Towards a Model-defined Cloud-of-Clouds

Bo An¹, Zhicheng Cui¹, Mauricio Tsugawa², Ying Zhang¹, Xuanzhe Liu¹, Gang Huang¹, Joséa. B. Fortes²
¹ Peking University
² University of Florida,
{anbo, czcpku, zhangying, liuxuanzhe, hg}@pku.edu.cn
{tsugawa, fortes}@acis.ufld.edu

Abstract

With the growth in the number of Cloud Service Providers, many enterprises and organizations are now able to use multiple cloud platforms in order to achieve improved overall Quality-of-Service (QoS), reliability and cost efficiency. However, due to the diversity in architecture and functionalities among different cloud platforms, it is difficult to build a system that simultaneously manages multiple clouds, i.e., a cloud-of-clouds. This paper presents a model-defined approach to the development of a cloud-of-clouids management (abbreviated as CCMan) system. The runtime model of a CCMan system that meets custom management requirements is constructed through model construction, model merging and model transformation. Each step of the approach is presented in detail in terms of an example. Evaluation of the approach from several perspectives shows that the efforts needed to both develop the CCMan system and operate its services are significantly reduced with negligible performance loss.

Keywords: model-defined, cloud of clouds, runtime model

1. Introduction

A cloud management system is a collection of tools and interfaces that allow providers and consumers of cloud services to efficiently provision multiple IT systems and operate cloud-hosted IT environments, respectively. For cloud providers, it provides the ability to manage a pool of heterogeneous resources. For cloud consumers, it provides abilities to deploy their applications and use cloud resources. A number of cloud management systems have emerged in recent years, such as OpenStack, CloudStack, VMware vCloud Suite, Eucalyptus and OpenNebula.

With the growth in the number of Cloud Service Providers, many enterprises and organizations can use and combine services from multiple providers. The use of multiple clouds brings many advantages: cost optimization, Quality of Service (QoS) improvements, high availability, avoiding vendor lock-in, disaster recovery and so on (Petcu, 2013). Such multi-cloud systems are also referred as inter-cloud, sky computing, federated clouds and cloud of clouds. Due to differences in cloud services and their exported interfaces (APIs), current practice is to interact with each cloud separately: e.g., use CloudStack tools to interact with CloudStack clouds and use VMware vCloud Suite to interact with VMware clouds. While this approach does not require additional software development, it has many disadvantages: administrators may need to continually switch among two or more completely different cloud systems, a global and unified view of all available resources is not provided, and different clouds do not interact with each other.

In order to efficiently use resources and services from multiple cloud providers, there is a need for a management system that can interact with different clouds and offer a unified view of the entire system. In this work we address three key challenges: (1) each cloud system has its own way of organizing resources, and exposes different management interfaces, making it difficult to develop an integrated and unified management system; (2) cloud providers are physically located on geographically separate sites, and a networking system that can be dynamically reconfigured is needed to enable the interaction among different clouds; and (3) there are different custom management requirements consisting of specific scenarios and appropriate management styles.

Our approach to address the above challenges relies on Model Driven Engineering (MDE), a branch of software engineering that raises the level of abstraction in program specification and increases automation in program development. It enables developers to work at a high level of abstraction and focus on cloud concerns rather than implementation details. In this paper, we present a model-defined approach to the development of CCMan system. Runtime models of multiple cloud platforms and a virtual networking system (to enable and manage the connectivity among multiple clouds) are constructed. A “model at runtime”, also called a runtime model, is a causally connected self-representation of the associated systems. We have developed a model-based runtime management tool called SM@RT (Supporting Model AT Run Time (Song et al., 2010) (Song et al., 2011)), which provides the synchronization engine between a runtime model and its corresponding running system. ViNe (Tsugawa et al., 2009) (Tsugawa and Fortes, 2006) has been selected as the virtual-networking system due to its dynamic network reconfiguration ability and availability of programmatic interfaces. These run-time models are merged to form a composite model. Finally, the composite model is transformed into a customized model that meets the personalized requirements of an enterprise or an organization.

