Towards Uniform Management of Cloud Services by applying Model-Driven Development

Toni Mastelić, Ivona Brandić
Institute of Information Systems
Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{toni, ivona}@infosys.tuwien.ac.at

Andrés García García
Instituto de Instrumentación para Imagen Molecular (I3M)
Centro mixto CSIC - Universitat Politècnica de València - CIEMAT
Camino de Vera s/n, 46022 Valencia, España
angarg12@upv.es

***Abstract*—Popularity of Cloud Computing produced the birth of Everything-as-a-Service (XaaS) concept, where each service can comprise large variety of software and hardware elements. Although having the same concept, each of these services represent complex systems that have to be deployed and managed by a provider using individual tools for almost every element. This usually leads to a combination of different deployment tools that are unable to interact with each another in order to provide an unified and automatic service deployment procedure. Therefore, the tools are usually used manually or specifically intergrated for a single cloud service, which on the other hand requires changing the entire deployment procedure in case the service gets modified.**

In this paper we utilize Model-driven development (MDD) approach for building and managing arbitrary cloud services. We define a metamodel of a cloud service called CoPS, which describes a cloud service as a composition of software and hardware elements by using three sequential models, namely Component, Product and Service. We also present an architecture of a Cloud Management System (CMS) that is able to manage such services by automatically transforming the service models from the abstract representation to the actual deployment. Finally, we validate our approach by realizing three real world use cases using a prototype implementation of the proposed CMS architecture.

Keywords*—*Cloud Computing; Model-Driven Development; Cloud Service Model; Cloud Management System;

I. Introduction

Cloud Computing represents a new paradigm where arbitrary IT products, such as software applications, development environments and processors are integrated and offered as part of on-demand online services. According to the National Institute of Standards and Technology [1], Cloud Computing defines three service layers, including Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS), also referred to as SPI service models. Unlike IaaS that usually offers a single distinct product, i.e., a virtual machine (VM), upper layers such as PaaS and SaaS provide services that are composed out of arbitrary software products such as database applications, web servers and storage platforms. This leads to an explosion of services such as Storage-aaS, Database-aaS, Identity-aaS and Computing-aaS, thus producing the birth of Everything-as-a-Service (XaaS) concept [2].

Building and managing such diverse services requires separate deployment tools [3] for each element that is part of the service, usually being distinguished by its deployment layer, e.g., deploying VMs is done by using OpenNebula [4] on an infrastructure layer, while a database can be deployed using CloudFoundry [5]. These tools are mostly used manually and separately for each layer, or in a best case scenario they are specifically integrated and customized for a single Cloud service. However, if the service changes the deployment and management procedure has to change as well, which may require reintegrating deployment tools from the start. Fox example, Microsoft offered a standard PaaS with their Azure [6], until they granted a VM access to their customers by introducing VM roles, hence reaching towards elements on an infrastructure layer. Problem with such approach is that the deployment procedure for each service is being built from the ground up, without reusing procedures from a service that uses the same elements.

In order to reuse existing deployment procedures, Cloud service elements should be abstracted with a higher-level model [7] capable of describing any service with regards to its composition [8]. Furthermore, a Cloud Management System (CMS) must implement an uniform deployment procedure that utilizes this abstraction, while actual deployment is executed by a lower-level tools specifically designed for a targeted piece of software. For example, a database application can be part of SaaS such as Facebook [9], where customer has no access to it. It can be part of PaaS such as Amazon RDS [10], where it is accessed by a customer, but managed by a provider. Finally, it can be part of Amazon EC2 [10] as an example of IaaS, where it is both accessed and managed by a customer. In all three cases the database application could be deployed with the same procedure using different configuration.

In this paper we introduce an uniform approach for deploying and monitoring arbitrary Cloud services. We utilize Model-driven development (MDD) [11] for defining a Cloud service metamodel called CoPS, in order to get uniform representation of a Cloud service. The CoPS follows the Model-driven architecture (MDA) scheme proposed by OMG [12], which defines three levels of models that describe a system on the abstract level, on the structural level and finally on the implementation level. We refer to these models as **S**ervice, **P**roduct and **Co**mponent, which form the abbreviation CoPS in a reverse order. The models can be sequentially transformed between each other going from the abstract representation to the actual

deployment. Additionally, the CoPS allows partitioning of the models so the models for individual service elements can be reused in other services.

We present the Cloud Management System (CMS) architecture capable of managing Cloud services described with CoPS. Management of a service and its elements is performed on the structural level since service components are represented through templates as black boxes, while transformation to the implementation model and final deployment is performed via plugins. Additionally, modularity of the CoPS models allows the CMS to reuse its templates and plugins for multiple Cloud services. The modules and interfaces of the architecture are described with Unified Modeling Language (UML) [13].

The prototype architecture is implemented using the Cloudcompaas [14], a Cloud manager framework for the dynamic management of Cloud resources, and M4Cloud [15], a plugin-based monitoring tool capable of monitoring arbitrary metrics. The prototype is used for validating both CoPS metamodel and the CMS architecture on the three use cases based on real world scenarios.

The rest of the paper is organized as follows. Three use cases used throughout the paper are described in Section II. Section III gives a motivation for the approach taken in this paper. Section IV introduces the CoPS model. Section V provides a detailed description of the architecture, while Section VI describes the implementation of its prototype. Section VII gives validation of the CoPS metamodel and the architecture by realizing the three use cases. Section VIII describes relevant related works. Finally, section IX concludes the paper and proposes the future work.

II. Use case scenarios

We depict three use cases based on real world scenarios to emphasize of the issues related to management of Cloud services.

- **Online Course**: Students of an online course require a specific software stack to complete exercises of the course, similar to [16]. The software stack includes a large number of tightly coupled Python libraries, some of them specific to the course topic and not widely available. In order to avoid the cumbersome operation of manually installing the libraries, the course offers an online service for instantiating a VM with a customizable software stack.
  This is a basic IaaS scenario where a provider such as Amazon [10] offers VMs. Additionally, the provider also offers ready to use VM images with preinstalled software [17]. Another examples of such a service are AWS Marketplace [18] and Vagrantbox [19], where developers can configure and upload VM images for customers to deploy and use.
- **Genomics**: A group of scientists want to run their scientific applications in the Cloud. They utilize distributed genomic applications that work on large datasets similar to [20]. However, since not every instance works over a complete dataset, but only a small subset of it, transferring the data to each machine incurs a large overhead in terms of time and cost. Therefore, they want to utilize arbitrary number of VMs with an access to a third party online Bio-Database such as GenBank [21] or EMBL Nucleotide Sequence Database [22] allowing customers to retrieve only a subset of genomic data they are interested in.
  This use case represents the collocation of VMs, which are on an infrastructure layer, with a database offered on a software layer, thus providing an unique platform for running genomic applications. A similar scenario can be found in Amazon Web Services [10] where a customer can utilize additional hardware and software products along with the deployed VMs. Heroku also provides a large set of add-ons [23], software products that can be referenced from within customer applications.
- **Web hosting**: A web developer wants to migrate his web application to the Cloud in order to benefit from a highly available and scalable environment. However, the customer does not want to manually configure and manage the environment, but rather have it all done automatically.
  This use case represents a traditional PaaS where a customer deploys the application on a targeted platform. Google App Engine [24] enables deploying Java and Python web applications on a managed Cloud platform that is completely transparent to the customer. Heroku [23] offers a Ruby on Rails environment deployed on top of Amazon EC2. The platform is completely managed by the provider, while customers only interface with provided runtimes, databases and add-ons. Windows Azure [6] supports deployment of web and non-web applications in a Windows runtime for a variety of programming languages.

