SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **00**:1–34
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

CloudPick: A Framework for QoS-aware and Ontology-based Service Deployment Across Clouds

Amir Vahid Dastjerdi *, Saurabh Kumar Garg†, Omer F. Rana‡, and Rajkumar Buyya *

SUMMARY

The cloud computing paradigm allows on-demand access to computing and storage services over the Internet. Multiple providers are offering a variety of software solutions in the form of virtual appliances and computing units in the form of virtual machines with different pricing and Quality of Service (QoS) in the market. Thus, it is important to exploit the benefit of hosting virtual appliances on multiple providers to not only reduce the cost and provide better QoS but also achieve failure resistant deployment. This paper presents a framework called CloudPick to simplify cross-cloud deployment and particularly focuses on QoS modeling and deployment optimization. For QoS modeling, cloud services have been automatically enriched with semantic descriptions using our translator component to increase precision and recall in discovery and benefit from descriptive QoS from multiple domains. In addition, an optimization approach for deploying networks of appliances is required to guarantee minimum cost, low latency, and high reliability. We propose and compare two different deployment optimization approaches: genetic-based and Forward-Checking-Based Backtracking (FCBB). They take into account QoS criteria such as reliability, data communication cost, and latency between multiple Clouds to select the most appropriate combination of virtual machines and appliances. We evaluate our approach using a real case study and different request types. Experimental results suggest that both algorithms reach near optimal solution. Further, we investigate effects of factors such as latency, reliability requirements, and data communication between appliances on the performance of the algorithms and placement of appliances across multiple Clouds. The results show the efficiency of optimization algorithms depends on the data transfer rate between appliances.
Copyright © 2014 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Cloud Computing; Cloud Service Composition; Quality of Service; Multi-Clouds

1. INTRODUCTION

The advantages of cloud computing platform, such as cost effectiveness, scalability, and ease of management, encourage more and more companies and service providers to adopt it and offer their solutions via cloud computing models. According to a recent survey of IT decision makers of large companies, 68% of the respondents expect that by the end of 2014 more than 50% of their companies' IT services will be migrated to cloud platforms [1].

*A.V Dastjerdi and R Buyya are with Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Parkville, VIC 3010, Australia. R. Buyya also serves as a Visiting Professor for the University of Hyderabad, India; King Abdulaziz University, Saudi Arabia; and Tsinghua University, China.
Email: amir.vahid@unimelb.edu.au and rbuyya@unimelb.edu.au
†S.K. Garg is withDepartment of Computing and Information System, Faculty of Engineering and ICT, University of Tasmania. Email:Saurabh.Garg@utas.edu.au
‡Omer F. Rana is with School of Computer Science. Cardiff University. Wales, United Kingdom.
Email:o.f.rana@cs.cardiff.ac.uk

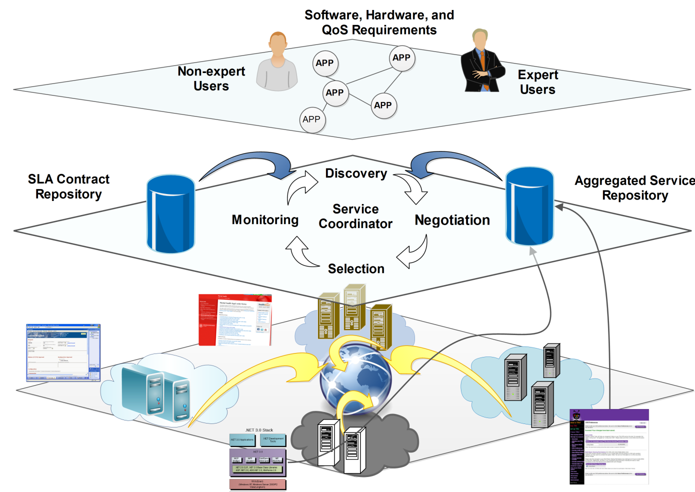


Figure 1. Service coordination in a multi-cloud environment.

In order to offer their solutions in the cloud, service providers can either utilize Platform-as-a-Service (PaaS) offerings such as Google App Engine [2], or develop their own hosting environments by leasing virtual machines from Infrastructure-as-a-Service (IaaS) providers like Amazon EC2 [3]. However, most PaaS services have restrictions on the programming language, development platform, and databases that can be used to develop applications. Such restrictions can encourage service providers to build their own platforms using IaaS service offerings.

One of the key challenges in building a platform for deploying applications is to automatically select and configure necessary infrastructures. If we consider the deployment requirements of a web application service provider, it will include security devices (e.g. firewall), load balancers, web servers, application servers, database servers, and storage devices. Setting up such a complex combination of applications is costly and error prone even in traditional hosting environments [4], let alone in clouds. Virtual appliances can provide an elegant solution for this problem.

A virtual appliance is a virtual machine image that has all the necessary software components to meet a specific business objective pre-installed and configured [5] and can be readily used with minimum effort. Virtual appliances will not only eliminate the effort required to build these appliances from scratch, but also will avoid any associated issues such as incorrect configuration. To overcome deployment problems such as root privilege requirements and library dependencies, virtual appliance technology is adopted as a major cloud component.

As multiple providers are offering different software solutions (appliances) and virtual machines (units) with different pricing in the market, it is important to exploit the benefit of hosting appliances on multiple providers to reduce the cost and provide better QoS. However, this can be only possible if high throughput and low latency could be guaranteed among different selected clouds. Therefore, the latency constraint between nodes has to be considered as key QoS criteria in the optimization problem. Amazon EC2, GoGrid, Rackspace, and other key players in the IaaS market, although they constitute different deployment models using virtual appliances and units (computing instances), do not provide a solution for composing those cloud services based on users functional and non-functional requirements such as cost, reliability and latency constraints.

If we make the assumption that service providers prefer IaaS and multi-cloud, they have to go through a process to select the most suitable cloud offerings to host their services. This process, which is called cloud service coordination, consists of four phases, namely discovery, Service Level Agreement (SLA) negotiation, selection, and SLA monitoring as shown in Figure 1. In the service discovery phase, users with different level of expertise provide their requirements as input for discovering the best suited cloud services among various repositories of cloud providers. For SLA Negotiation, discovered providers and the user negotiate on the quality of services. A set of SLA

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

contracts is selected from a set of made agreements. Then, the acquired services are continuously monitored in the SLA monitoring phase.

The first step to enable cross-cloud deployment optimization is to model the appliances, virtual units, and QoS requirements of users. Currently, there is no single directory that lists all the available virtual appliances and units. Hence, we need an approach to automatically build a directory of aggregated commonly described virtual appliance and unit information. In the next step, we have users' requests (group of connected appliances) with different latency, reliability and budget constraints, and the objective of minimizing the deployment cost, in one hand and in another hand we have various combinations of appliances and virtual units in the aggregated repository. The problem is to find a composition that adheres to user constraints and minimizes the cost of deployment. After the composition is selected, and the appliances are deployed, a standard cloud-agnostic format is required for storing the deployment configurations. This format can later be used for discovering and reconfiguring alternative deployments in the case of failure. To address the aforementioned challenges, in this work, which is a significant extension of our previous conference paper [6], we propose a novel framework called CloudPick.

The **major contributions** of this paper are: 1) proposing an effective architecture that utilizes ontology-based discovery and deployment descriptor and optimization techniques to simplify service deployment in multi-cloud environments, 2) proposing an approach to automatically build an aggregated semantically enriched cloud service (along with their non-functional properties) repository, 3) modeling of relevant QoS criteria, namely latency, cost (data transfer cost, virtual unit, and appliance cost), and reliability for selection of the best virtual appliances and units in cloud computing environment, and 4) presenting and evaluating two different selection approaches, genetic-based and Forward-checking-based backtracking, (as the major focus of performance evaluation section) to help users in deploying network of appliances on multiple clouds based on their QoS preferences.

The remainder of this paper is organized as follows. In the next section, a brief introduction to necessary concepts related to the paper is given. Related work in contexts of SOA, Grid and cloud computing is discussed in Section 3 following by Set of questions that motivate our work in Section 4. Then, we present description of CloudPick components that are addressing cross-cloud deployment challenges in Section 5. Section 6 contains a translation approach to decrease the human intervention in the process of converting virtual appliance meta-data to ontology-based annotations. Section 7 presents QoS criteria and algorithms required for the optimization. Section 8 focuses on building an experimental testbed and using it to compare the optimization algorithms' performances and study appliances placement patterns. Finally, Section 9 concludes the paper and presents future research directions.

2. PRELIMINARIES

In this section, concepts related to our approach, e.g. Web Service Modeling Ontology (WSMO) and virtual appliance are described.

2.1. Web Service Modeling Ontology (WSMO)

Web Service Modeling Ontology (WSMO) [7] defines a model to describe Semantic Web Services, based on the conceptual design set up in the Web Service Modeling Framework (WSMF). WSMO identifies four top-level elements as the main concepts:

- Ontologies: They provide the (domain specific) terminologies used and is the key element for the success of Semantic Web services. Furthermore, they use formal semantics to connect machine and human terminologies.
- Web services: They are computational entities that provide some value in a certain domain. The WSMO Web service element is defined as follows:
  - Capability: This element describes the functionality offered by a given service.

Copyright © 2014 John Wiley & Sons, Ltd.

Softw. Pract. Exper. (2014)
DOI: 10.1002/spe

  - Interface: This element describes how the capability of a service can be satisfied. The Web service interface principally describes the behavior of Web Services.

- Goals: They describe aspects related to user desires with respect to the requested functionality, i.e. they specify the objectives of a client when consulting a web service. Thus they are individual top-level entities in WSMO.
- Mediators: They describe elements that handle interoperability problems between different elements, for example two different ontologies or services. Mediators can be used to resolve incompatibilities appearing between different terminologies (data level), to communicate between services (protocol level), and to combine Web services and goals (process level).

Besides these main elements, non-functional properties such as cost, deployment time, performance, scalability, and reliability are used in the definition of WSMO elements that can be used by all its modeling elements. Furthermore, there is a formal language to describe ontologies and Semantic Web services called WSML (Web Service Modeling Language) [8] that contains all aspects of Web service descriptions identified by WSMO. In addition, WSMX (Web Service Modeling eXecution environment) [9] is the reference implementation of WSMO, which is an execution environment for business application integration.

2.2. Virtual Appliance

Virtual appliances are pre-configured and read-to-run virtual machine images that can be run on top of a hypervisor. The main objective of virtual appliances is decreasing the cost and labor associated with installing and configuring complex stacks of softwares in Cloud computing environments. In recent designs and implementations of virtualization systems, virtual appliances get the most attention. The idea has been initially presented [5] to address the complexity of system administration by making the labor of applying software updates independent of number of computers on which the software runs. Overall, the work develops the concept of virtual networks of virtual appliances as a means to reduce the cost of deploying and maintaining software. VMware [10] introduces a new generation of virtual appliances which are pre-installed, pre-configured, and ready to run. However, in practical scenarios, pre-configured solutions can not satisfy varying requirements of users. In addition, those pre-configured virtual appliances require large amount of storage space, if the system supports variety of operating system and software combinations. And it is not feasible for all range of users to have huge storage devices to store all those appliances shaped based on their configuration. Moreover, Amazon has launched AWS Marketplace, which enables customers to search for appliances from trusted vendors, pay for them in a pay-as-you-go manner, and run them on the EC2 [3] infrastructure.

3. RELATED WORK

The concept of virtual appliances was originally introduced to simplify their deployment and management of desktop personal computers in enterprise and home environments [5]. Then they have been adapted in Grid and Cluster Computing environments to simplify the deployments [11]. With the emergence of cloud Computing, which utilizes virtualization to provide elastic usage of resources, virtual appliances are becoming the preferred technology to deploy applications on virtual machines with minimum effort. Hence, virtual appliance deployment has been investigated in industry and academia from various angles which includes planning, modeling, QoS-based deployment optimization, and service selection.

Sun et al. [4] showed that, by utilizing virtual appliances, the deployment process of virtual machines can be made simpler and easier. Wang et al. [12] presented a framework to improve the efficiency of resource provisioning in large data centers using virtual appliances. Similarly, a framework for service deployment in cloud based on virtual appliances and virtual machines has been introduced in our previous work [13]. That research focused on selecting suitable virtual

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

machines using ontology based discovery model, packaging, and deploying them along with virtual appliances in the cloud platform, and monitoring the service levels using third parties. In this work, we are concentrating on QoS-based virtual unit and appliance composition where multiple appliances need to be deployed across multiple clouds with acceptable latency and reliability to achieve users' business objectives.