The rest of this paper is structured as follows. Section II motivates our work through an example. Section III presents
our proposed approach. Section IV evaluates our approach through a real case study. Section V discusses related work. Section VI draws our conclusions and indicates future work.

2. A Motivating Example

Consider a company/organization that owns two datacenters to build two private clouds: one using CloudStack and another using VMware vCloud Suite. Possible reasons to use different cloud management systems include vendor lock-in avoidance and the use of different services and/or features of these two systems. These private clouds might be unable to meet the needs of the entire company, due to insufficient capacity, increased Quality-of-Service (QoS) demands and/or availability requirements for services in multiple regions. One solution to these problems is to enlarge the scale of private clouds, which may be economically impractical. Another solution is to use public clouds (such as Amazon Web Services and Aliyun.com) and rent resources from them, which could be less expensive and provide better flexibility and availability. The company chooses the latter solution and now the company needs to manage and/or interact with three different clouds: the CloudStack cloud, the vCloud cloud and the public cloud. As the management systems of the three clouds are different, many problems arise. First, the lack of a unified way to manage all the resources requires the administrator to login to each cloud management system to interact with the corresponding resources, to learn how to use many different cloud management systems and to continually switch among these systems. Second, a unified view of all available resources is not provided. Third, different clouds do not interact with each other.

In order to deal with these problems, the company decides to develop a CCMan system to manage all the resources. The requirements of the CCMan include: (1) all the resources from different clouds can be managed in a unified way; (2) different clouds can interact with each other, i.e., the network connection can be configured on demand between virtual machines of different clouds. (3) Different management views (including a set of managed resources and a set of management interfaces) are provided for users in different roles. For instance, the root administrator can manage all the resources and invoke all the management interfaces. A domain administrator only has the authority to manage the resources in her/his domain and to invoke the management interfaces operating on the resources in this domain; and the domain user only has the authority to view and use the resources in the domain. The word domain refers to a division or department of an organization in charge of their own resources.

3. Our Approach

In this paper, we present a model-defined approach to develop a CCMan system. Let’s start by giving an overview of the approach, as illustrated in Figure 1. First, the runtime models of each cloud platform and a virtual networking system (to enable and manage the connectivity among multiple clouds) are constructed. Second, these runtime models are merged to form a composite model. Third, the composite model is transformed into different customized models that meet the personalized management requirements (e.g., three different management views for users in different roles) of an enterprise or an organization. After the three steps, all the participant clouds can be managed in a unified and personalized manner through operations of the customized model. Network connectivity among all virtual machines on these clouds can be enabled. Management tasks such as VM scheduling, fault tolerance, load balancing and cloud monitoring can be carried out by writing and executing different QVT (Query/View/Transformation) programs on the customized model, without the need to interact with the management interfaces of the underlying cloud systems.

The rest of this section presents the details of each step. For simplicity, we use two clouds for this study: a private cloud managed by the CloudStack system and a public cloud managed by the OpenStack system. Note that there is no loss of generality since there are no essential differences between managing two, three, or more clouds when using the proposed approach.

3.1 Model Construction

The runtime models in our approach are abstracted from the underlying systems (a cloud management system or ViNe). The input to the model construction procedure includes an architecture-based meta-model specifying what kinds of elements can be managed in the system, and an access model of the configurations specifying how to use the management APIs to monitor and modify those managed elements. Then the architecture-based runtime model of the target system is automatically constructed by SM@RT, and the correct synchronization between the runtime model and the running system is enabled. For instance, in Figure 2 the synchronization engine builds a model element in the runtime model for each virtual machine. When a model element of the type Virtual Machine is deleted, the synchronization engine will detect this change, identify which virtual machine this model element represents and finally invoke the management interface to delete it in the real system, and vice versa.

