Towards Architecture-based Management of Platforms in Cloud

Gang Huang, Xing Chen, Ying Zhang, Xiaodong Zhang

Towards Architecture-based Management of Platforms in Cloud

Gang Huang, Xing Chen, Ying Zhang, Xiaodong Zhang
Key Laboratory of High Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
hg@pku.edu.cn; { chenxing08, zhangying06, zhangxd10 }@sei.pku.edu.cn

***Abstract*—System management becomes increasingly complex and brings high costs, especially with the advent of Cloud Computing. In a Cloud, numerous platforms like Virtual Machines (VMs) and Middleware have to be managed to make the whole system work cost-effectively after an application is deployed. For controlling the management cost, in particular the manual management cost, many computer programs have been developed to take over manual management tasks or reduce their complexity and difficulty. These programs are usually hard-coded by languages like Java and C++, which bring enough capability and flexibility but also cause high programming effort and cost. This paper proposes an architecture based approach to developing the management programs in a simple but powerful enough manner. First of all, the manageability (such as APIs, configuration files and scripts) of a given platform is abstracted as a runtime model of the platform's software architecture, which can automatically and immediately propagate any observable runtime changes of the target platforms to the corresponding architecture models, and vice versa. Then the management programs will be developed using modeling languages, instead of those relatively low-level programming languages. Such architecture-level management programs bring many advantages related to the performance, interoperability, reusability and simplicity. The experiment on a real-world cloud and the comparison with the programming language approach demonstrate the feasibility, effectiveness and benefits of the new approach for management program development.**

Keywords - Cloud Management; Software Architecture; Models at Runtime.

I. INTRODUCTION

Nowadays, more and more software applications are built or migrated to run in a cloud, with the goal of reducing IT costs and complexities. The layers of cloud computing can be divided into three kinds, including Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service, which sit on top of one another. Other "soft" layers can be added on top of these layers as well, with elements like cost and security extending the size and flexibility of the cloud [1]. This trend brings unprecedented challenges to system management of Cloud. The increasingly efforts of platform (note that there's no consensus on the definition of platforms in cloud, while we consider the virtual machines, operating systems and middleware as platforms in this paper) management mainly come from the following two aspects:

On one hand, the virtualization makes the physical resources easier to share and control but increases the complexity of management mainly because the virtualized resources are much more and less reliable than the physical ones. For instance, given an application that uses 10 nodes, cloud administrators have to manage the required 10 VMs as well as the physical nodes hosting these VMs. On the other hand, the service oriented natures of cloud make the management much more complex than the product centric natures of traditional datacenters because cloud applications can use different types of platforms and require resources on demand. For instance, a 3-tier JEE (Java Enterprise Edition) application typically has to use the web server, EJB server and DB server. These servers have different management mechanisms. An EJB server should comply with JMX management specification and rely on the JMX API, while a DB server is usually managed through the SQL-like scripts. In addition, the EJB server can usually sustain the running of several applications simultaneously. What's more, all of the platforms are in a resource sharing and competing environment. Administrators have to carefully coordinate each part to make the whole system work correctly and cost effectively.

To tame the complexity of manual system management, many programs are built to carry out management tasks automatically. Such a management program usually uses with the four-stage autonomic loop proposed by IBM [2]: monitor the runtime system and collect the critical data concerned, analyze the collected data to find if the system needs a reconfiguration, plan a proper reconfiguration procedure, and execute reconfigurations on the system. Such a management program is usually implemented in general purpose programming languages like Java and C/C++, which can bring enough power and flexibility but also cause high programming effort and cost. For instance, the existing VM and middleware platforms have already provided adequate proprietary APIs (e.g., JMX) to be used by monitoring and executing related codes. Administrators first have to be familiar with these APIs and then build programs upon them. Such a work is not easy due to the heterogeneity of platforms and the huge numbers of APIs provided. In a management program, proper APIs have to be chosen for use and different types of APIs (e.g., JMX and scripts) have to be made interoperable with each other. Such "boring" work is not the core of the management logics comprised by analyzing and planning related codes, but it has to be done to make the whole program run effectively. During this procedure, the irrelevant APIs as well as the collected low-level data can sometimes make administrators exhausted

and frustrated. Furthermore, as the programs are built on the codes that directly connect with the runtime systems, they are not easy for reuse. Administrators have to write many different programs to manage different cloud applications and their platforms even the management mechanisms are the same. In addition, hard-coding the analyzing and planning related codes will also bring high costs. Although many advanced techniques such as model checking [3] can help to mitigate the complexity, administrators have to adapt the codes to the requirements of various model checkers. Therefore, the finally generated codes are very long and difficult to understand.

