TOWARDS SEMANTIC-BASED CLOUD APPLICATION MANAGEMENT

Kyriakos Kritikos and Dimitris Plexousakis
ICS-FORTH
{kritikos,dp}@ics.forth.gr

Abstract

Cloud computing promises to transform applications and services on the web into elastic and fault-tolerant software. To aid at this target, various research prototypes and products have been already proposed. However, especially with respect to the design phase of cloud-based applications, such prototypes do not enable the appropriate composition of cloud services at different levels to realise not only the functionality but also the underlying infrastructure support for such applications. Moreover, most existing prototypes and products lack the appropriate semantics to guarantee that the respective design product is the most suitable and accurate one according to the various types of user requirements posed. To this end, this article proposes a semantic cloud application management framework that addresses the aforementioned issues by relying on ontologies to semantically describe cloud service capabilities and application requirements, on semantic cloud service matchmakers considering both functional and non-functional aspects as well as on a novel cloud service composition approach which is able to perform concurrently service concretisation and deployment plan reasoning, thus catering for the different levels involved in a cloud environment and their respective dependencies by also satisfying all types of user requirements posed. The service composition approach is experimentally evaluated deriving quite promising results indicating that the state-of-the-art is advanced.

Keywords: [cloud, service, composition, semantic, ontology, QoS, constraint programming, requirements]

1. INTRODUCTION

Cloud computing has revolutionized the deployment and provisioning of applications by promising an infinite amount of underlying and cheap resources to enable applications to scale at any type of demand. To this end, many major software, application and business process vendors have migrated their business to the cloud. Moreover, various research prototypes and commercial products have been proposed which enable application designers to discover the most suitable cloud services and assist in the cloud-based application deployment.

The services offered in cloud computing lie in different levels. There exist software services (SaaS), platform services (PaaS) and infrastructure services (IaaS). Different levels can provide support to different phases in the application life-cycle. The design of an application can rely on SaaS in order to have the means to realise the application functionality, while the application deployment can rely on PaaS and IaaS services. Moreover, for an application which exploits cloud services at different levels, its quality of service (QoS) depends on the respective quality and characteristics at lower-levels of abstraction. Thus, there are actually dependencies between the different levels which should be taken into account in an integrated and non-isolated manner.

However, the existing prototypes and products, which focus on the design and/or deployment phases, not only fail to consider such dependencies but also do not produce a design and deployment solution which is accurate and optimised according to the application requirements. The latter problem is mainly due to the lack of semantics in the description of the cloud services and requirements which then maps to their non-accurate discovery before they are actually composed.

To remedy the above issues, this article presents a semantic framework for the management of cloud-based applications. This framework relies on novel ontology-based language to describe application requirements at different levels, a semantic matchmaker able to discover services which accurate fulfill both the functional and non-functional requirements of the application and a cloud service composition component which solves a combined design and deployment optimisation problem by considering all possible cloud levels. The rest of the lifecycle phases are covered via components which attempt to enable the adaptive deployment and provisioning of the cloud application by building on existing research work and open-source software.

The cloud service composition component advances the state-of-the-art as, apart from composing cloud services at different levels, it exhibits the following features: (a) it considers unary and binary component placement constraints indicating one component's location or the relative location between two components either at the same VM or cloud, respectively; (b) it considers high- and low-level security requirements in terms of security controls and Service Level Objectives (SLOs), respectively; (c) it exploits non-linear functions able to map high- to low-level quality capabilities; (d) it even takes decisions on whether to use in-house software components or external SaaS for a particular application task; (e) it is able to address multiple objectives which span quality, cost and security metrics.

The cloud service composition component has been experimentally evaluated against baseline cloud deployment approaches. The evaluation results show that this component produces in a faster way suitable cloud service

composition solutions which optimally satisfy the application requirements compared to the solution quality and performance of the baseline approaches.

The rest of the article is structured as follows. Section 2 introduces a use case to appropriately motivate the proposed work. Section 3 presents the semantic cloud application management framework. Section 4 introduces a novel application requirement language. Section 5 analyzes the cloud service composition approach. Section 6 discusses the main results of the experimental evaluation. Section 7 reviews the related work. Finally, Section 8 concludes the article and draws directions for further research.

2. USE CASE

This case concerns the design of a real-world traffic management application [Baryannis et al. 2013] that regulates traffic at particular areas of a city. This application comprises the following three main tasks:

- *Monitoring Task (MT).* It monitors traffic conditions in a particular city area as well as air pollution and noise levels.
- *Analysis Task (AT).* It analyzes all monitored information and produces traffic regulation plans that optimally address the current traffic situation.
- *Traffic Configuration Task (TCT).* It enforces the traffic regulation plans derived by *AT*. This can involve changing traffic lights frequency, informing drivers about congested places and emergency personnel about accident placement and the particular actions to follow.

Table 1 . The components mapping to application tasks

Task	Component Name	Component Description
MT, AT, TCT	*Con*	It hosts the three main servlets of the application
MT	*MC* (choice)	It realises *MT*'s functionality
AT	*AC*	It realises *AT*'s functionality
AT	*DC*	A DB storing the information used for the analysis
TCT	*TCC*	It realises *TCT*'s functionality
	WO (choice)	It orchestrates the application workflow

Various software components/services have been developed or are required to realize this application, where either one or more map to a particular task. Moreover, a service oriented architecture (SOA) is chosen to realize the application tasks, so some components have to be hosted on servlet containers. Table 1 clearly shows the respective task-to-component mapping.

For some components, there is a choice of either selecting an existing realization (developed in house by the end-user or purchased/downloaded) or exploiting an external service. This choice is indicated in parenthesis after the respective component's name.

Concerning service performance and cost, Table 2 indicates the respective offerings along with information on which cloud provider offers them if they are external. The performance of internal services was determined via benchmarking which also lead to their eventual VM requirements (given later in this section). In addition, these services' cost is zero as it maps to an already purchased hosting infrastructure. The symbols used in this table have the following meaning: *RT* maps to response time, *Av* to availability and *Thr* to throughput. Cost information is per month based on the providers' cost model.

Table 2. The QoS and cost features of the services

Comp. Name	Service Name	QoS/cost chars	Provider Name
MC	*MonService*	*RT* ≤ 4 sec, *Av* ≥ 99.99%, *Thr* ≥ 10 reqs/sec, *cost* = 10$	CP1
MC	*TraffService*	*RT* ≤ 8 sec, *Av* ≥ 99%, *Thr* ≥ 5 reqs/sec, *cost* = 5$	CP2
MC	*MC-Internal*	*RT* ≤ 8 sec, *Av* ≥ 99.99%, *Thr* ≥ 6 reqs/sec, *cost* = 0$	
AC	*AC-Internal*	*RT* ≤ 1.5 min, *Av* ≥ 99.99%, *Thr* ≥ 6 reqs/sec, *cost* = 0$	
TCC	*TCC-Internal*	*RT* ≤ 0.5 min, *Av* ≥ 99.999%, *Thr* ≥ 12 reqs/sec, *cost* = 0$	
WO	*Orchestrator*	*Av* ≥ 99.99%, *Thr* ≥ 12 reqs/sec, *cost* = 19$	CP1
WO	*WFEngine*	*Av* ≥ 99%, *Thr* ≥ 8 reqs/sec, *cost* = 15$	CP2
WO	*WO-Internal*	*Av* ≥ 99.99%, *Thr* ≥ 8 reqs/sec, *cost* = 0$	

The end-user also requires the satisfaction of the following types of application requirements:

- Deployment Requirements:
  - There is a communication requirement from *WO* to all application servlets, i.e., *MC*, *AC*, and *TCC*, and from AC to DC.
  - *MC*, *AC* and *TTC* should be deployed on *Con*. These components will be hosted at the same instance of *Con* only when it is decided that they will be collocated.
  - *AC* and *DC* require a "high" VM, *WO* and *MC* a "medium" VM while *TCC* a "small" VM.

- *AC* and *DC* should be co-located while *AC* should not be co-located with any other component (apart from *Con* that hosts it).

