CAM-D: A Description Method for Multi-cloud Marketplace Application

Hoang-Long Huynh[1], Duc-Huu Nguyen[1], Trong-Vinh Le[2], Quyet-Thang Huynh[1]
[1] Hanoi University of Science and Technology, Hanoi, Vietnam
[2] University of Science, Vietnam National University, Hanoi, Vietnam
Correspondence: Hoang-Long Huynh, longlove1232002@yahoo.com

Communication: received 30 November 2020, revised 27 January 2021, accepted 31 January 2021
Digital Object Identifier: 10.32913/mic-ict-research.v2020.n2.943

Abstract: **Multi-cloud Marketplace facilitates to create a diverse ecosystem for cloud software and cloud resource services provided by many stakeholders. To leverage the advantage of multi-cloud environment, cloud application could be a composition of software components which are able to be distributed across various cloud providers. So the cloud application is therefore a complex system. Consequently, an important key problem remained in research is to define multi-cloud application in a particular form and to construct his description. In this paper, we particularly focus on developing a description method that can be taken to tackle the lack of description for multi-cloud application by designing description templates for CAM which was developed in [1][2][3], called CAM-D. A completed application description can be synthesized from individual component descriptions. Our experimentation is expressed through the transformation of CAM-D template to TOSCA specification illustrated by case study. In addition, we also develop an flattening algorithm to assist in mapping to TOSCA.**

Keywords: *Cloud Computing, Multi-cloud, Multi-cloud Marketplace, Multi-cloud Marketplace Application, Composable Application Model (CAM), Software as a Service, Description Method.*

I. Introduction

At present, Cloud marketplace is an effective method for cloud providers to delivery Software as Service (SaaS) in a convenient way to consumers. Nevertheless, almost existing cloud marketplaces are owned by vendors themselves, and SaaS development is tightly coupled to Application Programming Interfaces (APIs) and tools of individual cloud providers. To overcome vendor lock-in problem, our ideal is about a new Multi-cloud marketplace model, called O-Marketplace [4], his ecosystem is depicted in Figure 2 including four actors: Cloud Consumers, Cloud App Vendors/Developers, Cloud Providers and O-Marketplace Administrator. SaaS supplied by O-Marketplace should be tailored to meet the characteristics of multi-cloud by separating cloud application into two parts, i.e. *cloud software* and *cloud platform* (runtime system), and cloud software could designed beyond a certain cloud provider. Ideally, cloud software is decomposed into components which are independently developed by various developers and can be deployed across different clouds. The overall idea of Multi-cloud application is illustrated in Figure 1 and shaped in our previous studies [1][2], called Composable Application Model (CAM).

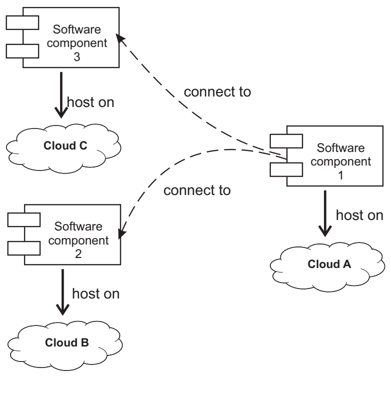


Figure 1. Multi-Cloud Application

In another aspect, multi-cloud marketplace has required manageability of cloud application. As such, there is an urgent need to have a written-related multi-cloud application description that predate, or have been developed. Unfortunately, there are very few existing related works to resolve this issue. There are three main approaches to describe multi-cloud application. The first one is a much-mentioned approach which is to use Domain Specific Language (DSL) to describe cloud application presented by K. Sledziewski et al. [5], E. Brandtzaeg et al. [6], G. Baryannis [7], G. Sousa [8], and N. Ferry [9]. The second