Definition of the Meta-Model. The meta-model specifies what kinds of elements can be managed. In our approach, the architecture-based meta-model is constructed as an Ecore meta-model (Merks et al, 2003). Each Ecore meta-model is stored as a file in XMI format and it can be displayed in a tree view or in a UML-like view called Ecore diagram. For better presentation, We use the Ecore diagram to present the Ecore meta-model, as shown in Figure 3, Figure 4 and Figure 5.

Figure 3 shows the architecture-based meta-model of the public cloud managed by OpenStack. This represents a public cloud that is operated and managed by a Cloud Service Provider. We can only view and use part of its resources that are allocated to our account, not necessarily all resources
managed by OpenStack. As shown in Figure 3, each kind of managed elements is represented as a class with some attributes. This public cloud provides each user with a resource unit called tenant. Each tenant is allocated with a certain amount of resources, and there are a group of images, servers, volumes, flavors, security groups, snapshots and floating IPs in it. Image denotes a virtual machine template providing a standardized group of hardware and software settings that can be used repeatedly to create new virtual machines configured with those settings. Server denotes a virtual machine. Flavor denotes the configuration template of CPU, memory and disk for creating a virtual machine. Floating IPs denote the public IP addresses allocated to a tenant, which can be associated with virtual machines.

Figure 4 illustrates the architecture-based meta-model of the private cloud managed by CloudStack. The attributes of each class are not shown for better clarity. As the CloudStack cloud is a private cloud, we can manage all the resource elements including the physical infrastructure and logical organization. For the physical infrastructure, the cloud platform consists of several datacenters. Each datacenter consists of several secondary storages (for storing templates, ISOs and snapshots), several system services (e.g., load balance service, console login service etc.), several physical networks (for providing network infrastructure), several virtual routers (for providing routing services for virtual machines) and several clusters. Each cluster consists of several primary storage units (for providing volumes) and a
set of physical hosts which are virtualized with the same kind of hypervisor. A group of virtual machines are deployed and run on each host. For the logical organization, the CloudStack cloud platform consists of a set of domains (e.g., each domain can denote a department of the company). A certain amount of resources (CPU, memory, IP addresses and so on) are allocated to each domain. Domain administrators have the authority to manage all the resources within the domain, while domain users can only view and use such resources. Each domain consists of several projects, each of which is a smaller resource unit created by domain administrators. Each project consists of a set of templates, snapshots, ISOs, security groups, networks, public IP addresses, firewalls and virtual machines. Besides the above resources, the CloudStack cloud also consists of a group of compute offerings (CPU and memory configuration templates), root administrators (who have the authority to manage all the cloud resources), and so on.

Figure 5 shows the architecture-based meta-model of ViNe. ViNe is software developed at the University of Florida for deployment and management of user-level virtual networks. A ViNe network consists of a set of Virtual Networks (VNs). Each VN consists of a set of Virtual Routers (VRs). Any machine deployed with ViNe software can serve as a VR, and VRs in the same VN have network connectivity.

Definition of the Access Model. The architecture-based meta-model specifies what kinds of elements can be managed. However, it does not specify how to manage these elements. Although there may be many management interfaces in a cloud system, we derive a unified access model by specifying how to invoke the interfaces to manage each element. The operations are summarized in Table 1 which include “Get”, “Set”, “List”, “Insert”, “Remove”, “Lookfor”, “Identify” and “Auxiliary”. Let’s take several important management actions of virtual machines as examples. “Create a VM” can be implemented by using an “Insert” operation. “Terminate a VM” can be implemented by using a “Remove” operation. “Pause a VM” and “Unpause a VM” both can be implemented by using “Set” operation. “Migrate a VM” can be implemented by using an “Auxiliary” operation.

By using the SM@RT tool, the access model can be easily constructed. As illustrated in Figure 6, the left part shows the access model of the OpenStack cloud. First, it imports all the managed elements from the architecture-based meta-model of the OpenStack cloud and illustrates the elements in a tree view. Second, we add related operations in the view to manage each element and its attributes. Third, we specify the implementation code for each operation. For instance, we add a “Get” operation (added as the <Logic:Get> tag in Figure 6) and a “Set” operation (added as the <Logic:Set> tag in Figure

Figure 4 Ecore diagram representing the meta-model of the private CloudStack cloud.

