Matchmaking for Multi-Cloud Marketplace Application

Huynh Hoang Long[1], Nguyen Huu Duc[1], Le Trong Vinh[2]
[1] Hanoi University of Science and Technology, Hanoi, Vietnam
[2] University of Science, Vietnam National University, Hanoi, Vietnam
Correspondence: Nguyen Huu Duc, ducnh@soict.hust.edu.vn
Communication: received 19/05/2019, revised 07/07/2019, accepted 14/07/2019
Online early access: 14/07/2019, Digital Object Identifier: 10.32913/mic-ict-research.v2019.n1.854
The Area Editor coordinating the review of this article and deciding to accept it was Prof. Le Hoang Son

Abstract: Software as a Service has been developing according to the trend that leverages the advantage of multi-cloud environment to avoid vendor lock-in problem. To consume cloud resource provided by various providers, cloud software should be decomposed into components which can be deployed across different clouds. Ideally, components of a component-based cloud software are independently developed and offered through multi-cloud marketplace. To bring the highest benefit to consumers, there should be an efficient cost strategy for finding a group of compatible cloud platforms for their cloud softwares. In this paper, after redefining and developing simple formal definition for Composable Application Model (CAM), we present a matchmaking method that could be useful for verifying the correctness of software composition as well as for checking the correct deployment of a software composition on specified cloud platforms. As an illustration, we experimentally transform the cloud application model achieved by our matchmaking method into TOSCA-based specification template, which is known as a standard representation for multi-cloud applications.

Keywords: *Cloud computing, multi-cloud, multi-cloud marketplace, composable application model (CAM), software as a service, matchmaking.*

I. Introduction

In recent years, the Software as a Service delivery model (SaaS) is increasingly used and it has become a reasonable choice to consumers [1]. A simple way for a consumer to find a suitable SaaS for his needs is through a cloud marketplace where cloud softwares are developed and provided by vendors themselves and/or individual developers.

Usually, a cloud provider publishes a set of application programming interfaces (APIs) and tools to developers for implementing applications on the top of their cloud resources. Thus, cloud applications are often tightly coupled to the cloud services at different levels and restrictions. Developers are totally bound to the ecosystem of technology from the cloud provider where they want to host their on-developing application. As a result, the vendor lock-in problem arises, SaaS development is single, isolated and discursive in narrow ranges. There are some limitations of that approach which are identified as follows: (i) the lack of integration mechanisms among cloud providers making the development of a cloud application whose components are designated to be hosted on different cloud infrastructures become infeasible; (ii) the capability limit of cloud providers preventing the development of large-scale software system which requires a tremendous amount of resources; (iii) lack of standards supporting a uniform development of cloud applications.

To overcome these limitations, cloud application is designed in a fashion that they consume various types of resource offered by cloud providers. A good idea is that a cloud application in a multi-cloud environment should be decomposed into different components which can be developed and distributed across heterogeneous cloud platforms, it was mentioned in some papers such as Guillén *et al.* [2] and Baryannis *et al.* [3]. This view becomes clearer through Copil *et al.* [4] proposed a generic composition model of cloud service to represent the entire cloud application or system which can be further decomposed into service topologies and service units. Service units represent individual software or cloud offering services, and can be grouped into a cloud service topology for establishing semantically connections. Also from this point of view and combine with the component-based approach presented in [5–8], in our previous studies [9] and [10], we proposed a component-based cloud application model in which the cloud software of multi-cloud marketplace is composed from software components, each of them can be independently developed by different developers and can reside in different clouds.

Developers could be free to develop cloud software components as they wish without being dependent on any cloud API of a certain cloud provider, and cloud software development could be not tightly coupled to any cloud provider. This brings some benefits as follows: (i) this enables the distribution of software components on heterogeneous multi-cloud platforms; (ii) this separates the cloud software development from cloud providers and so minimizes vendor lock-in problem; (iii) this helps consumers exploit the most advanced features from a cloud provider by only choosing its best platform services to host some appropriate components instead of a full SaaS solution; (iv) developers do not need to pay too much attention to cloud infrastructure information runtime parameters such as CPU, RAM, Storage, middleware, network, etc.

In this approach, cloud software development is separated from cloud providers. Thus, we need an efficient way to make sure that a cloud software and its underlying execution platform are compatible. There are some studies related to this issue. Guillén *et al.* [2] presented a cloud development framework for developing managing cloud application that is separated from the source code and managed, source code of application could be deployed on multiple cloud platforms. The framework is intended to be applied upon applications targeted towards IaaS and PaaS clouds. Baryannis *et al.* [3] presented a cloud service composition approach for multi-cloud applications so that would be able to find the most optimal resource by different cloud providers for satisfying all end-user requirements. Kolb [11] introduced an approach for matching web application among PaaS providers based on their profiles. The matchmaking is successful if and only if all web application properties exactly match with a compared PaaS profile. Zeng *et al.* [12] proposed a Wordnet-based matching algorithm that considers the semantic similarity of the concepts mapping to the I/O parameters of the services. QoS information is utilized to rank the search. Zilci [13] introduced an idea to compare services based on their quality of service (QoS) requirements in cloud service marketplaces by using constraint programming to solve the service matchmaking problem.

Garg *et al.* [14] proposed a framework and a mechanism for ranking Cloud services based on their performance on QoS properties and the weights given to these properties, by exploiting an Analytical Hierarchy Process (AHP)-based algorithm. The matchmaking solution of Cloud4SOA proposed by D'Andria *et al.* [15], allows searching among the existing PaaS offerings those that best match the user requirements and ranks them based on the number of satisfied user preferences. The matchmaking mechanism uses semantic technologies to align the user requirements and the compatible PaaS offerings. García-Gómez *et al.* [16] introduced an ideal for matchmaking via the use of blueprints. A set of alternative Abstract Resolved Blueprints (ARBs) are created in design process of software developer based on the offerings of other available third-party source blueprints that can be queried and purchased from the marketplace. Each ARB is a possible combination of blueprints constituting a Cloud application. Elshareef [17] propose a matchmaking strategy between the incoming requests and various resources in the cloud environment to satisfy the requirements of users and to load balance the workload on resources.