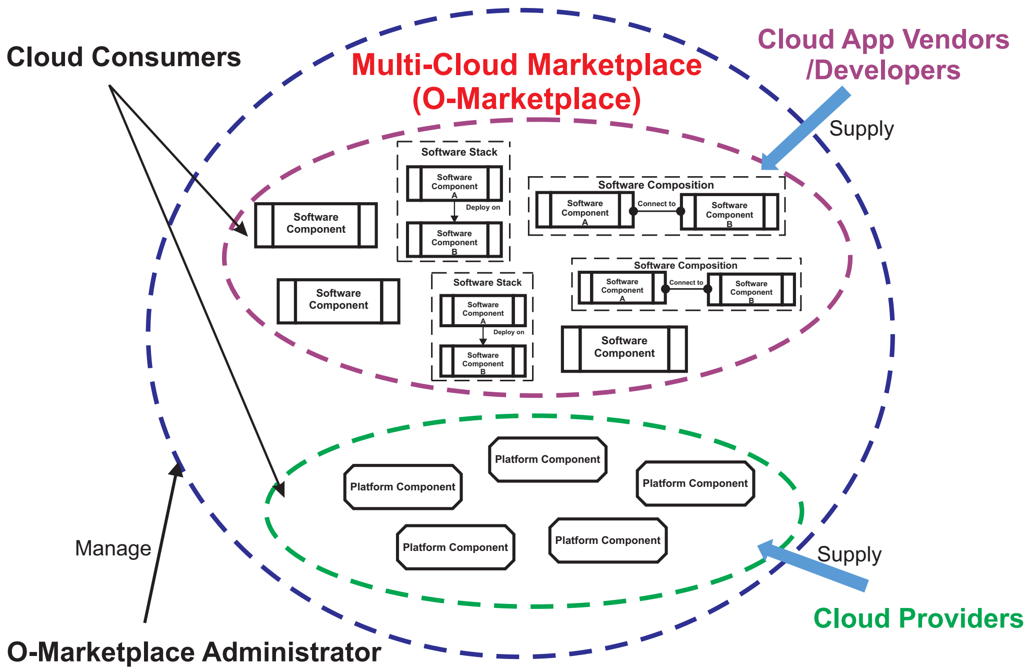


Figure 2. O-Marketplace Ecosystem

approach is to use open source solutions such as Heat [10] and Juju [11] for describing and modeling composite cloud applications and supporting deploying them on particular cloud providers. The last approach is to adopt TOSCA specification to distribute a cloud application across multiple clouds introduced by G. Tricomi [12], J. Carrasco [13], K. Alexander and [14], and K. Saatkamp [15].

In general, two first approaches could be done on a single cloud, and despite two notable efforts refined TOSCA [16] to enable a customized distribution of the components of an application to different cloud providers introduced in [13][15] which mentioned in the last, these proposals have limitation as follows: (i) there is no proposal to independently describe components within a multi-cloud application; (ii) the lack of solutions that are capable of combining individual component descriptions to form a complete description of a multi-cloud application. (iii) the multi-cloud application specification validation mechanism has not been built in yet based on proposed specification/description methods.

To tackle these limitations, our key approach is to develop a description method which is able to cover multi-cloud application in a uniform. This supports the distribution of software components on heterogeneous multi-cloud platforms. In this paper, we define a specification method for multi-cloud application which is modeled by CAM introduced in previous papers [4][1][2][3]. Our work towards the following contributions:

- Defining the description templates for CAM.
- Presenting a novel approach for creating the multi-cloud application description: the completed description of multi-cloud application is synthesized from the individual component descriptions.
- For experimentation, we propose an algorithm for flattening the description to TOSCA-based specification.

The rest of the paper is organized as follows: the core of our work in this paper is presented in Section II, we focus on making up description templates for CAM. In Section III, the illustration of transformation from CAM-D description template to TOSCA specification is shown. The advantages of CAM-D is highlighted in Section V. Finally, we conclude and summarize the contributions of the paper in Section VI.

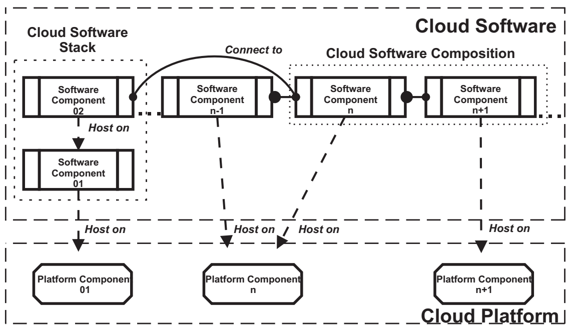


Figure 3. Composable Application Model [2].

II. CAM-based Description Templates

With the motivation to standardize the multi-cloud application description in a uniform, we propose a description method that is not only capable of expressing its behaviours and properties in standardized description patterns, but also meet the multi-cloud characteristics in general and O-Marketplace proposed in [4] in particular. To achieve this goal, our work focuses on developing description templates for entire CAM and its components, which were defined in our recent paper [2] including: Simple Cloud Software Component, Cloud Software Stack, Cloud Software Composition, Cloud Platform Component, and CAM depicted in Figure 3. The description of CAM application can be built from individual descriptions based on the *matching rules* proposed in [2]. In addition, an interesting point is that CAM is defined as a nested structure, so the description method has to express this special feature.

1. CAM-based Application

Before going further into details of the CAM description templates, several conventions have been made as follows:

- All templates are written in a YAML-like format.
- We assume that all the defined templates are available so that they can be referred via their ids.
- Since CAM components are designed in a nested manner, we use dot-notation for accessing to an inner element and/or property of a component. For example, `c.capabilities[cid].p` stands for the property `p` of the capability `cid` of the component `c`.

```
application <app id> =
  software : <software component template>
    arguments:
      - <arg> : <value>
      - ...
  platforms :
    - <platform> : <platform service
         template>
      arguments:
        - <arg> : <value>
        - ....
    - ...
  dependencies:
    - <dependency>:
      type: HOST-ON
      from: <software>.<requirement>
      to: <platform>.<capability>
end
```

Figure 4. CAM-based Application Template

A CAM-based multi-cloud application can be described by an application script using the application template depicted in the Figure 4. Each application should specify a CAM-based cloud software component as well as a cloud platforms on which we prefer to host the application. The cloud software component and the platform are characterized by sets of parameters which serve as user preferences. The application script should also indicate the correspondences between the software requirements and the platforms in the section `dependencies`.