A single virtual appliance on a virtual unit will not be able to fulfill all the requirements of a business problem. Inevitably, we will require more than one virtual appliance and unit working together to provide a complete solution. Hence, it is important to develop compositions of virtual unit and appliances. Konstantinou et al. [14] proposed an approach to plan, model, and deploy virtual appliance compositions. In their approach, the solution model and the deployment plan for virtual appliance composition in cloud platform are developed by skilled users and executed by unskilled users. As discussed by them, the contribution has not proposed an approach for selection of virtual appliance and machine providers. In our work, however, we consider that users will be only aware of the high level components that are required for the composition to address their business objectives and our solution provides an approach to select the best composition based on their functional and QoS requirements. Similarly, Chieu et al. [15] proposed the use of composite appliances to automate the deployment of integrated solutions. However, in their work, QoS objectives are not considered when building the composition.

Characteristics of the deployment optimization and service selection and composition in cloud differ from works done in other contexts such as Grid and web services. Grid Computing aims to "enable resource sharing and coordinated problem solving in dynamic, multi-institutional virtual organizations" [16]. Therefore, the QoS management and composition works in this context mainly focus on load balancing (applying queuing theory and market driven strategy [17]) and fair distribution of resources among service requests [18, 19]. Most of these works proposed Constraint Satisfaction based Matchmaking Algorithm (CS-MM) and other artificial intelligence-based optimization techniques to improve the performance of scheduling. However, In Service Oriented Architecture's (SOA) context, the main concern is defining a QoS language [10, 20] to express user preferences and QoS properties of the service (e.g. semantic-based QoS description [21]). In this context, for automated web services composition, various techniques such as workflow and AI planning have been adapted [22].

However, in the context of cloud computing, the deployment optimization's objective is not fair distribution of resources between requesters. Instead, cloud customers have emphasized more on QoS dimensions such as reliability and cost. Therefore, in this work we present a novel way to measure composition reliability and suitability based on Service-Level Agreements (SLA). In addition, the data transfer cost is also included in our deployment cost. The importance of modeling data transfer cost can be realized by the example of deployment in Amazon cloud where data transfer costs approximately $100 per terabyte. These costs quickly add up and become a great concern for the administrator. In the context of cloud computing, there are several works that have focused on deployment optimization challenges such as Optimis, Mosaic, and Contrail.

Optimis's [23] main contribution is optimizing the whole service life cycle, from service construction and deployment to operation. The considered QoS criteria are trust, risk, eco-efficiency and cost. In Optimis, the evaluation of cloud providers is accomplished through an adoption of Analytical Hierarchy Process (AHP). In comparison with our approach, works that applied AHP and Multi-Attribute Utility Theory (MAUT) [24] can only perform well when the number of explicitly given service candidates is small and the number of objectives is limited. In contrast, as shown in Section 8.2, our approach can deal efficiently with a large number of cloud services in the repository.

Mosaic project [25] is proposed to develop multi-cloud oriented applications. In Mosaic, cloud ontology plays an essential role, and expresses the application's needs for cloud resources in terms of SLAs and QoS requirements. It is utilized to offer a common access to cloud services in cloud federations. Compared to our work, which also adopts ontology, Mosaic is not able to create ontology automatically from information provided through API calls to clouds. In addition, the provided semantic cloud services in contrast to our work do not contain QoS information.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Contrail [26] is another project which builds a federation that allows users to utilize resources belonging to different cloud providers. Like our previous work, they use OVF meta-data (in the format of XML) to acquire resources from multiple cloud providers. However they have not considered deployment optimization by considering criteria such as cost, latency, and reliability.

CloudGenius [27] is a framework that focuses on migrating single tier Web application to the cloud by selecting the most appealing cloud services for users. CloudGenius considers different sets of criteria and dependencies between virtual machine services and virtual appliances to pick up the most appropriate solution. Like the majority of the works in the cloud computing context, it chooses AHP for ranking cloud services. Since pair-wise comparisons for all cloud services are computing intensive, the selection criteria were restricted to numerical criteria.

4. MOTIVATION: SCENARIO AND CHALLENGES

To study user requirements and concerns for deploying a network of appliances on clouds, we give an example of a real world case study with known network traffic between appliances. A good example of network of virtual appliances (a set of appliances in the form of a connected graph which have data communication among them) is multi-tier applications supporting web-based services. Each tier has communication requirements as characterized by Diniz Ersoz et al. [28]. They considered a data center with 11 dedicated nodes of a 96-nodes Linux cluster and host an e-business web site encompassing 11 appliances: 2 front-end Web-Servers (WS) in its web tier, 3 Databases (DB) in its database tier and 6 Application Servers (AS) in between. As they have characterized network traffic between tiers, we selected their work to build our case study. Assume that the administrator of the e-Business web site might be interested in migrating the appliances to the cloud in order to save on upfront infrastructure and maintenance costs, as well as to gain the advantage of on-demand scaling. In addition, to allow disaster recovery and geography-specific service offering, one may prefer multiple cloud deployment. For such deployment, the administrator faces several challenges such as:

1. How to automatically build an integrated repository of cloud services so that their functional and QoS properties are understood by all parties (users, cloud service providers, monitoring service providers) to avoid low precision and recall in cloud service discovery?
2. What is the best strategy for placing appliances across cloud providers? Should they be placed based on the traffic they exchange, therefore placing those with higher connectivity closer to each other to decrease latency and data transfer cost?
3. Is it economically justifiable?
4. If appliances are placed across multiple providers, how the latency between different providers affects the performance?
5. How can the most reliable cloud services be selected for the deployment?
6. If all appliances and their related deployment meta-data such as auto-scaling policies and security configuration are placed on the same provider, and that provider fails, the access to deployment information would not be guaranteed. Consequently, the recovery process would be significantly delayed.

To address aforementioned issues and enabling cross-cloud service deployment, we introduce CloudPick.

5. CLOUDPICK ARCHITECTURE

The proposed architecture is depicted in Figure 2 and its main components are explained below:

1. **User Portal:** All services provided by the system are presented via the web portal to clients. This component provides graphical interfaces to capture users' requirements such as software, hardware, QoS requirements (including maximum acceptable latency between tiers, minimum

Copyright © 2014 John Wiley & Sons, Ltd.

Softw. Pract. Exper. (2014)
DOI: 10.1002/spe

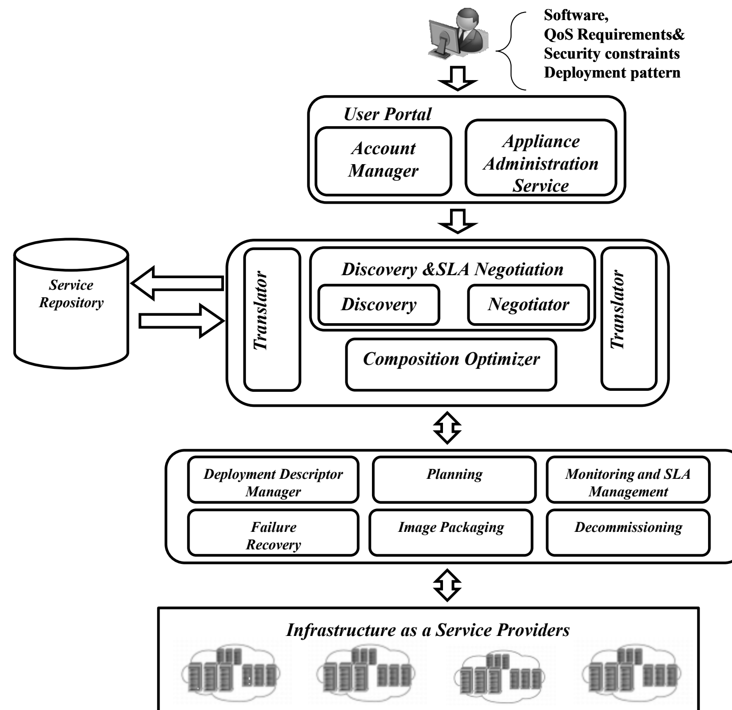


Figure 2. CloudPick's main components that enable cross-cloud deployment of virtual appliances.

acceptable reliability, and budget), firewall, and scaling settings. In addition, it transforms user requirements to WSMO format in the form of goals which are then used for cloud service discovery and composition. Moreover, it contains an account manager, which is responsible for user management. For more details regarding the format of goals, readers can refer to our previous work [13]

2. **Translator:** Since Web Service Modeling Ontology (WSMO) is used for service discovery, cloud services information is translated to the Web Service Modeling Language (WSML) format by the Translator component. This component takes care of building and maintaining an aggregated repository of cloud services and is explained in detail in Section 6.1.
3. **Cloud Service Repositories:** They are represented by appliance and virtual unit service repositories in Figure 2 and allow IaaS providers to advertise their services. For example, an advertisement of a computing instance can contain descriptions of its features, costs, and the validity time of the advertisement. From standardization perspective, a common metamodel that describes IaaS provider's services has to be created. However, due to the lack of standards, we developed our own metamodel [13] based on previous works and standards in this area using WSMO.
4. **Discovery and Negotiation Service:** Non-logic based discovery systems in grid and cloud (IBM Smart Cloud Catalog search, Amazon EC2 image search) require exact match between a client's goal and a provider's service description. In a heterogeneous environment such as cloud, it is difficult to enforce syntax and semantics of QoS descriptions of services and user requirements. Therefore, applying symmetric attribute-based matching between requirements and a request is impossible. Building semantics of cloud services, user requirements, and data would provide an inter-cloud language which helps providers and users share common understanding regarding the cloud service functionalities, QoS criteria, and their measurement units. A semantic service that is built by the translator component is a result of a procedure in which logic-based languages over well-defined ontologies are used to describe functional and non-functional properties of a service. This allows our ontology-based discovery technique to

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

semantically match services with user requirements and avoid low recall caused by lack of common service functionalities and QoS understandings.
As cited by Faratin et al. [29], time-dependent negotiation tactics are a class of functions that compute the value of a negotiation issue by considering the time factor. They are particularly helpful for our scenario, where we have to acquire services by a deadline. Therefore, our negotiation service uses a time-dependent negotiation strategy that captures preferences of users on QoS criteria to maximize their utility functions. In addition, since in parallel negotiation a party makes a decision based on the presented QoS values in SLA offers, our Negotiation Service provides a way to know how reliable the provider is in delivering those promised QoS values. To this end, the recorded data from monitoring services is analyzed and converted to reliability information of offers. The monitoring is based on the copy of the signed SLA, which is kept in the SLA repository. The proposed negotiation strategies are described in detail in our previous work [30].

5. **Composition Optimizer:** Once the negotiation completes and eligible candidates are identified, the composition component, which is the focus of this paper, builds the possible compositions candidates. Then Composition Optimizer evaluates the composition candidates using the users' QoS preferences. The Composition Optimizer takes advantage of the proposed selection algorithms that are explained in Section 7.3 to provide an elegant solution to the composition problem.
6. **Planning:** The Planning component determines the order of appliance deployment on the selected IaaS providers and plans for the deployment in the quickest possible manner.
7. **Image Packaging:** The Packaging component builds the discovered virtual appliances and the relevant meta-data into deployable packages, such as Amazon Machine Image (AMI) or Open Virtualization Format (OVF) [20] packages. Then the packages are deployed to the selected IaaS provider using the deployment component.
8. **Deployment Component:** It configures and sets up the virtual appliances and computing instances with the necessary configurations such as firewall and scaling settings. For example in a web application, specific connection details about the database server need to be configured.
9. **Deployment Descriptor Manager:** This component persists specifications of required services and their configuration information such as firewall and scaling settings in a format called Deployment Descriptor. Besides, it includes the mapping of user requirements to the instances and appliances provided by the cloud. The mapping includes instance description (e.g. name, ID, IP, status), image information, etc. This meta-data is used by the appliance administration service to manage the whole stack of cloud services even if they are deployed across multiple clouds. Formally described using WSML, the Deployment Descriptor is located in our system (as a third party service coordinator), and in a cloud-independent format that is used for discovering and configuring alternative deployments in case of failures. An example of a Deployment Descriptor is shown in Appendix B. It identifies how firewall and scaling configurations have to be set for Web server appliances. In addition, Deployment Descriptor helps to describe the utility function of users for provisioning extra cloud services when scaling is required. This helps to create scaling policies that utilize the optimization component on the fly to provision services that maximizes the user's utility functions. For example, providers that have the lowest price, latency, and highest reliability are going to be ranked higher.
10. **Appliance Administration Service:** After the deployment phase, this component helps end users to manage their appliances (for example starting, stopping, or redeploying them). It uses the Deployment Descriptor to manage the deployed services.
11. **Monitoring and SLA Management:** This component provides health monitoring of deployed services and provides the required inputs and data for failure recovery and scaling. A monitoring system is provided by this component for fairly determining to which extent an SLA is achieved as well as facilitating a procedure taken by a user to receive compensation when the SLA is violated. The monitoring is based on the copy of signed SLA, which

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls

is kept in SLA repository. The component provides an approach to discover and rank necessary third party monitoring services. Third party monitoring results can be similar to what the CloudStatus* service reports. Hyperic's CloudStatus is the first service to provide an independent view into the health and performance of the most popular cloud services, including Amazon Web services and Google App Engine. CloudStatus gives users real-time reports and weekly trends on infrastructure metrics including service availability, response time, latency, and throughput that affect the availability and performance of cloud-hosted applications. More details on this component is provided in our previous paper [31].