Figure 5 Ecore diagram representing the meta-model of VINE.
to the property “tenant_name” of the class “Tenant”. For the “Get” operation, we specify the code to invoke the API that gets the tenant_name attribute (the code is specified as the value of the <Feature:Primary> tag in Figure 6). For the “Set” operation, we specify the code to invoke the API that changes the name of the tenant in the real system. We also
add a “List”, an “Insert” and a “Remove” operations to the property “tenant_servers” of the class “Tenant”. For the “List” operation, we specify the code to invoke the API that gets all the servers in this tenant (the code fragment is shown in Figure 6). For the “Insert” and “Remove” operations, we specify the code to invoke the API that create a new server or remove an existing server in the real system. After all the operations are added and the implementation codes are specified, the SM@RT tool can automatically generate all the synchronization code between the runtime model and the running system, without the need for manual modifications.

Figure 7 A view of the runtime model of the public cloud

Figure 7 shows a view of the runtime model which is implemented in Eclipse. All the model elements are illustrated in a tree view. When you click at one element, the “Properties” window shows the values of all the properties of this element. The runtime model is synchronized with the real system. On one hand, when you make operations such as adding an element, removing an element and changing the value of a property, it will invoke corresponding codes to make these operations work in the real system. On the other hand, when the elements or properties are changed in the real system, it will be reflected in the runtime model as well.

3.2 CONSTRUCTION OF THE COMPOSITE MODEL

In order to manage all the participant clouds in a unified way, we construct a composite model by merging all the runtime models of each cloud and ViNe. The composite model provides a global view of all clouds, so that all the elements and data are aggregated into one model. After getting the composite model, we also have to guarantee the data synchronization between the composite model and the distributed models.

Model Merging. The common way of model merging is to merge model MA (short for model A) that conforms to meta-model MMA (short for meta-model A) and model MB (short for model B) that conforms to meta-model MMB (short for meta-model B) into a model MC (short for model C) that conforms to meta-model MMC (short for meta-model C). In our approach, the composite model is just a simple aggregation of all distributed models, so the meta-model of the composite model is also an aggregation of all distributed meta-models, specifically, it’s an aggregation with systematic renaming of classes from different models so to ensure unique names in the meta-model of the composite model. We developed a tool to generate the meta-model of the composite model. Based on a configuration file created by a developer, the tool automatically generates the composite meta-model. Figure 8 shows one example of the configuration file. The URI, NAME and ROOT defines the namespace URI, name and root class of the target metamodel. ECOREPATHS defines the file path of each meta-model. ROOTCLASSES defines the root class of each meta-model. As there may be classes with the same name in multiple meta-models, we add a unique prefix to the classes in each meta-model, which is defined by PREFIXES. Finally, the TARGETPATH defines the save path of the target meta-model. The composite meta-model generation process is quite simple. First, we specify the namespace URI of the composite meta-model and create the root class. Second, we rename all the classes in each distributed meta-model by adding a unique prefix. Third, we add the main contents of each renamed distributed meta-model to the composite meta-model as son nodes of the root class. Then the composite meta-model is constructed. Figure 9 shows a fragment of the meta-model of the composite model. We can see that the meta-models of CloudStack Cloud, OpenStack Cloud and ViNe all become son nodes of the root class “Composite”, and the names of the classes have been changed by adding

Figure 8 A sample configuration for generating the meta-model of the composite model

Figure 9 A fragment of the meta-model of the composite model
prefixes “CS_”, “OP_” and “VN_”, while the attributes of each class and the relationship between the classes keep unchanged.

**Data Synchronization.** Data synchronization between the composite model and the distributed models is done by transferring the model operations to their original models. On the one hand, when the administrator operates on the composite model, the model operations are transferred to the corresponding distributed models. On the other hand, changes of the distributed models are automatically discovered through periodic comparisons with their previous copies, and then the changes are transferred to model operations, which will be executed on the composite model.