2. Platform Service Template

Platform services (*Cloud Platform Component* depicted in Figure 5) are provided by cloud providers. In order to integrate their services into our CAM-based system, cloud providers should publish their services in form of Platform Services Template as shown in the Figure 6.

Platform Component
Caps

Figure 5. Cloud Platform Component [2].

```
platform <platform id> =
  capabilities:
    - <capability> : <port signature>
    - ...
  parameters:
    - <param> : <type>
    - ...
  parameter mappings:
    - <capability>.<property> as <param>
        default <value>
    - ...
  methods :
    - <method> : <method type>
      parameters:
        - <arg> : <type>
        - ....
    - ...
end
```

Figure 6. CAM-based Platform Service Template

Each cloud platform service defines a set of capabilities to provide to the cloud software. These capabilities should match with the requirements of the software for the successful deployment of the application. In order to perform the matching algorithm, each capability and/or requirement is associated to a port signature (illustrated in Figure 7) which describes the properties and the operations provided by the capability and/or needed for the requirement. Please note that the a port signature is not only specifies the dependency between a cloud software requirement and the platform capability, but also is used to describe the dependency among software components in a composition. We distinguish different kinds of dependency on a port signature by port type which is either HOST-ON or CONNECT-TO.

```
signature <port signature id> =
  type: <port type>
  properties:
    - <property> : <type>
    - ...
  methods :
    - <method> : <method type>
      parameters:
        - <arg> : <type>
        - ....
    - ...
end
```

Figure 7. Port Signature

3. General Structure of a Cloud Software Component Template

CAM-based cloud softwares and components are defined in a uniform way using the Cloud Software Component template (Figure 8).

Each component may have its own inner structure which consists of a set of sub-components and the dependencies (described in the Section `Dependencies` of the template) among them. A CAM runtime should check the well-form of a component by matching requirements to capacities according to the specified dependencies. In a special case, when the component is a Simple Software Component, this inner structure is skipped.

The outer view of a CAM-based component consists of a set of capabilities (described in the Section `Capabil-ities` of the template) specifying what the component provides, and a set of requirements (described in the Section `Requirements` of the template) specifying what are needed for managing and running the component.

Properties of these capabilities are mapped into a set of parameters in the section `parameter mapping` so that programmers will know how to characterize the components when they are in use. In some special cases, the component itself has extra parameters defined in the section `parameters`. This is worth to mention that the properties of sub-components requirements can be taken from the properties of their corresponding capabilities from platforms and/or other sub-components.

A component also provides a set of operations indicated by methods in the `methods` section. The component script should describes the set of arguments to a method as well as its execution code, normally written in a special scripting language.

```
component <component id> =
  type: <component type>
  # Inner structure of the component
  sub-components:
    - <component> : <software component
        template>
    - ...
  dependencies:
    - <dependency>:
      type: <port type>
      from: <component>.<requirement>
      to: <component>.<capability>
    - ...
  # outer interface
  capabilities:
    - <capability> map-to
        <component>.<capability>
    - <capability> : <port signature>
    - ...
  requirements:
    - <requirement> map-to
        <component>.<requirement>
    - <requirement> : <port signature>
    - ...
  parameters:
    - <param> : <type>
    - ...
  parameter mappings:
    - <capability>.<property> as <param>
        default <value>
    - ...
  methods :
    - <method> : <method type>
      parameters:
        - <arg> : <type>
        - ...
      code: <script>
    - ...
end
```

Figure 8. General Structure of a Cloud Software Component Template

4. Simple Software Component Template

Simple Cloud Software Component (SSC) is the atomic entity or the basic element of CAM. A SSC packs its code and data together with the necessary requirements and capabilities for deployment and operation. The model of SSC is depicted in Figure 9. SSC can only reside in a single cloud platform.

As mentioned before, a SSC doesn't have its own inner structure. Instead, programmers should define component's capabilities and requirements by specifying corresponding port signatures. All of the code and data to manage and run the component can be defined in the `methods` section.

A Simple Software Component can therefore be defined using the template shown in Figure 10.

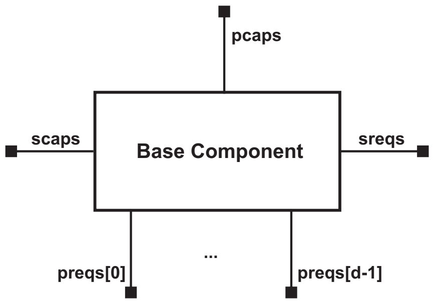


Figure 9. Simple Cloud Software Component [2].

```
component <component id> =
  type: SimpleComponent
  capabilities:
    - <capability> : <port signature>
    - ...
  requirements:
    - <requirement> : <port signature>
    - ...
  parameters: ...
  parameter mappings: ...
  methods: ...
end
```

Figure 10. Simple Software Component Template

5. Cloud Software Stack Template

Cloud Software Stack (SS) represents a series of software components established on each others deployed on a single cloud platform. For simplicity, we define a SS with two elements: a top component and a base component. In the nested manner, we restrict top component to be a simple component while base component can be either a simple component or another stack.

Composing elements of SS depicted in Figure 11 and we define the the description template for SS as showed in Figure 12.