12. **Failure Recovery:** It automatically backs up virtual appliance data and redeploys them in the event of cloud service failure.
13. **Decommissioning:** In the decommissioning phase, cloud resources are cleaned up and released by this component.
14. **IaaS Providers:** They are in both fabric and unified resource level [16] and contain resources that have been virtualized as virtual units. A virtual unit can be a virtual computer, database system, or even a virtual cluster. In addition to virtual units, IaaS providers offer virtual appliances to satisfy software requirements of users.

5.1. Execution workflow of CloudPick

Consider a user request that includes two machines: A and B. The machine A is required a minimum CPU capacity of 2 GHz, RAM capacity of 2 GB, Hard Disk capacity of 200 GB, and AIX operating system. The machine B has similar requirements, however it entails a minimum RAM capacity of 4 GB, and a UNIX-based operating system. The maximum acceptable latency between two machines is set to 5 ms.

5.2. Initial phases

First, every user should have an account in the system. The account is used for userâĂŹs authentication and authorization; besides, it stores all user information regarding their requests. In CloudPick, information regarding the machine A and B requirements, network and firewall settings are stored in the form of Deployment Descriptor. This information can help the systems to offer better quality of service to the user. For example consider a scenario that a user face the failure in deploying his appliances on a specific Cloud in the previous interaction with system, this information which is stored in a Cloud agnostic format passes to the deployment service to rapidly provision resources in another service provider. As mentioned earlier, it is necessary to build a service repository which contains semantic description of Cloud services, such as their capabilities (pre-conditions, post-conditions, assumptions and effects), interfaces (choreography) and non-functional properties. This is the place for all IaaS providers to advertise their virtual units as a service. Ontology repository is built up to contain ontologies for describing semantics of particular domains. Any components might wish to consult ontology, but in most of the cases ontologies will be used by the mediator related components to overcome data and process heterogeneity problems. In our case, semantic has to be described for operating systems, virtual hardwares, and other QoS domains, etc.

5.3. Execution phases

Once user requirements in the form of Deployment Descriptor is received, it may just describe some of needed resources for example only CPU and storage. In this situation, default values for other requirements are assigned by the portal. These default values are presented by the portal and could be assigned according to the software requirements and previous requested virtual units of users. In the next phase the Deployment Descriptor for machine A and B are used by Discovery component as an input for searching the best suited virtual appliances and machines. The Discovery component

---
* Hyperic. `http://www.hyperic.com/products/Cloud-monitoring.html`

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

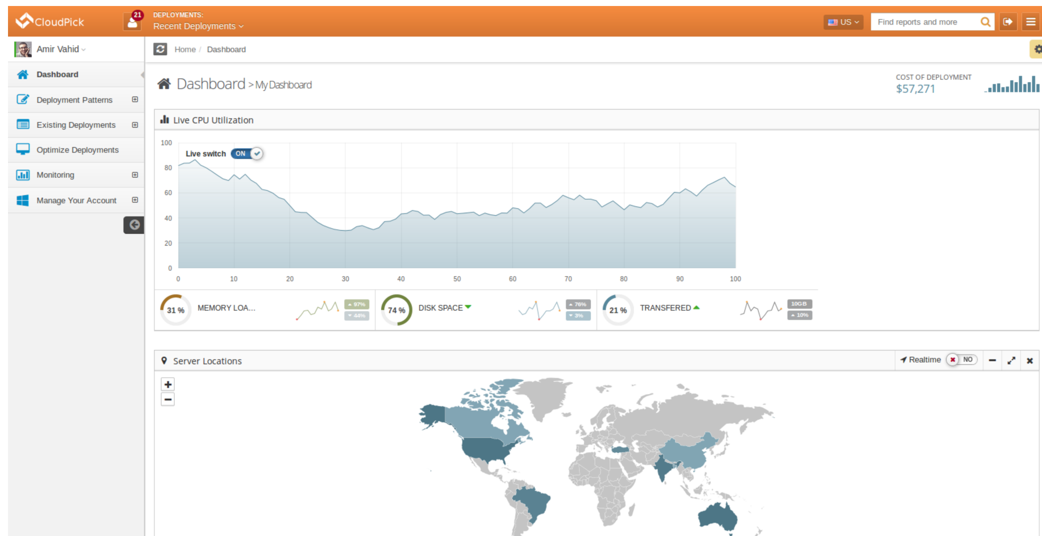


Figure 3. CloudPick's dashboard.

as explained in the previous section checks the capabilities of virtual units and appliances against the resource requirements in the Deployment Descriptors of machine A and B. For machine B, Since the knowledge base (KB) specifies that both Linux family and OpenSolaris are types of Unix, therefore not only X (supplying Linux virtual appliances) but also Y (supplying OpenSolaris virtual appliances) IaaS provider, pass the virtual appliance and virtual machine requirement criteria. For machine A, only provider Z can supply AIX virtual appliance in its infrastructure. After the discovery phase, if providers support bargaining, the Negotiation Service is called to negotiate for the minimum cost and the highest QoS with provider X, Y and Z. The result of negotiation along with achieved QoS and cost are passed to the Composition Optimizer component. Composition optimizer then, using the proposed optimization techniques, search the problem space. It returns X and Z as they could stratify latency constraints of 5 ms and the total cost of deployment is minimum among the other candidates (that is less than the budget in the Deployment Descriptor). Once the cloud services are selected they are passed to Deployment Manager to be mapped to deployment descriptor requirements. After that deployment manger provision the cloud services and configure them based on user preferences and calls monitoring services with Service Level Objectives obtained during negotiation. In a case of any SLA violations and when real source of failure is detected, the monitoring service updates related QoS information of services in the repository.

5.4. *Implementation*

In order to realize the proposed architecture, a number of components and technologies are utilized.

- **Development Framework:** CloudPick is built using Spring MVC Framework [†] and it benefits from Spring Security and Data projects to develop a extensible, secure, and modular web application. It is worth mentioning that major components of CloudPick are designed to expose their functionalities via RESTful services [32] using Spring MVC framework. This provides a standard way and enforces simple and yet powerful rules for communicating to users and also other services.

[†]Spring MVC Framework `http://spring.io/`

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

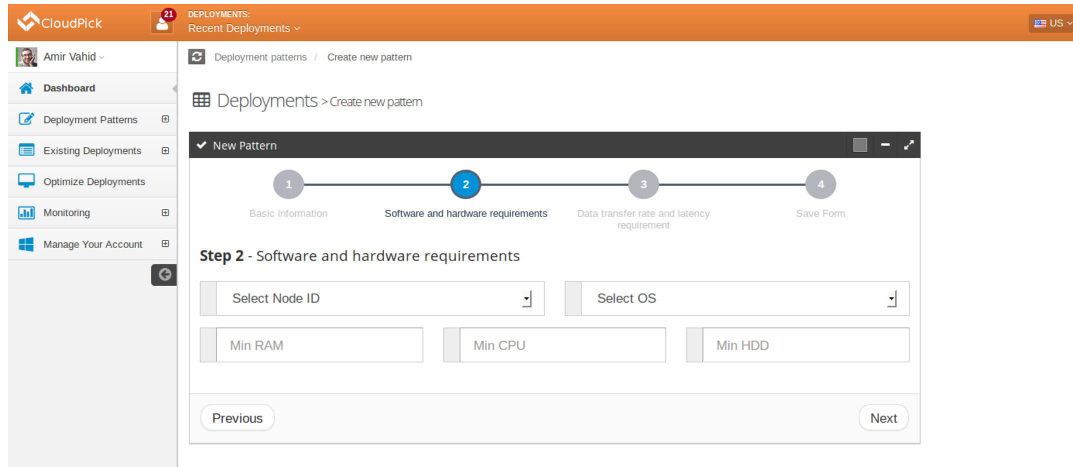


Figure 4. Capturing user requirements in CloudPick.

- **Portal:** A light-weight portal is built for CloudPick using Twitter Bootstrap ‡ and jQuery § to create an elegant interface on every device as well as speed up development time. Figures 3 and 4 demonstrate how the dashboard and a form for acquiring user requirements are designed in CloudPick.
- **Process Management:** As the deployment of appliances has to be accomplished through a multi-step process Bonita [33] is used to orchestrate and compose aforementioned tasks. In cloudPick, Bonita helps us to manage a process that coordinates between end users, our frameworks, and utilized service, maintain process state, and log all process events.
- **Cloud Service Discovery and Translation:** The WSMO Discovery Engine [34] is utilized to provide dynamic cloud service discovery. It exploits WSMO formal descriptions of user requirements and services that is built by the translator component (refer to Section 6.1 for more details).
- **Connecting to Multiple clouds:** To deploy selected appliances on the selected IaaS provider, we have to utilize cloud APIs (either in the form of command line or web service requests). Although there are efforts to derive standard APIs to access and configure cloud services, those standards have not yet resulted in a dependable product. To resolve that issue, we adopted the jclouds API, which provides an option to use either portable abstractions or cloud-specific features. It supports a number of cloud providers including Amazon, GoGrid, Azure, vCloud, and Rackspace. It is an open source library that helps users to easily manage the public and private cloud platforms using their existing Java and development skills.
- **Image Packaging:** This component is implemented to utilize sets of APIs provided by cloud providers through jclouds (dynamically and based on the source and destination providers) to create virtual appliance packages and convert them to different formats. For example, if it is required to deploy a VMDK virtual machine image on Amazon EC2, the component uses the Import/Export Tools¶ of Amazon EC2 to convert it to AMI format.
- **Monitoring:** This service use the CloudHarmony RESTful API ‖ to monitor cloud services. More specifically, as we are particularly interested in collecting information regarding availability, the "*getAvailability*" service is used to obtain information regarding outages that occurred over the specified time period. The collected information includes downtime which is the total number of minutes for a particular outage.

‡Twitter Bootstrap. `http://bootstrapdocs.com/v3.0.0/docs/`
§jQuery. `http://jquery.com/`
¶EC2 Import/Export Tools. `https://aws.amazon.com/ec2/vm-import/`
‖CloudHarmony. `http://cloudharmony.com/ws/api`

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

- **Optimization:** For Implementation of our optimization algorithm based on Genetic Algorithm, Java Genetic Algorithm Package (JGAP) [35] is used. It offers a number of fundamental genetic mechanisms that can be used to apply evolutionary principles to our cloud service deployment optimization problem.

6. CLOUD SERVICE MODELING

There are two major phases in the cloud deployment optimization process. First, the cloud virtual units and appliances information, including their QoS values, has to be collected, aggregated, and translated to the format which is commonly understood by all the parties. As discussed by Kritikos [36], this can be achieved by the adoption of semantic services, which is known as the most expressive way of describing QoS. For this purpose, we extended WSML to support description of cloud service QoS. Currently, virtual appliances and units meta-data are defined in the form of XML, however to get the advantages of Ontology-based discovery, they have to be described conceptually using WSMO ontologies in the form of WSML. The manual translation of cloud appliance and virtual unit offerings' descriptions is not a feasible approach. Therefore, we propose an approach that minimizes human intervention to semantically enrich cloud offerings.

6.1. Automated Construction of Semantic-Based Cloud Service and Their Quality of Services

Currently, there is no integrated repository of semantic-based services for virtual appliances and units. The first step towards describing services and their QoS is to communicate with clouds and the cloud monitoring services through their APIs and gather required meta-data for building the repository. The process of metadata translation is demonstrated in Figure 5. The components involved in this process are:

6.1.1. Integrity Checking This component first merges output messages of API calls for acquiring cloud services description using Extensible Stylesheet Language Transformations (XSLT)[**] and then compares them with the previously merged messages using a hash function. If the outputs of the hash function are not equal, the component triggers the Sync component to update the semantic repository.

6.1.2. Sync Component The goal of this component is to keep the semantic-based repository consistent with the latest metadata provided by cloud providers. As the synchronization is computing intensive, it is avoided unless the integrity checking component detects any inconsistency. In this case, the component receives the output message that is required for synchronization and finds the corresponding semantically rich services and updates them with the output of the translator component.

6.1.3. Translator Component During the communication of a semantic-level client and a syntactic-level web service, two directions of data transformations (which is also called grounding) are necessary: the client semantic data must be written in an XML format that can be sent as a request to the service, and the response data coming back from the service must be interpreted semantically by the client. We use our customized Grounding technique on WSDL operations (that are utilized to acquire virtual appliance and unit metadata) output to semantically enrich them with ontology annotations. WSMO offers a package that utilizes Semantic Annotations for WSDL (SAWSDL) for grounding [37]. It provides two extensions attribute namely as Lifting Schema Mapping and Lowering Schema Mapping. Lowering Schema Mapping is used to transfer ontology to XML and lifting Schema Mapping does the opposite. In our translator component, the lifting mapping extension has been adopted to define how XML instance data obtained from clouds API calls is transformed to a semantic model.