The fundamental challenge faced by the development of management tasks is the conceptual gap between the problem and the implementation domains [4]. To bridge the gap, using approaches that require extensive handcraft implementations such as hard-coding in general purpose programming languages like Java will give rise to the programming complexity. Software architecture acts as a bridge between requirements and implementations. It describes the gross structure of a software system with a collection of managed elements and it has been used to reduce the complexity and cost mainly resulted from the difficulties faced by understanding the large-scale and complex software system recently [5]. So it is a natural idea to understand management tasks through modeling the architecture of the cloud. Current research in the area of model driven engineering (MDE) supports systematic transformation of problem-level abstractions to software implementations [6]. The complexity of bridging the gap could be tackled through developing automated programs based on the model that describes the architecture of the cloud and through the automated support for transforming architectural models to running systems and vice versa. What's more, many model-centric analyzing and planning methods and mechanisms are already developed for use [6], such as architecture styles, constraints and model checkers. Programs based on models can benefit from these techniques to build their own analyzing and planning parts.

This paper proposes a runtime architecture-based approach to managing the platforms such as middleware and VMs of the cloud. We construct an architecture-based model of the cloud for platform management and ensure the proper synchronization between the system and its model. Any change of the runtime model will be immediately propagated into the runtime system and vice versa. Then we evaluate our approach by comparing it to the hard-coding approach in terms of the performance, interoperability, reusability and simplicity. The evaluation results prove that the runtime architecture-based management is cost-effective and promising in the cloud environment.

The rest of this paper is organized as follows: Section II presents an example to show the importance and necessity of architecture-based management of platforms in Cloud. Section III and IV describe our approach in detail. Section V and VI report the evaluation and related work of our approach. Section VII concludes this paper and identifies our future work.

II. MOTIVATING EXAMPLE

Many automated programs have been built to tame the complexity of manual management and most of them are hard-coded in general purpose languages like Java. However, it may result in several difficulties to develop management programs in such general purpose languages. For instance, platforms in the cloud consist of different types of resources which need to be managed collaboratively. Administrators have to be familiar with the management APIs and then build programs upon them. While developing a management program, they have to choose proper APIs for use and make different types of APIs interpretable with each other, as shown in Figure 1.

```
/*
 * JAVA: To get the value of the maxThreads of a JOnAS
 * through the JMX.
 */
public int getMaxThreads(String port)
{
    //To prepare to invoke the interface
    String objName = "jonas:type=Connector,port=" + port;
    String attributeName = "maxThreads";
    MBeanServerConnection mbeanServerConn = null;
    try
    {
        JMXServiceURL connURL = new JMXServiceURL(
        "service:jmx:rmi://localhost/jndi/rmi://localhost
        :1099/jrmpconnector_jonas");
        JMXConnector connector = JMXConnectorFactory.
        newJMXConnector(connURL, null);
        connector.connect(null);
        mbeanServerConn = connector.getMBeanServerConnection();
    }
    ...
    //To invoke the specific management interface
    try
    {
        attributeValue = (Integer) mbeanServerConn.
        getAttribute(obj, attributeName);
    }
    catch (AttributeNotFoundException e)
    {
    ...
```

```
/*
 * JAVA: To set the value of the memory of a virtual machine
 * through the script
 */
public String setMem(String imgIp, String mem)
{
    try
    {
        String[] args = new String[4];
        args[0] = "sh";
        args[1] = "/opt/xen/setMem.sh";
        args[2] = imgIp;
        args[3] = mem;
        ProcessBuilder builder = new ProcessBuilder(args);
        Process process = builder.start();
        ...
```

Figure 1. An Example for Programming in Java

Such code fragments are not the core of management logics compared with the analyzing and planning related codes, but it has to be developed to make the whole program run effectively. Many similar code fragments are required for a simple task. As shown in Figure 1, the code fragment for fetching the value of the "maxThreads" attribute in a JOnAS (a popular open source Java application server) through JMX API is more than 20 LOC (Line of Code).

What's more, administrators have to construct the adapters to invoke the different types of management interfaces such as JMX and Script.

When using our approach, the programs become much simpler and shorter. Figure 2 shows the architecture-based program doing the similar management task, which is written in QVT [7], a widely adopted modeling language. The architecture-based model can shield programmers from the relatively low-level details of the managed platforms.

```
/*
 * QVT: To do memory allocation depending on
 * the value of the thread pool
 */
modeltype Cloud use "http://Cloud";
transformation Check(inout cloudModel:Cloud);
mapping Platform::check()
    when{self.usedThreadPool > 300 and self.Memory < 2048;}
    {
        self.Memory := 2048;
    }
main()
{
    cloudModel.objectsOfType(Platform)->map check();
}
```

Figure 2. An Example for the Languages of QVT

With the help of the runtime architecture-based model, administrators can focus on the management targets (e.g. VMs and middleware) and program in the architecture level, without developing code fragments to invoke management APIs. The architecture-based model is abstracted from the underlying infrastructure of Cloud as shown in Figure 3, and the synchronization engine is needed to reflect the cloud into a model and ensures a bidirectional consistency between the system and the model. For instance, in this scenario, the synchronization engine must build a model element for the JOnAS platform in the runtime model. When the management program deletes the model element of JOnAS, the synchronization engine must detect this change, identify which platform this removed element stands for and finally invoke the script to shut down the JOnAS platform.

III. An Architecture-based Model for Platform Management in Cloud

A. Approach Overview

We provide an architecture-based runtime model for administrators to develop automated programs of platform management in the architecture level, and the correct synchronization between the model and the runtime system is ensured. The inputs of our approach include a meta-model for platform management specifying what kinds of elements can be managed in Cloud and an Access Model of the configurations specifying how to use the management APIs to monitor and modify those manageable elements. Then the runtime software architecture of the target system is automatically constructed by the code generated by SM@RT tool, which is proposed in [8].