A description template of SS is well-formed if it satisfies the following validation conditions:

- Top component and base component should be well-formed;
- Type of the top component should be “SimpleComponent";
- Type of the base component should matches either “SimpleComponent" or “SoftwareStack";
- The dependencies between the top component and the base component defined in the section `dependencies` should be guaranteed, i.e. signatures of the top component requirements should match with the signatures of the base component capabilities;

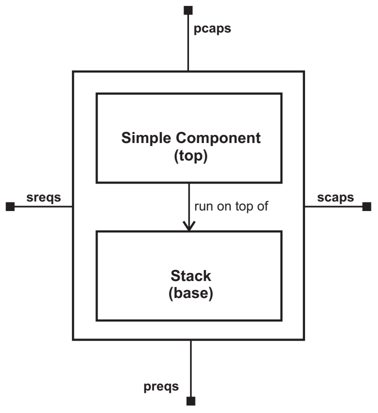


Figure 11. Cloud Software Stack [2].

```
component <component id> =
  type: SoftwareStack
  # inner structure
  sub-components:
    - top: <Component template>
    - base: <component template>
  dependencies:
    - <dependency>:
      type: HOSTON
      from: top.requirements[<requirement>]
      to: base.capabilities[<capability>]
    - ...
  # outer interface
  capabilities: ...
  requirements:
  parameters: ...
  parameter mappings: ...
  methods: ...
end
```

Figure 12. Software Stack Component Template

6. Cloud Software Composition Template

Cloud Software Composition (SC) represents a distributed multi-cloud software composition including cloud components which are able to deploy on various cloud platforms. Each component may connect to others via software protocols. In our CAM model, we defined such software protocols as port signatures of type “CONNECT-TO".

For simplicity, we define a SC with 2 components: left and right. Composing elements of SC is depicted in Figure 13, and description template of Cloud Software Composition is shown in Figure 14.

The description template of a SC is well-formed if it satisfies the following validation conditions:

- All of its sub-components (i.e. left and right) are well-formed.

- The dependencies between the left component and the right component defined in the section `dependencies` should be guaranteed, i.e. signatures of the left component requirements should match with the signatures of the right component capabilities;

Please note that we can extend the template for software composition consisting more than two sub-components. In this case, a dependency between sub-components is a mapping from requirement of a sub-component to a capability of its counter part. This extended template will be used for describing *flattening composition* representing in the next section.

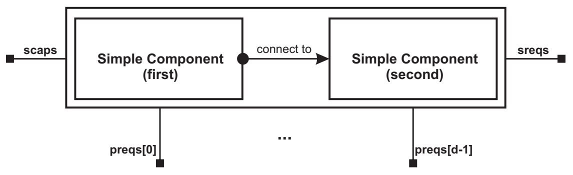


Figure 13. Cloud Software Composition [2].

```
component <component id> =
  type: SoftwareComposition
  # inner structure
  sub-components:
    - left: <Component template>
    - right: <component template>
  dependencies:
    - <dependency>:
      type: CONNECT-TO
      from: left.requirements[<requirement>]
      to: right.capabilities[<capability>]
    - ...
  # outer interface
  capabilities: ...
  requirements:
  parameters: ...
  parameter mappings: ...
  methods: ...
end
```

Figure 14. Software Component Composition Template

III. Transforming from CAM-D to TOSCA Specification

In order to prove the feasibility of our approach, it is necessary to have a runtime system which is able to parse our CAM-D application script and deploy the application on specified cloud platforms. While this is still on our on-going research, in this paper, we employ an alternative method: translating CAM-D application script to well-known TOSCA cloud application model and then using existing TOSCA-based runtime for deploying the application.

In general, our CAM-D application description template is very similar to TOSCA topology template. In terms of topology, the concept of Node in TOSCA template is equivalent to our CAM-D software component, and the concept of Relationship in TOSCA is equivalent to our CAM-D dependency. Different from TOSCA, CAM-D represents a cloud application in a nested manner for which software components can be independently developed and composed by different developers.

To transform a CAM-D application script to TOSCA topology template, we apply a two-step process:

1) Flattening the given CAM-D application script into a *flattening composition*;
2) Transforming the *flattening composition* into TOSCA topology template

In the following sub-sections, we will give a short representation about TOSCA as well as our two steps of the transformation algorithm.

1. Topology and Orchestration Specification for Cloud Application

Topology and Orchestration Specification for Cloud Applications (TOSCA) [16] is an open standard built by OASIS that defines the inter-operable description of cloud application hosted on the cloud. TOSCA will enable the inter-operable description of application and infrastructure cloud services, the relationships between parts of the service, and the operational behavior of these services independent of the supplier creating the service, and any particular cloud provider (Figure 15).

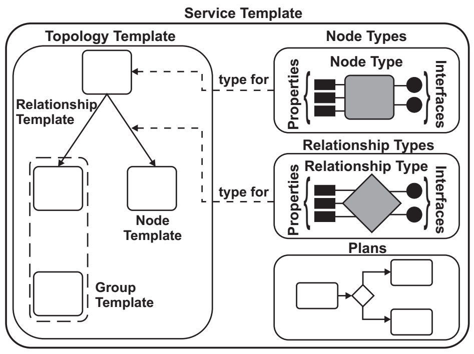


Figure 15. TOSCA Service Template overview [16].