Particularly, only the “Set”, “Add” and “Remove” operations can lead to changes of the runtime models. Figure 10 shows how the operation of creating a server is transferred from the composite model to the runtime model of the OpenStack cloud. The code transferred to the runtime model details the operation to be executed:

a) Query: Find the parent element – a tenant whose “tenant_id” is “9e850a...”.

b) Add: Create an element of “Server”.

c) Set: Carry out the configurations of the new “Server” element.

After the new “Server” element is created, the synchronization engine will propagate the new change of the runtime model to the real system.

![Figure. 10 The operation transferred from the composite model to the distributed model](image)

### 3.3 Model Transformation

As described in the motivating example, there are different management requirements, each of which consists of specific management scenarios and appropriate management styles. For instance, users in different roles have different authority and limitations on using and managing cloud resources. Root administrators can manage all the resources and invoke all the management interfaces. Domain administrators only have the authority to manage resources within their domains by invoking management interfaces operating on the resources in such domains. Domain users only have the authority to view and use the resources that belongs to their authorized domains. To meet these personalized management requirements and provide personalized management views for different types of users, we define the customized model. Different from the composite model which is an aggregation of the distributed runtime models, the customized model is constructed through model transformation from the composite model according to the personalized management requirements.

The common form of model transformation is transforming a model MA (source model) that conforms to meta-model MMA to a new model MB (objective model) that conforms to meta-model MMB. The transformation is based on a set of mapping rules between the two models. The mapping rules describe the mapping relationship between the elements of two models. Every attribute of the element in the customized model is related with one of the elements in the composite model. All changes on the customized model will be transformed to operations on the composite model, and vice versa. We define the description methods of mapping rules (main keywords are shown in Table 2) and implement a model transformation tool which can generate transformation codes (written in QVT) automatically based on the mapping rules.

**Table 3** presents three types of basic mapping rules:

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Descriptions</th>
<th>Keywords</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>helper</td>
<td>Mapping Rules</td>
<td>type</td>
<td>Types of Mapping Rules</td>
</tr>
<tr>
<td>mapper</td>
<td>There is a mapping relationship between the attributes of the objective element and the source element.</td>
<td>query</td>
<td>There is a mapping relationship between the attributes of the objective element and the element that is related to the source element.</td>
</tr>
<tr>
<td>src</td>
<td>Elements or Attributes of Source Models</td>
<td>obj</td>
<td>Elements or Attributes of Objective Models</td>
</tr>
<tr>
<td>condition</td>
<td>Preconditions</td>
<td>node</td>
<td>Types of Objective Elements</td>
</tr>
</tbody>
</table>

Table 2 Keywords of the Description of Mapping Rules

relationships between model elements: One-to-One, Many-to-One and One-to-Many. For example, the element of class “OP_Image” in the composite model is transformed to the element of class “Template” in the customized model, as well as their attributes. It is a One-to-One mapping. The element of class “CS_VirtualMachine” and the element of class “VN_VirtualRouter” in the composite model are transformed to the element of class “VirtualMachine” in the customized model. Some of the attributes of “VirtualMachine” class are
taken from “CS_VirtualMachine” class, such as “vm_id”, “vm_name”, “vm_privateip” and “vm_macaddr”. The others are taken from “VN_VirtualRouter” class such as “vm_vrid” and “vm_virtualip”. Therefore, this is a Many-to-One relationship. The element of class “CS_Volume” in the composite model is transformed to the element of class “RootVolume” or the element of class “DataVolume” according to the value of its attribute “volume_type”. Therefore, this is a One-to-Many mapping relationship. The description of mapping rules and generated QVT code of the three examples are shown in Figure 11.