- Cost requirements: Application cost must be no more than 380 $ per month.
- Quality requirements:
  - Application duration should not be longer than 2.5 minutes.
  - *MC*, *AC* and *TCC* should have throughput greater than or equal to 10, 6 and 5 reqs/sec, respectively.
  - *MC* and *AC* should have availability of 99.99% while *TCC* of 99.999%.
- Security requirements:
  - The security controls (*https://cloudsecurityalliance.org/research/ccm*) to be supported for the application must be: AAC-02 (independent reviews and assessment of provider at least annually), DSI-01 (data \& service classification), DSI-05 (data leakage prevention), TVM-02 (timely vulnerability detection) and SEF-05 (monitoring \& quantification of security incident type, volume and cost).
  - Meantime between incidents [Pannetrat 2013] should be 6 months (*mti* > 6)

Table 3. The offerings of the four cloud providers

Provider	Offered VM	Security Control	Security SLO
CP1	(A) 2 core, 7.5GB, 32GB →0.140$ (B) 4 core, 15GB, 80GB →0.280$ (C) 4 core, 7.5GB, 80GB →0.210$	(A) AAC-02 (B) AAC-03 (C) DSI-01 (D) DSI-05 (E) EKM-03 (F) TVM-02 (G) SEF-05	(A) *mti* ≥ 8 (B) *ir (99%)* ≤ 3
CP2	(A) 2 core, 2GB, 10GB →0.06$ (B) 2 core, 4GB, 50GB →0.12$ (C) 4 core, 8GB, 130GB →0.24$	(A) AAC-02 (B) AAC-03 (C) DSI-01 (D) DSI-05 (E) EKM-03 (F) TVM-02 (G) SEF-05	(A) *mti* ≥ 6 (B) *ir (99%)* ≤ 2
CP3	(A) 1 core, 2GB, 10GB →0.02$ (B) 2 core, 4GB, 20GB →0.04$ (C) 4 core, 4GB, 40GB →0.1$	(A) AAC-02 (B) AAC-03 (C) DSI-05 (D) EKM-03 (E) TVM-02	(A) *mti* ≥ 6 (B) *ir (99%)* ≤ 4
CP4	(A) 1 core, 2 GB, 20GB→0.2$	(A) AAC-02 (B) DSI-01 (C) DSI-05 (D) TVM-02 (E) SEF-05	(A) *mti* ≥ 4 (B) *ir (99%)* ≤ 4

Let us now consider four cloud providers, namely CP1, CP2, CP3 and CP4 which offer particular cloud services/VMs and realize a certain set of security controls. **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.** shows the VMs satisfying the end-user requirements offered by these providers (along with cost information), a subset of security controls supported by these providers and the security SLOs promised.

Real values for VM characteristics and cost were considered by collecting them from cloud provider web pages. So, we are as realistic as possible, provided that cloud providers do not usually advertize SLO and security information. Thus, we have opted for an idealized use case matching the real world in the near future, when cloud providers decide to advertize the latter information due to the main benefits that this will provide to them.

By considering all above information, a common cloud service composition approach would not consider the alternative design choices and end-user's security requirements. As such, we can assume that the end-user would not specify that his/her components can be realized via external services. Thus, in the end, the respective approach would solve a simple optimization problem to produce a concrete IaaS composition. The outcome of such an approach would be a solution mapping application components to the following VMs (i) *AC* + *DC* → CP3 (C), (ii) *MC* → CP3 (B) and (iii) *HO* + *TCC* → CP3 (B), where CP*i* (*X*) means the *X* offering of Cloud provider *i*. Co-location of *HO* and *TTC* in CP3 VM of type (B), while not imposed by any direct constraint, is proposed as *TCC* does not demand strict VM requirements so that it can be supported via a VM with better characteristics that suit the *HO*'s resource requirements.

Table 4. Cost, QoS and security features of the solutions

Solution	Cost	Availability	Duration	Security
1	130$	99.99%	128 sec	no
2	286$	99.99%	128 sec	yes
3	129.8$	99.99%	124 sec	no

Table 4 summarizes the QoS, cost and security features of three solutions. The first solution maps to the common cloud service composition approach. The second solution, produced by our approach, is the best one as it considers all possible information and user requirements, including the security ones. It maps to selecting the external services offered by CP1 for all choices, the CP1 (C) VM for *AC* + *DC* and the CP2 (B) offering for *TCC*. To enable a more fair comparison, we also consider the third solution produced

via our approach by not considering security requirements, which is identical to the solution of the common service composition approach with the sole exception that *MC* and *HO* are not mapped to SaaS services. Please note that we assume that the common composition approach (as in the case of our approach) will produce the best possible solution based on the input provided to it as it will use a solving technique guaranteeing optimality (see Section 7).

The third solution, while not considering security requirements, is still more optimal than the first. The comparison between the second and third solution indicates the trade-off between security and cost that must be considered to produce the best possible solution based on user requirements.

The last two solutions propose a multi-cloud application design product spanning over two cloud providers (CP1 and CP2 for the second and CP1 and CP3 for the third). The second solution has filtered the remaining cloud providers as they do not satisfy the user security constraints: CP4 violates the SLO for mean time between incidents while CP3 does not support DSI-01 and SEF-05 security controls.

3. Semantic Cloud Application Management Framework

The architecture of the semantic cloud application management framework can be seen in Figure 1. This framework spans both the design as well as the adaptive deployment and provisioning of a cloud application. It comprises the following components:

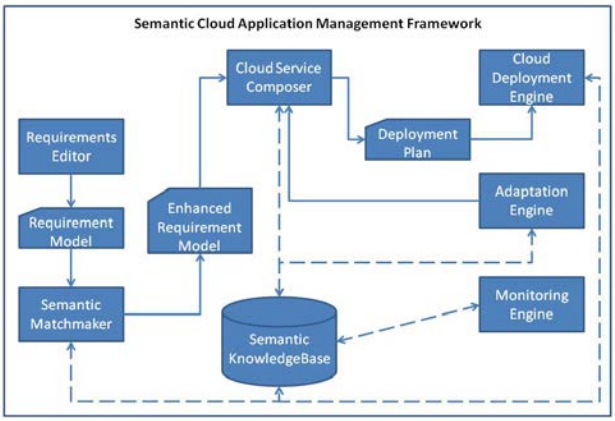

Figure 1. The architecture of the semantic cloud application management framework

- The *Requirements Editor* is a User Interface (UI) component which interacts with users to obtain their requirements. Users, through this component, are guided in providing different types of requirements at different levels. These requirements are then transformed into a *requirement model* described via the semantic requirement language analyzed in Section 4.
- The *Semantic Matchmaker* attempts to match the user requirements against semantic cloud service descriptions. It then enriches the user's requirement model through indicating which requirements are met by which cloud service alternatives, thus producing an *enhanced requirement model*. The SaaS matchmaking exploits particular techniques for both functional and non-functional aspects. Functional SaaS matchmaking relies on semantic input/output (IO) matching [Klusch et al. 2006]. Non-functional SaaS matchmaking, performed after the functional one, exploits particular matching metrics and respective constraint solving techniques [Kritikos & Plexousakis 2014] (see Section 7.2.1). IaaS matchmaking uses the same techniques as in non-functional SaaS matchmaking as IaaS offerings can be regarded comprising sets of constraints on VM features.
- The *Cloud Service Composer* obtains the *enhanced user requirement model* and transforms it into a constraint optimisation problem. This problem is then solved based on particular constraint solving techniques. In the end, the solution is transformed into a *deployment plan* specified in CAMEL [Rossini et al. 2014]. Section 5 provides a detailed analysis about this component.
- The *Cloud Deployment Engine* retrieves the *deployment plan* produced by the Cloud Service Composer and executes it. The Cloudiator framework [Domaschka et al. 2015a] has been used to realise its functionality. This framework not only abstracts away from the technical peculiarities of different clouds but is also capable of deploying applications in multiple clouds. In fact, in our opinion and also shown by the use case in Section 2, multi-cloud application deployment should be the way to go forward due to the following reasons: (a) leads to a more optimal satisfaction of application requirements as the most suitable alternative cloud services are selected with the best functional and quality capabilities; (b) the hurdle of vendor lock-in is surpassed enabling applications to be deployed in multiple and across different clouds.
- The *Semantic Knowledgebase* (SKB) contains information that spans the lifecycle of a cloud application as well as the main capabilities offered by different cloud providers. The SKB also includes the description of semantic rules operating over its content which can drive the adaptation behavior of an application by reacting on incoming application measurement and contextual information. An

existing triple store (Virtuoso[1]) with a reasoner on top of it (Pellet[2]) has been used to realise the SKB.

- The *Monitoring Engine* is responsible for monitoring the cloud-based application in each and across all clouds in which it has been deployed. The *MetricsCollector* component [Domaschka et al. 2015b] has been used to realise its functionality. This component has been enhanced in order to operate over the semantic description of metrics defined in OWL-Q [Kritikos & Plexousakis 2006].
- The *Adaptation Engine* is responsible for adapting the cloud-based application when critical situations occur. Such situations are detected based on the findings of the Monitoring Engine and the content of the adaptation rules. The latter indicate which pattern of events represent these situations and what are the adaptation actions required to resolve them. The description of adaptation rules relies on SRL [Kritikos et al. 2014] while the *Adaptation Engine* was realised based on the work in [Zeginis et al. 2015]. It should be noted that when a current critical situation cannot be any more coped by the existing adaptation rules, the *Adaptation Engine* informs the *Cloud Service Composer* to propose a new cloud solution for the application at hand which should be able to overcome this situation.

This framework, as can be easily understood, exhibits various features that enable it to be distinguished from other research prototypes and products. More importantly, it is able to deal with multiple cloud levels as well as address the whole lifecycle of a cloud-based application, going from its design and deployment until its adaptive provisioning. Moreover, the different components involved in this framework can well be exploited by other prototypes and products in order to enable the latter to function at different cloud levels. For instance, the *Requirements Editor*, the *Semantic Matchmaker* and the *Cloud Service Composer* could be exploited by a product for the derivation of concrete deployment plans based on the initial user requirements. Then, such a product could employ its own components for the actual deployment and adaptive provisioning of an application.