These three use cases cover scenarios of Cloud services that combine software and hardware components from different layers and require separate management tools and access permissions. The use cases are used throughout the paper in order to illustrate how existing real world issues are represented and managed by the proposed models and the architecture.

III. Model-driven development

Model-driven development (MDD) includes the construction of a system model that can be transformed into an implementation [11]. This procedure shown in Figure 1 usually includes defining a metamodel first, which describes how the model should be defined and what it needs to include. Furthermore, the model of a targeted system is constructed by following the metamodel instructions. Finally, an instance of the system is created based on the model.

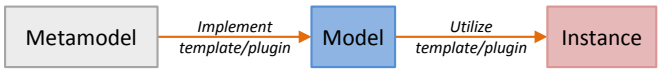

Fig. 1. Model-driven development (MDD) standard procedure

Special case of MDD is a Model-driven architecture (MDA) presented by OMG [12], which defines a model as a description or specification of the system and its environment for some targeted use. For this purpose, MDA defines three

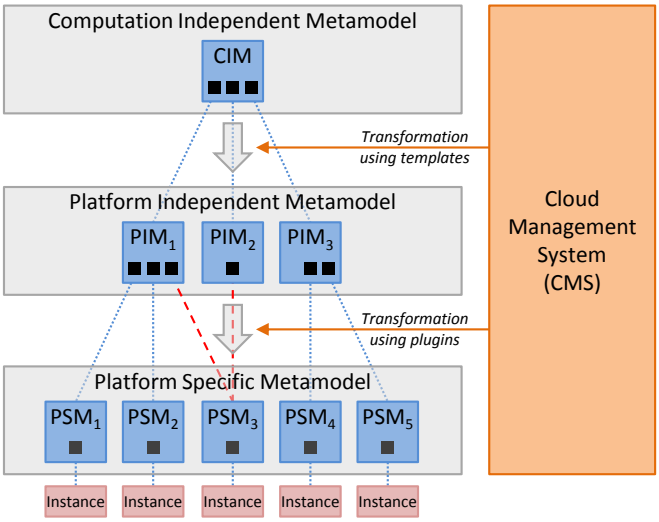

Fig. 2. Model-driven architecture (MDA) model in correlation with a Cloud service

types of models [12], going from the more abstract one to the model that closely describes details of the system:

- Computation Independent Model (CIM) is a view of a system that does not show any details. It should be independent of the system's structure, thus showing only requirements of the system and the way the system is used.
- Platform Independent Model (PIM) describes the system and its structure, however without implementation details, or the details of the platform where the system is deployed.
- Platform Specific Model (PSM) defines an implementation of the system that was previously described with PIM and its implementation details. This includes all the specific details of the platform where the system is deployed.

Transforming a higher-level model such as PIM to a lower-level model such as PSM is done using transformations, which can be manual, using a profile or templates, or they can be completely automatic.

The idea of using the MDD, or more specifically MDA for deploying Cloud services comes from the fact that a Cloud service is a system composed of multiple software and hardware elements, which are integrated in order to provide the service. For example, a Cloud service for the Online Course use case (Section II) is composed of a physical machine, hypervisor, operating system and python libraries. Here we exclude notion of developing new Cloud elements such as applications, and rather assume that all elements have been developed. Therefore, we can focus on integrating those existing elements into unique Cloud services.

Modeling an arbitrary Cloud service by applying MDA approach would result in separate models for each service, i.e., customized deployment procedures specialized for a single Cloud service. Therefore, in order to achieve uniformity of the deployment procedure, a metamodel of a Cloud service is required, which describes how CIM, PIM and PSM models should be defined, as shown in Figure 2. With this approach we get uniform models for Cloud services, where we can implement transformations for deploying a service instance by the CMS (Figure 2).

However, we would still not be able to reuse existing models of the service elements that are used in other services. Therefore, as shown in Figure 2, instead of transforming CIM to a single PIM, and PIM to a single PSM, we partition the models for each service element so they can be used separately. This allows us to reuse already existing service element models and their transformations, i.e., templates and plugins, for different services and implementations. Example is given in Figure 2 where PSM_3 could represent model of an operating system used as an underlying platform for database and web server Cloud products.

Finally, since we consider only existing service elements such as Hadoop and Ubuntu, rather then developing new ones, the deployment procedure is not required to *understand* internal structure of the service elements. Consequently, a service element model is not required to contain this information, except information about functionalities and configuration parameters of the service element so that it can be properly configured.

IV. The CoPS Model

This section introduces the CoPS metamodel able to describe Cloud services using three sequential models going from the most abstract one to the real deployment, similar to the MDA models described in the Section III. These three models include (1) **Service model** that defines service requirements and its usage, (2) **Product model** that defines structure and the composition of the service, and (3) **Component model** that defines configuration of each component of the service. Unlike MDA models that only extend the previous model by adding additional information, the CoPS models also partition this information for each element separately. For example, if a Cloud service comprises a database and a web server, the lower level model will include two new separate models instead of one, namely the database and the web server, each being described independently, as shown in Figure 2. This allows CoPS to build up services from components in a modular fashion. However, it also requires a deployment tool to be aware of all three models in order to deploy a final service, by sequentially transforming one model into another.

A. CIM - Service model

(1) **Service** represents CIM model of MDA, described in Section III. It defines a *service* as a composition of one or more *products* described in form of high-level requirements and relations between the products, e.g., the BioDatabase product grants the access to the VM products of the same customer for the Genomics service use case in Figure 3. The Service model does not contain any details about the structure of the products, rather it represents them as black boxes.