Figure 12, Figure 13 and Figure 14 illustrate the meta-models of the customized models for root administrators, domain administrators and domain users. In all the customized models, the elements of “OP_Tenant” are transformed into elements of “Project” belonging to a special element of “Domain”. The elements of “CS_ComputeOffering” are transformed into elements of “Flavor”. The elements of “CS_Volume” and “OP_Volume” are both transformed into elements of “RootVolume” and “DataVolume”. The elements of “CS_VirtualMachine” and “VN_VirtualRouter” are transformed into elements of “VM”. The elements of “VN_VirtualNetwork” are transformed into the elements of “VirtualNetwork”. In the root administrator’s customized model, all of the elements can be managed. In the domain administrator’s customized model, only the elements under this domain can be viewed and managed. In the domain user’s customized model, only the virtual machines in this domain can be viewed and used. Through model transformation from the composite model to customized models, personalized management requirements are easily fulfilled.

After the customized model is constructed, management tasks such as VM scheduling, VM placement and high availability can be carried out through execution of different QVT programs on the customized model, without the need to interact with the management interfaces of underlying cloud systems.

3.4 INTER-CLOUD NETWORK CONNECTION
In our approach, every VM serves as a VR – this enables full flexibility of network management, but can potentially exhibit negative impact on network performance (Tsugawa et al., 2009). The elements of “VN_VirtualRouter” and the elements of “OP_Server” or “CS_VirtualMachine” are transformed to the elements of “VM”. Every time an element of “VM” is created on the customized model, an element of “VirtualRouter” is created on the runtime model of ViNe, and the ViNe software is configured automatically on this VM. By using our approach, the administrators don’t have to perform complex and error-prone virtual network configuration.

Table 3 THREE TYPES OF MAPPING RELATIONSHIPS

<table>
<thead>
<tr>
<th>Classes in Source Model</th>
<th>Classes in Objective Model</th>
<th>Examples of Transformation Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OP_Image</strong></td>
<td><strong>VirtualMachine</strong></td>
<td><code>vm_virtualip := vmodel.objectsOf(Type(VN_VirtualRouter)) -&gt;select(vr_macaddr=self.vm_macaddr) -&gt;selectOne(true).vr_virtualip</code></td>
</tr>
</tbody>
</table>
| **CS_VirtualMachine**   |                           | `if (self.volume_type=="ROOT") {
  return object RootVolume {...}
} else if (self.volume_type=="DATA"){
  return object DataVolume {...}
}` |

Figure 12 The meta-model of the customized model for root administrators
4. EVALUATION

The previous sections have given a detailed description of our model-defined approach. In this section, we present a set of experiments to evaluate our approach. The experiment is performed on two cloud environments: one cloud managed by CloudStack deployed at the University of Florida, and another cloud managed by OpenStack deployed at the University of Chicago. We have applied our model-defined approach to the development of a CCMan system. We evaluate our approach from the following four aspects.

4.1 FEASIBILITY OF OUR APPROACH

A runtime model is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective. It has been broadly adopted in the runtime management of software systems. All the resources are modeled as managed elements in the runtime model and all the management interfaces can be abstracted as operations on the model elements. When making operations such as adding or removing an element or changing the value of a property, it will invoke the corresponding APIs to make these operations work in the real system. Conversely, when the elements or properties are changed in the real system, this is reflected in the runtime model as well. Management tasks, such as creating new VMs, deleting VMs, checking VM state, can be done by the model-level programs. For instance, “Pause a VM” can be implemented by using “Set” operation. When the state of VM is set to “Stopped”, our system will invoke the APIs to Pause the virtual machines. From this perspective, there is no difference between development based on system management interfaces and development based on runtime models in regards to feasibility.

4.2 DIFFICULTY OF PROGRAMMING USING MODEL-LEVEL LANGUAGES

In our approach, management tasks are developed by using model-level languages such as QVT. To evaluate the programming difficulty of using model-level languages, we compared with the programming difficulty of directly using management interfaces. We develop an automated program to check resource utilization of the physical servers and find the free or busy.