TOSCA will enable portable deployment to any compliant cloud and enhance the portability multi-cloud provider applications. So TOSCA is also very efficient in software provisioning, deployment and management of cloud application. It has been supported by many partners like IBM, Reb Hat, Cisco, Citrix, EMC, etc. Thus, we utilize this standard as a target of the transformation to demonstrate the feasibility of our proposed description method.

2. Flattening Algorithm

The first step of the transformation algorithm from CAM-D application script to TOSCA topology template is to flatten the nested composition of the CAM-D software component into a flat composition where all sub-components of the flat composition should be of type SimpleComponent. The transformation should keep all external view of the component, i.e. all requirements, capabilities, parameters, and methods. The things should be change are internal structure of the composition as well as the mapping of outer requirements, capabilities, parameters to its corresponding inner elements.

This would be done easily by using a recursive algorithm shown as below:

```
Flattening  Algorithm (CAM−D script);
Input:
  − A nested software component
Output:
  − A flattening  software component
  − A mappings from capabilities / requirements / parameters of input sub−components
     to the correspondences of the output simple sub−components.
Begin
   If c.type = "SimpleComponent"
   Then
         M = {e−>c.e, for each capability / requirement / parameter e of c}
         Return (c,M)
   Else If c.type = "SoftwareStack"
   Then
    Begin
         (new−top, top−mappings) = Flatten(c.top)
         (new−base, base−mapping) = Flatten(c.base)
         mappings = AddPrefix("top", top−mapping) ⋃
                    AddPrefix("base", base−mappings)
         sub−components = new−top.sub−components ⋃ new−base.sub−components
         dependencies = new−top.dependencies ⋃ new−base.dependencies ⋃
                        UpdateDependencies(c.dependencies, mappings)
         requirements = UpdateRequirements(c.requirements, mappings)
          capabilities = UpdateCapabilities(c. capabilities , mappings)
         parameter−mappings = UpdateParameters(c.parameter−mappings, mappings)
         methods = UpdateMethods(c.methods, mappings)
         new−mappings = MakeMappings(requirements, capabilities, parameter−mappings)
         Return (FlatCompositions(sub−components, dependencies,
                    requirements, capabilities , c.parameters, parameter−mappings, methods),
                new−mappings)
    End
   Else If c.type = "SoftwareComposition"
  Then
    Begin
         (new−left, left−mappings) = Flatten(c. left)
         (new−right, right−mapping) = Flatten(c. right)
         mappings = AddPrefix("left", left−mapping) ⋃
                    AddPrefix("right", right−mappings)
         sub−components = new−left.sub−components ⋃ new−right.sub−components
         dependencies = new−left.dependencies ⋃ new−right.dependencies ⋃
                        UpdateDependencies(c.dependencies, mappings)
         requirements = UpdateRequirements(c.requirements, mappings)
          capabilities = UpdateCapabilities(c. capabilities , mappings)
         parameter−mappings = UpdateParameters(c.parameter−mappings, mappings)
         methods = UpdateMethods(c.methods, mappings)
         new−mappings = MakeMappings(requirements, capabilities, parameter−mappings)
         Return (FlatCompositions(sub−components, dependencies,
                     requirements, capabilities , c.parameters, parameter−mappings, methods),
                new−mappings)
     End
End;
```

In this algorithm, the function `AddPrefix(s, mappings)` add a component access prefix `s` to all origins of `mappings` to avoid ambiguity. The functions `UpdateDependencies`, `UpdateRequirements`, `UpdateCapabilities`, `UpdateParameters`, `UpdateMethods` are just used for updating all references to the requirements/capabilities/parameters of sub-components of the input component by the corresponding elements of the SimpleComponent sub-components in the target component. The function `MakeMappings` just accumulates all modified mappings of requirements, capabilities, and parameters. Due to the limitation of the paper, we omit the presentation of these functions.

Correctness: The flattening algorithm is correct in the following principles:

- All sub-components of a resulting flat component are components of type SimpleComponent;
- Dependencies between the top and the base sub-components of any SofwareStack inside the structure of input component should be reflected in the dependencies of resulting flat component.
- Dependencies between the left and the right sub-components of any SofwareComposition inside the structure of input component should be reflected in the dependencies of resulting flat component.
- All mappings (requirement mappings, capability mappings, parameter mappings) of the input component should be changed in the output flat component so that they maps their original external names to specific names of requirements/capabilities/parameters of corresponding SimpleComponents.

The correctness of the algorithm can be easily proved by reduction on the hierarchical structure of the input nested component.

Complexity: The flattening algorithm is proceeded recursively on the hierarchical structure of the input nested component. Thus the complexity of the algorithm is linear to the number of the sub-components inside that structure, i.e. O(n) for n is the number of sub-components.

3. Mapping to TOSCA

As a result of the above flattening algorithm, a CAM-D application will be transformed into a flat model in which all sub-components are of type SimpleComponent.

