Multi-Cloud Application Design through Cloud Service Composition

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

***Abstract*—While various platforms are offering facilities for single-cloud application design, deployment and provisioning, there is a need to move to multiple clouds in order to achieve cost-effectiveness and avoid vendor lock-in. Apart from not supporting multi-cloud application management, many platforms usually focus on the deployment and provisioning phases of the cloud-based application lifecycle by neglecting the design phase. However, the design selection of the best possible cloud service composition affects the provisioning phase, as the more distant from optimality is the selected solution, the more adaptation actions will be enacted. To this end, there is a high need for cloud application design tools and methods which can select the best possible cloud service composition based on user requirements. This paper satisfies this need by proposing a cloud service composition approach able to optimally compose different types of cloud services by simultaneously satisfying various types of user requirements. These types, not concurrently supported by any cloud application design tool, include quality, deployment, security, placement and cost requirements. Moreover, the proposed approach addresses a particular design choice type not currently considered in literature.**

I. Introduction

As Cloud Computing promises the cost-effective and on-demand use of resources to support application execution and provisioning, it has been embraced by the application design and development community. As such, various applications and business processes are currently designed and ported to the cloud. To support moving towards the cloud, various commercial and open-source platforms promise the rapid development and management of cloud applications.

However, such platforms are limited to managing single-cloud applications. Sometimes, they also do not enable switching to other cloud providers as the application code is hardwired in exploiting a specific cloud provider API. To exploit the cloud's real benefits, avoid vendor lock-in and achieve cost-effectiveness, the move to multi-clouds must be supported [1]. Multi-cloud applications, in this way, will be capable of exploiting those cloud services which better suit their requirements by also respecting cost constraints independently from which cloud provider offers them.

To this end, there is now a trend towards supporting multi-cloud applications. This can be derived by observing that some cloud platforms, driven by real user needs, support hybrid-cloud deployments. This is also reflected in the European Commission's priorities in research as well as the acceptance of particular European projects aiming at supporting the whole multi-cloud application management lifecycle, such as PaaSage (http://www.paasage.eu) and ModaClouds (http://www.modaclouds.eu). The current commercial and research offerings address well some lifecycle management activities, including application deployment, provisioning, and adaptation, but not the design and development ones. As such, they cannot be used for selecting software-as-a-service (SaaS) solutions to realize application components. They also tend to neglect all possible end-user requirements, such as security or high-level quality of service (QoS) requirements, but focus more on low-level resource ones. They finally do not currently support other requirements types concerning, e.g., the co-location of application components in the same virtual machine (VM) or cloud.

Concerning application design, the cloud service composition approaches proposed can compose one or more types of cloud services. Such approaches usually exploit optimization techniques and consider both functional and QoS requirements. However, they seem to neglect other requirements types, such as component placement and security requirements. In addition, they do not consider some alternative design choices which can well happen in real situations. In particular, during application design, there can be trade-offs on the application level QoS and cost between using and deploying an internal service or exploiting an external service with no maintenance or deployment costs.

To remedy for the above limitations, this paper presents a cloud service composition approach for multi-cloud applications able to find the most optimal solution such that all end-user requirements of any type are satisfied by also considering all possible design choices. The types of requirements considered include: (a) high- and low-level security requirements, (b) high- and low-level QoS requirements, (c) resource requirements and (d) component (co-)location requirements. These requirements are structured in an abstract deployment model, which is concretized by our approach by also considering the set of cloud provider offerings and internal SaaS realizations of the requester organisation.

This approach was realized by specifying an optimization problem which is subsequently solved via using constraint satisfaction optimization problem (CSOP) solving techniques [2]. It has also been applied in a particular use case to highlight its main benefits over the state-of-

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

Table I: The components mapping to application tasks

the-art. The approach experimental evaluation shows that its performance is satisfactory and appropriate in realistic circumstances.

The rest of the paper is structured as follows. Section 2 provides a use case used to highlight the main benefits of the proposed approach which is analyzed in Section 3 and experimentally evaluated in Section 4. Related work is analyzed in Section 5. Finally, the last section concludes the paper and draws directions for further research.

II. Use Case

This case concerns designing a real-world traffic management application [1] that regulates traffic at particular areas of a city. This application comprises three main tasks which are analyzed below:

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

To realize this application, various software components/services have been developed or are required, where either one or more map to a particular task. Moreover, a service oriented architecture (SOA) is chosen to realize the application tasks, such that some components need to be hosted on particular servlet containers. Table I 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 or just downloaded) or exploiting a particular external service. This choice is indicated in parenthesis after the respective component's name.

Concerning service performance and cost, Table II indicates the respective offerings along with information on which cloud provider offers them, when they are external. The internal services performance was determined via benchmarking which lead to the eventual VM requirements for them (given later on in this section). In addition, these

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

Table II: The QoS and cost features of the services

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 cost model of the providers.