Figure 15 describes the code fragments to find the free servers whose memory utilization is less than 40% and busy servers whose memory utilization is more than 80%, which are written in Java and QVT respectively. In the Java program, we traverse all the physical servers in the cloud and check each one’s memory utilization. The memory utilization of each physical server is got from invoking a script (/opt/xen/getUsedMem.sh) and parsing the result. While in the QVT program, with the help of the architecture-based model, the low-level details such as the parameters of APIs and data processing are shielded. The runtime model provides the runtime information of all the managed elements and is synchronized with the underlying system, which makes the developers focus on the logic of management tasks. From the amount of code view, the QVT program is shorter and more easily understandable than Java program. In addition, the modeling language provides operations such as “select”, “sum” and so on in the model level, which makes it simpler to do programming. It is much easier to programming using model-level languages.

4.3 DEVELOPMENT COST OF OUR APPROACH

We defined the architecture-based meta-model and the access model of each cloud, ViNe, and other management systems using the Eclipse Modeling Framework (EMF). The runtime model was generated by the SM@RT tool. So, in addition to being simple to build, the distributed runtime models only need to be constructed once.

In order to construct the composite model, a developer only needs to specify some configurations and the meta-model of the composite model. The composite model itself is generated automatically. The synchronization between the composite model and the distributed models is also provided.

Transformation of a composite model to customized models is accomplished by defining the meta-model of the customized model and mapping rules according to
personalized requirements. The transformation code is generated automatically.

To sum up, we have developed tools that can automate or semi-automate the development of CCMan systems. Input required from developers includes architecture-based meta-models, the access models, and transformation mapping rules. As the model is an abstraction of the underlying system, it is easier to develop at model-level compared to code-level.

4.4 Execution Efficiency of Model-Level Programs

We have made some experiments to compare the execution efficiency of management interface invocations (Java) with model-level programs (QVT) showed in Table 4. Four groups of management tasks were used: creating new VMs, deleting VMs, getting the “usedMemory” attribute of the VMs, and setting the “name” attribute of the VM. For instance, creating new VMs is an ‘Insert’ operation that inserts the VM elements into the value list of this property, as we mentioned. For each task, we made three experiments containing different numbers of VMs, and recorded execution time and data delay time. The execution time is the average time cost of each group of management tasks, and the data delay is the average delayed time to obtain the data. The results of experiments are summarized in Figure 16.

For the “create new VMs”, “delete VMs” and “set ‘name’ attribute” management tasks, the execution time of Java programs is lower than the QVT ones. The main reason is that both programs are based on the same management APIs and the runtime model-based approach requires extra operations to ensure the synchronization between the runtime model and real system. To be specific, our programs would transform the QVT and invoke the APIs which made our execution time longer, and our performance was influenced by the scale and complexity of the runtime system. These reasons made our approach a little slower and in fact, the difference is sufficiently small and can be considered acceptable.

For the “get ‘used-memory’ attribute” management tasks, the execution time of Java programs is longer than the QVT ones, but the data delay of QVT programs is longer than the Java ones. There are two main reasons: (1) Java programs query the attributes of appliances by directly invoking the management interfaces, so the execution time increases...
linearly with the number of the appliances and the data delay is very small; (2) the runtime model is equivalent to the
snapshot of system metrics and getting the attributes of appliances just needs a read operation, so the execution time
of the QVT programs is shorter. The runtime model is synchronized with the running system requiring the traversing of all the metrics of the running system, so the data delay increases linearly with the size of the model. For our experiments, when the number of VMs continued to increase, the data delay of our approach is almost unchanged and the execution time is still short while Java programs’ time would increase linearly and finally much longer than ours.