B. PIM - Product model

(2) **Product** represents PIM model of MDA, described in Section III. Each *product* of a *service* is describes separately as a composition of *components* deployed on top of each other in form of parent-child relations as shown in Figure 4a, where physical machine is a parent to its child operating system. Since some components can be multi-tenant, i.e., supporting

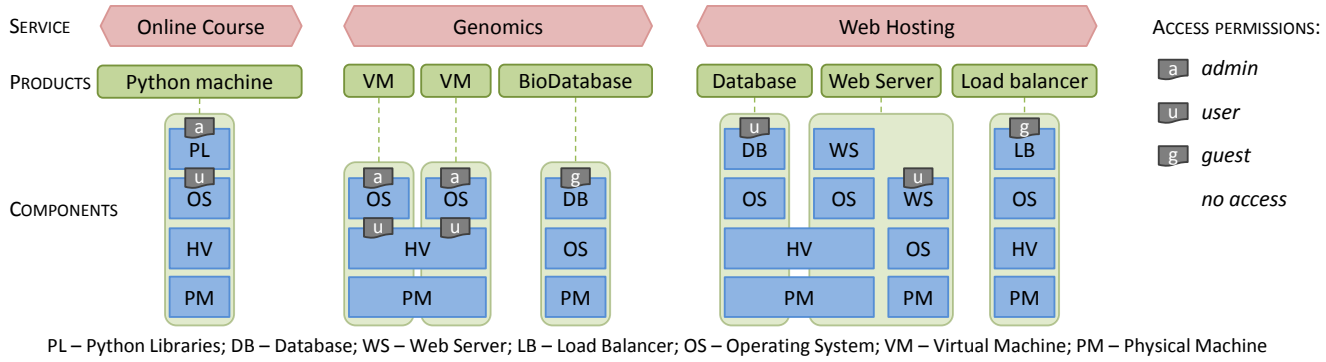

Fig. 3. Three use cases from the Section II represented using the CoPS cloud service model

more than one customer, a parent component can have more than one child. Fox example, in Figure 4b, operating system has two children, namely a web server and a database.

The parent-child relation corresponds to the *application-platform* relation of MDA described in Section III. Since the Product model defines a component as a black box, it only defines its external functions, namely *dependencies* and *access permissions*, as shown in Figure 5. Therefore, implementations of dependencies and access permissions are defined in the Component model. Dependencies represent requirements of a child component towards its parent, i.e., functionalities and interfaces that it has. Access permissions define four access levels to a component bound to a single tenant using the component, as discussed next.

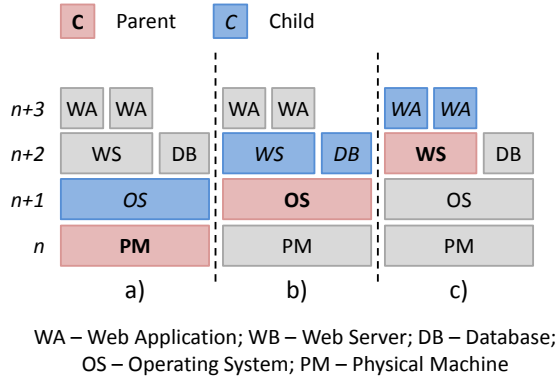

Fig. 4. Parent-child relations between components

- **No access**: A customer has no access to a component, rather it is completely managed by a provider. Fox, example, any of the use cases in Figure 3 where physical machines are accessible only to the provider.
- **Guest**: A customer has only a limited access to a component through its externally exposed functions. A provider is still managing the component and is responsible that the functions are failsafe. Fox example, the Genomics use case, where a customer can only read and write to the database component. However, he cannot change the table structure of the database or configure the component itself.
- **User**: A customer can upload his own code or deploy a child component in order to customize its functionalities. A provider is still responsible for the component. However, he cannot guarantee for the customization done by the customer. Fox example, the Web Hosting use case in Figure 3, where the customer can upload his own web application to the web server component.
- **Admin**: A customer owns a component and thus can change whatever he wants, while a provider does not have any responsibility regarding it. Fox example, the Online course use case in Figure 3, where a customer has *admin* access to the python library component.

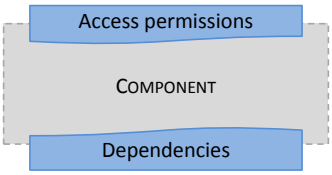

Fig. 5. Black box representation of a component in the Product model

C. PSM - Component model

(3) **Component** represents PSM model of MDA, described in Section III. In the real life, components represent instances of hardware and software items that serve as building blocks of a functional product, e.g., an operating system or a hypervisor component, as shown in Figure 3, Online Course use case. Similar to a product, each component is defined separately with its own specific implementation details. However, since a goal of our approach is not to develop new components that will be used as part of a Cloud service, but rather to deploy existing ones, the components are represented as gray boxes. It defines *parameters* and *access data* specific to each component, as shown in Figure 6. For example, an operating system component can have parameters such as `name`, `type`, `version` and `open_ports`. Due to the diversity of components, parameters should be defined using Domain Specific Languages. Furthermore, dependencies defined in the Product model are implemented using the parameters in the Component model, e.g., web server requires an operating system with parameter `name=Linux` and `kernel_version=3.11.1`.

Access permissions are configured with the component's parameters, and provided to the customer or other components in form of access data. Fox example, for an operating system with *no access* a customer receives no access data, while with *guest* access he receives the IP address, username and password of a guest account not allowed to install or modify anything. For *user* access, a customer receives IP address, username and password of a superuser account that is allowed to modify or deploy his own child components on top of the operating system. Finally, admin access permission assumes user or admin access to a parent component, so the customer

can install, modify, upgrade or completely remove the operating system.

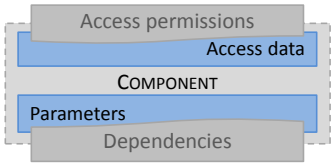

Fig. 6. Gray box representation of a component in the Component model

D. Service customization

Customization of a service is defined by a provider through its composition and high level requirements that map to component parameters. In the Online Course use case, a customer is able to fully customize python libraries by selecting which libraries he wants to use. His customization is then translated into the deployment parameters of the component. In the Genomics use case, the provider is able to offer different sizes of a virtual machine product, such as small, medium and large, by defining the set of parameters used by a hypervisor for instantiating new virtual machines. They include number of cpu cores, amount of memory, storage size, etc. In the Web Hosting use case a customer is able to choose between different implementations of the web server product, such as Apache or II7. This defines which product template should be deployed, the one with the Apache as a web server component or the II7.

By introducing three sequential models, the CoPS metamodel is able to describe any Cloud service (Service model) with regards to its composition (Product model), including all information required for actual deployment (Component model). Therefore, CoPS provides a basis for the deployment and management of customizable Cloud services composed out of arbitrary software and hardware elements, which may or may not be directly accessible to a customer as part of his service.