An interesting approach of incorporating Domain Specific Languages (DSL) which facilitate to model cloud application and support between matchmaking deployment requirements and infrastructure descriptions were presented by Sledziewski *et al.* [18] and Brandtzaeg *et al.* [19]. Baryannis [3] introduced an ideal to matchmaking application with cloud infrastructure. Constraint satisfaction rules are employed in order to match requirements with existing infrastructure descriptions/capabilities, resulting in one or more proposed plans for deployment. The matchmaking process is conducted through the matchmaker engine of which inputs are requirement specifications provided from application developer, cloud infrastructure descriptions are derived from knowledge base, and constraint satisfaction rules are derived from a rule base. In addition, there are some open source solutions such as Heat [20] and Juju [21] which describes and model composite cloud applications and support deploying them on several cloud providers. However, these proposals could only be done on a single cloud. An effort for distributing cloud application on many clouds is showed in a recent study which was presented by Saatkamp [22], he introduced the Split and Match Method for TOSCA specification. His method splits TOSCA topology according to the specified targets providers and matches the resulting topology fragments with the cloud provider services to support an automated deployment of the application to multiple clouds. The goal of the Split and Match Method is to enable a customized distribution of the components of an application to different cloud providers.

In general, these matching solutions have limitations as follows: (i) the compatibility and the dependencies among components in a multi-component cloud application is not considered; (ii) these matching solutions have not been fully defined in a specific multi-cloud application pattern; (iii) a cloud software and its runtime systems were not described in a standard specification model so that cloud software components can be distributed across various clouds.

In this paper, we present a novel matchmaking method for cloud application of multi-cloud marketplace as the

next step of our previous works in [9] and [10]. Our contributions are summarized as follows:

- We improve Composable Application Model (CAM) proposed in [9] that organizes multi-component cloud software in a nested structure. Then, we define a simple formal definition for abstract model of CAM.
- We propose an approach to explicitly specify the technological constraints among software components and between a component and its expected underlying runtime platforms by matching rules. We use these constraints to verify the correctness of software composition and deployment.
- We develop a matchmaking algorithm to retrieve a group of compatible cloud platforms for a component-based cloud software that suits with a criteria set ordered by cloud consumer.

Multi-cloud marketplace also plays a role of a service broker. Therefore, our matchmaking method aims to create an effective broker mechanism that support consumers to find suitable cloud platforms that satisfy their demands for each cloud software that independently developed by developers and released through multi-cloud marketplace. The output of this work is a group of compatible cloud platforms for a component-based cloud software; on this basis, a multi-cloud application defined by CAM is made up. Relying on a case study a case study presented in Section IV, we validate our proposal by experimenting a transformation from a specification of CAM into TOSCA-based specification template which is used for representing multi-cloud application structure.

The rest of the paper is organized as follows. Section II introduces our work for improving Composable Application Model. Matchmaking algorithm is presented in Section III. The illustration of transformation from CAM specification to TOSCA application template is shown in Section IV. Finally, we conclude and summarize the contributions of the paper in Section V.

II. Composable Application Model

In our approach, the development of cloud software should not be bound to any specific cloud provider. A cloud software should be able to deploy on compatible cloud platforms to form a cloud application system without having to re-engineer or to re-develop. Thus, we divide a cloud application into two separated parts: a cloud software and an underlying runtime system which is provided by specific cloud providers. We also adopt the concept of component-based application model in which a cloud software is decomposed into software components, each of them can be hosted on separated cloud platforms. This

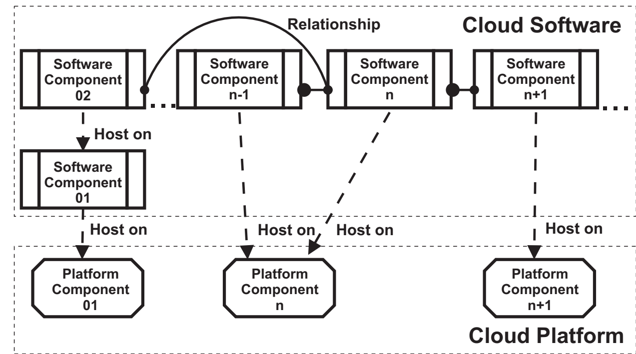

Figure 1. Cloud Application.

component-based approach requires the dependency among software components to be explicitly clarified for the sake of separate development and testing of the components. We call this cloud application model is *Composable Application Model* (CAM) and are going to redefine the model in following sub-sections.

1. General Concepts

Before going to further details of CAM, we need to clarify several related concepts as follows:

1) Cloud application: A cloud application is a runtime service that offers specific business functionalities. To customers, this refers to a similar concept to Software as a Service (SaaS). Internally, a cloud application may be constructed as a distributed system whose elements are compute servers running specific softwares and located at specific cloud providers.

We divide a cloud application into two separated parts: a cloud software and an underlying runtime system (which is a single or a group of cloud platforms) as illustrated in Figure 1. This separation allows us avoiding vendor lock-in problem. The software part can be developed separately from the underlying cloud platforms. All dependencies between the cloud software and the cloud platforms should be explicitly described as they will be used to check the compatibility at the time of deployment. Depends on the number of cloud platforms are used, we classify cloud applications into two categories:

- *Single-platform application:* A single platform application is a cloud application operating on a single cloud platform.
- *Multi-platform application:* a multi-platform application is a cloud application operating on a group of cloud platforms which may be distributed among different cloud providers.