The end-user also requires the satisfaction of requirements of the following types for his/her application:

- 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 not be more than 380 $ per month.
- Quality requirements:
  - Duration of the whole application 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

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

Table III: The offerings of the four cloud providers

detection) and SEF-05 (monitoring & quantification of security incident type, volume and cost).
- Meantime between incidents [3] should be 6 months ($mti \geq 6$)
- 99% of critical incidents are reported within 4 hours ($ir\,(99\%) \leq 4$)

A "high" VM requirement maps to at least 4 cores, 4 GBs of RAM and 40 GBs of hard disk, a "medium" VM requirement maps to at least 2 cores, 2GBs of RAM and 20 GBs of hard disk and a "small" VM requirement maps to 1 core, 2 GBs of RAM and 10 GBs of hard disk.

Let us now consider four cloud providers, namely CP1, CP2, CP3 and CP4. These providers offer particular cloud services/VMs and realize a particular set of security controls. Table III 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. Please consider that real values for VM characteristics and cost were considered by collecting them from cloud provider web pages. Thus, 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 such 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 will 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 by just selecting a IaaS services composition. The outcome of such an approach would be a solution mapping application components to the following VMs (i) *AC* + *DC* $\rightarrow$ CP3 (C), (ii) *MC* $\rightarrow$ CP3 (B) and (iii) *HO* + *TCC* $\rightarrow$ CP3 (B). 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.

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

Table IV: Cost, QoS and security features of the solutions

Our approach, as considers all possible information, will lead to selecting a better solution all user requirements, including the security ones, which 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 a solution produced by our approach not accounting the security requirements, identical to the solution of the common service composition approach with the sole exception that *MC* and *HO* are not mapped to SaaS services. Table IV summarizes the QoS, cost and security features of the three solutions.

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 left out the remaining cloud providers as they do not satisfy the user security constraints: CP4 violates the security SLO for mean time between failure while CP3 has not realized the DSI-01 and SEF-05 security controls.

III. Cloud Service Composition Approach

To cater for all possible design choice alternatives, we model out of the existing cloud service space and end-user requirements posed a particular optimization problem which, when solved, can discover the most optimal solution satisfying all user requirements posed irrespectively of their type. The approach followed was inspired by the service concretization work in [4]. 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.

$$uf_q(x) = \begin{cases} a_q + \frac{v_q^{max} - x}{v_q^{max} - v_q^{min}} \cdot (1 - a_q), & v_q^{min} \leq x \leq v_q^{max} \\ \mathrm{m}\left(a_q - \frac{v_q^{min} - x}{v_q^{max} - v_q^{min}} \cdot (1 - a_q), 0\right), & x < v_q^{min} \\ \mathrm{m}\left(a_q - \frac{x - v_q^{max}}{v_q^{max} - v_q^{min}} \cdot (1 - a_q), 0\right), & x > v_q^{max} \end{cases} \tag{2}$$

$$uf_q(x) = \begin{cases} a_q + \frac{x - v_q^{min}}{v_q^{max} - v_q^{min}} \cdot (1 - a_q), & v_q^{min} \leq x \leq v_q^{max} \\ \mathrm{m}\left(a_q - \frac{v_q^{min} - x}{v_q^{max} - v_q^{min}} \cdot (1 - a_q), 0\right), & x < v_q^{min} \\ \mathrm{m}\left(a_q - \frac{x - v_q^{max}}{v_q^{max} - v_q^{min}} \cdot (1 - a_q), 0\right), & x > v_q^{max} \end{cases} \tag{3}$$

A. *Cloud Service Composition Problem Formulation*

To formulate the optimization objective of the problem, we rely on the Analytical Hierarchy Process (AHP) [5] to derive the relative importance of the quality parameters and cost to the end-user. The result of this process is an assignment of weights to all of these parameters, indicating their relative importance, whose sum should equal to one. We also follow Simple Additive Weighting (SAW) technique [6] 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:

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

The utility function of each parameter is formulated based on the formulas in [4] which cater for slightly violating some problem constraints to address over-constrained user requirements. Two similar formulas (see Equations (2-3)) can be expressed, depending 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.

The value v_q that a parameter q can take depends on the type of parameter and its derivation can be application-specific. To this end, in the general case, we consider that a particular user-specified function is provided taking as input the respective parameter values of the application components. In other terms:

$$val_q = f_q(val_i^q) \tag{4}$$

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

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 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 so as to be able to specify co-location constraints.

It is apparent that two cases exist for each 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 either the condition for the first case should hold or that only one external SaaS must be selected for realizing the component. Both cases can lead to requiring the satisfaction of the following constraint:

$$\sum_j \sum_k x_{ijk} + \sum_l z_{il} = 1 \tag{5}$$

While this is obvious for the second case, it is also true for the first if we regard that y_i is fixed to be one and z_{il} to be zero for this case.