[**]XSLT. `http://www.w3.org/TR/xslt`

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

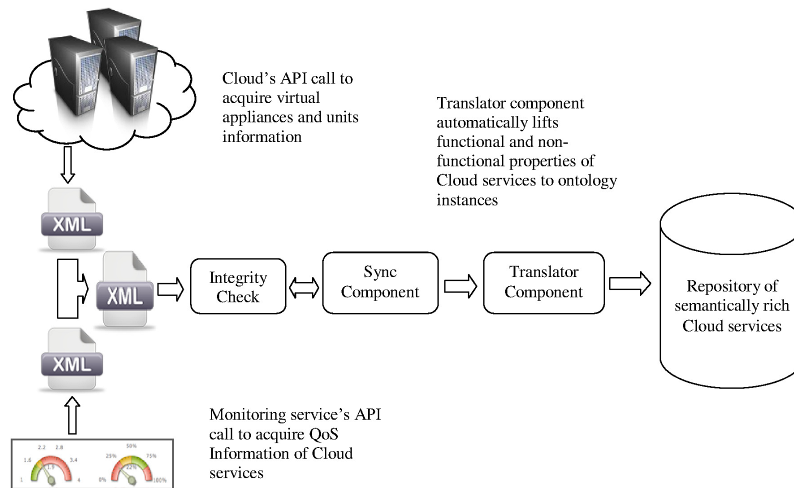


Figure 5. The process of translation of virtual appliances and units descriptions to WSML.

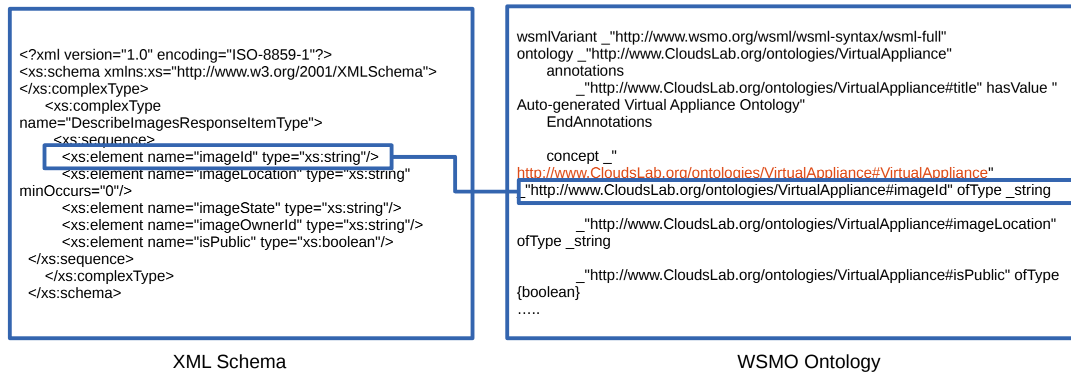


Figure 6. The mapping of the XML Schema to Virtual Appliance ontology concept.

As the first step in grounding, from output message schema, the necessary ontology is created for virtual units and appliances. The basic steps to build the ontology from XML schema using WSMO grounding is explained by Kopecky et al. [37]. In our implementation, we defined conceptual mappings between the XML Schema conceptual model and the WSMO Ontology model and build an engine that uses these mappings and automatically produces cloud service WSMO ontology out of an acquired XML Schema. The implemented engine maps the primary types of XML Schema elements to WSML-supported types. A simple mapping of such kind is provided in Figure 6. In this step our contribution lies on building the ontology from multiple output message schemas. It means that the monitoring service output message schema is used to extend the ontology to encompass non-functional properties. This can be accomplished by merging two schemas to construct an output message that describes the format of the elements that has functional and non-functional properties such as price and reliability.

Having the ontology available, the next step is to add the necessary Mapping URI for all element declarations. For this purpose Modelreferences are used, which are attributes whose values are lists of URIs that point to corresponding concepts in the constructed ontology. Subsequently, we need to add schema mappings that point to the proper data lifting transformation between XML data and semantic data. For this purpose, two attributes, namely liftingSchemaMapping and loweringSchemaMapping, are offered by SAWSDL. These aforementioned attributes are then utilized to point from cloud virtual appliance meta-data schema to a XSLT, which shows how meta-data is transferred from XML to WSML.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

We tested this approach for cloud service repositories with variety of sizes, and present the experimental result in Section 8.2.1. The ontology listed in Appendix A was partially created by the described translator component. For example, it shows how an appliance meta-data with ID of "aki00806369" has been translated to WSMO format.

Semantic service toolkits and libraries based on OWL-S and WSMO use XML based grounding. This XML mapping approach cannot deal with the growing number of cloud provider's interfaces that use non-SOAP and non-XML services. The main reason that we have used XML is to follow the path that was suggested by WSMO, standard libraries and documentation provided by WSMO, and that major IaaS providers currently have a full support for XML-based services. For alternative approaches of grounding for non-XML services, readers can refer to studies conducted by Lambert et al. [38]. It is worth mentioning that there are other specifications such as Open Cloud Computing Interface (OCCI) [39] that aims at providing a standard way for describing cloud resources.

7. DEPLOYMENT OPTIMIZATION

After the discovery phase –which is explained in our previous works [6,10,13] along with semantic-based virtual appliance and units description in WSML– the discovered services are passed to the deployment optimization component. The deployment optimization step consists of finding the composition of appliances and virtual units for the customers that minimizes the deployment cost and adheres to reliability and latency constraints. The deployment problem maps to multi-dimensional knapsack problem due to multiple QoS constraints. The Multidimensional Knapsack problem is classified as an NP-hard optimization problem [40]. It consists of selecting a subset of alternatives in a way that the total profit of the selected alternatives is maximized while a set of knapsack constraints are satisfied. First, the QoS criteria are described and then the optimization problem is formally defined.

7.1. QoS Criteria

The three QoS criteria considered in the deployment optimization problem are reliability, cost, and latency.

1. **Reliability:** For measuring cloud providers reliability, we introduce SLA Confidence Level (SCL), which is a metric to measure how reliable are services of each provider based on the SLAs and their performance history. SCL values are computed by a third party that is responsible for monitoring the SLA of providers based on Equation (1):

$$SCL = \sum_{j=1}^{k} (I_j \times SCL_j) \tag{1}$$

Where SCL_j is the SLA confidence level for QoS criteria j of a cloud service; I_j is the importance of the criteria j for the user; k is the number of monitored QoS criteria.
We utilized the beta reputation system [41] to assess the SCL for each criterion. The reason is that the Monitoring Outcome (MO_{jt}) of a particular quality of service criteria j in the period t in the SLA contract can be modeled as shown in Equation (2), and therefore it is a binary event. Consequently, the beta density function, which is shown in Equation (3), can be efficiently used to calculate posteriori probabilities of the event. As a result, the mean or expected value of the distribution can be represented by Equation (4).

$$MO_{jt} = \{SLAnotviolated, SLAviolated\} \tag{2}$$

$$\begin{gathered} f(x|\rho,\tau) = \frac{\Gamma(\rho+\tau)}{\Gamma(\rho)\Gamma(\tau)} x^{\rho-1}(1-x)^{\tau-1} \\ where \quad 0 \le x \ge 1, \rho < 0, \tau > 0, \\ \text{and } \rho \text{ , and } \tau \text{ are beta distribution parameters} \end{gathered} \tag{3}$$

$$\mu = E(x) = \rho / (\rho + \tau) \tag{4}$$

As mentioned earlier in Section 5, in our architecture a component is responsible for monitoring SLA contracts. If we assume that the monitoring component has detected that SLA violation occurred v times for provider p (for the total number of n monitored SLAs). Considering that $p = n - v + 1$ $\tau = v + 1$ and, the SCL is equal to the probability expectation that SLA is not going to be violated and is calculated as shown in Equation (5).

$$SCL_j = \frac{n - v + 1}{n + 2} \tag{5}$$

We modeled availability for SCL generation, as current cloud providers only include availability in their SLAs. The reliability in our work is considered as a user constraint for each cloud service.

2. **Cost:** Cost is a non-functional requirement of a user who wants to deploy a network of appliances. In our problem, minimization of deployment costs is considered as the objective of users. The deployment cost includes monetary cost of leasing virtual units as well as appliances and communication costs. The communication monetary cost for connected virtual appliances depends on how much data they exchange and can be determined by the following factors: 1) One time communication message size and 2) Communication rate (how often two appliances communicate), which can be calculated based on request inter-arrival rate.
   In this work, we focused on computing and data transfer cost. However, a comprehensive cost model can take into consideration many other forms of costs including:
   - Storage: This includes replication and backup cost and can vary based on number of reads and writes operation.
   - Content Delivery Network (CDN): The CDN cost is generally calculated based on per GB of data transferred through CDN edges and the geographical location of edges.
   - Load balancing: This cost is also calculated based on volume of data transferred through the load balancer.
   - Monitoring: The monitoring cost grows based on number of instance and monitoring frequency.
3. **Latency:** Latency can have a significant impact on e-Business web sites performance and consequently on the end user experience. Therefore, we have considered it in the problem as one of the users' constraints. It is assumed that customers have different constraints for the latency between appliances that have to be satisfied with the selection of proper cloud providers.

7.2. *Deployment Problem Formulation*

7.2.1. Provider model Let m be the total number of providers. Each provider is represented in Equation (6).

$$P_k : (\{a\}, \{vm\}, Cdata_{internal}(Pk), Cdata_{in}(Pk), Cdata_{out}(Pk)) \tag{6}$$

Where a, vm, $Cdata_{internal}(Pk)$, $Cdata_{in}(Pk)$, and $Cdata_{out}(Pk)$ denotes appliance, virtual machine, cost of internal data transfer and cost of external data transfer to and from cloud respectively. A virtual appliance a can be represented by a tuple of four elements: appliance type, cost, license type, and size as represented in Equation (7).

$$a : \{ApplianceType; Cost; LicenseType; Size\} \tag{7}$$

A virtual machine vm can be formally described as a tuple with two elements as shown in Equation (8).

$$vm : \{MachineType; Cost\} \tag{8}$$

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

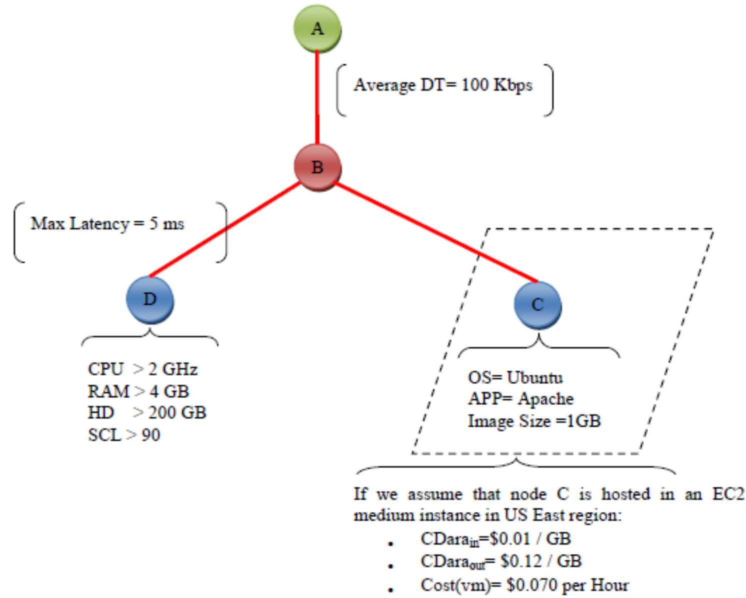


Figure 7. An Example of Request Graph.

7.2.2. User request model The user request for deployment of his application can be translated into a graph $G(V,E)$ where each vertex represents a server (virtual appliance running on a virtual unit). Server corresponding to a vertex v is represented in Equation (9).

$$S_v = \{appliance, virtualunit\} = \{a_v, vm_v\}, \forall v \in V \tag{9}$$

Each edge $e\{v,v'\}$ indicates that vertex v and v' are connected. The data transfer between these connected vertices (i.e., one server to another) is given by "*DSize*". An example of a user request (for 3 nodes) with its major attributes is illustrated in Figure 7. The objective of a user is to minimize the deployment cost of his whole application on multiple cloud providers' infrastructures, given a lease period of T and budget B. Users have constraint for reliability (SCL_v) of the provider on which server should be hosted and also latency constraint $(L(e\{v,v'\}) where \quad v,v' \in V)$ that represents the maximum acceptable latency between servers. The cost of renting a server includes the cost of virtual unit and virtual appliance.