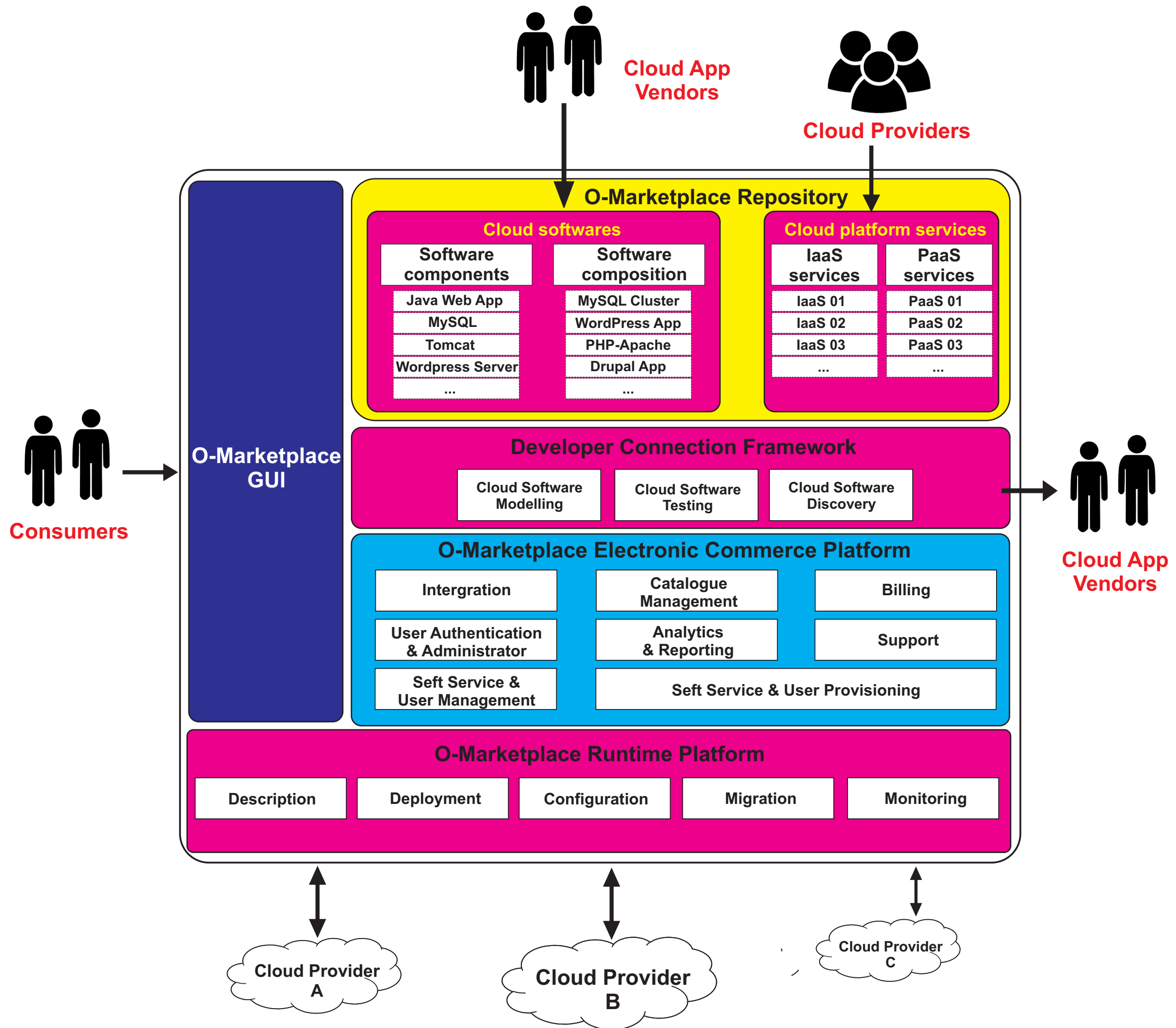

Figure 2. O-marketplace.

2) Cloud software: A cloud software is a collection of artifacts (i.e., code and data) grouped as a software bundle in a standard format. There are several types of cloud software. In the simplest form, cloud software is just a single component which can resides in a single cloud platform. In more complex situations, a cloud software is a composition of software components that could be distributed across many cloud platforms. With this component-based approach, developers can build up an application by just incorporating existing components into their own software solution. This would help us to increase re-usability of the software components and reduces unnecessary efforts in the software development process.

Cloud Software Component, or *component* for short, is a basic element of CAM. Components are used to build up more complex cloud softwares. Beside the content of code and data, the definition of a component should specify necessary conditions for the component to be integrated with others and to be deployed on a specific platform and/or a platform group. A component may be an atomic entity or a combination of other components. We classify cloud software components into three following types:

- *Simple component* type stands for a set of atomic entities, the building blocks for cloud softwares. A simple component packs its code and data together with the necessary requirements for running properly. It also explicitly defines the capabilities which other components may need when combining to form a cloud software.
- *Cloud Software Stack*, or *stack* for short, is a special kind of multi-component cloud composition. It defines a sequence of cloud software components in which a component consumes the software services provided by the following components in the sequence, and in the same time it sets up necessary environment for the previous components in the sequence. Dependencies

among layers of components should be satisfied when developing a software stack. A stack can only be deployed on a single platform.

- *Cloud Software Composition*, or *composition* for short, is a more generic multi-component cloud composition. Different from cloud software stack, which can only be deployed on a single platform, a cloud software composition can host its components on multiple platforms (may be from different cloud providers). Dependencies among software components should be satisfied according to the composition specification.

3) Cloud platform: Cloud platform is a kind of runtime system provided by cloud providers for hosting cloud software components. We also refer the term *cloud platform* as a model of delivery IaaS (Infrastructure as a Service) and/or PaaS (Platform as a Service). Various cloud providers can develop and provide the same kind of cloud platform service, but with different price, QoS, resource capacity, policies, etc. In this paper, we only consider the technical capabilities of cloud platforms for matching them with the proposed software model.

4) Cloud marketplace: Cloud marketplace is known as a kind of marketplace for selling or leasing cloud softwares which can be able to deploy on and to deliver from existing cloud providers. Most current cloud marketplaces are operated by a single cloud provider. They normally provide a complete application solution and/or a set of proprietary tools and APIs for developers to develop softwares on the top of their cloud environment. That obviously leads to the vendor lock-in problem.

To avoid this problem, in [10], we proposed a model of multi-cloud marketplace, which was called *O-Marketplace* (Figure 2). O-Marketplace targets to an open environment for developers who do not own cloud infrastructure, especially to startups whose resources are limited but their creativity is very plentiful. Developers are free to evolve their cloud software without any technology restriction from cloud providers. In this model, cloud software components are developed by individual developers, are published and sold on the marketplace, and can be able to deploy on compatible cloud platforms suggested by the marketplace.

To facilitate the suggestion feature of the marketplace, we develop a matchmaking algorithm. This algorithm would help us to find suitable platforms to host a specific cloud software by matching the platform requirements of the cloud software to the capabilities of the cloud platforms registered in their profiles. More details of the algorithm will be given in the Section III.