V. ARCHITECTURE

This section introduces an architecture that utilizes the CoPS models for enabling the management of arbitrary Cloud services. Figure 8 shows the UML component diagram that is technology and implementation agnostic. It is composed of six modules that interact with one another by means of well defined information flow through their interfaces. The information is structured following the transformations between the CoPS models, namely Component, Product and Service.

The transformation of the Service and Product models is realized with predefined templates of the products containing their structure, access permissions and dependencies of their components. Transformation of the Product and Component models is done via separate plugins. The use of plugins is required due to a lack of standardization, i.e., every software component is implemented and configured differently. And by the definition of PSM, platform specific information has to be included in this phase. The information also includes customer information related to provided service elements. A detailed deployment procedure is shown in Figure 7 with UML sequence diagram showing the information flow between the modules of the architecture.

a) SERVICE: The **Central Management System** module receives a customer *service* request through the **iService** interface with $m1$. With $m2$ and $m3$, the module retrieves a template of the *service* as CoPS representation through the **iData** interface exposed by the **Database** module. The Database module contains information of an entire infrastructure. It also includes customer data, monitoring data, CoPS resource templates and other arbitrary information required for the management of the infrastructure. Once the template is retrieved, $m4$ applies the parameters received from the customer request to the template, thus customizing the *service*.

b) PRODUCT: Sequence *product_loop* is executed for each product defined by the service template. Based on the customized service template, $m5$ locates compatible parent *components* where the **product** can be deployed. Afterwards, $m6$ and $m7$ retrieve monitoring data of the selected components. Monitoring data is retrieved through the **iMonitoring** interface exposed by the **Monitoring System** module responsible for collecting live monitoring data from the deployed resource elements. It also stores this data to the Database module for later analysis. Based on the product template, available parent components and their monitoring data $m8$ performs a scheduling operation by choosing components where the product will be deployed.

c) COMPONENT: Deployment procedure begins by repeating the sequence *component_loop* for every component of the product that needs to be deployed. Messages $m9$ and $m10$ retrieve data for the parent component including its current parameters and access data, in order to locate and manage it. Message $m11$ sends deployment data to the **Component Manager** module through its **iComponent** interface. The Component Manager keeps a local copy of data of the managed components. In order to deploy the component, the data is used for translating CoPS representation of the component into a format understandable by the **Component Plugin** module shown in Figure 8. The Component Plugin module, such as OpenNebula is built for managing one or more component types, and is excluded from the sequence diagram in Figure 7 as it only receives messages delegated from the Component Manager through the **iPlugin** interface. The Component Manager must implement all the iPlugin interfaces of the Component Plugins in order to use their functionalities. After the deployment is performed with $m12$, $m13$ returns the access data of each component. Along with the component and parent IDs and its parameters, the access data is forwarded to the Monitoring System module with $m14$. The Monitoring System starts monitoring with $m15$ by initializing **Metric Plugin** modules through their **iMetric** interfaces. Metric Plugins support monitoring specific metrics for the targeted components. Component ID is used for initializing monitoring, while parent ID is used for monitoring metrics that are monitored from its parent. Similar to the Component Plugin module, the Metric Plugin module receives delegated messages from the Monitoring System, and is thus excluded from the sequence diagram in Figure 7.

After all components are deployed, access data of all elements is updated and stored back to the Database module with $m17$. Finally, the customer receives service data in $m18$ comprising access data for all components with *admin*, *user* or *guest* permissions, and may start using the service with $m19$.

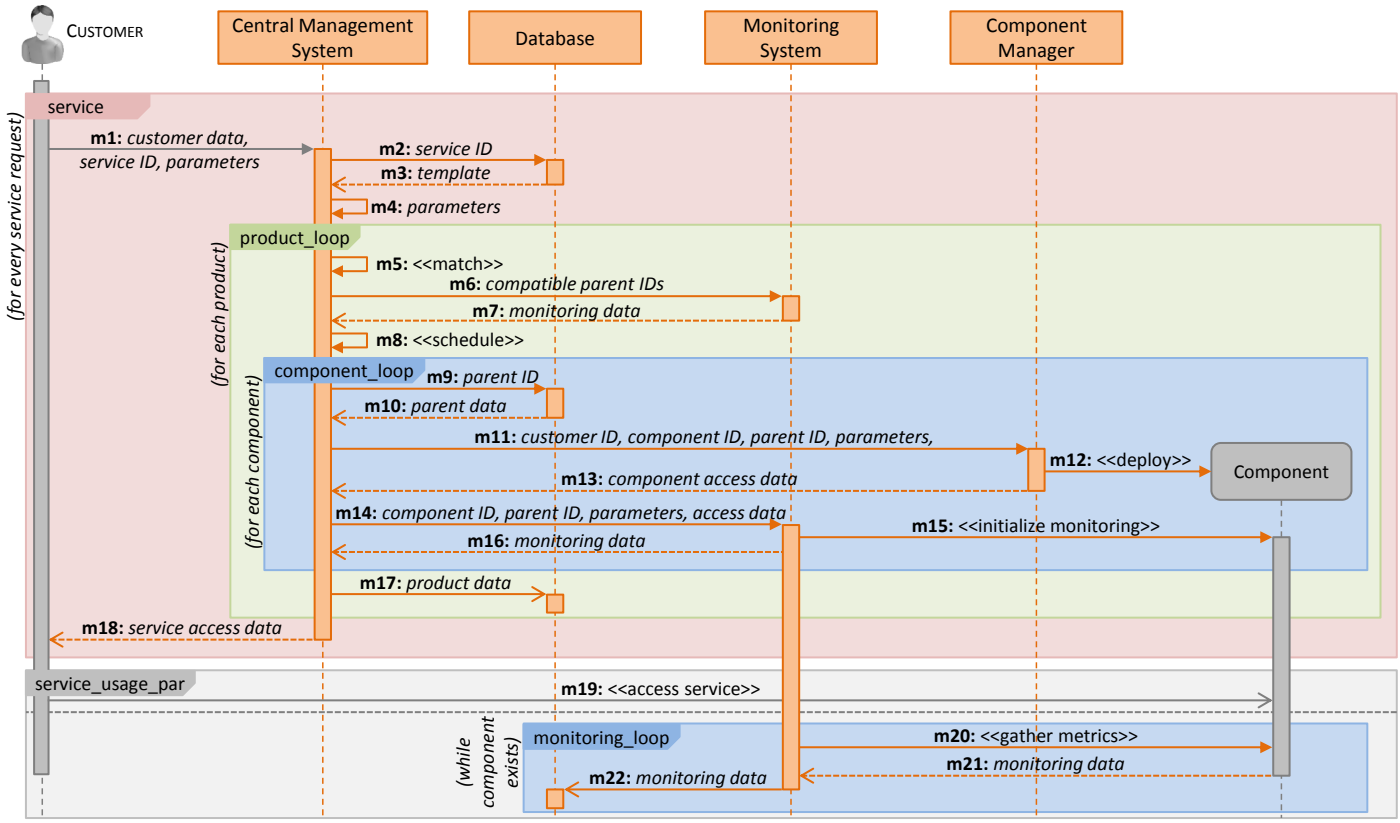

Fig. 7. UML sequence diagram showing the information flow for service deployment

d) USAGE and MONITORING: In parallel with the service usage, sequence *monitoring_loop* performs monitoring over the resource elements used by the customer. The Monitoring System gathers metrics using the deployed Metric Plugins modules, and stores the monitoring data into the Database module with $m20$, $m21$ and $m22$, respectively. The *monitoring_loop* is executed during the lifetime of the components.