The approach is applicable on the following premises. First, the SM@RT tool is not intrusive, that is, neither instructs non-manageable systems nor extends inadequate APIs. As a result, the managed elements in Cloud such as virtual machines, operating systems and middleware should provide their own management mechanisms, API or script. This premise is feasible for the popular and well-developed platforms. Second, we reflect a direct model for the cloud (that means the model is homogeneous with the architecture of the cloud: each model element stands for one managed element in the runtime system). Note that SM@RT supports automatically just-in-time synchronization between two heterogeneous models [23] and then cloud administrators can define their own architectural models and the mapping to our built-in models.

B. The Architecture-based Meta-Model

As shown in Figure 3, physical nodes are the basic elements to compose the foundation of the cloud. The virtualization handles how images [1] of operating systems, middleware, and applications are pro-created and allocated to the given physical machines. The images could be moved around and put into production environment on demand. The virtual machines occupy resources such as computing power, memory and so on from physical machines. Upon them, different types of operating systems organize resources to support the basic environment for software running and network accessing. There is always only one middleware product in a virtual machine for the reason of isolation. The virtualized resources, operating system and middleware compose the platform and several platforms are organized properly to provide the runtime environment for a whole system. These elements above should be managed collaboratively. Therefore, we construct an architecture-based meta-model of the cloud for platform management as

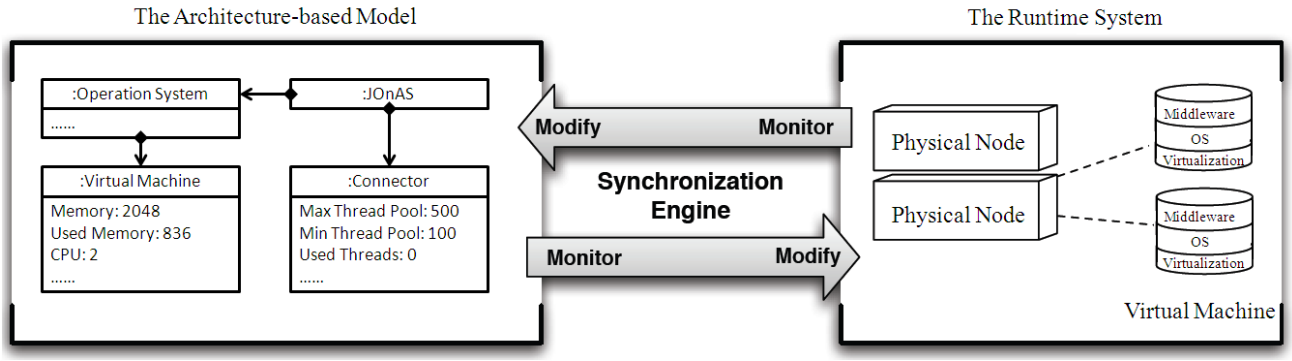


Figure 3. A Common Structure of the Synchronization Engine between the Architectural Model and Runtime System

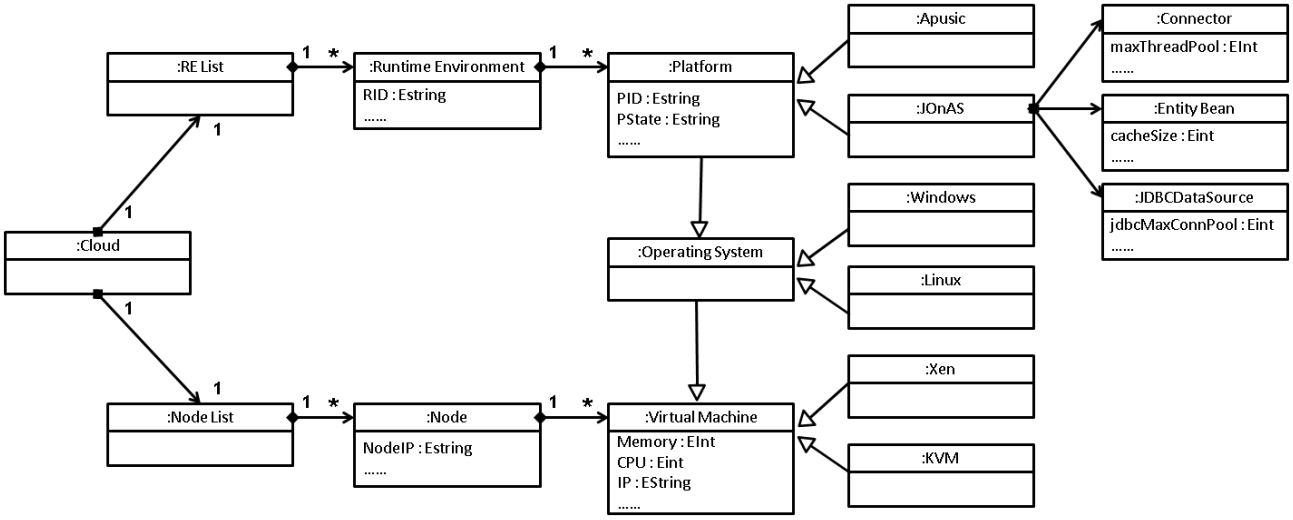


Figure 4. The Architecture-based Meta-model for Platform Management in Cloud

shown in Figure 4, whose instantiation is the runtime model.