4. APPLICATION REQUIREMENTS ONTOLOGY LANGUAGE

In order to express different types of application requirements at different levels, a particular semantic language has been developed based on OWL, the de-facto standard in ontology description. This language relies on OWL-S [Sycara et al. 2003] for the functional description of the SaaS services and OWL-Q [Kritikos & Plexousakis 2006] for the non-functional and cost description of any kind of cloud service. We should note here that we consider that an application, being a composition of cloud services in the end, is also considered as a SaaS service. The class diagram in Figure 2 shows the main classes and their respective relationships for this language.

The main class for describing an application or a SaaS service is *Service* in OWL-S for which the process model part (cf. *ProcessModel* class) can be used to indicate the domain classes to which its I/O maps for matchmaking purposes. Concering the service composition description, we have semantically-enhanced the part of the component meta-model in [Zeginis et al. 2015] dedicated to the modelling of composite services. In that meta-model, a service can be distinguished into a CompositeService or a SingleService. A CompositeService is associated to a data flow and a control flow, where the former indicates the data bindings between the inputs and outputs of the composite services' components while the control flow explicates the order of execution between the component services according to particular composition patterns. As such, through the re-use of this meta-model part, the application structure at the highest level can be provided. Obviously, in case users do not require to provide such a flow as they are not modelling a pure service and workflow-based application, they can just indicate via exploiting the proposed ontology the components that their application consists of.

Each *Service* is associated to a non-functional profile named as *QoSDemand* in OWL-Q. This profile maps to a logical combination of quality constraints indicating the set of conditions on particular quality metrics that are required for this service. Through using OWL-S and OWL-Q and by assuming that the SaaS services offered by cloud providers are described in an equivalent way, we enable the semantic matchmaker to combine the existing functional and non-function service matchmaking techniques in order to cover service discovery at the SaaS level.

The remaining types of requirements not covered by OWL-Q and OWL-Q are encapsulated as sub-classes of the *Requirement* class which is directly associated to a *Service*. This class comprises the *DeploymentRequirement* and *SecurityControlRequirement* sub-classes which represent deployment and security control requirements, respectively. A deployment requirement is further distinguished into service deployment, component deployment, communication and placement requirements.

[1] http://virtuoso.openlinksw.com/
[2] https://github.com/complexible/pellet

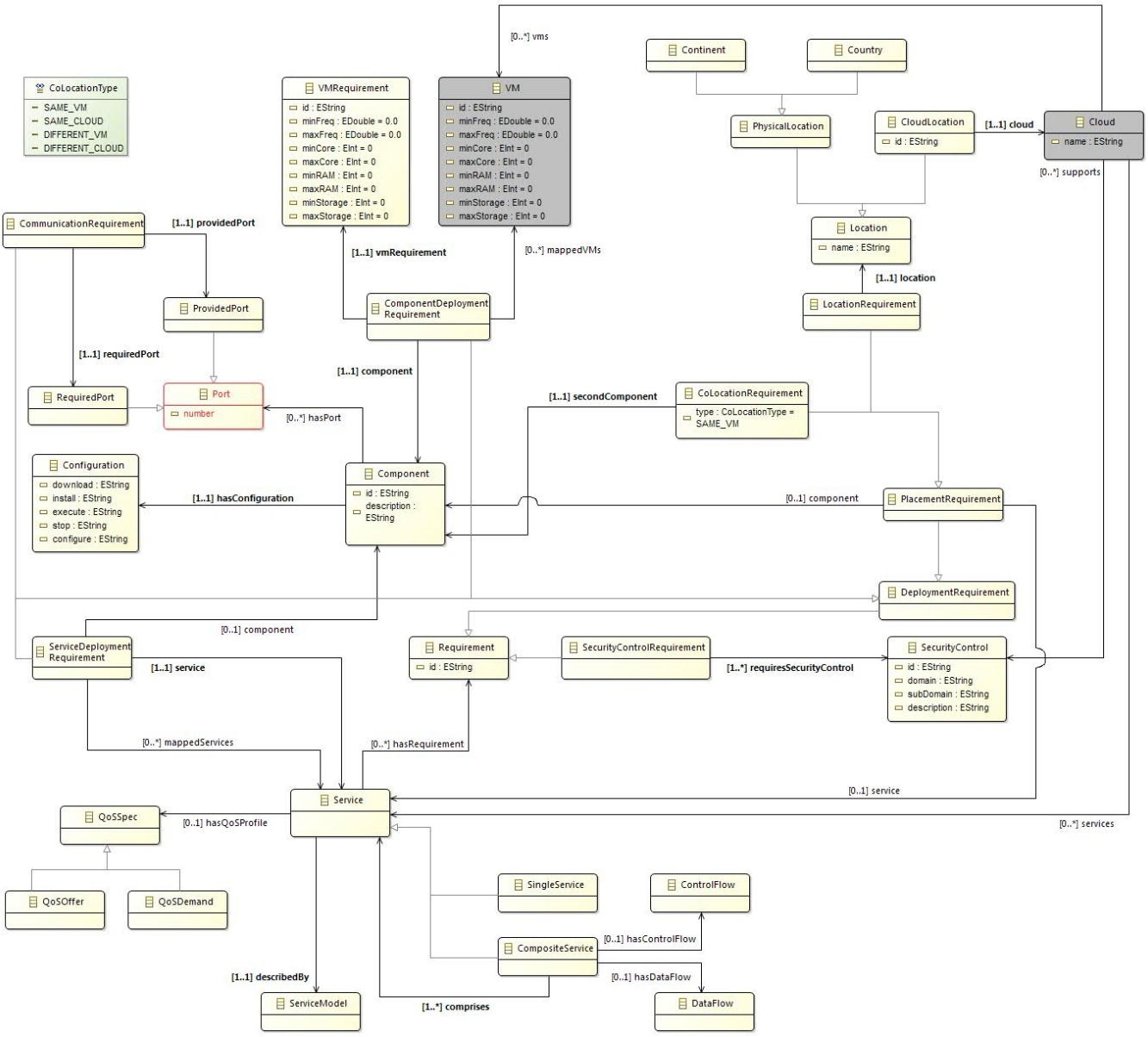

Figure 2. The class diagram of the requirements language

A service deployment requirement indicates whether a particular service component of the application will be realised via an external SaaS or an internal software component which has been either developed in house or has been purchased. Such a requirement can also leave both options open such that the proposed framework can decidesthe one that best satisfies the user requirements.

In case of external SaaS realisation, either the external SaaS service is fixed by pointing to its actual description or SaaS matchmaking will be performed to discover SaaS external services realising the respective service component's functionality.

In case of a software component realisation (or of both options), a pointer to the description of the software component has to be provided. To this end, the *SoftwareComponent* class has been modelled which has a name and short description as attributes. It is related to *Configuration* information which maps to the handling of its lifecycle by indicating how the code of the component can be downloaded, configured, installed, executed and stopped. A software component may also have particular ports through which it can communicate with other components.

As it can be understood, this requirement specification is quite flexible. It can either rely on the framework to address the mapping of an application component to any kind of internal or external software or on the user to determine a concrete external SaaS and/or software component to address this mapping.

A component deployment requirement indicates the VM requirements for the IaaS service which will be used to host a software component which realises the functionality of an application component. The requirements for a VM span constraints over its main characteristics which are the amount of cores, the CPU frequency, the size of the main memory and the size of the hard disk, and are modelled via the *VMRequirement* class. The collection of a set of different constraints into a single class enables the re-use of VM requirements across different component deployment requirements.

A communication requirement actually connects two components indicating that they should communicate to each other. By also explicating the ports exposed by these components which will be used for their communication, the deployment of the application can be configured correctly such that there are no errors in the establishment of the communication links between these pair of components.

A placement constraint can indicate a particular physical (e.g., Europe) or virtual location (e.g., Amazon eu-west-1) in which a component can be placed or pair-wise placement constraints between components. As such, we can cater for cases where conformance to laws and regulations should be guaranteed as well as for cases where we desire to guarantee that the communication requirements between a pair of components are met. Pair-wise placement constraints can be of the following type: (a) *SAME_VM*: the same VM should be used to host the pair of components; (b) *SAME_CLOUD*: the same cloud should be used for the deployment of these two components; (c) *DIFFERENT_VM*: these two components should be placed in different VMs; (d) *DIFFERENT_CLOUD*: these two components should be placed in different clouds.

A *SecurityControlRequirement* actually maps an application component or the application itself to a set of security controls which have to be supported by the cloud provider(s) of the respective cloud service(s) selected to realise the application component(s) functionality. As can be easily understood, such a cloud service can be either an external SaaS or a IaaS used to host the internal software component realising the current application component's functionality. Each *SecurityControl* is described via a domain, a sub-domain, its id and a particular textual description. The set of security controls which is currently supported has been derived from the Cloud Control Matrix (CCM) of Cloud Security Alliance (SCA)[3].