In case a product supports multi-tenancy, an existing product is matched with $m5$, and its monitoring data is gathered with $m6$ and $m7$. Furthermore, if existing product can support more tenants, the Central Management System retrieves its access data with $m9$ and $m10$ and instructs the Component Manager to modify existing component, instead of deploying a new one. Finally, sequence *product_loop* is periodically repeated for optimizing the resource usage on the infrastructure by analyzing monitoring data and rescheduling the components.

The described sequence diagram represents the deployment procedure agnostic to a service or its components that are being deployed. It uses product abstraction in form of templates to connect individual components into an unique Cloud service.

VI. Prototype Implementation

This section describes a prototype implementation of the architecture presented in the previous section that utilizes plugin-based approach to deploy different components and the CoPS metamodel to manage them as a single service. The prototype is built using Cloudcompaas [14], a Cloud manager framework for the dynamic management of Cloud resources, and M4Cloud [15], a plugin-based monitoring tool capable of monitoring arbitrary metrics, as shown in Figure 8.

- **Central Management System** module is implemented as a Java webservice for the Apache Axis2 server [25] with two modules of the Cloudcompaas framework, namely the (C1) *SLA Manager* and the (C2) *Orchestrator*. The SLA Manager handles the translation of input instructions in form of WS-Agreement [26] received through the iService interface, which is implemented as a RESTful [27] interface. The interface uses HTTP Basic authentication for validating the input instructions. Once the input is translated to the CoPS model, it is delegated to the Orchestrator. The Orchestrator maintains a global vision of state of a Cloud infrastructure, and performs scheduling and allocation of resource elements. When an allocation request arrives, the Orchestrator queries the Database for service template, available resources and their utilization data from the Monitoring System in order to fulfill a customer request.
- **Database** module of the architecture is implemented with the (C3) *Catalog* module of the Cloudcompaas framework. The Catalog includes an embedded HSQLDB database for storing and retrieving data, as well as a RESTful implementation of the iData interface. The interface only allows simple operations on single tables, and does not allow performing arbitrary SQL queries. This is why complex monitoring data, such as product calculable metrics [15] is acquired directly from the Monitoring System module.
- **Monitoring System** module is implemented with the (M1) *Application Level Monitoring* module of the M4Cloud tool. The module is implemented as a Java application with a socket-based iMonitoring interface.

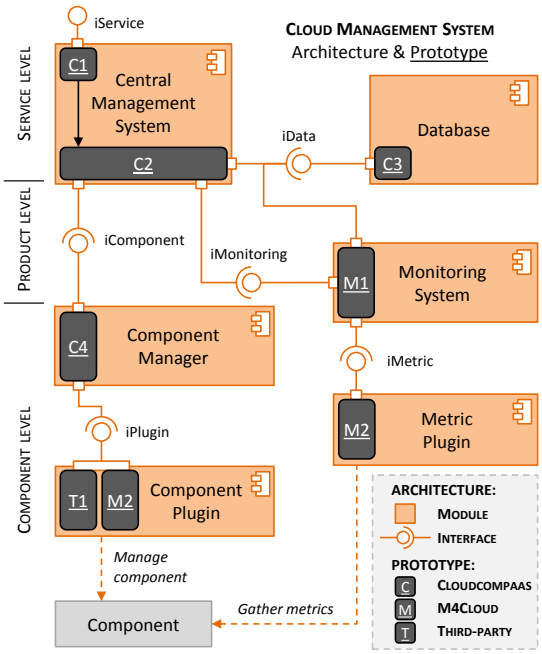

Fig. 8. Architecture of the Cloud Management System for automatic management of Cloud services

The instructions are received and sent as Java serialized objects.

- **Component Manager** module is implemented with the (C4) *Cloud Connector* module of the Cloudcompaas framework. The Cloud Connector translates CoPS deployment instructions received from the Central Management System to operations understandable by the Component Plugins. Unlike the Orchestrator that keeps a global state information of the system, the Cloud Connector keeps a local state information of available components deployed by its Component Plugins.

- Two **Component Plugins** are supported by the Cloud Connector, namely (T1) OpenNebula that exposes XML-RPC API as the iPlugin interface, and the (M2) Application Deployer of the M4Cloud tool that exposes socket connection as the iPlugin interface. OpenNebula is an open source infrastructure layer management tool capable of managing two types of components, namely HV for allocating new VMs and OS in form of VM images. The Application Deployer is part of the M4Cloud tool capable of deploying arbitrary applications on top of Linux-based operating systems using customizable shell scripts.

- **Metric Plugin** is also implemented as part of the (M2) *Application Deployer*. It includes Sigar library [28] for monitoring system and process metrics. It is implemented in Java with multiple socket connections The first connection is used as an iPlugin interface, and the second one as an iMetric interface. It receives and sends instructions in form of Java serialized objects.

VII. Concept Validation

In order to validate our models (Section IV) and the architecture (Section V) we use the prototype implementation described in Section VI for realizing the three use cases presented in Section II. Each validation begins by providing a high level description of the requirements. Following this description, the first use case is described with a step by step procedure following the sequence diagram in Figure 7, while other two use cases are referred to the first one by describing differences in the procedure.

Online Course use case: A customer requests a service allocation comprised of a single machine with a pre-installed software stack, referred to as Python machine. The customer describes the machine product by customizing the software stack from a set of available packages by selecting `scipy` and `numpy` as additional libraries.