2. Matching Definition

In CAM, a cloud software is composed from independent software components in a composition, a software component is deployed on a platform component which is a cloud resource service bundle such as IaaS or PaaS offered from different cloud providers. To depict the relationships of components within a cloud application modeled by CAM, we specify two types of dependencies in a cloud application model: *software dependence* and *platform dependence*. Software dependence denotes the interconnection between two software components. The platform dependence denote deployment capability of a pair of software components or a pair of a software component and a cloud platform. We define two elements to make up a dependence: *Requirement* and *Capability*. Relying on these elements, matching rules is constructed to denote the dependencies within a cloud application. Requirement specification, capability specification, and matching condition is defined as follows:

- Requirement specification: *requirement* denotes the dependency of a component on other another component. It poses the necessary conditions for component to perform its functionality and behaviors. Requested interfaces and properties of a component are encapsulated in requirement. We specify two types of requirement: *software requirement* and *platform requirement*. Software requirement *(sreq)* is a constraint on the functionality and behavior of another component. Platform requirement *(preq)* is a constraint on the environment and technology of another component.
- Capability specification: *capability* denotes the available capacity that can meet the request from outside. its functionality and behavior are performed when it satisfies the condition from another external component. Responded interfaces and properties of a component are encapsulated in capability. We specify two types of capability: *software capability* and *platform capability*. Software capability *(scap)* is a supply of the functionality and behavior of a component. Platform capability *(pcap)* is a supply of the environment and technology of a component.
- Matching condition: we define a dependence between two components is valid if *requirement* of a component fully match with *capability* of the other.

3. An Abstract Model of CAM

Our ultimate goal is to develop a full specification for the proposed composable cloud application. This development requires a great effort on various aspects including: (1) defining standards and frameworks for coding cloud software components; (2) specifying the deployment and management operations of composable cloud application; (3)

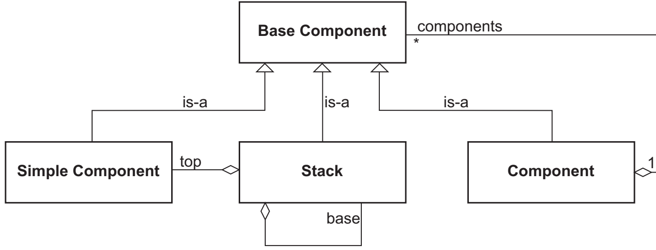

Figure 3. Overall structure of CAM.

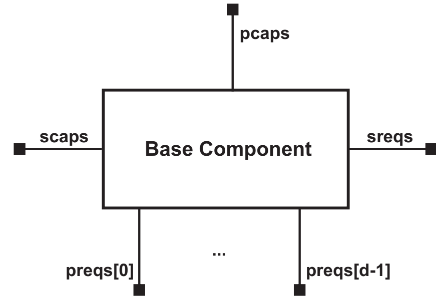

Figure 4. Base component.

verifying the correctness of cloud software composition; (4) matching composable cloud software to existing cloud platforms. In this paper, we limit our study to the latter two problems. We develop an abstract model of CAM in which we ignore the implementation details of the components but focus on the relationship among components inside a software composition and the relationship between a component and its underlying platforms.

1) Overall structure: We define the abstract model of CAM as a nested structure where each component may be a composition of other components. Some of them may again be compositions of other smaller components. This approach helps increase the degree of re-usability since the implementation of a complex component may be reused in other compositions. This also motivates the cloud software marketplace where developers can develop and sell their small components instead of complete software solutions.

According to the concepts presented in section II-C, we classify components in the model into the following three types: (1) *Simple Component*, (2) *Stack*, and (3) *Composition*. Figure 3 shows the overall structure of CAM and the relationship among these three types of the components. Details for each type of component will be explained in the following sections.

2) Base component: As shown in Figure 3, three types of software components are modeled as sub-types of the Base Component. This formalism allows us to treat components of different types uniformly as specializations of the Base Component. Each component has its own properties. We use the dot-notation to refer to a property of a component. For example, $X.y$ refers to the property y of the component X. Since we focus on the relationship among components inside a composition and the relationship between a component and its underlying platforms, the properties of a component should explicitly describe its requirements and capabilities. More specifically, as shown in Figure 4, a component should have at least the following properties: (1) description of software services provided by the component (software capabilities - *scaps*), (2) the establishment of runtime environment providing to upper components in a stack manner (platform capabilities - *pcaps*), (3) descriptions of extra software services required for running the component properly (software requirements - *sreqs*), (4) and the necessary requirements for underlying platforms to host the component (platform requirements - *preqs*). The specifications of components, i.e,. the description of component's properties, are used for verifying the correctness of software composition and for checking the compatibility between a component and its underlying platforms. These specifications are written by developers, and they are not necessary to collate the actual implementation of the components. Instead, developers can select the requirements and capabilities of a component from a predefined set of terms.

Note that a component may be hosted on multiple cloud platforms from different cloud providers. The platform requirements should be specified for each platform. We call *degree* of a component to be the total number of underlying platforms for hosting the component. Thus, the platform requirements for the platform i is defined as $preqs[i]$.

For a convenient development, we also use the following derived properties:

- *preqs*: set of all platform requirements,

$$preqs = \bigcup_{i \in [0 \ldots d-1]} preqs[i].$$

- *reqs*: set of all software requirements and platform requirement,

$$reqs = preqs \bigcup sreqs.$$

- *caps*: set of all software requirements and platform requirement,

$$caps = pcaps \bigcup scaps.$$

3) Simple Component: Simple component is an atomic type of components in CAM. In a full definition, a simple component should specify its code and data as well as the necessary specification of requirements and capabilities. A simple component is hosted on a single platform, i.e., ($degree = 1$). If the set of requirements of a simple

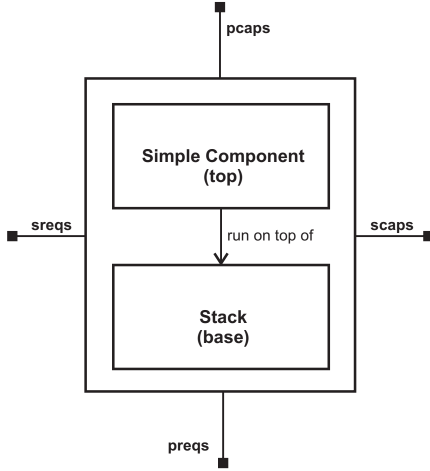

Figure 5. Cloud software stack.

component is empty ($regs = \emptyset$), the component can represent a complete cloud software. In general cases, a simple component can combine with other components to form a more complex component (i.e., a stack or a composition).

4) Stack: A cloud software stack, or stack for short, denotes a sequence of simple components deployed on top of each other vertically. For simplicity, we define a Stack in nested manner. Each stack has two elements: (1) *top* element is a simple component lay on top of the stack, (2) and *base* which is either a simple component or another stack representing the remain components in the sequence. Similar simple component, a software stack can only be deployed on a single platform ($degree = 1$). Beside the advantage of re-usability, the combination of multiple components into a single form of stack would help us reducing the cost of deployment and management by combining corresponding operations from the stack's elements. We will discuss this interesting problem in another study from our on-going research.