We should highlight here that VM requirements should be matched with VM capabilities. To this end, we have created another ontology-based *semantic provider language*, which is used to describe VM capabilities as well as the respective cloud providers that offer them. The VM capabilities are described in an equivalent, symmetric manner with respect to the respective requirements. The matching of them relies on a simple constraint problem derivation which is similar to the one used for non-functional SaaS matching. In this way, there is a uniform way in which non-functional and VM requirements are handled.

The *semantic provider languge* also connects a cloud service provider to the SaaS services that it offers as well as to the set of security controls that it supports. In this way, we also enable the filtering of the cloud provider space according to the high-level security requirements posed.

The whole *requirement model* which can be expressed by our proposed ontology language can then be enhanced by the *Semantic Matchmaker* by indicating which application components can be realized by which cloud services. In this respect, the requirement model is updated with pointers to the descriptions of the respective cloud services in service deployment and component deployment requirements. The service deployment requirements will point to the external SaaS that can be used to realise an application component's functionality, while the component requirements will point to those IaaS that can be used to host the respective software component of a certain application component.

To summarize, the proposed requirement ontology language is quite flexible and expressive to specify any kind of cloud-based application requirement. It is also complemented with a cloud capability language to enable the appropriate matching of requirements to respective cloud service offerings. The final product derived through the modelling and matchmaking of requirements is an *enhanced requirement model* which drives the cloud service composition and the respective deployment plan production which will be dealt with in the next section.

5. CLOUD SERVICE COMPOSITION APPROACH

All possible design choice alternatives incarnated into the *enhanced requirement model* are transformed into a particular optimization problem by the *Cloud Service Composer* which, when solved, can discover the most optimal cloud service solution satisfying all user requirements posed irrespectively of their type. The approach followed was inspired by the service concretization work in [Ferreira et al. 2009]. In the following, we analyze the way the constraint problem is modelled in a step-wise manner, starting from optimization objectives and going down to the formulation of the high- and low-level constraints mapping to user requirements. Then, we check the complexity and possible constraint solving technologies for this problem.

5.1 CLOUD SERVICE COMPOSITION PROBLEM FORMULATION

To formulate the optimization objective of the problem, we rely on the Analytical Hierarchy Process (AHP) [Saati 1980] to derive the relative importance of the quality

[3] https://cloudsecurityalliance.org/group/cloud-controls-matrix/

parameters and cost to the user. The result of this process is an assignment of weights to all these parameters, indicating their relative importance, whose sum should equal to one. We also follow Simple Additive Weighting (SAW) technique [Hwang & Yoon 1981] which maps the optimization of all criteria considered to a single optimization objective which is equal to the weighted sum of the application of the global value derived for each parameter (QoS and cost) on its utility function posed. More formally, the objective is formulated as follows:

$$maximize\left(\sum_{q=1}^{Q} w_q * uf_q\left(val_q\right)\right)$$

The utility function of each parameter is formulated based on the formulas in [Ferreira et al. 2009] which cater for slightly violating some problem constraints to address over-constrained user requirements. The following combined expression represents these formulas in which the first two cases depend on the monotonicity of the respective parameter (i.e., the first for negatively monotonic parameters like *cost* and the second for positively monotonic parameters like *availability*), where m is the max function, v_q^{max} and v_q^{min} are the maximum and minimum values requested by the end-user for the parameter q and a_q is a real number in [0.0,1.0] used to regulate the percentage of values allowed outside the user-requested range.

$$uf_q(x)=\begin{cases} a_q+\dfrac{v_q^{max}-x}{v_q^{max}-v_q^{min}}\cdot\left(1-a_q\right), & v_q^{min}\le x\le v_q^{max}\wedge x\downarrow \\ a_q+\dfrac{x-v_q^{min}}{v_q^{max}-v_q^{min}}\cdot\left(1-a_q\right), & v_q^{min}\le x\le v_q^{max}\wedge x\uparrow \\ \mathrm{m}\left(a_q-\dfrac{v_q^{min}-x}{v_q^{max}-v_q^{min}}\cdot\left(1-a_q\right),0\right), & x<v_q^{min} \\ \mathrm{m}\left(a_q-\dfrac{x-v_q^{max}}{v_q^{max}-v_q^{min}}\cdot\left(1-a_q\right),0\right), & x>v_q^{max} \end{cases}$$

The value v_q that a parameter q can take depends on the type of this parameter and its derivation can be application-specific by depending on the application's structure [Ardagna & Pernici 2007]. To this end, in the general case, we consider that a particular application-specific function is provided taking as input the respective parameter values of the application components. In other terms: $val_q = f_q\left(val_i^q\right)$ where val_i^q is the parameter value for application component *i*. The use of a function covers all possible cases in parameter value derivation. In this way, by considering the running example, the application availability equals the product of availabilities of the three main components (i.e., *MC*, *AC* and *TCC*), while application cost is equal to the cost of all components (thus mapping to the respective cost of the infrastructure-as-a-service (IaaS) or SaaS exploited).

Before specifying the problem constraints, we introduce the main decision variables mapping to three variable arrays:

- y_i indicating whether the internal software component or the external SaaS services will be used to realize (application) component *i*
- x_{ijk} which indicates whether for the internal software of an application component *i*, the IaaS offering *k* of the cloud provider *j* has been selected (internal service selection case) to host it.
- z_{il} indicating whether SaaS service *l* has been selected to realise the functionality of application component *i*.

We differentiate between IaaS and SaaS services as they map to different formulas indicating how their parameter values can be mapped to the respective values at the (application) component level. While it could be argued that there is no need for explicating which IaaS offerings are provided by which cloud provider, we need to make this differentiation to be able to specify pair-wise placement constraints.

It is apparent that two exclusive cases exist for each application component: (a) there is no choice for realizing but just for deploying it and (b) there is indeed a realization choice. In the first case, it is enough to enforce that only one cloud provider and respective offering can be selected. In the second case, we need to indicate that only one external SaaS must be selected for realizing the component. Both cases lead to requiring the satisfaction of the following constraint:

$$\sum_j \sum_k x_{ijk} + \sum_l z_{il} = 1$$

Apart from the above constraints, we need to go down to the level of application components and indicate how their parameter values are derived from those of the offerings selected for them. We first assume that an application component's parameter value either maps to a one-to-one manner to the respective software component value which is computed as a function over the resources exploited (memory, CPU and storage), or is computed from the respective parameter value of the external SaaS realizing it. More formally:

$$val_i^q = y_i * f_i^q(core_i, mem_i, store_i) + (1 - y_i) * \left(\sum_l z_{il} * val_{il}^q \right)$$

where f_i^q is the function over the resources for parameter *q* of (application) component *i* while val_{il}^q is the parameter value for the *l* external SaaS of component *i*.

In case of internal software component deployment in the cloud the above (left part of the) computation is valid as the usual way of deriving high- from low-level requirements is either via benchmarking, simulation, or performance model learning [Xiong et al. 2013] such that we can map different service levels of application components to different resource levels.

Thus, we regard that the end-user has exploited one of the three possible approaches to produce the respective functions for those quality parameters of interest. We also envisage a step-wise approach to performance modelling. First, performance models for components are generated and then we go up to the level of the application. In this way, the component performance models will be more precise and will also lead to more accurate application performance models rather than attempting to map immediately the application performance to the underlying resources.

In this sense, we only need now to specify how the low-level resource values are produced for a particular component[4]. This maps to the following three formulas:

$$core_i = \sum_{jk} x_{ijk} * core_{jk}$$

$$mem_i = \sum_{jk} x_{ijk} * mem_{jk}$$

$$store_i = \sum_{jk} x_{ijk} * store_{jk}$$

where $core_i$, mem_i, and $store_i$ are the variables mapping to the component's *i* number of cores, main memory size and storage size, respectively, while $core_{jk}$, mem_{jk}, and $store_{jk}$ are the corresponding but fixed resource values for the concrete VM offering *k* of provider *j*.

The cost of each component is calculated by considering the next formula:

$$cost_i = y_i * \sum_{jk} x_{ijk} * cost_{jk} + (1 - y_i) * \sum_l z_{il} * cost_{il}$$

where $cost_{jk}$ is the cost of IaaS offering *k* of provider *j* and $cost_{il}$ is the cost of SaaS *l*. Thus, a component's cost equals the cost of the IaaS or SaaS it exploits.

The unary location constraints can regard either a physical or a virtual location. The latter maps to a specific cloud so this means that the *enhanced requirement model* will be already filtered according to such locations. There can be two types of physical locations, either countries or continents, where the former can obviously be included in the latter. By relying on the Food and Agriculture Organisation of United Nations (FAO) physical location ontology[5], we have covered all continents and countries by also modelling all their respective inclusion relationships. In this way, functions operating on the respective location elements can be employed in order to enforce the following two location constraints on IaaS and SaaS services, respectively.

$$\text{equalOrIn}(loc_{jk}, loc_i) == \text{true}$$

$$\text{equalOrIn}(loc_{il}, loc_i) == \text{true}$$

where equalOrIn is a function indicating whether the first location equals or is included in the second location, loc_{jk} is the location of the *k* VM offering of provider *j*, loc_{il} is the location of service *l* for component *i* and loc_i is the required location of component *i*. We should note here that we do not cover the case that the location of a cloud service includes the required location of the component as it is not certain that the cloud provider will be able to offer the respective service in the required country of the supported continent. Such a provider might support some but not all of the countries inside that particular continent.