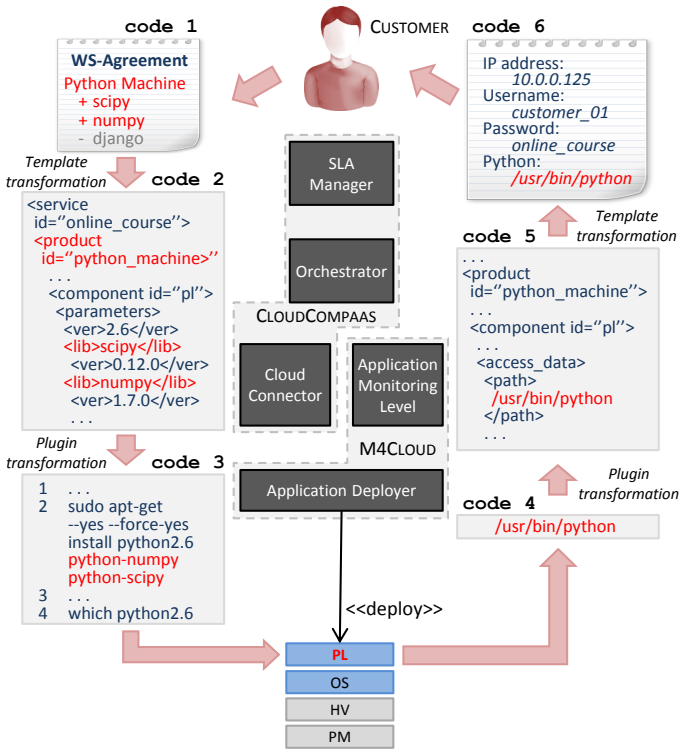

Fig. 9. Delegation of service parameters used for deploying PL component in the Online Course use case, and returning its access data to the customer

Customer's service request containing the selected libraries is encoded in the WS-Agreement format and received by the SLA Manager as shown in Figure 9. The SLA Manager extracts the service requirements (`code 1` in Figure 9) from the WS-Agreement and forwards them to the Orchestrator in form of CoPS parameters. The Orchestrator retrieves the template of the requested service from the Catalog and applies the customized parameters. `Code 2` snippet in Figure 9 shows the template with the customized parameters in red color. Version of Python and its libraries, as well as the underlying operating system is already defined in the template. In this use case we use UbuntuServer 12.04 LTS for the operating system, and Python 2.6, Scipy 0.12.0 and Numpy 1.7.0 as shown in `code 2` snippet.

Since the customer has requested a single product, the *product_loop* from the sequence diagram in Figure 7 is iterated only once, while *component_loop* for the product is iterated twice for the operating system (OS) component and for the python libraries (PL) component. In the first iteration the operating system is deployed by the OpenNebula Component Plugin by allocating a new hypervisor tenant, i.e., a VM, and deploying a VM image of UbuntuServer 12.04 LTS on top of it. OpenNebula returns the VM identifier, IP address and user credentials. In a second iteration the Orchestrator forwards the required parameters for deploying PL component to the Cloud Connector, including the customized libraries. The Cloud Connector translates the CoPS parameters (`code 2` snippet in Figure 9) into deployment instructions understandable by the Application Deployer shown with `code 3` snippet. We assume that the deployed VM image comes with pre-installed Application Deployer.

Once the python libraries are deployed, the Application Deployer returns its installation path (`code 4` in Figure 9). The path is sent by the Cloud Connector to the Orchestrator in form of access data (`code 5` in Figure 9). The Orchestrator then updates the Catalog module with the newly deployed product and returns the access data to the customer. The data includes the IP address of the machine, username and password for accessing it, as well as Python installation path for further usage, as shown in `code 6` snippet in Figure 9.

Since the operating system is deployed with *user* permissions, it means that the customer can deploy new child components on top of it, i.e., install new software or remove existing one. However, he has limited access to system files and has no option to replace the operating system since he has *no access* to the underlying hypervisor. On the other hand, the python libraries can be removed, updated and modified by the customer just like any other child component of the operating system, since he has *admin* access permissions for it.

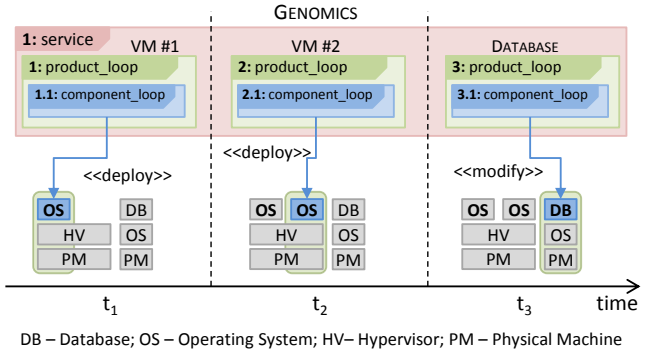

Fig. 10. Timeline representation of the infrastructure deployment for the Genomics use case

Genomics use case: A customer requests a service allocation comprising two VMs and the access to a database containing genomic data. The customer selects the size of the VM products from a set of predefined offerings (medium size VMs), along with a reference to a database product containing genomic data (BioDatabase).

Similar to the Online use case, the service request is received by the SLA Manager, which forwards the requirements to the Orchestrator. The Orchestrator then applies them to the service template and iterates three times through the *product_loop*, once for each product, namely two VM instances and the BioDatabase, as shown in Figure 10. Figure 11 shows parameters conversion for the VM products where the customer has selected medium size instances. The parameter `medium` is applied to the VM product template, which defines number of cpu cores, memory, etc., as shown in `code 2` snippet in Figure 11. Finally, the Cloud Connector translates the CoPS parameters to OpenNebula instructions (`code 3` in Figure 11) that deploys the VMs. The BioDatabase is allocated by adding a new tenant to the existing BioDatabase product, i.e., the Application Deployer adds a new user to the DB component that is implemented with MySQL database in this use case. Additionally, it grants access to it from IP addresses of newly allocated VMs.

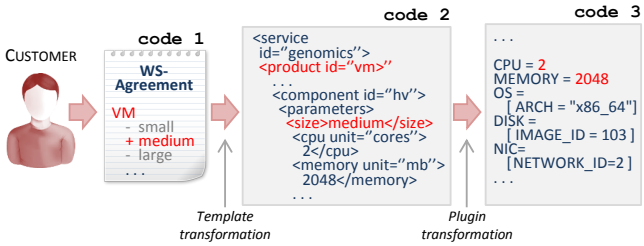

Fig. 11. Delegation of service parameters used for deploying VM products in the Genomics use case

Unlike the Online Course use case where new products have to be deployed, the Genomics use case also requires a modification of the existing product for fulfilling the customer request. Finally, when the customer requests the finalization of the service, the VMs are deallocated and the BioDatabase product is reconfigured to stop providing access to the customer.

Web hosting use case: A customer requests a Web Hosting platform service for deploying a web application. A provider offers service customization by providing different Web Server products, such as Apache or II7. The service also includes a database and a load balancer for user requests between web server instances.