The *NodeList* class (lower left) represents the list of physical nodes in the cloud, which compose the shared infrastructure. The *Node* class and the *VirtualMachine* class separately represent physical nodes and virtual machines. Different types of virtualization products such as Xen and KVM are similar in management APIs, although they are different in implementations. So the *Xen* class and the *KVM* class (lower right) may be regarded as the subclasses of the *VirtualMachine* class. Therefore, the elements of nodes and virtual machines, and the relations between them in the model reflect the working conditions of the shard infrastructure in the cloud.

The *RuntimeEnvironment* class (upper) represents the runtime environment for a whole system, which may contain more than one platform. The *Platform* class represents the platform which is the main managed element in the model. Platform, OS and VM consist of VM appliance [9] which may be regarded as one entire managed element. Therefore, the *Platform* class inherits the *OperatingSystem* class and the *VirtualMachine* class. Through the model elements of *Platform*, the attributes of the operating system and the virtualized resources may be accessed as well. And the subclasses of the *Platform* class represent different types of middleware products such as *JOnAS* and *Apusic*, whose architecture we have discussed in [8].

The architecture-based meta-model specifies what kinds of elements can be managed in Cloud and would help administrators understand their management tasks.

C. *Runtime Changes*

Given the architecture-based meta-model, we also need to identify the changes enabled by the model. Depending on the nature of the initiating agent, the changes can be classified as external changes (initiated by external entities) or internal changes (applied by the management system). For example, it is an external change that a node in the cloud does not work properly. Then the internal changes should be adopted to adjust it.

In this context, it becomes clear that for management purposes it is important to provide a comprehensive identification of the internal changes in Cloud, as they define the scope of potential actions that can be applied by the automated programs. The cloud platform can be managed at the levels of middleware, operating systems and virtualized resources. Figure 5 provides a short list of the primitives which includes several main types of management operations. For each operation we detail the management operation names, the required arguments and the changes it causes to the configuration when applied (including value changes and the existence or not of the elements) and the management operations. It is possible to remove existing elements, and

Name	Argument	Post Condition
CreateAVM	Node rn, Image ri, Property[] props	$\exists$ VM rv, rv $\in$ rn.vms $\wedge$ rv instanceof ri $\wedge$ props $\subseteq$ rv.properties
ShutdownAVM	Node rn, VM rv	rv $\notin$ rn.vms
MigrateAVM	Node rnI, VM rv, Node rnO	rv $\notin$ rnI.vms $\wedge$ rv $\in$ rnO.vms
PauseAVM	Node rn, VM rv	rv $\in$ rn.vms $\wedge$ rv.state = STOPED
UnpauseAVM	Node rn, VM rv	rv $\in$ rn.vms $\wedge$ rv.state = STARTED
ConfPProp	RuntimeUnit ru, Property[] props	props $\subseteq$ ru.properties

Figure 5. Definition of some Runtime Changes

Name	Meta element	Parameter	Description
Get	Property(1)	-	Get the value of the property
Set	Property(1)	newValue	Set the property as newValue
List	Property(*)	-	Get a list of values of this property
Add	Property(*)	toAdd	Add toAdd into the value list of this property
Remove	Property(*)	toRemove	Remove toRemove from the list of this property
Lookfor	Class	Condition	Find an element according to condition
Identify	Class	Other	Check if this element equals to other
Auxiliary	Package	-	User-defined auxiliary operations

Internal Changes	Manipulation
CreateAVM	Add
ShutdownAVM	Remove
MigrateAVM	Auxiliary
PauseAVM	Set
UnpauseAVM	Set
ConfPProp	Set

Figure 6. All Kinds of Manipulations

instantiate new runtime elements defined. The attributes of the existing units can also be adjusted, with controlling over their property configurations.

Although there are hundreds of management APIs in the cloud, we could model them into the Access Model [8] through specifying how to invoke the APIs to manipulate each type of elements. Where meta-element is the set of all the elements in the architecture-based meta-model (classes, attributes, etc.), the manipulation is the set of all types of manipulations, which are summarized in Figure 6. The management operations in Figure 5 are also classified according to the rules.

IV. Implementations of the Runtime Architecture-based Model

We define the architecture-based meta-model and the Manipulation on the Eclipse Modeling Framework (EMF) [10], and then generate the synchronization codes to maintain the causal links between model and system through our SM@RT tool [8]. The tool is an extension of EMF, and it generates the model listener, model proxy, and system proxy specific to the target system. Specifically, it generates a Java class for each of the MOF classes in the system meta-model, implementing the EObject interface defined by Ecore.