We provide a specific formulation for pair-wise placement constraints depending on their type. A SAME_VM pair-wise placement constraint is formulated as follows:

$$x_{ijk} = x_{i'jk}$$

where *i* and *i'* are the two components for which the co-location constraint is posed. This constraint indicates that the decision for both components should coincide. Thus all values for respective array parts in which *i* and *i'* are fixed should be equal.

[4] Please note that in the case of internal software deployment, we use the term component from now on to indicate both the application component and the software used to realise it as there is a one-to-one mapping between their respective parameters.

[5] Available at: http://www.fao.org/countryprofiles/geoinfo/geopolitical/resource/

A DIFFERENT_VM placement constraint is expressed as follows:

$$if\left(x_{ijk} == 1\right) \Rightarrow x_{i^{'}jk} = 0$$

indicating that if a particular offering *k* of a cloud provider *j* is selected for component *i*, then this provider's offering cannot be selected for component *i'*.

The SAME_CLOUD pair-wise placement constraint is formulated as follows:

$$\sum_{k} x_{ijk} = \sum_{k} x_{i^{'}jk}$$

where *i* and *i'* are the two components for which the pair-wise placement constraint has been posed. This constraint indicates that for both components the same cloud has been selected which maps to requiring that the sum of values of the decision variables (mapping to the provider's offerings) for each cloud provider to be equal for these components.

The DIFFERENT_CLOUD placement constraint can be expressed as follows:

$$if\left(\sum_{k} x_{ijk} == 1\right) \Rightarrow \sum_{k} x_{i^{'}jk} = 0$$

indicating that if any offering of cloud provider *j* is selected for component *i*, then no offering from this provider can be selected for component *i'*.

To conclude formulating the problem, we need to cater for the user security requirements which can be separated into high-level in terms of security controls and low-level in terms of SLOs. In the first case, we introduce set variables and enforce set operations to address the respective requirements. In particular, we enforce that if a particular cloud provider has been selected, then this provider should have realized all security controls required by the end-user. This is translated to the following complex constraint:

$$\text{if}\left(y_i \wedge \sum_{k} x_{ijk} == 1\right) \Rightarrow cc - ccp_j = \varnothing$$
$$\text{elseif}\left(\neg y_i \wedge z_{il} == 1\right) \Rightarrow cc - ccp_{z_{il}.provider} = \varnothing$$

where *cc* is a fixed set variable mapping to all required security controls, ccp_j is a fixed set variable mapping to the security controls supported by provider *j*, and $z_{il}.provider$ is the index of provider which offers SaaS *l* for component *i*. We consider that the security control requirements should hold for any provider whose service is selected. In case such requirements are posed at the component level, the above formula can be remodelled by replacing *cc* with cc_i mapping to the fixed set variable for component *i* equal to the security controls to be realized by the provider whose service is used to realize or support this component.

In case of low-level security requirements, a similar constraint is posed:

$$\text{if}\left(y_i \wedge \sum_{k} x_{ijk} == 1\right) \Rightarrow seq_j^p \geq seq^p$$
$$\text{elseif}\left(\neg y_i \wedge z_{il} == 1\right) \Rightarrow seq^p_{z_{il}.provider} \geq sec^p$$

where seq^p is the low required threshold for security property *p* while seq_j^p is the respective property value promised by provider *j*. This formula is meaningful for positively monotonic security properties. The opposite case can be easily derived but due to space limitations is not shown. If the user provides both low and upper thresholds, the constraints introduced for both security property types must be enforced.

5.2 COMPLEXITY & SOLVING TECHNIQUES

The common cloud service composition problem is NP-Hard [Jula et al. 2014]. While we use additional sets of constraints, especially non-linear ones, and variables, the general problem formulation showed in previous sub-section is still NP-Hard.

Due to the nature of this problem, Mixed-Integer Programming (MIP) techniques cannot be actually used. Thus, non-linear constraint solving techniques must be checked, from which we have selected the Constraint Solving Optimization Problem (CSOP) ones, as they seem the perfect candidate for our case. These techniques can address not only non-linear constraints but can also cater for the use of different variables, such as boolean, integer, and set variables. However, real variables are not natively supported. To this end, the current workaround that seems to work well in many circumstances is to combine the use of CSOP with either MIP or Constraint Programming techniques focusing on interval arithmetic. In fact, many hard and real-world problems are now solved through the combined use of these techniques [Timpe 2002; Milano 2003].

In our current implementation, we have used a well-known and free CSOP solver called Choco (*choco-solver.org*) which is also supported by a very active community, while performs well and even competes with proprietary solvers. Apart from supporting all types of variables required, Choco has implemented well-known

state-of-the-art constraint types (e.g., *all different*) and various search strategies. Choco also includes an explanation engine indicating in case of over-constrained requirements which user constraints are hard to satisfy.

To address real variables, Choco exploits the Ibex constraint programming engine (*www.ibex-lib.org*). Ibex has been realized as a C++ library, relies on both interval and affine arithmetic, and is able to address non-linear constraints, handle roundoff errors, and declaratively build strategies via the contractor programming paradigm.

6. EXPERIMENTAL EVALUATION

We have conducted a preliminary experimental evaluation of our approach performance which aimed at assessing the effect of an increasing number of cloud provider offerings and placement constraints. To this end, two separate experiments were performed evaluating the effect that each different factor has. Three CSOP approaches were actually evaluated: (a) *RESOURCE* mapping to the common IaaS composition method used as a baseline where only resource constraints are considered and just one optimization parameter (cost), (b) *RESOURCE_SEC* which is same as previous method but enriched with security and placement constraints and (c) *FULL* which is the actual proposed approach.

The evaluation metric was the average solving time whose value was generated over 30 runs in order to minimize various interference types in the measurements, such as those attributed to the running OS. The computer on which the experiments were performed had the following characteristics: 1.7 GHz CPU, 2GB of main memory and 500 GB of disk.

The input given to the three approaches was randomly generated but only realistic values were considered. For instance, the core number was given values from 1 to 8 while main memory from 512 to 8192 for a particular cloud provider IaaS. Security capabilities were formed by randomly assigning a specific percentage of all possible security controls for each cloud provider, while a respective smaller percentage was used as the application requirement. Placement constraints were formed by randomly picking up their type and component pair on which they should hold. Then, each approach exploited this input, created the respective CSOP problem and solved it. In the CSOP formulation, a linear function from resources to QoS attributes was utilized for each component. It was also assumed that the composition of values for execution time & cost, throughput and availability at the global application level exploited additive, minimum or multiplicative functions, respectively.

The initial values for the experiment configuration parameters were: application component number → 5, cloud provider number → 10, IaaS/SaaS number per provider → 5 and placement constraint number → 5. In the first experiment, we increased the value of the IaaS/SaaS offerings per provider in units of 5 until the value of 25. In this way, we simulate the case where either an increased number of offerings is supplied by each provider or an increased number of providers occurs. The evaluation results are shown in Figure 3. As it can be seen, due to the nature of the problem, all approaches exhibited an exponential behavior. However, our approach had a better performance than the others. This can be certainly justified by the fact that while slightly increasing the variable number, the constraint number is also increased. As such, the constraint solving algorithm more deeply cuts the search space to find the most optimal solution. The same holds when comparing *RESOURCE_SEC* and *RESOURCE* where again the increased number of constraints leads to a better performance. We have posed a limit of 3 minutes to the solving time so as to be acceptable by an application designer which justifies the first approach behavior.

The second experiment focused on examining the effect on increasing the placement constraint number from 1 to 5 (but not greater due to the small number of application components). Figure 4 shows the respective evaluation results only for the last two approaches that are indeed capable of considering such constraints. The same linear decreasing behavior is observed for both approaches. This is expected as placement constraints reduce the offering space to be explored. Again, *FULL* had a better performance than *RESOURCE_SEC* as it considers also high-level constraints.