The second step of the transformation algorithm is to translate this flat application model together with its undeying platforms into TOSCA topology template. The translations is rather simple. Each sub-components and/or platforms will be transformed into a TOSCA node. Each dependency among components and between a component and a platform will be translated into a TOSCA relationship. We do not give further details of this translation algorithm. Instead, result from an experimentation which will be given in the next section will demonstrate our mapping method.

IV. Experimentation with Case Study

In this section, we demonstrate the feasibility of CAM-D by translating flattening description template of CAM-D into TOSCA specification template with a case study.

1. Case Study

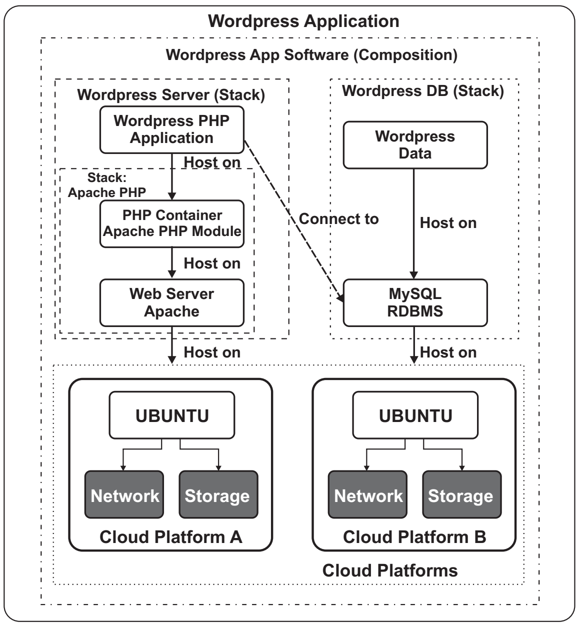


Figure 16. Wordpress Application modeled by CAM

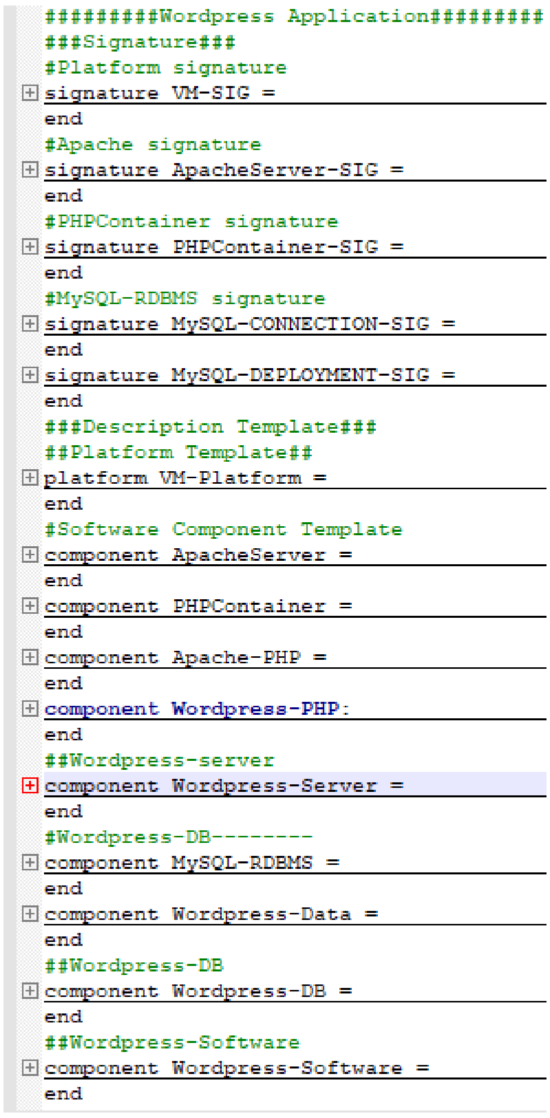
```
application Wordpress-Application =
   software: Wordpress-Software
      arguments:

   platforms :
         p1 : VM-Platform
            arguments:
               provider: "dsg@openstack"
               instanceType : "000001920"
               baseImage :
                   "a82e054f-4f01-49f9-bc4c
               -77a980457 39c"
         p2 : VM-Platform
            arguments:
               provider: "dsg@flexiant"
               instanceType : "000001920"
               baseImage :
                   "a82e054f-4f01-49f9-bc4c
               -77a980457 39c"
   dependencies:
      d1:
         type: HOST-ON
         from: software.requirements[R1]
         to:  p1.capabilities[vm]
      d2:
         type: HOST-ON
         from: software.requirements[R2]
         to:  p2.capabilities[vm]
end
```

Figure 17. Wordpress Application Description Template

For validate our idea, we deploy a WordPress application on two Clouds: OpenStack and Flexiant. We use CAM to model Wordpress application which is depicted in Figure 16. There are five software components of cloud software: Wordpress PHP Application, PHP Container, Apache, Wordpress DB, MySQL covered in two stack, and two Platform components are cloud infrastructures: OpenStack and Flexiant. These software components modeled in three software stack and one software composition. The CAM-D template of Wordpress Application showed in Figure 17.