We have also constructed some architecture-based runtime model of virtualization products like Xen and JEE servers like JOnAS and Apusic, as shown in Figure 7. It should be noted that the size of these models just reflects the manageability, not the size or functionality, of these products. Using these architecture-based models, administrators can cost-effectively develop automated programs in modeling languages for platform management in Cloud.

Runtime System	Number of Elements	Number of Manipulations
JOnAS	28	271
Apusic	26	351
Xen	8	33

Figure 7. Information about Some Architecture-based Models

V. Evaluation

The previous sections have given the detailed description of our runtime model of the cloud. In this section, we present a set of experiments to evaluate our approach. The experiment is done on a cloud environment: Internetware Test Bed [11]. It is a research cloud project supported by the Nation Key Basic Research and Development Program of China. The Internetware Cloud provides on-demand VMs as well as middleware infrastructures (e.g. JEE server and DB server) for cloud users. We have applied the architecture-based model to the cloud and written a set of automated programs in QVT on the model.

We evaluate our approach by comparing it with the hand-coding approach in two scenarios. In the first experiment, we take anti-pattern detection [12] for example to prove that our approach has advantages related to the performance and the reusability. In the second experiment, we take VM states checking for example to validate the advantages of our approach on the interoperability and simplicity.

A. Anti-pattern Detection

A pattern is a kind of conclusion of well-known experience, which describes an effective solution to repeated problems. As an extension, an anti-pattern describes a commonly occurring solution that generates decidedly negative consequences. With the architecture-based model, administrators can easily develop programs of anti-pattern detection in a simple way compared with hard-coding in the languages like Java. As shown in Figure 8, six classic anti-patterns about JSP and Servlet in JEE applications are concluded and classified. We separately develop management programs in Java and QVT to detect these anti-patterns in benchmark JEE application “Ecperf”. The results and the executing time are also shown in the figure.

The two groups of automated programs have the same results of anti-pattern detection. It is easy to see that the executing time of the Java programs is less than the QVT ones. The main reason for this is that the two sets of programs are based on the same management APIs and there are some extra operations in architecture-based approach, which are aimed to ensure the synchronization between the model and the runtime system. There are complex factors that affect the performance of synchronizers: first, the execution time of synchronization process is constituted of the time spent on QVT transformation and the API invocations. Second, the performance is affected by both the complexity and the scale of the runtime system architecture.

NO	Anti-pattern about JSP and Servlet	Ecperf	Executing Time(ms)			Line of Code		
			Java	QVT	Extra Time	Java	QVT	Reduced LOC
1	Ignoring Reality	√	46	91	98%	23	5	78%
2	Too Much Code	√	431	709	65%	22	5	77%
3	Embedded Navigational Information	√	293	453	55%	32	5	84%
4	Too Much Data in Session		775	1135	46%	26	5	81%
5	Ad Lib Taglib		735	1172	59%	25	5	80%
6	Accessing Entities Directly	√	53	87	64%	23	5	78%

//Programming in Java

```
1.  //JAVA: Too Much Data in Session
2.  EList<JSP> jsps;
3.  //To get the list of JSPs
4.  jsps = new EObjectResolvingEListForWrapping<JSP>(
5.          JSP.class, this,
6.          JOnASPackage.WEB_MODULE_JSPS,
7.          JOnASPackage.eINSTANCE.getJSP());
8.  /*
9.   * To count the number of the pattern of
10.  * "session.setAttribute"in each JSP
11.  */
12. ((EObjectResolvingEListForWrapping<JSP>) jsps)
13. .refreshWrap();
14. for(JSP j: jsps)
15. {
16.     java.util.regex.Pattern pattern = java.util.regex.
17.     Pattern.compile("session.setAttribute");
18.     java.util.regex.Matcher matcher = pattern.matcher(
19.             j.getCode());
20.     int count = 0;
21.     while(matcher.find())
22.         count ++;
23.     if(count > 10)
24.         return 0;
25. }
26. return 1;
```

```
1.  //JAVA: Ad Lib Taglib
2.  EList<JSP> jsps;
3.  //To get the list of JSPs
4.  jsps = new EObjectResolvingEListForWrapping<JSP>(
5.          JSP.class, this,
6.          JOnASPackage.WEB_MODULE_JSPS,
7.          JOnASPackage.eINSTANCE.getJSP());
8.  /*
9.   * To count the number of the pattern of
10.  * "taglib" in each JSP
11.  */
12. ((EObjectResolvingEListForWrapping<JSP>) jsps)
13. .refreshWrap();
14. for(JSP j: jsps){
15.     java.util.regex.Pattern pattern = java.util.regex.
16.     Pattern.compile("taglib");
17.     java.util.regex.Matcher matcher = pattern.matcher(
18.             j.getCode());
19.     int count = 0;
20.     while(matcher.find())
21.         count ++;
22.     if(count > 3)
23.         return 0;
24. }
25. return 1;
```

//Programming in QVT

Ignoring Reality	`when{self.deploymentDescriptor.find("form-error-page") = 0}`
Too Much Code	`when{self.jsps[JSP]->exists(jsp | jsp.code.length() > 5000)}`
Embedded Navigational Information	`when{self.jsps[JSP]->exists(jsp | jsp.code.find(".jsp") <> 0 or jsp.code.find(".JSP") <> 0)}`
√ Too Much Data in Session	`when{self.jsps[JSP]->exists(jsp | jsp.sessionAttributeSize > 10)}`
√ Ad Lib Taglib	`when{self.jsps[JSP]->exists(jsp|jsp.tagLibSize >= 3)}`
Accessing Entities Directly	`when{self.deploymentDescriptor.find("<ejb-ref-type>Entity</ejb-ref-type>") <> 0}`

Figure 8. Programs of Anti-pattern Detection in the Languages of Java and QVT

However, their difference in executing time is very small and it could be ignored from the aspect of system management.