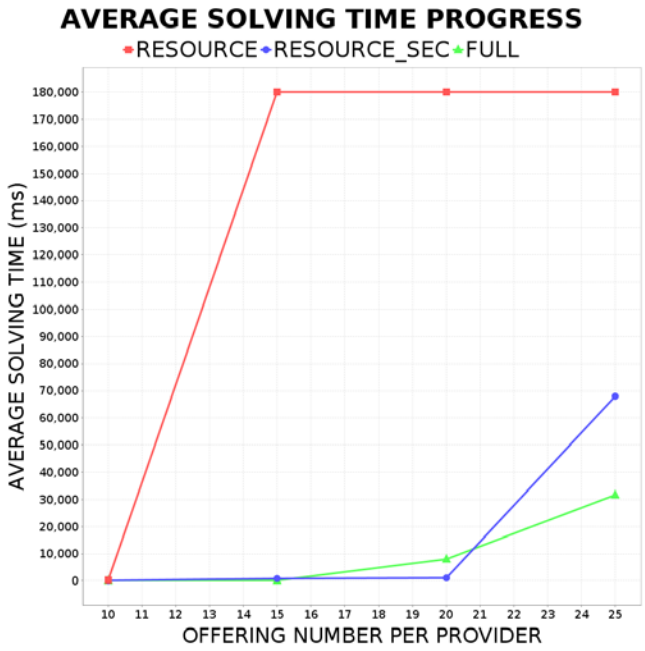

Figure 3. Average solving time per offer number

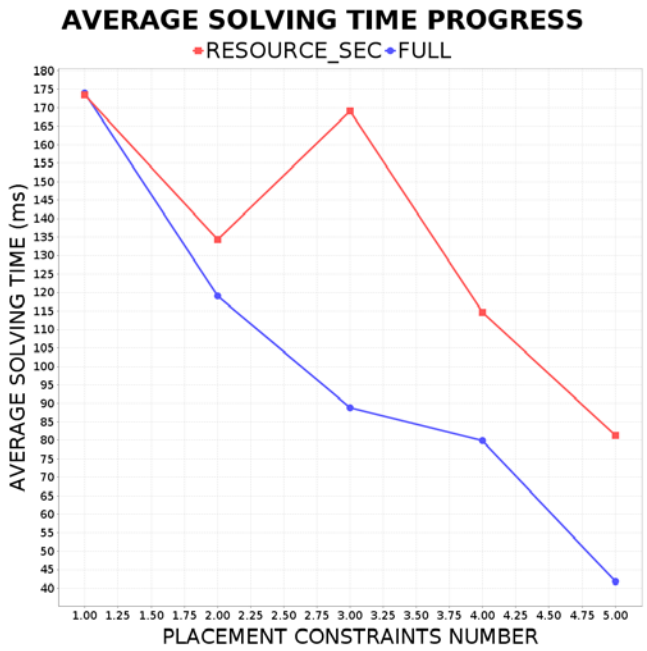

Figure 4. Average solving time per placement constraint number

7. RELATED WORK

7.1 SERVICE MODELLING

7.1.1 SOFTWARE SERVICE MODELLING

WSDL [Christensen et al. 2001] is the de-facto standard for the description of the interface for web-services. However, it is a structural language and does not cover other information aspects, such as the service functionality and its non-functional capabilities.

USDL was a semi-formal language used for the description of business and software services. Recently, it has been transformed to a Linked-Data counterpart [Pedrinaci et al. 2014] in order to become more formal. USDL is capable of covering SLA, quality, security, cost and legal aspects. There was also an approach [Cardoso et al. 2013] focusing on the integration of USDL with TOSCA [Palma and Spatzier 2013] to link service selection with deployment such that the cloud application lifecycle is better supported.

OWL-S is a W3C recommendation for the semantic description of web services. It mainly focus on functional aspects, covering the semantic description of the service I/O and its abstract interface. It also proposes a particular grounding mechanism in WSDL. However, OWL-S does not cover the non-functional aspects and is not able to describe service orchestrations.

WS-BPEL [Alves et al. 2007] is a service orchestration language which has been widely adopted. It is relies on WSDL for the description of the component services of the orchestration but there have been research-based extensions which have relied on WSMO [de Bruijn et al. 2005], another semantic service description language with equivalent capabilities to OWL-S. WS-BPEL also comes with additional extensions towards covering service choreographies and human tasks. However, this language does not cover non-functional aspects.

SoaML [Amsden et al. 2012] is a UML-based language for specifying Service-Oriented Architectures (SOAs) able to define components and their inter-relationships at the business and service levels. However, SoaML cannot actually describe service orchestrations and to this end, it needs to be complemented by a service orchestration language, like WS-BPEL. Moreover, it does not cover non-functional aspects.

Concerning the non-functional description of services, various approaches have been proposed which can be distinguished according to their performance on various comparison criteria, such as their expressiveness, complexity, formality and extensiveness [Kritikos et al. 2013]. Among these approaches, OWL-Q can be considered as the one which has the best performance across all these criteria. OWL-Q is semantic, extensible and very expressive, covering all major aspects in service quality description, including quality attributes, metrics, units, and measurement formulas. It is able to specify both service quality models that can be used for populating SLAs as well as quality-based service descriptions covering the non-functional profiles of services. Finally, OWL-Q can be connected to any functional service description language to enable the complete description of web services.

7.1.2 CLOUD SERVICE/APPLICATION MODELLING

TOSCA is considered as a de-facto standard for the deployment description of applications and seems to be widely used in research prototypes. However, it has particular shortcomings related to the non-coverage of the instance level which is required for dealing with runtime aspects, the lack of domain/cloud-specific constructs and the incomplete coverage of the non-functional aspects.

CAMEL is a multi-purpose DSL developed in the context of the PaaSage European project. It is able to cover many aspects related to the lifecycle management of cloud applications, such as the deployment, monitoring, scalability, organisation, security and cloud service ones. In terms of deployment, it relies on CloudML [Ferry et al. 2013] which follows the template-instance pattern and caters for runtime aspects through a models@runtime approach [Aßmann et al. 2011]. However, the latter DSL does not cover all types of placement constraints and is

mainly oriented towards IaaS services. The description of cloud services relies on the Saloon framework's [Quinton et al. 2013] generic DSL able to cover any possible cloud service offering. The main drawback of CAMEL as a whole is that it is semi-formal as it is Ecore-based. Thus, it does not have the formality level and reasoning capabilities of an ontology-based modelling approach.

The language in [Nguyen et al. 2011] is able to support the semi-formal description of Blueprint Templates which cover cloud-offerings at multiple abstraction levels and capture service capability, virtual topology and QoS & policy aspects. Apart from being a semi-formal language, it does not capture all lifecycle aspects as covered, e.g., in CAMEL. In addition, it is not capable of defining the quality terms required for specifying the quality capabilities of the respective service offerings.

A cloud meta-model is proposed in [Galán et al. 2009] which extends OVF[6] towards covering self-configuration, elasticity and performance monitoring. This meta-model is not capable of specifying placement constraints, component dependencies and quality capabilities and requirements.

Another OVF extension called service manifest has been proposed in [Rumpl et al. 2010] which covers placement and allocation constraints, security requirements and performance profiles according to the properties of trust, reputation, eco-efficiency and cost. This service manifest, however, stays mainly at the IaaS level and is not capable of describing component dependencies while does not cover additional quality attributes related, e.g., to performance.

The mOSAIC ontology [Moscato et al. 2011] has been developed in OWL and can be used for the semantic annotation of semi-formal cloud service descriptions. It is rich enough to cover various aspects, including cloud service requirements and resources, metrics, SLAs, components and policies.

7.2 Service Matchmaking

7.2.1 Software Service Matchmaking

The functional matchmaking of software services has mainly relied on the service I/O. The approaches proposed rely on employing either Information Retrieval [Dong et al. 2004] or ontology-based [Paolucci et al. 2002] or both types of techniques [Plebani & Pernici 2009]. While the latter two types of approaches seem to cater for better accuracy, they do not consider the service behavior. As such, the results produced will never be as accurate as possible. To remedy for this, few approaches [Sycara et al. 2002] have employed behavior-based service matching by relying on full input-output-precondition-effects (IOPE) service profiles. While these approaches reach even higher matchmaking accuracy levels have the main drawback that full IOPE service profiles do not exist in reality and require additional modelling effort by the cloud service provider.

Non-functional software service matching approaches can be separated into constraint-based, ontology-based or mixed. Constraint-based approaches [Cortés et al. 2005] express the quality description of service requirements and capabilities as a constraint model and then employ constraint solving techniques and particular matching metrics (e.g., subsumption [Cortés et al. 2005]) in order to perform the service matchmaking. These approaches assume that the quality-based service specification comprise terms which have been defined in a common quality term repository. Ontology-based approaches [Zhou et al. 2004] provide ontology languages for enabling the service quality description and exploit subsumption reasoning to infer whether service quality capabilities match the respective requirements posed. Such approaches have the drawback that can only involve the processing of unary quality-based service specifications which comprise one quality term per constraint. Finally, mixed approaches [Kritikos & Plexousakis 2014] combine the best from both worlds with respect to the previous approach types. They rely on ontologies to enable the service quality description, they then align the descriptions according to the quality terms and finally transform the aligned descriptions into constraint problems in order to use the techniques in the first approach type to perform the actual service matchmaking. The latter types of approaches exhibit better accuracy than the other types and are also able to cope with n-ary quality service specifications.