Let an appliance for S_v be rented from provider P_k and a virtual unit from provider P_l. The cost of server S_v as shown in Equation (10) is the cost of the appliance $(cost(a_{v,p_k}))$ and virtual unit $(cost(vm_{v,p_l}))$ plus cost of transferring the appliance if the appliance and virtual unit providers are not the same.

$$Cost(S_v) = \begin{cases} \left(Cost\left(a_{v,P_k}\right) + Cost\left(vm_{v,P_l}\right)\right) \times T & \text{if } k = l; \\ \left(Cost\left(a_{v,P_k}\right) + Cost\left(vm_{v,P_l}\right)\right) \times T + Size\left(a_{v,P_k}\right) * \\ Cdata_{out}\left(P_l\right) \quad \text{if } k \neq 1. \end{cases} \tag{10}$$

Let $S_v = \{a_{v,Pk}, vm_{v,pl}\}$ and $S'_v = \{a_{v',Pk'}, vm_{v',pl'}\}$ be two connected vertices (servers) by edge $e\{v,v'\} \in E$; and P_k , P_l, $P_{k'}$ and $P_{l'}$ are the providers using whose resources Servers S_v and $S_{v'}$ are deployed. The data transfer cost between the two servers is given by Equation (11). The data transfer between connected vertices can be measured from the current production environment (before migrating to cloud data centers) as shown in [28]. Alternatively, this data can be collected via emulation of user's inputs and mouse clicks.

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

$$DCost\left(e\left\{v,v'\right\}\right)=\begin{cases} DSize\left(e\right)\times\left(CData_{out}\left(p_l\right)+CData_{in}\left(p_{l'}\right)\right)\times T \\ \text{if } l\neq l' \\ DSize\left(e\right)\times CData_{internal}\left(p_l\right)\times T \\ \text{if } l=l' \end{cases} \tag{11}$$

where $CData_{in}$ counts for the cost of data transferred to a cloud provider; $CData_{out}$ is the cost of data transferred out of cloud provider (refer to node C in Figure 7); and $CData_{internal}$ stands for cost of internal data transfer.

Therefore, the total cost of hosting users application on the multiple clouds is given by Equation (12).

$$TC=\sum_{v\in V}Cost\left(S_v\right)+\sum_{\substack{e\in E\\ v,v'\in V}}DCost\left(e\left\{v,v'\right\}\right) \tag{12}$$

7.2.3. Deployment Optimization Objectives The objective of the user is to minimize the deployment cost of his whole application on multiple cloud infrastructures $(P_k \quad 0<k<m)$. Thus, the mathematical model is given by Equations (13), (14) , and (15).

$$Min\left(TC\right)Subject \quad to \quad 0<TC<B \tag{13}$$

$$for\,all\; e\left\{v,v'\right\}\; in\, E: \quad Latency\left(S_v,S_{v'}\right)<L\left(e\left\{v,v'\right\}\right) \tag{14}$$

$$for\,all\; v\; in\, V: \quad SCL\left(S_v\right)>SCL_v \tag{15}$$

Where, $Latency\left(S_v,S_{v'}\right)$is the latency between cloud infrastructures where server S_v and $S_{v'}$ are hosted, and $SCL(S_v)$ is the reliability of the cloud infrastructure where server S_v is hosted.

7.3. Deployment Optimization Algorithms

To tackle the aforementioned problem, one may consider a greedy selection algorithm [42]. By greedy selection algorithm, we mean a simple heuristic approach that for each node, the cloud service candidate that offers the highest score compared to the other candidates is selected. With this approach, it is not possible to consider a user's constraints which is applied to the whole service composition (such as budget) or even latency constraints between vertices. Another approach which can be used to solve the problem is finding all possible compositions using exhaustive search, comparing their overall cost, and selecting the composition with the lowest cost that satisfies budget, reliability, and latency constraints. This approach can find the optimal solution; however, the computation cost of the algorithm is high due to NP hardness of the problem [42]. In order to deal with the aforementioned challenges in following we describe two algorithms: Forward-checking-based backtracking (FCBB) and the genetic-based cloud virtual appliance deployment optimization.

7.3.1. Forward-checking- based -backtracking (FCBB) In FCBB, the process of searching providers begins from a start node (vertex) S_v which has minimum deployment cost (including appliance and virtual unit cost) and for all its children there can be found at least one provider that satisfies all constraints (partial forward checking) [Algorithm 2 lines:12-14]. The partial forward checking on the problem constraints is added to the algorithm to avoid back jumps in the circumstances where latency constraints of the users are comparatively tight.

Then, S_v is added to the processed node list. After that, the algorithm processes all the children of S_v which are not processed, and for each child of $S_{v'}$, providers are selected using the selection function [Algorithm 1 lines:8-11] such that 1) latency and SCL constraints are satisfied with all the connected processed nodes (backward checking), 2) they can pass forward checking and 3) they have minimum communication (to already processed nodes) and combination cost [Algorithm 2 lines:16-19]. After selection of all the unprocessed children of the start node $S_{v'}$, the similar search and selection process is applied recursively for all the grand children of start node S_v [Algorithm 1

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Algorithm 1: FCBB

```
Input: S_v, RequestG(V,E)
Output: selected[]
1  if S_v = theFirstStartNode then
2  |  S_v ← getStartNode(RequestG(V,E), processedSet);
3  |  processedSet ← processedSet ∪ S_v ;
4  |  selected[S_v] ← selection(S_v);
5  |  if selection(S_v') = null then
6  |  |_ backtrack;
   |_
7  connectedNotProcessed ←
   getConnectedNotProcessed(parentNode, RequestG(V,E), processedSet);
8  foreach S_v' in connectedNotProcessed do
9  |  selected[S_v'] ← selection(S_v');
10 |  if selection(S_v')=null then
11 |  |_ backtrack;
   |_
12 foreach S_v' in connectedNotProcessed do
13 |_ FCBB(S_v');
```

lines: 12-13]. If the selection function does not find any set of providers, it moves back and replaces the parent node with the second best set of providers in the *Combination* list (Backtrack) [Algorithm 1 lines: 7 and 11].

7.3.2. Genetic-Algorithm based Virtual Unit and Appliance Provider Selection Since genetic approaches have shown potential for solving optimization problems [43], this class of search strategies was utilized in our problem. The adoption of genetic-based approaches for the deployment problem involves 4 steps.

The *first* step is to plan the chromosome, which consists of multiple genes. In our problem, each vertex in the graph of request is represented by a gene. The *second* step is to create the population, hence each gene represents a value which points to a combination of virtual unit and appliance service (which satisfies requirements of corresponding vertex) in a sorted (based on the combination cost) list. Implementation of fitness function is the *third* step. The fitness values are then used in a process of natural selection to choose which potential solutions will continue on to the next generation, and which will die out. The fitness function as shown in Equation 16 is equal to the total cost of the solution. However, if constraints are violated, the penalty function is applied.

Designing penalty function for genetic-based approach is not a trivial task. Several forms of penalty functions have been proposed in literature [44], including rejection of infeasible solutions or giving the death penalty. However, those solutions could make the search ineffective when the feasible optimal solutions are close to infeasible solutions. For our problem, the penalty function is constructed as a function of the sum of the number of violations for each constraint multiplied by constants as shown in Equation 17. In the penalty function, *Age* is the age of chromosome, *ki* constant, *NVi* is number of cases that violates the constraints, and *NNVi* is the number of cases that do not violate the constraints. In addition, to discard the infeasible solutions in early generations (for our case where we have adequate sampling of the search space), infeasible solutions with lower age are penalized heavier. We realized that using a modest penalty in the early stages, although ensures larger sampling, leads to infeasible solutions more frequently. Finally, the last step is the evolution of the population based on the genetic operator. The genetic operator adopted for our work is the Java Genetic Algorithm Package (JGAP) natural selector [35].

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

Algorithm 2: Selection

```
Input: S_v
Output: selectedCombination
1  minCost ← ∞; constraintsViolated ← false; feasible ← true; selectedCombination ← null;
2  foreach combination in getAllCombinationSorted(S_v) do
3      ▷ getAllCombinationSorted returns combinations sorted using quick sort.
4      if SCL(S_v) < SCL(combination.getVUProvider()) and
       SCL(S_v) < SCL(combination.getAppProvider()) then
5          connectedProcessed ←
           getConnectedProcessed(startNode, RequestG(V,E), processedSet);
6          if connectedProcessed = null then
7              foreach S_v' in connectedProcessed do
8                  if Latency(S_v, s_v') > L(e{S_v, s_v'}) then
9                      constraintsViolated ← true;
10             if constraintsViolated = false then
11                 connectedNotProcessed ←
                   getConnectedNotProcessed(startNode, RequestG(V,E), processedSet);
12                 foreach s_v' in connectedNotProcessed do
13                     if ∉ combination in getAllCombinationSorted(S_v) that
                       Latency(S_v, s_v') > L(e{S_v, s_v'}) then
14                         feasible ← false          ▷ Forward Checking;
15                 if feasible = true then
16                     cost ← communicationCost + combination.getCost();
17                     if cost < minCost and cost + totalCost < request.getBudget() then
18                         minCost ← cost;
19                         selectedCombination ←
                           {combination.getVUProvider(), combination.getAppProvider()};
20 return selectedCombination;
```

$$fitness = \begin{cases} \left(\sum_{i\in V} Cost\left(Gene_i\right) + \sum_{\substack{e\in E\\ i,j\in V}} DCost\left(e\left\{Gene_i, Gene\,j\right\}\right)\right) * T & \text{if constarints are not viloated} \\ \left(\sum_{i\in V} Cost\left(Gene_i\right) + \sum_{\substack{e\in E\\ i,j\in V}} DCost\left(e\left\{Gene_i, Gene\,j\right\}\right)\right) * T + Penalty\left(\right) & \text{if constraints are violated} \end{cases} \tag{16}$$

$$Penalty\left(\right) = \sum_{i=1}^{n} \left(\frac{NVi}{NVi + NNVi} \times ki\right) \times \left(\frac{1}{Age}\right) \times fitnessvalue \tag{17}$$

7.3.3. Additional issues One may require the optimization algorithm to take into account latency constraints between end users (in different geographical locations) and particular servers. This can be easily modelled by considering users a special case of server that has latency requirements but no software, hardware, and reliability requirements.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Table I. Latency between clouds and SCL input data.

Cloud A	Cloud B	Latency (ms)	Cloud B Monitored Availability	Cloud B Promised Availability
Ec2	Rackspace	49.8	99.996%	100%
Ec2	GoGrid	8.9	99.996%	100%
Ec2	Lindoe	5.01	99.996%	100%

Table II. Request types.

Request Type	Request Graph Density	Request Inter Arrival Rate DB $\leftrightarrow$ AS	Request Inter Arrival Rate WS $\leftrightarrow$ As
Strongly connected	0.85	Log-normal (1.4719,2.2075)	Weibull (0.70906,10.185)
Moderately connected	0.5	Log-normal (1.1695,1.9439)	Weibull (0.41371,1.1264)
Poorly connected	0.25	Log-normal (0.8912,1.6770)	Weibull (0.24606,0.03548)

8. EXPERIMENTAL TESTBED CONSTRUCTION AND PERFORMANCE EVALUATION

To evaluate the proposed algorithms and study the placement of appliances, essential input data using real experiments was collected. The collected data can be classified either as data for providers modeling or data for user request modeling.

1. Providers modeling: A set of 12 real cloud providers are selected, namely: Amazon, Zerigo, Softlayer, VMware, Bitnami, rpath, Turnkeylinux, Rackspace, GoGrid, ReliaCloud, Lindoe, and Prgmr. Their virtual units and appliances have been modeled in our system. In addition, latency data between cloud providers and SCL for each of them have been measured. The following subsections describe the data collected in detail.
2. Virtual unit and appliance modeling: We built an aggregated repository of virtual appliances and virtual unit services based on the advertised services by cloud providers. Services contain information regarding cost, virtual appliance size, and data communication cost inside and outside of clouds.
3. Latency and reliability (SCL) calculation: We first setup testing nodes in 12 different infrastructure/server clouds as mentioned earlier. Next, the nodes initiate latency network tests (hourly) with each of the other nodes that are placed in other infrastructures. This includes pinging to other nodes to determine latency. For the experiment purpose, we calculate the mean from all of these tests that ran for three months. Table I shows mean latency between EC2 and 3 different virtual unit providers as an example. From the collected data, we can identify which clouds are best connected. For example, EC2 is best connected with Lindoe and GoGrid. Max, min and average of latency between providers are 58.94, 2.51 and 29.88 (ms) respectively. However, the real implementation of CloudPick uses Cloud Harmony RESTful API, which provides real-time latency information among more than 30 infrastructure providers. In addition, Panopta (a monitoring tool) is used to supply SCL input data. Table I demonstrates how a sample of SCL input data looks like for 3 cloud Providers for a 365 days period.

8.1. *Generation of requests for experiments*

The request generation involves three steps. Firstly, number of servers requested by the user and requirements of each server in terms of virtual unit and appliance types are determined. Next,

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

connected vertices in the request are identified. Finally, data transfer rates between connected appliances are identified. For experimental evaluation two classes of requests are used, i.e., a real case study and randomly generated requests.