The logics to detect these anti-patterns are not complex, since they just check a few attributes of JSP or servlet in the system. For example, the automated program to detect the forth anti-pattern is aimed to check if there are too many codes about data processing in any JSP and it is just needed to count the times how many the pattern of "session.setAttribute" appears in one JSP. From the fragments of the Java program, it is easy to see that most of codes are aimed to deal with data accessing, which is not the core of the management logics, compared with less than 10 percent of codes to express management logics. What's more, the similar codes of data access have to be repeated in different automated programs, which make administrators exhausted. By contrast, the QVT programs reduce about 80 percent of LOC. The runtime architecture-based model is modeling those management APIs, which reuses the codes of data access. Then administrators can develop programs based on the architecture-based model and do not have to invoke specific management APIs any more, which reduces programming costs. It is easy to see that our approach has the advantages related to the reusability.

B. VM States Checking

Load balance management in the cloud requires resource provision (e.g., CPU and memory) to be both stable and reliable, which is an important problem and a challenge in system management. The fundamental solution to this issue is to integrate and coordinate the resources in a global view.

//Fragment of Java Program

//Object: To check if there are nodes whose memory utilization is below 40 percents.
//1st Function - getAMemFreeNode(): It contains the main logic of the management task.
//2nd Function – getMemUtilizationOfNode(): It deals with the detailed implements of the runtime system.

```java
/*
 * JAVA: To get a node whose memory utilization is
 * less than 40%.
 */
public String getAMemFreeNode()
{
    String[] nodes = getAllNodes().split(";");
    int len = nodes.length;
    // To check every node.
    for(int i = 0; i < len; i ++)
    {
        String nodeId = nodes[i];
        String nodeIp = getNodeIp(nodeId);
        NodeClient nc = new NodeClient(nodeIp);
    /*
     * To get the nodes' memory utilization.
     */
        double memUltilization = Double.parseDouble(
                nc.getMemUtilizationOfNode());
        if(memUltilization < 0.4)
            return nodeId;
    }
    return null;
}
```

```java
/*
 * JAVA: To get a node's memory utilization
 * by invoking the management script.
 */
public String getMemUtilizationOfNode()
{
    try{
        String[] args = new String[2];
        args[0] = "/bin/sh";
        args[1] = "/opt/xen/getUsedMem.sh";
        ProcessBuilder builder =
            new ProcessBuilder(args);
        Process process = builder.start();
        BufferedReader br = new BufferedReader(
                new InputStreamReader(process.
                        getInputStream()));
        String line;
        int usedMem = 0;
        while((line = br.readLine()) != null)
        {
            //To get the value by parsing the string.
            String[] tokens1 = line.split(" ");
            if(!tokens[1].equals("ID"))
                usedMem += Integer.parseInt(tokens[2]);
        }
        process.waitFor();
        double memUtilization = usedMem / (double)mem;
        return String.valueOf(memUtilization);
    }
    catch(Exception e){
        return null;
    }
}
```

//Fragment of QVT Program

```
//Operations in QVT Language: select() – To return a list of the objects which are in a certain condition.
//To get the list of nodes in a free condition
var freeNodes = cloudModel.objectsOfType(Node)->select(VM.Memory->sum() < Memory * 0.4);
//To get the list of nodes in a busy condition
var busyNodes = cloudModel.objectsOfType(Node)->select(VM.Memory->sum() > Memory * 0.8);
```

Figure 9. Programs of Load Balance Management in the Languages of Java and QVT

Many automated programs are aimed for this problem [13]. One of the key challenges is to find if the current states of the cloud platform satisfy some specific conditions. In this experiment, we develop the automated program to check if there are physical nodes which are in a free or busy condition. Figure 9 describes some code fragments in Java and QVT programs.

As the figure shows, in the Java program, we traverse the list of physical nodes to check the conditions of each node in the first function, where the second function is invoked to count the memory utilization of a node. In the second function, a script is invoked to retrieve the information about memory allocation in a node and the result needs to be parsed. The administrators have to cope with the detailed implementations while hard-coding in Java, including the interaction between the Java program and the script, and the relatively low-level logics of data processing. Therefore, administrators need to understand the details of the managed system, which may make them exhausted.

By contrast, with the help of the architecture-based model, administrators focus on the logics of management tasks without handling the different types of APIs like scripts and low-level data processing. In addition, the modeling language provides operations in the model level, such as "select", "sum" and so on, which make it simpler to do programming.