When creating a stack, a developer must specify the *top* element and the *base* element (Figure 5). The developer must also specify the stack requirements and capabilities (i.e., *sreqs*, *preqs*, *scaps*, and *pcaps*), like those mentioned in the Base Component. A correct combination of a stack S should satisfy the following validation rules:

$$\begin{aligned} S.top.preqs &\subseteq S.base.pcaps \\ S.pcaps &\subseteq S.top.pcaps \bigcup S.base.pcaps \\ S.preqs &\supseteq S.base.preqs \\ S.sreqs &\supseteq S.top.sreqs \bigcup S.base.sreqs \\ S.scaps &\subseteq S.top.scaps \bigcup S.base.scaps \end{aligned}$$

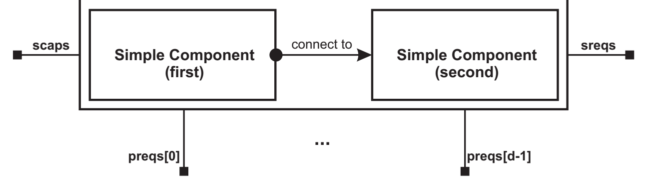

Figure 6. Cloud software composition.

5) Composition: Cloud software composition, or composition for short, denotes a set of software components combined in a single form of component to hide the complexity of its own inter-dependency. A software composition is considered as a directed graph whose vertex are either a simple component or a stack. An edge from a component A to a component B in a composition specifies that the component A consumes services provided by the component B at runtime.

Similar with stack, we apply the nested structure for defining compositions. Each composition consists of a set of components and some of them may be other compositions. For simplicity, we define a composition with two elements (Figure 6): the $first$, and the $second$. The interdependence of a composition is restricted to the software dependence of the $first$ to the $second$. Degree of the composition is calculated by the sum of degrees of the $first$ and the $second$. When creating a composition, developers need to specify its external requirements and capabilities. At the moment, we do not allow extra components to lay on top of a composition. So, the property *pcaps* should be empty. The following validation rules should be applied for verifying the correctness of a composition C:

$$\begin{aligned} C.scaps &\subseteq C.first.scaps \bigcup C.second.scaps \\ C.sreqs &\supseteq C.second.sreqs \bigcup \\ &\quad (C.first.sreqs \backslash C.second.scaps) \\ C.preqs[i] &\supseteq C.first.preqs[i] \\ &\quad \forall i \in [0 \ldots (C.first.degree - 1)] \\ C.preqs[i] &\supseteq C.second.pregs[i - d_1] \\ &\quad \forall i \in [d_1 \ldots (d_1 + d_2 - 1)] \\ &\quad d_1 = C.first.degree \\ &\quad d_2 = C.second.degree \end{aligned}$$

6) Cloud platform: Cloud platform, or platform for short, is used for modeling the cloud platform profile in a cloud marketplace. A specification of a single platform P should describe its capabilities (*pcaps*) in order to match with the platform requirements (*preqs*) of cloud software components which are aimed to deploy on it. Practically, the specification of a platform is provided by the cloud provider

or a service broker such as cloud marketplace. In this case, developers and cloud providers should agree on the same set of predefined terms of the platform requirements and capabilities.

A cloud software composition may require more than one platform. We define *platform group* as an order set of cloud platforms, and *compatible platform group* as a platform group that satisfies the platform requirements of a composition. The following predicates are used for checking this property.

- $compatibleWithReqs(P, preqs)$ if $P.pcaps \supseteq preqs$
- $compatibleWith(PG = [P_0, P_{d-1}], C)$
  if $d = C.degree$ and
  $\forall i \in [0 \ldots d-1].compatibleWithReqs(P_i, C.preqs[i])$

At the time of software deployment, platform compatibility should be checked to ensure that the software component will work fine on the underlying platform. The platform compatibility check is also a basis for the matchmaking algorithm presented in the next section.

III. Matchmaking Method for Cloud Application

In multi-cloud marketplace, besides providing a place for publishing and selling cloud softwares, the marketplace may also play a role of a service broker. Based on the specification of platform requirements of cloud software component, the marketplace would be able to suggest optimal solutions for renting compatible cloud platforms. The term "optimal" could be defined according to the user needs. Some user wants to find a platform solution with smallest price. Others may need a solution with the best quality of service. In the context of this paper, we do not dig deeper in this aspect.

We assume that a platform solution for hosting a cloud software component C is a compatible platform group PG to the component C, i.e. $compatibleWith(PG, C)$ is satisfied. There may be more than one solution existed since several cloud providers may offer a same kind of platform (but with different price and quality of service). Choosing an optimal solution means that we need to make a comparison among the solutions based on a designated cost function. We call $Cost(PG)$ for the cost function of a platform solution, i.e., the platform group PG. Some reasonable examples for the cost function would be:

- $Cost([P_0, \ldots, P_{d-1}]) = \sum_{i=0}^{d-1} Price(P_i)$. This cost function helps customers select a platform solution with optimal price.
- $Cost([P_0, \ldots, P_{d-1}]) = |\{P_0, \ldots, P_{d-1}\}|$. This cost function helps customers select a platform solution with a smallest number of platforms will be used.

Algorithm 1: Matchmaking(C, PS)

```
1  Inputs:
2      - A cloud software component C.
3      - A set of cloud platforms PS.
4  Output:
5      - A compatible platform group PG to the component
6        C whose members are selected from PS.
7  begin
8      d = C.degree ;
9      for i = 1 to d − 1 do
10         CPL[i] = ∅;
11         foreach P in PS do
12             if compatibleWithReqs(P, C.preqs[i]) then
13                 CPL[i] = CPL[i] ∪ {P};
14             end
15         end
16     end
17     min_cost = INFINITY;
18     best_solution = NULL;
19     Let S = (CPL[0] × CPL[1] × ... × CPL[d − 1]);
20     foreach PG in S do
21         if Cost(PG) < min_cost then
22             min_cost = Cost(PG);
23             best_solution = PG;
24         end
25     end
26     return best_solution;
27 end
```

Using such a cost function, we develop a matchmaking algorithm for suggesting compatible platform solution as presented in Algorithm 1.

The first part of the algorithm (lines 8-16) calculates a list of compatible cloud platforms $CPL[i]$ for each set platform requirements $preqs[i]$ of the component C. In the second part (lines 17-25), the algorithm select the best platform solution among all possible solutions, the Cartesian product $\mathcal{S} = CPL[0] \times CPL[1] \times \cdots \times CPL[d-1]$.