5. RELATED WORK

In recent years, the interest in CCMan systems has been increasing both in academia and industry. Some libraries such as Apache jclouds, Apache Libcloud, Apache Deltacloud provide abstraction layers facilitating the provisioning and deployment of multiple cloud systems through a unified interface. They support numerous IaaS providers such as OpenStack, CloudStack, Eucalyptus, Rackspace, vCloud and Amazon Web Services. Commercial products and research projects such as RightScale, EnStratus, and mOSAIC(Sandru et al., 2012) also adopt a similar approach. While these products and projects effectively foster their deployment and maintenance, they remain on the code-level, which makes redesign difficult and error-prone. Liu et al. (Liu et al., 2011) propose a Multi-cloud management platform that is located between cloud users and cloud sites and provides unified cloud services from the SOA perspective. N. Ferry et al. (Ferry et al., 2013) propose a model-based framework called CLOUDMF to manage multiple clouds. It uses a tool-support domain-specific modelling language to model the provisioning and deployment of Multi-cloud systems, and uses a models@run-time environment for enacting the provisioning, deployment and adaptation of these systems. Liu and N. Ferry’s work are on higher level than code-level, but they both lack the support of network connection between different clouds. Ines Houidi et al. (Houidi et al., 2011)
present a cloud broker framework to enable inter-cloud links between two different clouds by using OpenFlow technology, however, it needs the network infrastructure to support OpenFlow, which is currently not widely deployed. Pierre Riteau et al. (Riteau et al., 2006) present an approach to building dynamic computing infrastructures over distributed clouds and propose an inter-cloud live migration mechanism called Shrinker. It shows good inter-cloud network performance, however, it uses a single cloud management system (Nimbus) to achieve the unified management of multiple clouds, which is at code-level and not flexible to be redesigned to meet personalized management requirements.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a model-defined approach to the development of a CCMan system. Through model construction, model merging and model transformation, multiple clouds can be managed in a unified and customized manner. Evaluation of the approach from several perspectives shows that the efforts needed to both develop the CCMan system and operate its services are significantly reduced with negligible performance loss.

In a runtime model, all the resources are modeled as managed elements and all the management interfaces are abstracted as manipulations on the elements. From this perspective, there’s no difference between developing based on system management interfaces with developing based on runtime models in regards to feasibility. In our approach, we also guarantee the synchronization between the customized model, the composite model and the distributed models. Therefore, our approach is feasible.

In future, we plan to leverage the developed base system to implement advanced management tasks such as virtual machine placement in a Cloud of Clouds environment.

7. ACKNOWLEDGMENT

This work was supported by the High-Tech Research and Development Program of China under Grant No.2015AA01A202, the National Natural Science Foundation of China under Grant No.61421091, 61370020 and the National Science Foundation (NSF) of United States under Grant CNS-1265341. Any opinions, findings or conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and/or NSF-C. The authors also acknowledge the support from FutureSystems that provided the OpenStack cloud platform for the experiments.

8. REFERENCES


Authors

**Bo An** is a Ph.D. student in the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. His research interests include cloud computing and distributed system.

**Zhicheng Cui** is a master student in the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. His research interests include operating system and cloud computing.

**Mauricio Tsugawa** is a research professor of Electrical and Computer Engineering and Computer Science at the University of Florida. His research interests are in the areas of distributed computing, parallel processing, computer architecture and configurable logic design.

**Ying Zhang** received his PhD in electronics engineering and computer science from Peking University in 2012. He is a researcher at Peking University. His research interests are in the area of distributed computing with a focus on middleware, including the construction and management of middleware, software engineering with a focus on component based development, and mobile computing.

**Xuanzhe Liu** is an Associate Professor in the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. He is also with Beida (Binhai) Information Research. His research interests are in the area of services computing, mobile computing, and big data analytic.

**Gang Huang** is a Full Professor in Institute of Software, Peking University. He is also with Beida (Binhai) Information Research. His research interests are in the area of middleware of cloud computing and mobile computing. He is the corresponding author of this article.

**Joséa. B. Fortes** received the PhD degree in electrical engineering from the University of Southern California, Los Angeles, California. He is the faculty of the Department of Electrical and Computer Engineering, University of Florida, Gainesville, as Professor and AT&T Eminent Scholar. His research interests are in the areas of distributed computing, autonomic computing, computer architecture, parallel processing, and fault-tolerant computing. He is a fellow of the American Association for the Advancement of Science (AAAS).