VI. Related Work

Our architecture based runtime model is a general approach to platform management in Cloud. There are several industry cloud products to provide platform resources as a service, which are similar in architecture. For instance, Windows Azure [14] adopts Windows server to provide runtime environment to applications and relies on VM ware. Oracle Public Cloud [15] adopts the products of WebLogic Server and its infrastructure depends on Oracle VM. Though the products above support different types of applications,

they all contain virtualization based infrastructure and middleware software products.

Platform management is a key problem in Cloud. Although there are some relative administrative tools, such as Tivoli [16] and Hyperic [17], which are aimed to the factors of heterogeneity and distribution; the management still costs a lot, for the infrastructure fundament and middleware software products should be managed collaboratively and there are too many metrics of different types to manage manually. But there are no effective administrator tools to do the management automatically at present and automated programs are needed.

Other efforts have been made to improve the development of automated management programs. OpenStack Compute [18] is a cloud computing fabric controller. It is written in Python and utilizes many external libraries. It is easy to express management tasks in Python but programmers still need to be familiar with the management APIs and understand the architecture of runtime system. In previous work [19], we propose the solution of "Management as a Service" (MaaS) from the reuse point of view. We encapsulate functions, processes, rules and experiments in IT management into web services and regard them as reusable assets, which is to be presented, used and collaborate in a service-oriented style. However, programmers are hard to express complex management tasks in BPEL (Business Process Execution Language) and still need to deal with the operations on attributes.

Architecture-based approach is usually used in system management. For example, they are applied for automatically obtaining valid configurations of network equipment such as routers and bridges [20]. By modeling the relevant characteristics of every manageable element and defining their restrictions using propositional logic, these engines can automatically find correct configurations for each element, or diagnose the correctness of a preset configuration. We will mention another interesting application of architecture-based approach, in this case for generating test cases of a complex system configuration [21]. This work highlights how this technique can be applied to find efficiently solutions to a search space where multiple constraints over the correct solution are defined. However, none of the analyzed initiatives addresses the problem of automating platform management in Cloud, although it has been successfully applied to some relative problems.

We have made many researches in the area of model driven engineering. For a given meta-model and a given set of management interfaces, SM@RT [8] can automatically generate the code for mapping models to interfaces with good enough runtime performance. If users change the meta-model, SM@RT can re-generate the mapping code. More details can be found in our previous works [22]. If the management interfaces support remote invocations, the RSA can invoke them. In our previous work [23], we encapsulate hundreds of management interfaces of WebSphere, JOnAS, Tomcat, MySQL, Apusic, etc. into SOAP-based web services for the remote management. In addition, for the situation of incomplete formalized of modeling languages, our previous work [24] has provided an MOF meta-model extension mechanism with support for upward compatibility and automatically generates a model transformation for model integration, and the work we implemented on architecture-level fault tolerance [25] can also compensate for this to a degree. Our RSA is also able to translate system logs into elements of RSA and users should analyze the root cause based on RSA. An example can be found in our previous work [26]. We translate JEE application server's system logs into sequence diagrams and then use the automata to detect anti-patterns that cause the poor performance. The approach in this paper is built on our previous resources.

VII. Conclusion and Future Work

Platform management in Cloud brings high costs. Many automated programs thus have been built to tame the complexity of management. Hard-coding these programs in languages like Java can bring enough power and flexibility but also cause high programming effort and cost. It is trivial for administrators to be familiar with different types of APIs and understand the detailed implementations of the cloud environment. This paper proposes a runtime architecture-based approach to managing the platform facilities such as the middleware and VMs of Cloud. We construct an architecture-based model for administrators to develop automated programs of platform management at the architecture level, and the correct synchronization between the model and the runtime system is ensured. Then administrators may develop automated programs of management tasks in modeling languages. The cloud providers are the target users of our approach. Moreover, some cloud-based applications need to customize or even re-invent the management functions provided by the cloud. The developers of such applications can also use our approach to do their own management. We evaluate the approach by comparing it to the hard-coding approach in two management scenarios and the results prove that the architecture-based management is effective and promising in the performance, the interoperability, the reusability and the simplicity.

As future work, we plan to give more support for administrators to manage platforms in Cloud. At present, our approach has the bottle net of performance as other centralized management frameworks. We are searching the model-based solutions to this issue and have made some progress. We also plan to perform further analysis such as model checking to ensure a deeper correctness and completeness of the generated causal link between management tasks. In addition, our approach is an abstraction of any target system and supports any types of operations if there are corresponding management interfaces [27]. We will extend our architecture-based model to fulfill more requirements in cloud management, such as data storage managements and cloud application developments.

Acknowledgments

This work is sponsored by the National Basic Research Program of China under grant no. 2009CB320703; the National Natural Science Foundation of China under grant

no. 61121063, 60933003; the High-Tech Research and Development Program of China under Grant No. 2012AA011207; the European Commission Seventh Framework Programme under grant no. 231167; and NCET.

REFERENCES

[1] Evangelos Kotsovinos, Morgan Stanley. Virtualization: Blessing or Curse? Managing Virtualization at a large scale is fraught with hidden challenges. Communications of the ACM, 2010, 54(1): 61-65.

[2] Jeffrey O.Kephart, David M.Chess. The Vision of Autonomic Computing. IEEE Computer Society, 2003. 36(1): 41-50.