Correctness: Correctness of the algorithm is obvious since the optimal solution $best_solution$ is selected from a set of all compatible platform groups to the component C, i.e., $\mathcal{S} = CPL[0] \times CPL[1] \times \cdots \times CPL[d-1]$.

Performance analysis: Let n be the number of cloud platforms in PS. The first part of the algorithm (lines 8-16) requires $d \times n$ compatibility checks. Thus the complexity of this part is $O(d \times n)$. The second part of the algorithm (lines 17-25) requires n^d calculation for the $Cost$ function. Assume that $O(f(n))$ is the complexity of the $Cost$ function. The complexity of the second part would be $O(n^d \times f(n))$. In summary, the complexity of the algorithm is $O(n^d \times f(n))$.

The algorithm is rather simple but flexible since we always have chance to change the cost function according to the consumer needs.

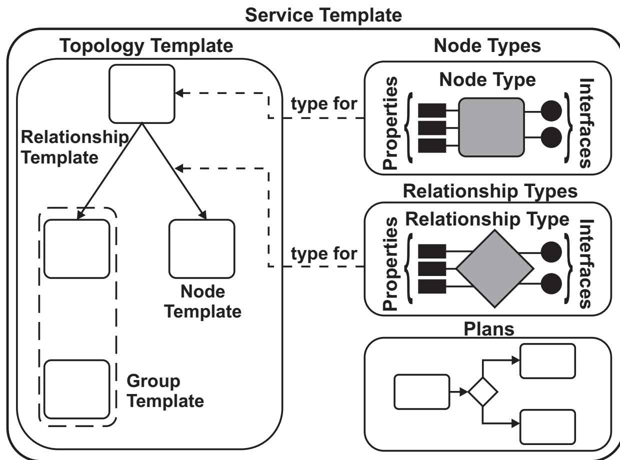

Figure 7. Structural elements of a service template and their relations.

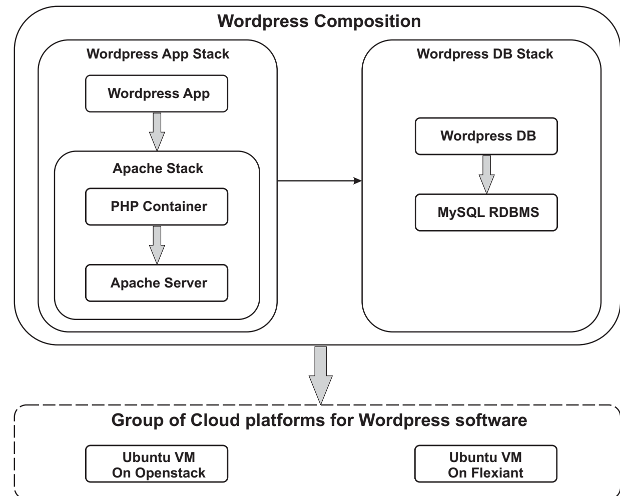

Figure 8. Wordpress Application represented in CAM.

IV. Mapping to TOSCA

In this section, we demonstrate the feasibility of our proposed composable application model (CAM) by translating a specification of CAM into TOSCA topology template, a standard specification for creating multi-cloud application.

1. Topology and Orchestration Specification for Cloud Application

Topology and Orchestration Specification for Cloud Applications (TOSCA) [23] is an open standard built by OASIS that defines the inter-operable description of cloud application hosted on the cloud; including its components, relationships, dependencies, requirements, and capabilities. TOSCA enables portability and automated management across cloud providers which have different underlying platform or infrastructure. Thus, the portable deployment of cloud applications could be done on any compliant cloud because the relationships among parts of the service and the operational behaviours of these services are independently described with any cloud provider.

TOSCA provides a meta model and packaging format to realize the automated implementation, deployment, configuration, management, and orchestration of cloud applications in an automated manner through a combination of two core concepts: Topology Templates and Management Plans (Figure 7). In order to support such packaging format. Target clouds have to have runtime environments that support TOSCA standard. Both IaaS and PaaS are feasible with this approach.

TOSCA is a middle-level language for the specification of the topology and orchestration of an IT service in the form of a service template. It is very efficient in software provisioning, deployment and management of cloud application. It enhances the portability of cloud application and has been supported by many partners like IBM, Reb Hat, Cisco, Citrix, EMC, etc.

Since TOSCA is currently a mature standard with a lot of supporting tools for designing and orchestration. In this paper, we select this standard as a target of the translation to demonstrate the feasibility of our proposed composable application model.

1) Method Overview: The major differences between CAM specification and TOSCA specification are:

- TOSCA specification represents a complete blueprint of a cloud application system including the software and the cloud infrastructure/platform parts while CAM separates these two parts into independent entities.
- TOSCA expresses the internal structure of a cloud service by a topology template which is a graph representing software/hardware components (as nodes), and relationship among these components (as edges). CAM represents a cloud application in a nested manner.

Thus, we design the translation from a CAM specification to TOSCA specification in a process of the following steps:

- First, we combine the cloud software and its compatible platform group into a single form of CAM specification.
- Second, we flatten the nested structure of the CAM specification into a graph representation. Nodes of the graph correspond to simple components and platform, and edges of the graph represent the relationships among simple components and platforms. This translation is done recursively based on the nested structure

```
Simple components:
   - Apache Server:
        type: software
        preqs: [Linux]
        pcaps: [Apache]
   - PhP Container:
        type: software
        preqs: [Apache]
        pcaps: [PhP, Apache]
   - WordpressApp:
        type: software
        preqs: [Apache, PhP]
        sreqs: [WordpressDB]
   - MySQLDB:
        type: software
        preqs: [Linux]
        pcaps: [MySQL]
   - WordpressDB:
        type: software
        preqs: [MySQL]
        scaps: [WordpressDB]
```

```
Stacks:
   - PhP-Apache Stack:
        type: stack
        top: PhP Container
        base: Apache Server
        preqs: [Linux]
        pcaps: [Apache, PhP]
   - WordpressApp Stack:
        type: stack
        top: WordpressApp
        base: Php-Apache Stack
        preqs: [Linux]
        sreqs: [WordpressDB]
   - WordpressDB Stack:
        type: stack
        top: WordpressDB
        base: MySQLDB
        preqs: [Linux]
        scaps: [WordpressDB]
```

```
Compositions:
   - Wordpress Composition:
        type: composition
        first: WordpressApp Stack
        second: WordpressDB Stack
        preqs:
            - [Linux]
            - [Linux]
```

```
Platforms:
    - UbuntuVM on Openstack:
        Type: os
        provider: OpenStack
        pcaps: [Linux]
    - UbuntuVM on Flexiant:
        Type: os
        provider: Flexiant
        pcaps: [Linux]
```

```
Wordpress Application:
   software: Wordpress Composition
   platforms: [UbuntuVM, UbuntuVM]
```