According CAM-D, the description template of Wordpress Application is created based on individual description templates which represent for the type of CAM-based components according to the degree of CAM. The overview of individual description templates of Wordpress Application is shown in Figure 18. The full code of Wordpress Application Templates is available at https://github.com/longlovehl/Descriptions/.



```
#########Wordpress Application#########
###Signature###
#Platform signature
signature VM-SIG =
end
#Apache signature
signature ApacheServer-SIG =
end
#PHPContainer signature
signature PHPContainer-SIG =
end
#MySQL-RDBMS signature
signature MySQL-CONNECTION-SIG =
end
signature MySQL-DEPLOYMENT-SIG =
end
###Description Template###
##Platform Template##
platform VM-Platform =
end
#Software Component Template
component ApacheServer =
end
component PHPContainer =
end
component Apache-PHP =
end
component Wordpress-PHP:
end
##Wordpress-server
component Wordpress-Server =
end
#Wordpress-DB--------
component MySQL-RDBMS =
end
component Wordpress-Data =
end
##Wordpress-DB
component Wordpress-DB =
end
##Wordpress-Software
component Wordpress-Software =
end
```

Figure 18. Wordpress Application Description Templates

2. Transforming CAM-D Template to TOSCA Specification

To transform CAM-D Template to TOSCA specification, we first use proposed flattening algorithm to flatten the Wordpress Application into a graph representation. Then, we map flattening description template of WordPress Application into TOSCA-based specification. 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. The result is showed in Figure 19.

```
<?xml version="1.0"?>
<ns2:Definitions id="WordpressApp"
      xmlns:ns2="http://docs.oasis-open.org/toscans/2011/12"
      name="WordpressApp">
<ns2:ServiceTemplate id="WordpressTopology">
 <ns2:TopologyTemplate>
   <ns2:RelationshipTemplate id="WordpressPHP_HostOn_PHPContainer"
      type="HOSTON">
    <ns2:SourceElement ref="PHPContainer"/>
    <ns2:TargetElement ref="WordpressPHP"/>
   </ns2:RelationshipTemplate>
   ...
   <ns2:RelationshipTemplate id="WordpressPHP_ConnecTTo_MySQLRDBMS"
      type="CONNECTTO">
    <ns2:SourceElement ref="MySQLRDBMS"/>
    <ns2:TargetElement ref="WordpressPHP"/>
   </ns2:RelationshipTemplate>
   <ns2:NodeTemplate id="ApacheVM" type="os" maxInstances="3" minInstances="1">
     <ns2:Properties>
      <MappingProperties>
       <MappingProperty type="os">
             <property name="provider">dsg@openstack</property>
             <property name="instanceType">000001920</property>
<property name="baseImage">a82e054f-4f01-49f9-bc4c-77a98045739c</property>
<property name="packages"/>
       </MappingProperty>
       </MappingProperties>
   ...
<ns2:NodeTemplate id="MySQLVM" type="os" maxInstances="2" minInstances="1">
      <ns2:Properties>
       <MappingProperties>
<MappingProperty type="os">
<property name="provider">dsg@flexiant</property>
<property name="instanceType">000001920</property>
<property name="baseImage">a82e054f-4f01-49f9-bc4c-77a98045739c</property>
<property name="packages"/>
</MappingProperty>
</MappingProperties>
      ...
    </ns2:NodeTemplate>
    <ns2:NodeTemplate id="MySQLRDBMS" type="software" maxInstances="1"
      minInstances="1">
      ...
      <ns2:Requirements>
       <ns2:Requirement id="MySQLVM" type="HOSTON"/>
      </ns2:Requirements>
      <ns2:Capabilities>
       <ns2:Capability id="WordpressPHP" type="CONNECTTO"/>
      </ns2:Capabilities>
      ...
 </ns2:NodeTemplate>
</ns2:TopologyTemplate>
</ns2:ServiceTemplate>
<ns2:ArtifactTemplate id="Artifact_93f8753b-17ab-43c5-8f11-e3ec98fe3224"
      type="sh">
    <ns2:Properties />
    <ns2:ArtifactReferences>
      <ns2:ArtifactReference
      reference="http://.../upload/files/wordpress/install_WordpressPHP.sh" />
    ...
</ns2:ArtifactTemplate>
...
</ns2:Definitions>
```

Figure 19. TOSCA-based Description of WordPress Application

V. The advantages of CAM-D

To evaluate the advantages of CAM-D in particular and CAM [2][3] in general, we make a comparison between CAM-D and TOSCA in Table I because TOSCA has been the well-known standard. However, TOSCA specification has not totally developed for multi-cloud application specification.

TABLE I
The comparison between CAM-D and TOSCA Specification

FEATURES	CAM-D	TOSCA Specification
Topology	Nested structure	Flat structure
Cloud software portability	YES	NO
Component-based cloud application description	YES	NO
Synthesized from component specifications	YES	NO
Multi-cloud service matchmaking	YES	NO

The above comparison showed significant differences of CAM-D compared with TOSCA Specification. CAM-D especially supports Cloud software portability, Component-based cloud application description, Synthesized of component specifications, and Multi-cloud service matchmaking. These features have proved the feasibility of our proposal with distinct advantages for cloud application development as follows:

- A cloud application may be constructed as a distributed system of which elements are compute servers running specific software and located at specific cloud providers.
- Developers are free to evolve their cloud software without any technology restriction from cloud providers. They also can develop and sell their small pieces instead of complete software solutions.
- Enhancing the ability to reuse the cloud software component, developers can build up an application by just incorporating existing components of others into their own software solution. This save cost and time in cloud application development.