[3] J. M. Rushby. Model Checking and OtherWays of Automating Formal Methods. In Position paper for panel on Model Checking for Concurrent Programs, Software Quality Week, San Francisco, May/June 1995.

[4] Garlan, D. Software Architecture: A Roadmap. Proceeding ICSE'00 Proceedings of the Conference on The Future of Software Engineering, pp. 91-101.

[5] Gang HUANG, Hong MEI, Fu-qing YANG. Runtime Recovery and Manipulation of Software Architecture of Component-based Systems. International Journal of Automated Software Engineering, 2006, 13(2): 251-278.

[6] Robert France, Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. Future of Software Engineering, 2007, pp. 37-54.

[7] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT

[8] Gang Huang, Hui Song, Hong Mei. SM@RT: Applying Architecture-based Runtime Management of Internetware Systems. International Journal of Software and Informatics, 2009, 3(4):439~464.

[9] Wikipedia. Virtual appliance. http://en.wikipedia.org/wiki/Virtual_appliance

[10] Eclipse. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/

[11] Peking University. Internetware Test Bed. http://edu-icloud.internetware.org

[12] LAN Ling, HUANG Gang, WANG Wei-Hu, MEI Hong. Anti-Pattern Based Performance Optimization for Middleware Applications. Journal of Software, 2008, 19(9): 2167-2180.

[13] Ying Zhang, Gang Huang, Xuanzhe Liu, and Hong Mei. Integrating Resource Consumption and Allocation for Infrastructure Resources on-Demand. 2010 IEEE 3rd International Conference on Cloud Computing, pp. 75-82.

[14] Microsoft. Windows Azure. http://www.windowsazure.com/

[15] Oracle. Oracle Public Cloud. http://cloud.oracle.com/

[16] IBM. IBM Tivoli Software. http://www-01.ibm.com/software/tivoli/

[17] SpringSource. Hyperic. http://www.hyperic.com/

[18] OpenStack. The Open Source Cloud Operating System. http://openstack.org/projects/

[19] Xing Chen, Xuanzhe Liu, Fuzhi Fang, Xiaodong Zhang, Gang Huang. Management as a Service: An Empirical Case Study in the Internetware Cloud. IEEE International Conference on E-Business Engineering, ICEBE 2010, pp. 470-473.

[20] S. Hallé, E. Wenaas, R. Villemaire, O. Cherkaoui, Self-configuration of Network Devices with Configuration Logic, Proceeding AN'06 Proceedings of the First IFIP TC6 international conference on Autonomic Networking, 2006, pp. 36-49.

[21] M.B. Cohen, M.B. Dwyer, J. Shi, "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach", IEEE Trans. on Software Engineering, 2008, 34(5): 633-650.

[22] Hui Song, Gang Huang, Franck Chauvel, Yingfei Xiong, Zhenjiang Hu,Yanchun Sun, Hong Mei. Supporting Runtime Software Architecture: A Bidirectional-Transformation-Based Approach. Journal of Systems and Software, Elsevier, 2011, 84(5): 711-723.

[23] Xing Chen, Xuanzhe Liu, Xiaodong Zhang, Zhao Liu, Gang Huang. Service Encapsulation for Middleware Management Interfaces. International Symposium on Service Oriented System Engineering, 2010, pp. 272-279.

[24] Xiangping Chen,Gang Huang,Franck Chauvel,Yanchun Sun,Hong Mei. Integrating MOF-Compliant Analysis Results. International Journal of Software and Informatics, 2010, 4(4):383-400.

[25] Junguo Li, Xiangping Chen, Gang Huang, Hong Mei and Franck Chauvel. Selecting Fault Tolerant Styles for Third-Party Components with Model Checking Support, International SIGSOFT Symposium on Component-based Software Engineering (CBSE), 2009, pp. 69-86.

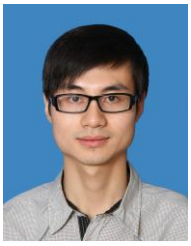
[26] Weihu Wang, Gang Huang. Pattern-Driven Performance Optimization at Runtime: Experiment on JEE Systems. 9th Workshop on Adaptive and Reflective Middleware (ARM2010), pp. 39-45.

[27] Hui Song, Gang Huang, Franck Chauvel, Wei Zhang, Yanchun Sun, Weizhong Shao, Hong Mei. Instant and Incremental QVT Transformation for Runtime Models. 14th international conference on Model driven engineering languages and systems, 2011, pp. 273-288.

Gang Huang was born in 1975. He received his Ph.D. in 2003 from the School of Electronics Engineering and Computer Science of Peking University. He is a professor at Peking University. Huang's research interests include system software and software architecture.

Xing Chen was born in 1985. He is a Ph.D. candidate at Peking University. His research interests include middleware, cloud management and software architecture.

ZHANG Ying was born in 1983. He received the PhD degree in Electronics Engineering and Computer Science from Peking University in 2012. He is a research staff 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.



Xiaodong Zhang was born in 1989. He is a Ph.D. candidate at Peking University. His research interests include middleware and cloud computing.