1. **Modeling user requests using a real case study** For the real case study example, we use the three-tier data centre scenario presented by Ersoz et al. [28]. The required virtual units and appliance types for each vertex is assigned based on the scenario. They implemented an e-Business web site that encompasses 11 appliances: 2 front-end web-servers (WS) in its web tier, 3 databases (DB) in its database tier, and 6 application servers (AS) in between. In their work, a three-tier data centre architecture was used to collect the network load between appliances. Two different workloads, RUBiS [45] and SPECjApp-Server2004, are used by them. However, our focus is on the RUBiS, which implements an e-Business web site. That web site includes 27 interactions that can be carried out from a client browser. Their analysis of experiments results has been represented by various distributions of request inter-arrival times, and data size between tiers for 15 minutes runs of the RUBiS workload with 800, 1600, and 3200 clients. This data, which is shown in Table II, is used to calculate the network traffic between connected appliances.
2. **Modeling user requests for extensive experiment study** Three classes of user requests (network of appliances) namely strongly, moderately, and poorly connected are created as shown in Table II, which differs from each other in communicated message sizes, message inter-arrival rates, and graph density (proportion of the number of edges in request graph to total possible number of edges) of the request graph. The reason for building 3 classes of requests is to study the effect of network traffic and request graph density on performance of algorithms and placement of appliances. It is worth mentioning that having variations in graph density, latency constraint, and data transfer rate can examine how effectively an algorithm can handle budget and latency constraints in different circumstances. For each vertex, we randomly assign a required virtual unit and appliance type, and then we use random graph generation technique to identify which vertices are connected. All generated network of appliances follow the topology presented by Ersoz et al. [28]. Based on appliances that are connected to each other, data transfer rates are assigned. For example, if one appliance is a database and the other one is an application server and the request is in category of strongly connected, then the request inter-arrival rate is Log-normal(1.4719, 2.2075). In addition, to investigate effects of message size, two classes of requests with different message sizes are created using workload "a" [28] (e-Business application with small message size) and "b" [46] (98 World cup with large message size).

8.2. *Experimental results*

The experiments aim at:

1. Evaluating the performance of the translation approach to find out how effectively it can build an aggregated semantically-enriched service repository for a multi-cloud environment;
2. comparing the proposed heuristics with Exhaustive Search (ES) using the real case study to determine how effectively CloudPick can satisfy user requirements;
3. evaluating effects of variation in request types on algorithms performance;
4. analyzing effects of variation in request types and constraints on deployment cost and distribution factor which shows how users' applications distributed across multiple cloud; and
5. investigating the effects of number of iterations an population size on the performance of the genetic algorithm to tune the optimizer component of CloudPick.

8.2.1. Performance of translation approach for different sizes of Cloud service repositories Major cloud providers have large repository of virtual appliance and unit services. For example, Amazon Web Service's repository's size alone is greater than 10.6 MB. To increase the efficiency of CloudPick we only perform synchronization when the translation service is triggered by the integrity

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

checking component. We increased the number of services in the repository by merging repositories from various cloud providers to investigate the scalability of our approach in terms of execution time needed for the translation. For each case of repository size, we repeated the experiment 30 times and the results are plotted in Figure 8. Regression analysis shows that there is positive and linear relationship between the repository size and the translation time. The evidence confirms that the regression coefficient is 0.6621, which suggests that if the data size to be translated increases by 1 MB, translation time increases roughly by 0.6 second. Consequently, synchronization function can be executed online in an acceptable time even if a considerable percentage of virtual appliance and unit properties is updated.

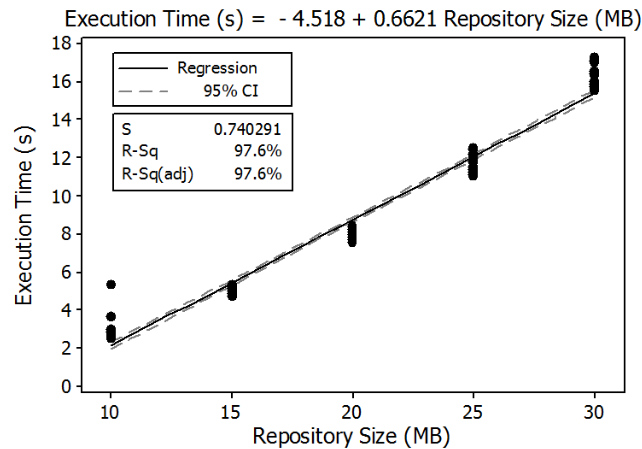


Figure 8. Execution time of translation for different repository sizes.

8.2.2. Comparison with Exhaustive Search (ES) Figuer 9 shows how close the proposed algorithms are to the Exhaustive Search (ES) for the case study. Both of them could reach the same solution achieved with ES. As evidenced by Table III, the mean execution time for finding the solution using exhaustive search of the solution space is extremely high comparing to our proposed algorithms. The execution time for the ES approach rises further exponentially as well as the computational effort for larger number of servers and providers. Therefore, it cannot be considered as a practical solution for the problem. To further examine the near-optimality of FCBB and the genetic approach, we conducted experiments with 10 different requests (in terms of service requirements, graph density, message size, and request inter-arrival time) for each category of 10, 15, and 20 servers. The results are shown in Table IV, where we observe that on average, the difference in deployment cost compared with ES is 7% for the FCBB and 1% for genetic approach. Therefore, both FCBB and genetic approach can reach a near-optimal solution without much computational cost.

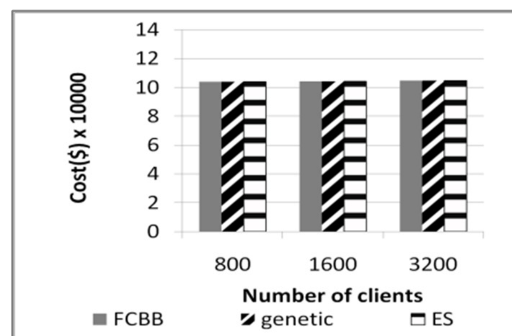


Figure 9. Performance Evaluation for Case Study.

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table III. Mean execution time for case study.

Algorithm	Mean Execution time(s)
FCBB	0.102
genetic	36.393
Exhaustive Search (ES)	3248.152

Table IV. Mean exhaustive search(es) costs/algorithms costs.

Algorithm	Number of servers		
	10	**15**	**20**
ES/FCBB	0.9841	0.9175	0.9013
ES/genetic	0.9952	0.9868	0.9923

Table V. Mean execution time (s).

Algorithm	Number of servers				
	10	**25**	**50**	**75**	**100**
FCBB	0.103	0.115	0.288	0.407	0.841
Discard subset	0.138	0.271	0.849	2.339	6.091
genetic	31.997	144.426	497.377	1288.056	1814.488

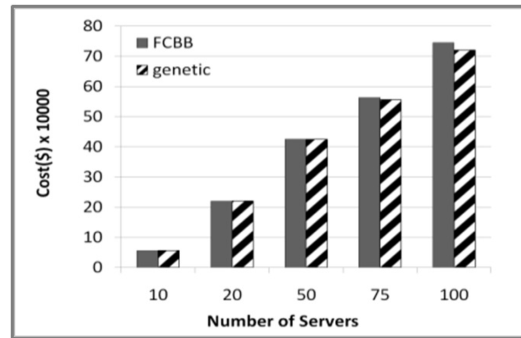
8.2.3. Impacts of variation in request types on algorithms performance and execution time Figures 10 and 11 depict the performance of the proposed algorithms for different request types (strongly, moderately, and loosely connected) with different number of servers. These experiments particularly examine the efficiency of CloudPick for applications with different workloads. Moreover, by varying the number of servers, the experiments investigate and compare the scalability of FCBB and the genetic algorithm. In the case of workload "a", as the message size is small, differences are comparatively small, except for strongly connected requests (Figure 10a and especially for the case of 100 servers where genetic-based approach can save approximately 3% of cost. In other cases of workload "a" when vertices are moderately or poorly connected, the genetic-based approach has better or relatively same performance (regarding the cost) compared to the FCBB algorithm. However, when the message size is larger (workload "b"), as shown in Figure 11a, the genetic algorithm in almost all cases outperforms the FCBB algorithm. In Table V, mean execution times for 20 experiments in relation to the number of servers for groups of requests are given. It shows that the execution time of FCBB is negligible compare to genetic's one. It also shows that adding "forwards checking" feature successfully decreases execution time, especially for the requests which require more than 10 servers and therefore it outperform the "discard subset" algorithm proposed in [42] (the algorithm proposed to solve the web service composition optimization problem with multiple constraints) regarding the execution time while they both could result in the same objective values for all cases.

Therefore, the performance of the algorithms differs from one workload to another and when there exists a workload with small message size (like the e-Business workload "a"), performance difference of algorithms is low. In such cases, FCBB can be used to save on execution time. However, when the message size increase, they show comparatively higher differences. As a result, when users look for minimizing cost instead of the execution time, the genetic-based approach is the most appropriate solution.

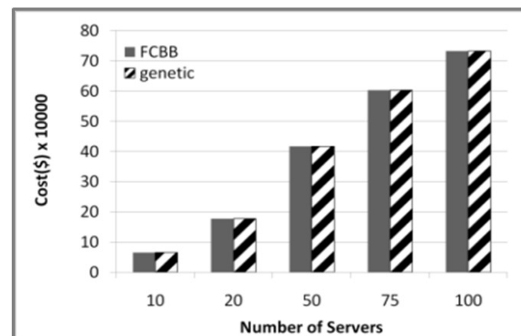
8.2.4. Effects of variation in request types and latency constraints on distribution factor in multi-cloud environments In this experiment, the objective is to study the possibility of placing a network of appliances on different providers rather than one in a multi-cloud environments when the only concerns are latency and deployment cost. For this purpose, a metric named "distribution factor"

Copyright © 2014 John Wiley & Sons, Ltd.

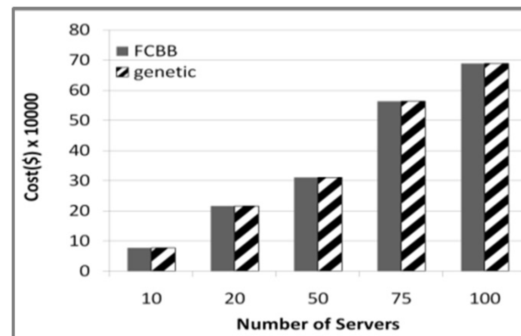
DOI: 10.1002/spe



(a) Strongly connected



(b) Moderately Connected



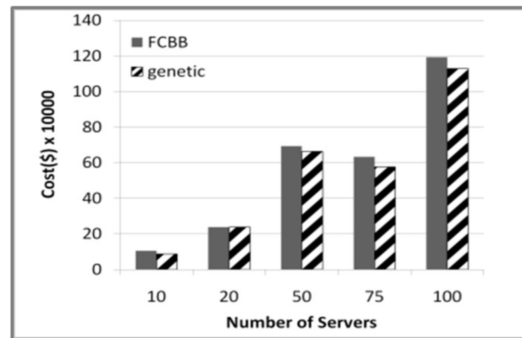
(c) Loosely Connected

Figure 10. Change in Connectivity for Workload a.

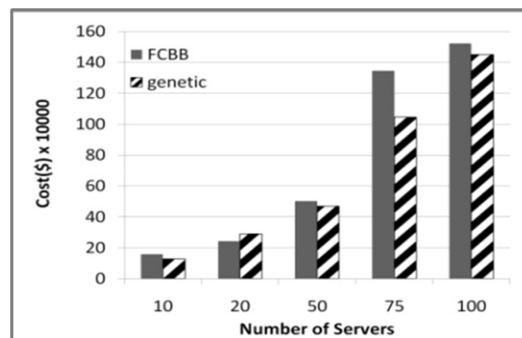
is designed, which shows the proportion of the number of different providers selected to the total number of providers. Table VI shows how a request type (data transfer rate and graph density, as explained in Table II) affects the distribution factor. For the loosely connected requests with loose latency requirement, we conclude that considering multiple cloud providers decreases the deployment cost while still maintaining the minimum performance requirements (by adhering to the latency constraint). For all cases from 10 to 100 servers, when there is a higher data transfer and number of connection between vertices, the distribution factor decrease dramatically. For the majority of cases, it decreases by more than 75%. It means that FCBB selection algorithms have a tendency to select the same virtual unit provider for all vertices to save on communication cost. The same trend can be observed for the genetic-based approach. However, when the latency constraints are tight, if we consider multiple providers for deployment, the cost will be lower. But still the distribution factor decreases by 25%. Consequently, the experiments show that network

Copyright © 2014 John Wiley & Sons, Ltd.

