bundling–for whose benefit? consumer valuation of pure bundling in the passenger transportation market. *Transportation Research Part A: Policy and Practice*, 2020.

[4] Xiong Zhang, Wei T Yue, and Wendy Hui. Bundling cloud software to fight piracy: an economic analysis. *Internet Research*, 2019.

[5] Thunyarat Amornpetchkul, Hyun-Soo Ahn, and Özge Şahin. Conditional promotions and consumer overspending. *Production and Operations Management*, 2018.

[6] Chenguang Wu, Chen Jin, and Ying-Ju Chen. Managing customer search via bundling. *Manufacturing & Service Operations Management*, 2022.

[7] Riccardo Trivisonno, Xueli An, and Qing Wei. Network slicing for 5g systems: A review from an architecture and standardization perspective. In *2017 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2017.

[8] John T Gourville and Dilip Soman. How packaging services can hurt consumption: The potential downside of bundling. *Cornell Hotel and Restaurant Administration Quarterly*, 2001.

[9] Mustafa Karataş and Zeynep Gürhan-Canli. When consumers prefer bundles with noncomplementary items to bundles with complementary items: The role of mindset abstraction. *Journal of Consumer Psychology*, 2020.

[10] Justin Pettit, Ben Pfaff, Joe Stringer, Cheng-Chun Tu, Brenden Blanco, and Alex Tessmer. Bringing platform harmony to vmware nsx, 2018.

[11] Imen Jerbi and Sami Bhiri. Definition and induction of a specification order relation between capabilities. In *2021 IEEE International Conference on Services Computing (SCC)*, 2021.

[12] Wassim Derguech, Sami Bhiri, and Edward Curry. Using ontologies for business capability modelling: describing what services and processes achieve. *The Computer Journal*, 2018.

[13] Jelena Zdravkovic, Janis Stirna, and Janis Grabis. Capability consideration in business and enterprise architecture frameworks. In *Capability Management in Digital Enterprises*. 2018.

[14] Nane Kratzke. A brief history of cloud application architectures. *Applied Sciences*, 2018.

[15] Sami Bhiri, Walid Gaaloul, Mohsen Rouached, and Manfred Hauswirth. Semantic web services for satisfying soa requirements. In *Advances in Web Semantics I*. 2008.

[16] Sonia Ben Mokhtar, Davy Preuveneers, Nikolaos Georgantas, Valérie Issarny, and Yolande Berbers. Easy: Efficient semantic service discovery in pervasive computing environments with qos and context support. *Journal of Systems and Software*, 2008.

[17] ETSI. Etsi gs nfv-man 001. Technical report, 2014.

[18] Committee Specification. Tosca simple profile for network functions virtualization (nfv) version 1.0. Technical report, 2017.

[19] IETF. Service function chaining (sfc) operation, administration and maintenance (oam) framework. Technical report, 2018.

[20] Mustafa M. Al-Sayed, Hesham A. Hassan, and Fatma A. Omara. An intelligent cloud service discovery framework. *Future Generation Computer Systems*, 2020.

[21] Qiang Duan. Automatic network service discovery and selection in virtualization-based future internet. In *2011 IEEE GLOBECOM Workshops (GC Wkshps)*, 2011.

[22] Qiang Duan. Network-as-a-service in software-defined networks for end-to-end qos provisioning. In *2014 23rd Wireless and Optical Communication Conference (WOCC)*, 2014.

[23] Nour el houda Nouar, Sami Yangui, Noura Faci, Khalil Drira, and Saïd Tazi. A semantic virtualized network functions description and discovery model. *Computer Networks*, 2021.