7.2.2 Cloud Service Matchmaking

Ruiz-Alvarez & Humphrey [2011] have proposed an approach which is able to discover cloud storage services that match the respective application requirements posed. Cloud storage requirements and capabilities are expressed via a semi-formal language. The framework in [Garg et al.] ranks cloud services according to their quality performance and weights given to each quality term according to the AHP process. Zeng et al. [2009] follow a Wordnet-based approach to measure the similarity of concepts in the I/O of cloud service specifications. The federated cloud environment in [Buyya et al. 2010] is able to match user quality requirements to cloud services. D' Andria et al. [2012] propose a PaaS matchmaking and selection framework which filters PaaS according to user requirements and ranks the remaining PaaS based on the number of user preferences satisfied. The approach in [García-Gómez et al. 2012] addresses blueprint (see Sub-section 7.1.2) matchmaking and is able to produce a composite blueprint document which comprises the cloud services that can be used for realising or supporting a cloud application. It does not propose though a concrete cloud

[6] https://www.dmtf.org/standards/ovf

service composition but just set of alternatives for different application elements at different levels of abstraction.

7.3 Service Composition

7.3.1 Software Service Composition

The successful SOA paradigm has led to a proliferation of available services. Such services can then be optimally combined to produce added-value functionality incarnated into respective applications. To this end, various service composition approaches have been proposed which usually focus either on the functional or QoS aspect. Most of the QoS-based work follows either a statistical [Canfora et al. 2005] or path-based approach [Ardagna & Pernici 2007] leading to an over-simplification or a pessimistic view of the problem. Some approaches employ a heuristic [Yu et al. 2007] or a QoS decomposition [Alrifai et al. 2009] approach to cater for better performance but sacrificing optimality. In addition, all these approaches regard QoS service offerings as simple QoS parameter values which is quite unrealistic if we also regard that many services run in quite dynamic environments. Moreover, these approaches fail to produce any result for over-constrained end-user requirements. One promising approach resolving most of the above issues was proposed in [Ferreira et al. 2009]. Some key aspects of this approach were exploited in our cloud service composition work.

7.3.2 Cloud Service Composition

The cloud service composition problem is harder than that of service selection as it involves composing different types of services with different characteristics and the synthesis is performed in different but inter-dependent levels such that the solution at one level impacts the solution at other levels. However, the cloud service composition approaches proposed usually focus on just one cloud service type. Even when they consider additional types, they either solve a limited case of the actual problem or a slightly different problem by also neglecting all possible user requirement types.

Concerning SaaS composition, the respective approaches can be separated into those which: (a) consider semantics [Zeng et al. 2009], (b) use heuristics to solve the respective optimization problem [Kofler et al. 2010], (c) address multi-tenant SaaS [He et al. 2012], (d) exploit feature models and multi-criteria decision making [Wittern et al. 2012] to find the most optimal SaaS compositions and (e) consider some other aspects, such as the network latency and the multiple instances that a particular SaaS service can have [Klein et al. 2012]. Although not clearly addressing IaaS services, the latter approach seems interesting and could be used for further extending our proposed work towards selecting only the appropriate instances for each SaaS selected.

The self-organizing agent-based cloud service composition method in [Gutierrez-Garcia et al. 2013] exploits distributed problem solving techniques, by also relying on the contract-net protocol, and is able to produce vertical, horizontal, one-time and persistent service compositions. Both SaaS and IaaS type of services are handled. However, this approach seems to cater only for functional and cost requirements.

In [Karim et al. 2013], an hierarchical quality model is proposed going from user requirements down to the QoS capabilities of IaaS services. This quality model is then used for ranking the service candidates across the different cloud levels. However, the ranking algorithm proposed seems to work on a different problem type where the end-user requires one or more SaaS services and then the providers of these services have to find suitable IaaS offerings for hosting their services. In addition, this algorithm does not consider placement constraints, while only low-level security requirements are taken into account. Finally, the algorithm seems to work only for sequential application workflow specifications.

8. Conclusions & Future Work

This article has presented a semantic cloud service composition framework which is able to address the whole lifecycle of an application in a multi-cloud environment. The genuine features of this framework is that it relies on semantics and constraint optimisation techniques which guarantee the quality of the cloud service composition derived. It is able to consider different types of requirements and different types of design choices across different levels for which their dependencies are accounted for. This framework is also able to dynamically adapt the application at runtime via exploiting adaptation rules as well as re-configuration opportunities provided by the *Cloud Service Composer* when quite critical situations occurs which cannot be addressed by such rules.

The framework's Cloud Service Composer advances the state-of-the-art as apart from considering a variety of different types of quite meaningful and critical application requirements, it is able to concurrently consider different cloud levels in order to produce the final cloud service composition product. Moreover, it does not sacrifice accuracy through the use of any kind of heuristics. This guarantees the optimality of the composition product and reduces the probability that such product has to be adapted at runtime. In addition, as shown from the experimental evaluation, it leads to a reduction in the composition execution time compared to common IaaS composition approaches, thus making the composition algorithm suitable for use even at runtime.

The following research directions are planned. First, a complete evaluation of the whole framework has to take

place. Second, the exploitation of PaaS services needs to be considered in order to make the solution even more complete with respect to the types of cloud services that can be exploited. Third, a complete UI spanning all the application lifecycle phases will be produced enabling application developers not only to pose requirements but also see the deployment progress and runtime performance of their applications and possibly interfere by e.g., enforcing adaptation actions or changing requirement models or plans. Fourth, it is planned to extend the functionality of the framework in order to deal with an additional level on the top mapping to the handling of business processes in order to realise the vision of BPaaS [Woitsch & Utz 2015].

9. ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 317715 (PaaSage) and from the European Community's Framework Programme for Research and Innovation HORIZON 2020 (ICT-07-2014) under grant agreement number 644690 (CloudSocket).

10. REFERENCES

Alrifai, M., Risse, T. (2009). Combining Global Optimization with Local Selection for Efficient QoS-Aware Service Composition. In *WWW*, 881–890.

Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guzar, A., Kartha, N., Kevin, C., Khalaf, R., Knig, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A. (2007). Web Services Business Process Execution Language. Technical report, Organization for the Advancement of Structured Information Standards (OASIS). Retrieved October 20, 2015 from http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf

Amsden, J., Athanasopoulos, G., Badr, I., Bauer, B., Belaunde, M., Benguria, G., Berre, A. J., Butler, J., Casanave, C., Covington, B., Cummins, F., Desfray, P., Ditze, A., Estefan, J., Fischer, K., Hahn, C., ystein Haugen, Hinton, P., Kolk, H., Larrucea, X., Lenoir, J., Lonjon, A., Mansour, S., Miyazaki, H., Mukerji, J., Odell, J., Pantazoglou, M., Rivett, P., Roman, D., Rosen, M., Roser, S., Shafiq, O., Seidewitz, E., Selic, B., Tsalgatidou, A., Hussey, K., Mervine, F. (2012). Service oriented architecture Modeling Language (SoaML). Technical report, Object Management Group (OMG). Retrieved October 20, 2015 from http://www.omg.org/spec/SoaML/1.0.1/PDF

Ardagna, D., Pernici, B. (2007). Adaptive Service Composition in Flexible Processes. *IEEE Transactions on Software Engineering*, 3(6), 369–384.

Aßmann, U., Bencome, N., Cheng, B. H. C., France, R. B. (2011). Models@run.time (dagstuhl seminar 11481). Technical Report 11, Dagstuhl Reports.

Baryannis, G., Garefalakis, P., Kritikos, K., Magoutis, K., Papaioannou, A., Plexousakis, D., Zeginis, C. (2013). Lifecycle Management of Service-based Applications on Multi-Clouds: A Research Roadmap. In *MultiCloud*.

Buyya, R., Ranjan, R., Calheiros, R. N. (2010). InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In *ICA3PP*, 13–31.

Canfora, G., Penta, M. D., Esposito, R., Villani, M. (2005). QoS-Aware Replanning of Composite Web Services. In *ICWS*, 121–129.

Cardoso, J., Binz, T., Breitenbücher, U., Kopp, O., Leymann, F. (2013). Cloud computing automation: Integrating USDL and TOSCA. In *CAiSE*, 1–16.

Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. (2001). Web Services Description Language (WSDL) 1.1. W3C, http://www.w3.org/TR/wsdl.

Cortés, A. R., Martín-Díaz, O., Toro, A. D., Toro, M. (2005). Improving the Automatic Procurement of Web Services Using Constraint Programming. *Int. J. Cooperative Inf. Syst.*, 14(4), 439–468.

D'Andria, F., Gorro˜nogoitia Cruz, J., Ahtes, J., Bocconi, S., Zeginis, D. (2012). Cloud4SOA: Multi-Cloud Application Management Across PaaS Offerings. In *MICAS*.

de Bruijn, J., Bussler, C., Domingue, J., Fensel, D., Hepp, M., Keller, U., Kifer, M., Knig-Ries, B., Kopecky, J., Lara, R., Lausen, H., Oren, E., Polleres, A., Roman, D., Scicluna, J., Stollberg, M. (2005). Web Service Modeling Language (WSMO). Technical report, World Wide Web Consortium (W3C). Retrieved September 12, 2015 from http://www.w3.org/Submission/WSMO/.