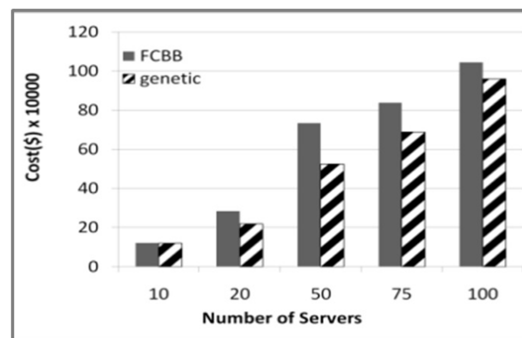
Prepared using speauth.cls
DOI: 10.1002/spe



(a) Strongly connected



(b) Moderately Connected



(c) Loosely Connected

Figure 11. Change in Connectivity for Workload b.

of appliances with higher graph densities and data transfer are less likely to be distributed across multiple providers and they are expected to have higher deployment cost.

8.2.5. Effects of variation of reliability constraints on deployment cost This experiment is designed to help us understand how characteristics of network of appliances affect the deployment cost when they are migrated to the cloud. As illustrated in Table VII, the deployment cost increases by almost 10% on average when latency requirement is tighter as less number of providers could satisfy such requirement (lower distribution factor). In addition, demanding providers with higher reliability slightly increases the cost of deployment, which is less than the increase in the case when the latency constraint is tighter.

8.2.6. Varying iteration number and population size Figure 12 and 13 represent the effects of increasing the number of iterations and population size on improvement of the objective function.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

Table VI. Distribution factor.

Request type	**Number of servers**				
	10	**25**	**50**	**75**	**100**
Loosely connected & loose latency	44%	55%	55%	55%	44%
Strongly connected	11%	11%	11%	11%	11%
Tight latency	22%	44%	33%	33%	33%

Table VII. Effects of the deployment constraints on the cost.

Request type	**Average percentage of cost increase**
High reliability	5.8117414
Tight latency	10.1966957

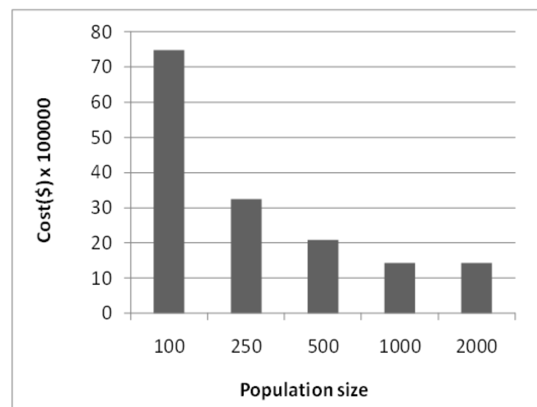


Figure 12. Population size versus cost.

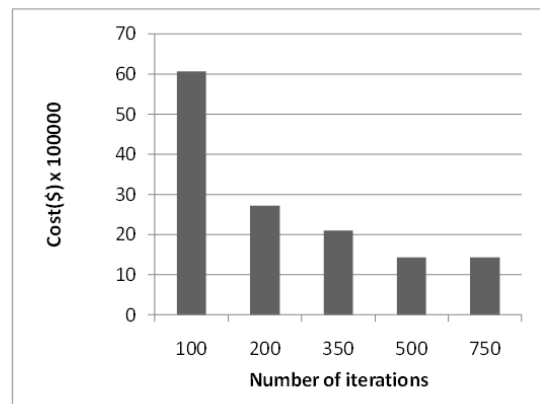


Figure 13. Number of iteration versus cost.

The examined request has 100 highly connected vertices from workload "b". The aim is to show to what extent increasing the number of iterations and population size improves performance of the genetic approach. It can be observed that increasing the number of iterations and population size contribute to the objective function. However, from a certain point (for population size 1000 and for iteration number 500), the improvement is marginal and negligible.

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

9. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we proposed a framework called CloudPick to simplify the process of service deployment in multiple cloud environments and mainly focused on its cross-cloud service modeling and deployment optimization capabilities. We investigated the cloud provider selection problem for deploying a network of appliances. We proposed new QoS criteria, and the problem of deployment is formulated and tackled by two approaches namely FCBB and genetic-based selection. We evaluated the proposed approaches by a real case study using real data collected from 12 cloud providers, which showed that the proposed approaches deliver near-optimal solution. Next, they were tested with different types of requests. The results show that when message size increases, approaches present comparatively higher differences, and if execution time is not the main concern of users, genetic-based selection in most cases achieves better value for the objective function. In contrast, if the massage size between appliances is small, FCBB can be used to save on execution time. Further, based on the conducted experiments, we found out that network of appliances with higher graph density and data transfer are less likely (in contrast to requests with lower data transfer) to be distributed across multiple providers. However, for requests with tight latency requirements, appliances are still placed across multiple providers to save on deployment cost. Further, we show how the iteration number and population size affect the performance of the genetic algorithm. And finally, the performance of the translation approach was measured for different repository sizes which demonstrates its scalability.

Future work will focus on identifying challenges in designing cross-cloud scaling policies when users have budget, deployment time, and latency constraints. We will further investigate the scaling optimization algorithm that not only minimizes the cost but also has the current knowledge of virtual appliance placement (inter-cloud latency and throughput) and maximizes the performance metrics such as end user response time. Another promising research topic is discovering and selecting resources for back up and then a deployment pattern that facilitates the recovery in a speedy and cost-optimal manner when failure happens.

By emergence of spot instances in cloud computing, IaaS providers like Amazon offer their virtual unit services with dynamic pricing. Therefore, the cross-cloud deployment optimization can investigate approaches for bidding and market selection that minimizes the deployment cost. In addition, as number of cloud services offered by IaaS are increasing, future research can investigate more detailed cost model to enhance the accuracy of provider selection algorithms.

Acknowledgments The authors wish to thank Rodrigo N. Calheiros and Kurt Vanmechelen for their constructive and helpful suggestions.

REFERENCES

1. Narasimhan B, Nichols R. State of cloud applications and platforms: The cloud adopters' view. *Computer* 2011; **44**(3):24–28.
2. Google. Google app engine. `http://code.google.com/appengine/`.
3. Varia J. Best practices in architecting cloud applications in the aws cloud. *Cloud Computing: Principles and Paradigms, Wiley Press, New Jersey, USA* 2011; :459–490.
4. Sun C, He QWL, Willenborg R. Simplifying service deployment with virtual appliances. *Proceedings of the 2008 IEEE International Conference on Services Computing*, 2008; 265–272.
5. Sapuntzakis C, Brumley D, Chandra R, Zeldovich N, Chow J, Lam M, Rosenblum M. Virtual appliances for deploying and maintaining software. *Proceedings of the 17th USENIX conference on System administration*, 2003; 181–194.
6. Dastjerdi AV, Garg S, Buyya R. Qos-aware deployment of network of virtual appliances across multiple clouds. *2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2011; 415–423.
7. Fensel D, Facca F, Simperl E, Toma I. Web service modeling ontology. *Semantic Web Services* 2011; :107–129.
8. De Bruijn J, Lausen H, Polleres A, Fensel D. The web service modeling language wsml: An overview. *in Proceedings of the 3rd European conference on The Semantic Web: research and applications (ESWC06)* 2006; :590–604.
9. Haller A, Cimpian E, Mocan A, Oren E, Bussler C. Wsmx-a semantic service-oriented architecture. *Proceedings of the IEEE International Conference on Web Services (ICWS)*, IEEE, 2005; 321–328.
10. VMWare. Virtual appliance marketplace. `http://www.vmware.com/appliances/`.

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

11. Keahey K, Freeman T. Contextualization: Providing one-click virtual clusters. *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, IEEE, 2008; 301–308.
12. Wang X, Du Z, Chen Y, Li S, Lan D, Wang G, Chen Y. An autonomic provisioning framework for outsourcing data center based on virtual appliances. *Cluster Computing* 2008; **11**(3):229–245.
13. Dastjerdi AV, Tabatabaei SG, Buyya R. An effective architecture for automated appliance management system applying Ontology-Based cloud discovery. *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010; 104–112.
14. Konstantinou A, Eilam T, Kalantar M, Totok A, Arnold W, Snible E. An architecture for virtual solution composition and deployment in infrastructure clouds. *Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, ACM, 2009; 9–18.
15. Chieu T, Mohindra A, Karve A, Segal A. Solution-based deployment of complex application services on a cloud. *Service Operations and Logistics and Informatics (SOLI), 2010 IEEE International Conference on*, IEEE, 2010; 282–287.
16. Foster I, Zhao Y, Raicu I, Lu S. Cloud computing and grid computing 360-degree compared. *Grid Computing Environments Workshop, 2008. GCE'08*, Ieee, 2008; 1–10.
17. Wang X, Yue K, Huang J, Zhou A. Service selection in dynamic demand-driven web services. *Web Services, 2004. Proceedings. IEEE International Conference on*, IEEE, 2004; 376–383.
18. Guha T, Ludwig S. Comparison of service selection algorithms for grid services: Multiple objective particle swarm optimization and constraint satisfaction based service selection. *Tools with Artificial Intelligence, 2008. ICTAI'08. 20th IEEE International Conference on*, vol. 1, IEEE, 2008; 172–179.
19. Ludwig S, Reyhani S. Selection algorithm for grid services based on a quality of service metric. *High Performance Computing Systems and Applications, 2007. HPCS 2007. 21st International Symposium on*, IEEE, 2007; 13–13.
20. DMTF. Open virtualization format. `http://www.dmtf.org/standards/ovf`.
21. García J, Ruiz D, Ruiz-Cortés A, Parejo J. Qos-aware semantic service selection: An optimization problem. *IEEE Congress on Services-Part I*, IEEE, 2008.
22. Rao J, Su X. A survey of automated web service composition methods. *Semantic Web Services and Web Process Composition* 2005; :43–54.
23. Kiran M, Jiang M, Armstrong D, Djemame K. Towards a service lifecycle based methodology for risk assessment in cloud computing. *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, IEEE, 2011; 449–456.
24. Tran V, Tsuji H, Masuda R. A new qos ontology and its qos-based ranking algorithm for web services. *Simulation Modelling Practice and Theory* 2009; **17**(8):1378–1398.
25. Di Martino B, Petcu D, Cossu R, Goncalves P, Máhr T, Loichate M. Building a mosaic of clouds. *Euro-Par 2010 Parallel Processing Workshops*, Springer, 2011; 571–578.
26. Carlini E, Coppola M, Dazzi P, Ricci L, Righetti G. Cloud federations in contrail. *Euro-Par 2011: Parallel Processing Workshops*, Springer, 2012; 159–168.
27. Menzel M, Ranjan R. Cloudgenius: decision support for web server cloud migration. *Proceedings of the 21st international conference on World Wide Web*, ACM, 2012; 979–988.
28. Ersoz D, Yousif M, Das C. Characterizing network traffic in a cluster-based, multi-tier data center. *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, IEEE, 2007; 59–59.
29. Faratin P. Automated service negotiation between autonomous computational agents. PhD Thesis, University of London 2000.
30. Dastjerdi AV, Buyya R. An autonomous reliability-aware negotiation strategy for cloud computing environments. *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, IEEE, 2012; 284–291.
31. Dastjerdi AV, Tabatabaei S, Buyya R. A dependency-aware ontology-based approach for deploying service level agreement monitoring services in cloud. *Software: Practice and Experience* 2011; **42**(4):501–518.
32. Fielding RT, Taylor RN. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)* 2002; **2**(2):115–150.
33. Valdes M, Charoy F. Bonita: Workflow cooperative system. objectweb consortium 2004.
34. Keller U, Lara R, Lausen H, Polleres A, Predoiu L, Toma I. Wsmo discovery engine 2004.
35. Meffert K, Rotstan N, Knowles C, Sangiorgi U. Jgap-java genetic algorithms and genetic programming package. *URL: http://jgap. sf. net* 2008; .
36. Kritikos K, Plexousakis D. Semantic qos metric matching. *Web Services, 2006. ECOWS'06. 4th European Conference on*, IEEE, 2006; 265–274.
37. Kopeckỳ J, Moran M, Roman D, Mocan A. Wsmo grounding. *wsmo working draft v0. 1*, 2005. Available at: `http://www.wsmo.org/TR/d24/d24`.
38. Lambert D, Benn N, Domingue J. Integrating heterogeneous web service styles with flexible semantic web services groundings. *First International Future Enterprise Systems Workshop (FES2010) at The 3rd Future Internet Symposium (FIS2010) Berlin, Germany*, 20-22 Sept 2010.
39. Metsch T, Edmonds A, Nyrén R, Papaspyrou A. Open cloud computing interface–core. *Open Grid Forum, OCCI-WG, Specification Document. Available at: http://forge. gridforum. org/sf/go/doc16161*, 2010.
40. Chvatal V. A greedy heuristic for the set-covering problem. *Mathematics of operations research* 1979; **4**(3):233–235.
41. Jsang A, Ismail R. The beta reputation system. *Proceedings of the 15th bled electronic commerce conference*, 2002; 41–55.
42. Jaeger M, Rojec-Goldmann G. Seneca–simulation of algorithms for the selection of web services for compositions. *Technologies for E-Services* 2006; :84–97.
43. Coello C. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information systems* 1999; **1**(3):129–156.
44. Michalewicz Z. *Genetic algorithms+ data structures= evolution programs*. springer, 1996.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using speauth.cls

45. Cecchet E, Marguerite J, Zwaenepoel W. Performance and scalability of ejb applications. *ACM Sigplan Notices* 2002; **37**(11):246–261.
46. Arlitt M, Jin T. A workload characterization study of the 1998 world cup web site. *Network, IEEE* 2000; **14**(3):30–37.