The service template comprises three products. First product is a Database implemented with MySQL 5.6.14. that runs on a dedicated operating system (OS). A second product is a Web Server that supports auto scaling implemented with Apache 2.4.6 web server (WS) component. A final product, namely the Load Balancer is composed of Apache 2.4.6 as a load balancer (LB) component configured with `mod_proxy`, `mod_proxy_http` and `mod_proxy_balancer` parameters [29]. The service template defines deployment of the Database product first, then the Web Server and finally the Load Balancer, as shown in Figure 12.

Once the service is deployed, the Monitoring System periodically checks cpu utilization ($m20$, $m21$ and $m22$) of the Web Server in order to scale the product up or down, namely to deploy new or undeploy existing instances of a web server component. Since web server components are defined within the product template to run on a dedicated operating system, we use CPU utilization of the underlying operating system as an utilization metric. If all the web server components of

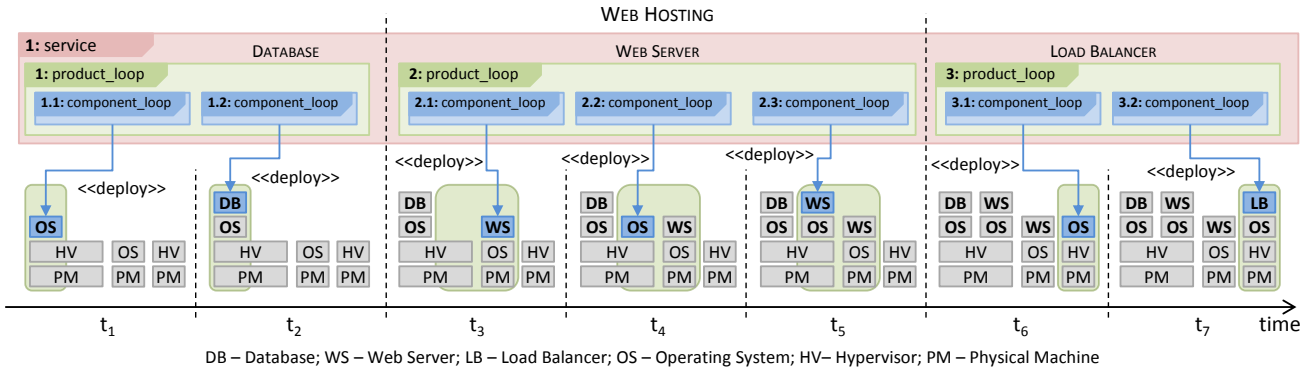

Fig. 12. Timeline representation of the infrastructure deployment for the Web Hosting use case

the Web Server product are over-utilized for a longer time period, the CMS deploys a new web server component. The new component is also registered at the Load Balancer product by adding its IP address to an IP pool.

VIII. Related work

Model-driven development (MDD) approach targets at developing arbitrary systems. Paper such as [30] and [31] utilize MDD for developing Cloud applications that can be executed on multiple Clouds. In [32] the authors include inverse engineering of existing software (legacy) to the new SaaS model using the model-driven approach. However, these works focus on developing new SaaS applications and required platforms, while our approach aims at deploying and monitoring existing applications and integrating them into a unique Cloud service.

In [33], the authors present a generic framework for the development of software applications using MDA. The approach used by the framework consists on decomposing software in services using SOA, and then describe each service as a model using MDA. Services are described using MDA in four different levels of abstraction, from platform specific, to models and metamodels. Finally, services can be deployed in the Cloud by assigning software components to the PSM of the service, executing these software components in Cloud infrastructures. Additionally, utilization of UML for defining the Platform Independent Model of SaaS applications is presented in [34]. The model is used to describe the implementation and technology details of the implementation of the service for an specific back-end. Authors in [35] focus on application deployment by using MDD for defining compliance requirements. Although, works [33], [34] and [35] utilize MDD for Cloud services, they focus only on deployment or describing its platform, while our aim is a management of the complete infrastructure.

Authors in [36] try to unify Cloud management by using service governance approach. However, they focus only on an infrastructure layer, i.e., VMs with a goal of achieving overall business level objectives. Rochwerger et.al [37] also focuses on infrastructure layer by offering a federation of Clouds. [38] uses SPI model for deploying and monitoring Cloud services. However, the SPI model is based on layers and not on the composition of a Cloud service. Therefore, they do not utilize benefits of modular deployment, nor are they able to manage services that spread across several layers. [39] introduces an idea of breaking up the current SPI monolithic approach by combining services from different layers. However, they still perform layer-based management as opposed to component-based one.

[40] proposes using MDA for the definition of SaaS software and their interoperability. By describing the interaction between software components at an abstract level, it provides a match between dependent software services. Using a PIM model for services, the authors argue that SaaS applications can be described independently by the underlying technology. Finally, the authors propose use of standardized service description methods, such as WSDL. However, they focus only on SaaS layer, without considering management of the entire infrastructure. An architecture for deploying elastic services in the Cloud and its prototype implementation are presented in [41]. This work is the closest to ours as the authors use component-based approach for deploying arbitrary software components on an underlying existing virtual infrastructure. Our work goes beyond this by defining the accessibility of the components by an end customer, and by including management of infrastructure components as well.

IX. Conclusion

This paper represents a first step towards the management of arbitrary Cloud services with a Cloud Management System that can utilize existing individual management tools in an uniform deployment and management procedure. For this purpose, we introduced CoPS metamodel that provides a constructive approach for composing Cloud services out of products, and products out of software and hardware components, all this in a modular fashion. We utilized the three models provided by the CoPS metamodel by defining an architecture of a Cloud Management System that is capable of managing Cloud services comprised out of arbitrary components. By realizing three use cases with the implemented prototype, we showed that management of Cloud services depends on the interaction and accessability of its elements, which was supported by our model and the architecture.

Future work includes defining an unified format of the data used for representing CoPS *services*, *products* and *components*, as well as a format for the parameters and access data that is used for configuring and accessing components. This will also allow us to strictly define interfaces and transformations within the architecture. Based on this, we plan to go beyond the prototype implementation and build the architecture modules from the ground up, with capabilities to plugin existing management tools such as OpenNebula.

ACKNOWLEDGMENT

The work presented in this paper has been funded by Vienna University of Technology under HALEY project (Holistic Energy Efficiency Management of Hybrid Clouds). The authors also wish to thank the financial support received from The Spanish Ministry of Education and Science to develop the project 'CodeCloud', with reference TIN2010-17804.

REFERENCES

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," tech. rep., July 2009.

[2] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B. Huberman, J. Manley, C. Patel, P. Ranganathan, and A. Veitch, "Everything as a Service: Powering the New Information Economy," *Computer*, vol. 44, no. 3, pp. 36–43, 2011.

[3] T. Forell, D. Milojicic, and V. Talwar, "Cloud Management: Challenges and Opportunities," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 881–889, 2011.

[4] Distributed Systems Architecture Research Group, "OpenNebula Project," tech. rep., Universidad Complutense de Madrid, 2009.