Figure 9. Wordpress application specification.

```
Nodes:
   - Apache Server:
        type: SimpleComponent
        preqs: [Linux]
        pcaps: [Apache]
   - PhP Container:
        type: SimpleComponent
        preqs: [Apache]
        pcaps: [PhP, Apache]
   - WordpressApp:
        type: SimpleComponent
        preqs: [Apache, PhP]
        sreqs: [WordpressDB]
   - MySQLDB:
        type: SimpleComponent
        preqs: [Linux]
        pcaps: [MySQL]
   - WordpressDB:
        type: SimpleComponent
        preqs: [MySQL]
        scaps: [WordpressDB]
   - UbuntuVMonOpenStack:
        provider: OpenStack
        pcaps: [Linux]
   - UbuntuVMonFlexiant:
        provider: Flexiant
        pcaps: [Linux]
```

```
Relationships:
   - from: WordpressApp
     to: PhP Container
     type: host-on
   - from: PhP Container
     to: Apache Server
     type: host-on
   - from: Apache Server
     to: UbuntuVMonOpenStack
     type: host-on
   - from: WordpressDB
     to: MySQLRDBMS
     type: host-on
   - from: MySQLRDBMS
     to: UbuntuVMonFlexiant
     type: host-on
   - from: WordpressApp
     to: WordpressDB
     type: connect-to
```

Figure 10. Wordpress application graph.

of CAM specification. The relationships are derived from the specifications of stacks and compositions.

- Finally, we map the resulting graph into TOSCA topology template. Here, the most cryptic part is that we have to map each node of the graph to a node template of TOSCA, and each edge of the graph to a relationship template of TOSCA.

The following sub-section demonstrating this process in a particular example – the Wordpress application.

2. Experiment with Wordpress Application

A typical Wordpress application is represented in CAM as shown in Figure 8. The specification of the Wordpress application is written in YAML as shown in Figure 9. We temporary ignore all implementation details of the cloud software components but only focus on the specification of their requirements and capabilities. If a property of a component is missed from the specification, we consider the value of this property is null or empty.

In the second step of the translation process, we flatten the Wordpress Application into a graph representation which is depicted in Figure 10. This translation ignores the specifications of requirements and capabilities of software stacks and software compositions. They are only used for verifying the correctness of the combination and for checking the compatibility between the software composition and its underlying platforms.

Finally, we map the graph representation to TOSCA topology template as shown in the following XML code:

```
<?xml version="1.0"?>
<ns2:Definitions  id="Wordpress"
      xmlns:ns2="http://docs.oasis-open.org/tosca/ns/2011/12"
      name="Wordpress">
<ns2:ServiceTemplate id="WordpressTopology">
<ns2:TopologyTemplate>
<ns2:RelationshipTemplate
      id="WordpressApp_HostOn_PHPContainer"
      type="HOSTON">
<ns2:SourceElement ref="PhP Container"/>
<ns2:TargetElement ref="WordpressApp"/>
</ns2:RelationshipTemplate>
<ns2:RelationshipTemplate
      id="PHPContainer_HostOn_ApacheServer"
      type="HOSTON">
<ns2:SourceElement ref="Apache Server"/>
<ns2:TargetElement ref="PhP Container"/>
</ns2:RelationshipTemplate>
<ns2:RelationshipTemplate
      id="ApacheServer_HostOn_UbuntuVM"
      type="HOSTON">
<ns2:SourceElement ref="UbuntuVM"/>
<ns2:TargetElement ref="Apache Server"/>
</ns2:RelationshipTemplate>
<ns2:RelationshipTemplate
      id="WordpressDB_HostOn_MySQLDB"
      type="HOSTON">
<ns2:SourceElement ref="MySQLDB"/>
<ns2:TargetElement ref="WordpressDB"/>
</ns2:RelationshipTemplate>
<ns2:RelationshipTemplate
      id="MySQLDB_HostOn_UbuntuVM" type="HOSTON">
<ns2:SourceElement ref="UbuntuVM"/>
<ns2:TargetElement ref="MySQLDB"/>
</ns2:RelationshipTemplate>
<ns2:RelationshipTemplate
      id="WordpressApp_ConnectTo_WordpressDB"
      type="CONNECTTO">
<ns2:SourceElement ref="MySQLDB"/>
<ns2:TargetElement ref="WordpressApp"/>
</ns2:RelationshipTemplate>
<ns2:NodeTemplate id="UbuntuVM" type="os" >
```

```
...
</ns2:NodeTemplate>
<ns2:NodeTemplate id="Apache Server" type="software">
...
</ns2:NodeTemplate>
<ns2:NodeTemplate id="PhP Container" type="software">
...
</ns2:NodeTemplate>
<ns2:NodeTemplate id="WordpressApp" type="software">
...
</ns2:NodeTemplate>
<ns2:NodeTemplate id="WordpressDB" type="software">
...
</ns2:NodeTemplate>
<ns2:NodeTemplate id="MySQLDB" type="software">
...
</ns2:NodeTemplate>
</ns2:TopologyTemplate>
</ns2:ServiceTemplate>
<ns2:ArtifactTemplate
    id="Artifact_93f8753b-17ab-43c5-8f11-e3ec98fe3224"
    type="sh">
...
</ns2: ArtifactTemplate >
...
</ns2: Definitions >
```

As seen in the code above, all nodes of the graph are mapped to Node Template of TOSCA, implementation details of the Node Template are omitted from the specification for brevity. Edges of the graph are mapped to Relationship Template. According to the original of the edges, i.e., from a stack or from a composition, the type of corresponding Relationship template will be given as HOSTON or CONNECTTO.

Although CAM has not been fully developed, this demonstration has proved the feasibility of the proposal with distinct advantages.