Domaschka, J., Baur, D., Seybold, D., Griesinger, F. (2015). Cloudiator: A Cross-Cloud, Multi-Tenant Deployment and Runtime Engine. In *9th Symposium and Summer School on Service-Oriented Computing*.

Domaschka, J., Hoppe, D., Kritikos, K., Sheridan, C., Yaqub, E., Baur, D., Griesinger, F., Seybold, D., Balis, B., Krol, D., Malawski, M., Zarioh, A. (2015). D5.1.2: Product Executionware. Paasage project deliverable.

Dong, X., Halevy, A., Madhavan, J., Nemes, E., Zhang, J. (2004). Similarity search for web services. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, 372–383.

Ferreira, A. M., Kritikos, K., Pernici, B. (2009). Energy-Aware Design of Service-Based Applications. In *ICSOC*.

Ferry, N., Chauvel, F., Rossini, A., Morin, B., Solberg, A. (2013). Managing multi-cloud systems with CloudMF. In *Solberg, A., Babar, M. A., Dumas, M., and Cuesta, C. E., editors, NordiCloud 2013: 2nd Nordic Symposium on Cloud Computing and Internet Technologies*, 38–45.

Galán, F., Vaquero, L. M., Clayman, S., Toffetti, G., Henriksson, D. (2009). Deliverable D4.1, D4.2 and D4.3 – Scientific Report. Reservoir project deliverable.

García-Gómez, S., Jiménez-Gañán, M., Taher, Y., Momm, C., Junker, F., Bíró, J., Menychtas, A., Andrikopoulos, V., Strauch, S. (2012). Challenges for the comprehensive management of Cloud Services in a PaaS framework. *Scalable Computing: Practice and Experience*, 13.

Garg, S. K., Versteeg, S., Buyya, R. (2011). SMICloud: A Framework for Comparing and Ranking Cloud Services. In *UCC*.

Gutierrez-Garcia, J. O. Sim, K. (2013). Agent-based cloud service composition. *Applied Intelligence*, 38, 436–464.

He, Q., Han, J., Yang, Y., Grundy, J., Jin, H. (2012). QoS-Driven Service Selection for Multi-Tenant SaaS. In *Cloud*, 566–573.

Hwang, C., Yoon, K. (1981). Multiple Criteria Decision Making. *Lect. Notes Econ. Math.*

Jula, A., Sundararajan, E., Othman, Z. (2014). Review: Cloud computing service composition: A systematic literature review. *Expert Syst. Appl.*, 41(8), 3809–3824.

Karim, R., Ding, C., Miri, A. (2013). An end-to-end qos mapping approach for cloud service selection. In *SERVICES*, 341–348.

Klein, A., Ishikawa, F., Honiden, S. (2012). Towards network-aware service composition in the cloud. In *WWW*.

Klusch, M., Fries, B., Sycara, K. (2006). Automated semantic web service discovery with OWLS-MX. In *AAMAS*, 915–922.

Kofler, K., Haq, I. U., Schikuta, E. (2010). User-centric, heuristic optimization of service composition in clouds. In *EuroPar*, 405–417.

Kritikos, K., Domaschka, J., Rossini, A. (2014). SRL: A Scalability Rule Language for Multi-cloud Environments. In *CloudCom*, 1-9.

Kritikos, K., Pernici, B., Plebani, P., Cappiello, C., Comuzzi, M., Benbernou, S., Brandic, I., Kertesz, A., Parkin, M., Carro, M. (2013). A Survey on Service Quality Description. *ACM Computing Surveys*, 46(1).

Kritikos, K., Plexousakis, D. (2006). Semantic QoS Metric Matching. In *ECOWS*, 265–274.

Kritikos, K. Plexousakis, D. (2014). Novel Optimal and Scalable Nonfunctional Service Matchmaking Techniques. *IEEE T. Services Computing*, 7(4), 614–627.

Milano, M. (2003). *Constraint and Integer Programming: Toward a Unified Methodology*. Kluwer Academic Publishers, Norwell, MA, USA.

Moscato, F., Aversa, R., Martino, B. D., Fortis, T., Munteanu, V. I. (2011). An Analysis of mOSAIC ontology for Cloud Resources annotation. In *Federated Conference on Computer Science and Information Systems - FedCSIS*, 973–980.

Nguyen, D. K., Lelli, F., Taher, Y., Parkin, M., Papazoglou, M. P., van den Heuvel, W. (2011). Blueprint Template Support for Engineering Cloud-Based Services. In *ServiceWave*, 26–37.

Palma, D., Spatzier, T. (2013). Topology and Orchestration Specification for Cloud Applications (TOSCA). Technical report, Organization for the Advancement of Structured Information Standards (OASIS). Retrieved October 20, 2015 from http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.pdf.

Pannetrat, A. (2013). D2.1: Security-aware SLA specification language and cloud security dependency model. Cumulus project deliverable.

Paolucci, M., Kawamura, T., Payne, T. R., Sycara, K. P. (2002). Semantic Matching of Web Services Capabilities. In *ISWC '02: Proceedings of the First International Semantic Web Conference on the Semantic Web*, 333–347.

Pedrinaci, C., Cardoso, J., Leidig, T. (2014). Linked USDL: A vocabulary for web-scale service trading. In *ESWC*, 68–82.

Plebani., P., Pernici, B. (2009). URBE: Web Service Retrieval Based on Similarity Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 21(11), 1629-1642.

Quinton, C., Romero, D., Duchien, L. (2013). Cardinality-based feature models with constraints: a pragmatic approach. In *Kishi, T., Jarzabek, S., and Gnesi, S., editors, SPLC 2013: 17th International Software Product Line Conference*, 162–166.

Rossi, F., van Beek, P., Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA.

Rossini, A., Nikolov, N., Romero, D., Domaschka, J., Kritikos, K., Kirkham, T., Solberg, A. (2014). D2.1.2 – CloudML Implementation Documentation (First version). PaaSage project deliverable.

Ruiz-Alvarez, A. Humphrey, M. (2011). An Automated Approach to Cloud Storage Service Selection. In *ScienceCloud*.

Rumpl, A., Rasheed, H., Waeldrich, O., Ziegler, W. (2010). Service Manifest: Scientific Report. Optimis project deliverable.

Saati, T. (1980). *The Analytic Hierarchy Process*. McGraw-Hill.

Sycara, K., Wido, S., Klusch, M., LU, J. (2002). Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *J. Auton. Agents and Multi-Agent Syst.*, 5, 173–203.

Sycara, K. et al. (2003). OWL-S 1.0 Release. OWL-S Coalition, http://www.daml.org/services/owl-s/1.0/.

Timpe, C. (2002). Solving planning and scheduling problems with combined integer and constraint programming. *OR Spectrum*, 24(4), 431–448.

Wittern, E., Kuhlenkamp, J., Menzel, M. (2012). Cloud service selection based on variability modeling. In *ICSOC*, 127–141.

Woitsch, R., Utz, W. (2015). Business Process as a Service - Model Based Business and IT Cloud Alignment as a Cloud Offering. In *ES*.

Xiong, P., Pu, C., Zhu, X., Griffith, R. (2013). vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *ICPE*, 271–282.

Yu, T., Zhang, Y., Lin, K.-J. (2007). Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web*, 1(1).

Zeginis, C., Kritikos, K., Plexousakis, D. (2015). Event Pattern Discovery in Multi-Cloud Service-Based Applications. *International Journal of Systems and Service-Oriented Engineering*, 5(4), 78-103.

Zeng, C., Guo, X., Ou, W., Han, D. (2009). Cloud computing service composition and search based on semantic. In *CloudCom*, 290–300.

Zhou, C., Chia, L.-T., Lee, B.-S. (2004). DAML-QoS Ontology for Web Services. In *ICWS*, 472– 479.

Authors

Kyriakos Kritikos received his B.Sc., M.Sc., and Ph.D degrees in Computer Science from the University of Crete. He was a Post-Doc Researcher at Politecnico di Milano and CNR in Italy as well as FNR in Luxembourg. He is currently a Researcher at the Information Systems Laboratory of the Institute of Computer Science, FORTH in Greece. His research interests span the following areas: Quality-aware service management; Cross-layer service monitoring and adaptation; Cloud-based application modelling and deployment; Ontology modeling and reasoning; Constraint and Mathematical Programming; Distributed (Information) Systems.

Dimitrios Plexousakis is a Professor at the Computer Science Department, University of Crete and a Researcher as well as the head of the Information Systems Laboratory of the Institute of Computer Science, FORTH in Greece. He received his B.Sc. degree in Computer Science from University of Crete and M.Sc. and Ph.D degrees in Computer Science from the University of Toronto. His research interests span the following areas: Knowledge Representation and Knowledge Base Design; Formal knowledge representation models and query languages for the Semantic Web; Formal reasoning systems with focus on dynamic action theories and belief revision; Business process and e-service modeling, discovery and composition. He is a member of ACM and IEEE.