[5] GoPivotal, Inc., "Cloud Foundry." http://www.cloudfoundry.com/, 2013.

[6] Microsoft Corporation, "Microsoft Azure." http://www.microsoft.com/azure, 2013.

[7] A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan, "OPTIMIS: A holistic approach to cloud service provisioning," *Future Generation Computer Systems*, vol. Vol. 28 (1), pp. 66 – 77, 2012-01-01 2012.

[8] L. Rodero-Merino, L. M. Vaquero, V. Gil, F. Galán, J. Fontán, R. S. Montero, and I. M. Llorente, "From Infrastructure Delivery to Service Management in Clouds," *Future Gener. Comput. Syst.*, vol. 26, pp. 1226–1240, Oct. 2010.

[9] Facebook, Inc., "MySQL at Facebook." https://launchpad.net/mysqlatfacebook, 2013.

[10] Amazon Web Services, Inc., "Amazon Web Services." http://aws.amazon.com/, 2013.

[11] S. J. Mellor, A. N. Clark, and T. Futagami, "Guest editors' introduction: Model-driven development," *IEEE Softw.*, vol. 20, pp. 14–18, Sept. 2003.

[12] J. Miller and J. Mukerji, "Mda guide version 1.0.1," tech. rep., Object Management Group (OMG), 2003.

[13] J. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual*. Addison-Wesley Professional, 1999.

[14] A. García, C. de Alfonso, and V. Hernández, "Design of a Platform of Virtual Service Containers for Service Oriented Cloud Computing," in *Cracow Grid Workshop '09 Proceedings*, (Cracow, Poland), pp. 20–27, ACC CYFRONET AGH, 2010.

[15] T. Mastelic, V. C. Emeakaroha, M. Maurer, and I. Brandic, "M4Cloud - Generic Application Level Monitoring for Resource-shared Cloud Environments.," in *CLOSER* (F. Leymann, I. Ivanov, M. van Sinderen, and T. Shan, eds.), pp. 522–532, SciTePress, 2012.

[16] Strawberry Canyon LLC, "Engineering Software as a Service Pre-Built VM." http://www.saasbook.info/bookware-vm-instructions, 2013.

[17] Canonical Ltd., "Ubuntu Pre-Built VMs." http://uec-images.ubuntu.com/, 2013.

[18] Amazon Web Services, Inc., "Amazon Web Services Marketplace." https://aws.amazon.com/marketplace, 2013.

[19] G. Rushgrove, "Vagrantbox." http://www.vagrantbox.es/, 2013.

[20] A. Matsunaga, M. Tsugawa, and J. Fortes, "CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pp. 222–229, 2008.

[21] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "GenBank.," *Nucleic acids research*, vol. 37, pp. D26–31, Jan. 2009.

[22] C. Kanz, P. Aldebert, N. Althorpe, W. Baker, A. Baldwin, K. Bates, P. Browne, A. van den Broek, M. Castro, G. Cochrane, K. Duggan, R. Eberhardt, N. Faruque, J. Gamble, F. G. Diez, N. Harte, T. Kulikova, Q. Lin, V. Lombard, R. Lopez, R. Mancuso, M. Mchale, F. Nardone, V. Silventoinen, S. Sobhany, P. Stoehr, M. A. Tuli, K. Tzouvara, R. Vaughan, D. Wu, W. Zhu, and R. Apweiler, "The EMBL Nucleotide Sequence Database," *Nucl. Acids Res.*, vol. 33, pp. D29–33, 2005.

[23] Heroku Inc., "Heroku." https://www.heroku.com/, 2013.

[24] Google Inc., "Google AppEngine." https://developers.google.com/appengine/, 2013.

[25] The Apache Software Foundation, "Apache Axis 2." http://axis.apache.org/axis2/java/core/, 2013.

[26] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web Services Agreement Specification (WS-Agreement)."

[27] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Taylor, Richard N.

[28] VMware, Inc., "Hyperic SIGAR." http://www.hyperic.com/products/sigar, 2013.

[29] The Apache Software Foundation, "Apache Axis 2." http://httpd.apache.org/docs/current/mod/mod_proxy.html, 2013.

[30] J. Guillén, J. Miranda, J. M. Murillo, and C. Canal, "Developing migratable multicloud applications based on mde and adaptation techniques," in *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, NordiCloud '13, (New York, NY, USA), pp. 30–37, ACM, 2013.

[31] D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu, A. Gericke, and C. Sheridan, "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds," in *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, pp. 50–56, June 2012.

[32] H. Bruneliere, J. Cabot, F. Jouault, *et al.*, "Combining model-driven engineering and cloud computing," in *Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010)*, 2010.

[33] R. Kumar, J. Bopaiah, P. Jain, N. Nalini, and K. C. Sekaran, "Model driven approach for developing cloud application.," *International Journal of Scientific & Technology Research*, vol. 2, no. 10, 2013.

[34] R. Sharma and M. Sood, "Cloud saas and model driven architecture," in *International Conference on Advanced Computing and Communication Technologies (ACCT11)*, pp. 978–81, 2011.

[35] I. Brandic, S. Dustdar, T. Anstett, D. Schumm, F. Leymann, and R. Konrad, "Compliant cloud computing (c3): Architecture and language support for user-driven compliance management in clouds," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 244–251, July 2010.

[36] M. Sedaghat, F. Hernandez, and E. Elmroth, "Unifying cloud management: Towards overall governance of business level objectives," in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pp. 591–597, 2011.

[37] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan, "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, pp. 4:1–4:11, July 2009.

[38] M. Litoiu, M. Woodside, J. Wong, J. Ng, and G. Iszlai, "A business driven cloud optimization architecture," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, (New York, NY, USA), pp. 380–385, ACM, 2010.

[39] S. Garcia-Gomez, M. Escriche-Vicente, P. Arozarena-Llopis, M. Jimenez-Ganan, F. Lelli, Y. Taher, J. Biro, C. Momm, A. Spriestersbach, J. Vogel, G. Le Jeune, A. Giessmann, F. Junker, M. Dao, S. Carrie, J. Niemoller, and D. Mazmanov, "4CaaSt: Comprehensive Management of Cloud Services through a PaaS," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pp. 494–499, 2012.

[40] R. Sharma and M. Sood, "A model-driven approach to cloud saas interoperability.," *International Journal of Computer Applications*, vol. 30, 2011.

[41] J. Kirschnick, J. M. Alcaraz Calero, P. Goldsack, A. Farrell, J. Guijarro, S. Loughran, N. Edwards, and L. Wilcock, "Towards an architecture for deploying elastic services in the cloud," *Softw. Pract. Exper.*, vol. 42, pp. 395–408, Apr. 2012.