A. PORTION OF DEVELOPED ONTOLOGY

```
1 wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
2 namespace { _"http://www.cloudslab.org/CloudProvider#"
3  }
4 ontology CloudProviderOntology
5 concept VMFormat
6 hasName ofType _string
7 concept place
8      hasName ofType _string
9 concept state subConceptOf place
10 concept country subConceptOf place
11 concept location
12 hasState ofType state
13 hasCountry ofType country
14
15 concept cloudService
16
17 concept monitoringMetric
18 hasName ofType _string
19 instance CPUUtilization memberOf monitoringMetric
20
21 concept loadBalancer subConceptOf cloudService
22 hasPort ofType _integer
23 hasProtocol ofType _string
24 isHealthcheckEnabled ofType _boolean
25 checkInterval ofType _integer
26 hasTimeOut ofType _integer
27 hasThreshold ofType _integer
28
29 concept virtualAppliance subConceptOf cloudService
30  _"http://www.CloudsLab.org/ontologies/VirtualAppliance#imageId" ofType
     _string
31
32              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                 imageLocation" ofType _string
33
34              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                 imageState" ofType _string
35
36              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                 imageOwnerId" ofType _string
37
38              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                 isPublic" ofType {_string, _boolean}
39
40              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                 architecture" ofType _string
41
42              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                 imageType" ofType _string
43
44              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                 kernelId" ofType _string
45
```

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

```
46              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    ramdiskId" ofType _string
47
48              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    platform" ofType _string
49
50              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageOwnerAlias" ofType _string
51
52              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#name"
                    ofType _string
53
54              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    description" ofType _string
55
56              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    rootDeviceType" ofType _string
57
58              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    rootDeviceName" ofType _string
59
60              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    virtualizationType" ofType _string
61
62              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    hypervisor" ofType _string
63
64              hasProvider ofType cloud
65              isCompatipleWith ofType virtualUnit
66              hasName ofType _string
67              hasFormat ofType VMFormat
68
69   instance  aki00806369  memberOf virtualAppliance
70              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#imageId
                    " hasValue "aki-00806369"
71              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageLocation" hasValue "karmic-kernel-zul/ubuntu-kernel
                    -2.6.31-300-ec2-i386-20091001-test-04.manifest.xml"
72              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageState" hasValue "available"
73              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageOwnerId" hasValue "099720109477"
74              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    isPublic" hasValue "true"
75              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    architecture" hasValue "i386"
76              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageType" hasValue "kernel"
77              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    rootDeviceType" hasValue "instance-store"
78              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    virtualizationType" hasValue "paravirtual"
79              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    hypervisor" hasValue "xen"
80              hasProvider hasValue {AmazonCalifornia}
81              hasName hasValue "aki00806369"
82              hasFormat hasValue AMI
83       instance aki00896a69 memberOf virtualAppliance
84              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#imageId
                    " hasValue "aki-00896a69"
```

Copyright © 2014 John Wiley & Sons, Ltd.

```
85              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageLocation" hasValue "karmic-kernel-zul/ubuntu-kernel
                    -2.6.31-300-ec2-i386-20091002-test-04.manifest.xml"
86              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageState" hasValue "available"
87              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageOwnerId" hasValue "099720109477"
88              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    isPublic" hasValue "true"
89              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    architecture" hasValue "i386"
90              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    imageType" hasValue "kernel"
91              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    rootDeviceType" hasValue "instance-store"
92              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    virtualizationType" hasValue "paravirtual"
93              _"http://www.CloudsLab.org/ontologies/VirtualAppliance#
                    hypervisor" hasValue "xen"
94              hasProvide hasValue {AmazonVirginia}
95              hasname hasValue    "aki00896a69"
96               hasFormat hasValue AMI
97 concept virtualUnit subConceptOf cloudService
98 hasName ofType _string
99 hasProvider ofType cloud
100
101 instance largeInstance memberOf virtualUnit
102 hasName hasValue "largeInstance"
103 hasProvider hasValue AmazonCalifornia
104
105
106 concept state subConceptOf place
107 concept country subConceptOf place
108
109 instance USA memberOf country
110 hasName hasValue "USA"
111 instance Singapor memberOf country
112 hasName hasValue "Singapor"
113 instance Canada memberOf country
114 hasName hasValue "Canada"
115 instance Ireland memberOf country
116 hasName hasValue "Ireland"
117 instance Brazil memberOf country
118 hasName hasValue "Brazil"
119 instance Japan memberOf country
120 hasName hasValue "Japan"
121 instance China memberOf country
122 hasName hasValue "China"
123 instance Australia memberOf country
124 hasName hasValue "Australia"
125 instance England memberOf country
126 hasName hasValue "England"
127
128 instance Toronto memberOf state
129 hasName hasValue "Toronto"
130 instance Tokyo memberOf state
131 hasName hasValue "Tokyo"
132 instance Dublin memberOf state
133 hasName hasValue "Dublin"
134 instance Virginia memberOf state
135 hasName hasValue  "Virginia"
```

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

```
136 instance Oregon memberOf state
137 hasName hasValue Oregon
138 instance California memberOf state
139 hasName hasValue "California"
140 instance Hongkong memberOf state
141 hasName hasValue "Hongkong"
142 instance Victoria memberOf state
143 hasName hasValue "Victoria"
144 instance London memberOf state
145 hasName hasValue "London"
146 instance Singapor memberOf state
147 hasName hasValue "Singapor"
148
149 concept cloud
150      hasName ofType _string
151 supportVmFormat ofType VMFormat
152 hasCountry ofType country
153 hasState ofType state
154
155 instance AMI memberOf VMFormat
156 hasName hasValue "AMI"
157
158 instance OVF memberOf VMFormat
159 hasName hasValue "OVF"
160
161 instance GSI memberOf VMFormat
162 hasName hasValue "GSI"
163
164 instance VMDK memberOf VMFormat
165 hasName hasValue "VMDK"
166
167 instance TerremarkCanada memberOf cloud
168 hasName hasValue "TerremarkCanada"
169 supportVmFormat hasValue {OVF, VMDK }
170 hasCountry hasValue Canada
171 hasState hasValue Toronto
172
173 instance TerremarkEngland memberOf cloud
174 hasName hasValue "TerremarkEngland"
175 supportVmFormat hasValue {OVF, VMDK }
176 hasCountry hasValue England
177 hasState hasValue London
178
179 instance TerremarkChina memberOf cloud
180 hasName hasValue "TerremarkChina"
181 supportVmFormat hasValue {OVF, VMDK }
182 hasCountry hasValue China
183 hasState hasValue Hongkong
184
185 instance TerremarkAuastralia memberOf cloud
186 hasName hasValue "TerremarkAustralia"
187 supportVmFormat hasValue {OVF, VMDK }
188 hasCountry hasValue Australia
189 hasState hasValue Victoria
190
191 instance AmazonVirginia memberOf cloud
192 hasName hasValue "AmazonVirginia"
193 supportVmFormat hasValue {AMI}
194 hasCountry hasValue USA
195 hasState hasValue Virginia
196
```

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

```
197 instance AmazonSingapor memberOf cloud
198 hasName hasValue "AmazonSingapor"
199 supportVmFormat hasValue {AMI}
200 hasCountry hasValue Singapor
201 hasState hasValue Singapor
202
203 instance AmazonIreland memberOf cloud
204 hasName hasValue "AmazonIreland"
205 supportVmFormat hasValue {AMI}
206 hasCountry hasValue Ireland
207 hasState hasValue Dublin
208
209 instance AmazonCalifornia memberOf cloud
210 hasName hasValue "AmazonCalifornia"
211 supportVmFormat hasValue {AMI}
212 hasCountry hasValue USA
213 hasState hasValue California
214
215 instance AmazonJapan memberOf cloud
216 hasName hasValue "AmazonJapan"
217 supportVmFormat hasValue {AMI}
218 hasCountry hasValue Japan
219 hasState hasValue Tokyo
220
221 relation compatible(ofType virtualAppliance, ofType virtualUnit )
222
223  axiom compatbilewith
224      definedBy
225
226 ?x memberOf virtualAppliance and  ?x[hasProvider hasValue ?p]  and ?y
        memberOf virtualUnit and ?y[hasProvider hasValue ?pvu] and
227 ?p[hasCountry hasValue ?capp] and ?pvu [hasCountry hasValue ?cvu] and ?capp[
        hasName hasValue ?cappName] and ?cvu[hasName hasValue ?cvuName] and
228 stringEqual(?cvuName,?cappName)
229 implies ?x[isCompatibleWith hasValue ?y].
```

B. DEPLOYMENT DESCRIPTOR

```
1 wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
2 namespace { _"http://www.cloudslab.org/Deployment#"}
3 ontology Deployment
4  importsOntology  _"http://www.cloudslab.org/CloudProvider#
        CloudProviderOntology"
5
6 concept ScalingPolicy
7 hasName ofType _string
8 hasUpperBoundThreshold ofType _integer
9 hasLowerBoundThreshold ofType _integer
10 hasPeriod ofType _integer
11 hasMetric ofType _"http://www.cloudslab.org/CloudProvider#monitoringMetric"
12
13 concept ScalingGroup
14 hasName ofType _string
15 hasVirtualUnit ofType _"http://www.cloudslab.org/CloudProvider#virtualUnit"
16 hasVirtualAppliance ofType _"http://www.cloudslab.org/CloudProvider#
        virtualAppliance"
17 hasMinSize ofType _integer
18 hasMaxSize ofType _integer
19 hasLocation ofType _"http://www.cloudslab.org/CloudProvider#location"
20 hasLoadBalancer ofType _"http://www.cloudslab.org/CloudProvider#loadBalancer
        "
```

Copyright © 2014 John Wiley & Sons, Ltd.

Softw. Pract. Exper. (2014)
DOI: 10.1002/spe

```
21 hasPolicy ofType ScalingPolicy
22 hasSecurityGroup ofType SecurityGroup
23
24 concept SecurityGroupRule
25 hasName ofType _string
26 hasProtocol ofType  _string
27 hasPort ofType  _integer
28 HasSourceIP ofType _string
29 hasAuthorizedSecurityGroup ofType SecurityGroup
30 isAuthorizingSecurityGroup ofType _boolean
31
32 concept SecurityGroup
33 hasName ofType _string
34 hasLocation ofType _"http://www.cloudslab.org/CloudProvider#location"
35 hasRules ofType SecurityGroupRule
36
37 concept DeploymentDescriptor
38 hasScalingGroup ofType ScalingGroup
39 hasSecurityGroup ofType SecurityGroup
40
41 instance WSRule1 memberOf SecurityGroupRule
42 hasName hasValue "WSRule1"
43 hasProtocol hasValue  "TCP"
44 hasPort hasValue  80
45 HasSourceIP hasValue "0.0.0.0/0"
46 isAuthorizingSecurityGroup hasValue false
47
48 instance WSSecurityGroup memberOf SecurityGroup
49 hasName hasValue WSSecurityGroup
50 hasLocation hasValue _"http://www.cloudslab.org/CloudProvider#California"
51 hasRules hasValue WSRule1
52
53
54 instance WSLoadBalancer memberOf _"http://www.cloudslab.org/CloudProvider#
       loadBalancer"
55 hasName hasValue "WShasLoadBalancer"
56
57 instance WSScalingPolicy memberOf ScalingPolicy
58 hasName hasValue "WSScalingPolicy"
59 hasUpperBoundThreshold hasValue 80
60 hasLowerBoundThreshold hasValue 10
61 hasPeriod hasValue 600
62 hasMetric hasValue _"http://www.cloudslab.org/CloudProvider#CPUUtilization"
63
64 instance webServerScalingGroup memberOf ScalingGroup
65 hasName hasValue "webServerScalingGroup"
66 hasVirtualUnit hasValue _"http://www.cloudslab.org/CloudProvider#
       largeInstance"
67 hasVirtualAppliance hasValue _"http://www.CloudsLab.
       org/ontologies/VirtualAppliance#aki-00806369"
68 hasMinSize hasValue 1
69 hasMaxSize hasValue 3
70 hasLocation hasValue _"http://www.cloudslab.org/CloudProvider#California"
71 hasLoadBalancer hasValue  WSLoadBalancer
72 hasPolicy hasValue WSScalingPolicy
73 hasSecurityGroup hasValue  WSSecurityGroup
74
75 instance DDsample memberOf DeploymentDescriptor
76 hasScalingGroup hasValue webServerScalingGroup
77 hasSecurityGroup hasValue WSSecurityGroup
```

Copyright © 2014 John Wiley & Sons, Ltd.

DOI: 10.1002/spe