V. Conclusion

In this study, we present a matchmaking method that supports to find suitable cloud platforms, whose profiles registered in a multi-cloud marketplace, to multi-component cloud software. The goal of this work is to verify compatibility among the components within a component-based cloud application, as well as between the components and their underlying runtime platforms. To implement this idea, firstly, we redefine *Composable Application Model* (CAM) that organizes multi-component cloud software in a nested structure. Secondly, we develop an abstract model of CAM which only covers the inter-dependency of the components. We proposed validation rules for the CAM components based on that type of dependency. Thirdly, We proposed matchmaking algorithm which is the core of brokerage mechanism of multi-cloud marketplace. Finally, our proposal is validated by experimenting a transformation from a specification of CAM into TOSCA specification.

To sum up, our work creates an effective brokerage mechanism to retrieve optimal compatible platform solutions for a specific multi-cloud software according to consumer's demand in the context of O-Marketplace. This result is a premise for our future research related to Service Level Agreement (SLA) in multi-cloud environment [24].

Acknowledgement

This work is supported by Hanoi University of Science and Technology under Project "Energy-aware workflow service in cloud computing environment", No. T2017-PC-077. Simulations and technical realization were achieved on the hardware equipment at the Center for Data and Computation Technology, Hanoi University of Science and Technology. We would like to thank all colleagues and domain experts for collaborations and consultations.

References

[1] A. Nayyar, *Handbook of Cloud Computing*, 04 2019.

[2] J. Guillén, J. Miranda, J. M. Murillo, and C. Canal, "A service-oriented framework for developing cross cloud migratable software," *Journal of Systems and Software*, pp. 2294–2308, 2013.

[3] G. Baryannis, P. Garefalakis, K. Kritikos, K. Magoutis, A. Papaioannou, D. Plexousakis, and C. Zeginis, "Lifecycle management of service-based applications on multi-clouds," in *Proceedings of the 2013 International workshop on Multi-cloud applications and federated clouds (Multicloud'13)*, 2013, pp. 13–20.

[4] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "Multi-level elasticity control of cloud services," in *Service-Oriented Computing*, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 429–436.

[5] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[6] B. Wallace, "A hole for every component, and every component in its hole," *Existential Programming*, 2010.

[7] Mcllroy and M. Douglas, "Mass produced software components," *Scientific Affairs Division, NATO*, p. 79, 1969.

[8] R. Niekamp, "Software component architecture," *Gestión de Congresos - CIMNE/Institute for Scientific Computing*, p. 4, 2011.

[9] H.-L. Huynh, H.-D. Nguyen, V.-T. Le, and T.-T. Nguyen, "A composable application model for cloud marketplace," *Journal of Vietnam Science and Technology*, vol. 16, no. 5, pp. 40–45, 2017.

[10] H.-L. Huynh, H.-D. Nguyen, V.-T. Le, and D.-H. Le, "Towards the cloud marketplace for multi-cloud infrastructures," in *18th Vietnam National Conference: Selected issues of information technology and communication*, 2015.

[11] S. Kolb and G. Wirtz, "Towards application portability in platform as a service," in *Proceedings - IEEE 8th International Symposium on Service Oriented System Engineering, SOSE 2014*, 2014.

[12] C. Zeng, X. Guo, W. Ou, and D. Han, "Cloud computing service composition and search based on semantic," in *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong, Eds.

Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 290–300.

[13] B. I. Zilci, M. Slawik, and A. Küpper, “Cloud service matchmaking using constraint programming,” in *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 2015, pp. 63–68.

[14] S. K. Garg, S. Versteeg, and R. Buyya, “SMICloud: A framework for comparing and ranking cloud services,” in *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, Dec 2011, pp. 210–218.

[15] F. DAndria, S. Bocconi, J. G. Cruz, J. Ahtes, and D. Zeginis, “Cloud4SOA: Multi-cloud application management across PaaS offerings,” in *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Sep. 2012, pp. 407–414.

[16] S. García-Gómez, M. Jiménez-Gañán, Y. Taher, C. Momm, F. Junker, J. Bíró, A. Menychtas, V. Andrikopoulos, and S. Strauch, “Challenges for the comprehensive management of cloud services in a PaaS framework,” *Scalable Computing: Practice and Experience*, vol. 13, 2012.

[17] W. Elshareef, H. A. Ali, and A. Y. Haikal, “A matchmaking strategy of mixed resource on cloud computing environment,” *International Journal OF Scientific and Technology Research*, vol. 4, 2015.

[18] K. Sledziewski, B. Bordbar, and R. Anane, “A DSL-based approach to software development and deployment on cloud,” in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, April 2010, pp. 414–421.

[19] E. Brandtzaeg, S. Mosser, and P. Mohagheghi, “Towards CloudML, a model-based approach to provision resources in the clouds,” in *8th European Conference on Modelling Foundations and Applications (ECMFA)*, 2012, p. 18–27.

[20] “OpenStack HEAT URL,” accessed: 2019-04-26. [Online]. Available: https://docs.openstack.org/heat/latest

[21] “Juju Charms URL,” accessed: 2019-04-26. [Online]. Available: https://jujucharms.com

[22] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, “Topology splitting and matching for multi-cloud deployments,” in *Service-Oriented Computing-ICSOC 2017 Workshops*, 2017, pp. 379–383.

[23] OASIS, “Topology and orchestration specification for cloud applications version 1.0,” *Organization for the Advancement of Structured Information Standards*, 2013.

[24] D. Kourtesis, K. Bratanis, A. Friesen, Y. Verginadis, A. J. H. Simons, A. Rossini, A. Schwichtenberg, and P. Gouvas, “Brokerage for quality assurance and optimisation of cloud services: An analysis of key requirements,” in *Service-Oriented Computing – ICSOC 2013 Workshops*, A. R. Lomuscio, S. Nepal, F. Patrizi, B. Benatallah, and I. Brandić, Eds. Cham: Springer International Publishing, 2014, pp. 150–162.

Huynh Hoang Long received B.Sc. degree from Nhatrang University in 2008 and M.Sc. degree from Hanoi University of Science and Technology in 2012. His research interest includes cloud computing.

Nguyen Huu Duc received Ph.D. degree in Computer Science from Japan Advanced Institute of Science and Technology (JAIST), Japan, in 2006. He is currently the director of the Center for Data and Computation Technologies, Hanoi University of Science and Technology.

His main research topics include compiler construction, high performance computing, distributed systems and big data.

Le Trong Vinh received Ph.D. degree in Computer Science from Japan Advanced Institute of Science and Technology (JAIST), Japan, in 2006. He is currently Associate Professor and the director of the Center for Information Technology and Communication, University of Science, Vietnam National University, Hanoi.

His main research topics include theory of algorithms, computer networks.