In addition, an interesting feature of CAM-D that is quite similar to a program written in a programming language. The application template is the main program, component templates are procedures and arguments of a procedure can refer to parameters of others. This is very convenient for developer to independently develop cloud software. He could package the software component as a “***Black Box***" and just express properties to the outer ports of such component. This is the special feature that CAM-D brings.

VI. Conclusion

In this study, we build a description method (CAM-D) that supports to describe multi-component cloud software. To implement this idea, firstly, we define CAM-based description templates for multi-component cloud software modeled in a nested structure. Secondly, we develop flattening algorithm to cover nested description to flattening description. Thirdly, As an illustration, we experimentally transform the description templates of CAM-D into TOSCA-based specification templates and demonstrate this process in a particular example – the Wordpress Application. Finally, the advantages of CAM-D is expressed. To sum up, the key of our work is to create a uniform for multi-cloud applications. The proposed concepts and templates can totally be applied to multi-cloud marketplace.

References

[1] 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.

[2] H.-L. Huynh, H.-D. Nguyen, and T.-V. Le, "Matchmaking for Multi-cloud Marketplace Application," *Journal of Information and Communications*, vol. 2019, no. 1, pp. 31–42, 2019.

[3] H.-L. Huynh, V.-D. Tran, H.-D. Nguyen, Z. Hu, T.-V. Le, and Q.-T. Huynh, "Auto-Updating Portable Application Model of Multi-cloud Marketplace through Bidirectional Transformations System," *International Conference on Intelligent Software Methodologies, Tools, and Techniques (SOMET 2019)*, pp. 11-24, Malaysia, Sep 2019. DOI:10.3233/FAIA190035. (SCOPUS)

[4] 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*, pp. 100-105, November 2015.

[5] 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*, pp. 414–421, April 2010.

[6] 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)*, pp. 18–27, 2012.

[7] 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)*, pp. 13–20, 2013.

[8] G. Sousa, W. Rudametkin, and L. Duchien, "Automated setup of multi-cloud environments for micro services applications," *in 2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pp. 327–334, 2016.

[9] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "CloudMF: Model-driven management of multi-cloud applications," *ACM Trans. Internet Technology*, vol. 18, Jan. 2018.

[10] "OpenStack HEAT URL." https://docs.openstack.org/heat/latest/. Accessed: 2019-04-26.

[11] "Juju Charms URL." https://jujucharms.com/. Accessed: 2019-04-26.

[12] G. Tricomi, A. Panarello, G. Merlino, F. Longo, D. Bruneo, and A. Puliafito, "Orchestrated multi-cloud application deployment in openstack with TOSCA," *in 2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 1–6, 2017.

[13] J. Carrasco, J. Cubo, and E. Pimentel, "Towards a flexible deployment of multi-cloud applications based on tosca and camp," *in Advances in Service-Oriented and Cloud Computing (G. Ortiz and C. Tran, eds.), (Cham)*, pp. 278–286, Springer International Publishing, 2015.

[14] K. Alexander, C. Lee, E. Kim, and S. Helal, "Enabling end-to-end orchestration of multi-cloud applications," *IEEE Access*, vol. 5, pp. 18862–18875, 2017.

[15] K. Saatkamp, U. Breitenbücher , O. Kopp, and F. Leymann, "Topology Splitting and Matching for Multi-Cloud Deployments," *in Service-Oriented Computing-ICSOC 2017 Workshops*, pp. 379–383, 2017.

[16] OASIS, "Topology and Orchestration Specification for Cloud Applications Version 1.0," *Organization for the Advancement of Structured Information Standards*, 2013.

Hoang-Long Huynh received B.S (2008) from Nhatrang University and M.S (2012) from Hanoi University of Science and Technology, Vietnam. His research interests in Cloud Computing.
Email: longlove1232002@yahoo.com

Huu-Duc Nguyen received PhD degree (2006) in Computer Science from Japan Advanced Institute of Science and Technology (JAIST), Japan. 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.
Email: ducnh@soict.hust.edu.vn

Trong-Vinh Le received PhD degree (2006) in Computer Science from Japan Advanced Institute of Science and Technology (JAIST), Japan. He is currently Associate Professor and the director of the Center for Information Technology and Communication, VNU University of Science. His main research topics include theory of algorithms, computer networks.
Email: vinhlt@gmail.com

Quyet-Thang Huynh received PhD degree (1995) in Information and Computer Sciences from Varna Technical University, Bulgaria. He is currently Associate Professor and the president of Hanoi University of Science and Technology. His main research topics include Software Quality, Software Testing, Methods in Software Development, Multi-Objective Optimization, Project Management, Big Data Processing and Analytics.
Email: thanghq@soict.hust.edu.vn