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

$$val_i^q = y_i * f_i^q(core_i, mem_i, store_i) + (1 - y_i) * \left(\sum_l z_{il} * val_{il}^q \right) \tag{6}$$

where f_i^q is the function over the resources for parameter q of component i while val_{il}^q is the parameter value for the l external SaaS of component i. The above assumption is valid even for the first case (internal service deployed on the cloud) if we consider that the usual way of deriving high-level requirements is either via benchmarking, simulation, or performance model learning [7] 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 rathen 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. This maps to the following three formulas:

$$core_i = \sum_{jk} x_{ijk} * core_{jk} \tag{7}$$

$$mem_i = \sum_{jk} x_{ijk} * mem_{jk} \tag{8}$$

$$store_i = \sum_{jk} x_{ijk} * store_{jk} \tag{9}$$

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:

$$cost_i = y_i * \sum_{jk} x_{ijk} * cost_{jk} + (1 - y_i) * \sum_{l} z_{il} * cost_{il} \tag{10}$$

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.

We provide a specific formulation for co-location constraints depending on their type. Two main types are considered: (a) two components must be co-located in the same VM and (b) they must be co-located at the same cloud. For each type, there is also the opposite case of requiring not co-locating two components. We consider that while the need for the first type is obvious, the second type is needed in cases where there is significant communication between two components but we do not need to co-locate them in the same VM due to interference issues. Thus, it is better to have both components at the same cloud where a high-communication bandwidth is quaranteed.

The first type of (positive) co-location constraint is formulated as follows:

$$x_{ijk} = x_{i'jk} \tag{11}$$

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. The negative case is expressed as follows:

$$if\ (x_{ijk} == 1) \Rightarrow x_{i'jk} = 0 \tag{12}$$

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 second type of (positive) co-location constraint is formulated as follows:

$$\sum_{k} x_{ijk} = \sum_{k} x_{i'jk} \tag{13}$$

where i and $i^{'}$ are the two components for which the co-location 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 negative case can be expressed as follows:

$$if\ \left(\sum_{k} x_{ijk} == 1\right) \Rightarrow \sum_{k} x_{i'jk} = 0 \tag{14}$$

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:

$$\begin{aligned} &\text{if } \left(y_i \wedge \sum_{k} x_{ijk} == 1\right) \Longrightarrow cc - ccp_j = \emptyset \\ &\text{else if } \left(\not y_i \wedge z_{il}\right) \Longrightarrow cc - ccp_{z_{il}.provider} = \emptyset \end{aligned} \tag{15}$$

where cc is a fixed variable set mapping to all required security controls, ccp_j is a fixed variable set 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:

$$\text{if } \left(y_i \wedge \sum_k x_{ijk} == 1\right) \implies seq^p_j \geq seq^p$$
$$\text{else if } (\not y_i \wedge z_{il}) \implies seq^p_{z_{il}.provider} \geq sec^p \qquad (16)$$

where seq^p is the low required threshold for security property p while seq^p_j 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 for both security property types must be enforced.

B. *Complexity & Solving Technologies*

The common cloud service composition problem is NP-Hard [8]. 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 [9], [10].

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 that can provide insight in case of over-constrained requirements on 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.

IV. 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.

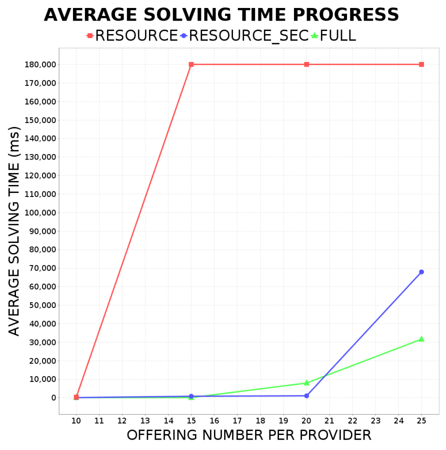

Figure 1: Avg. solving time per offer number

The evaluation metric was the average solving time whose value was generated over 30 runs in order to minimize various types of interference in the measurement, 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 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 $\rightarrow$ 5, cloud provider number $\rightarrow$ 10, IaaS/SaaS number per provider $\rightarrow$

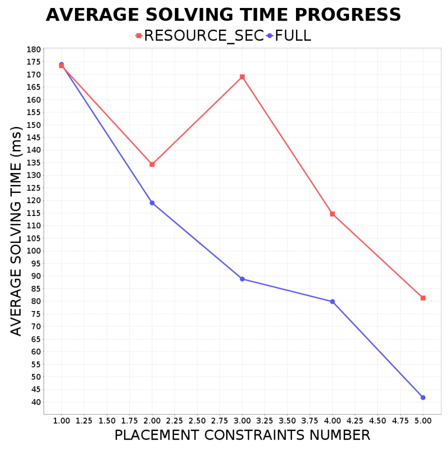

Figure 2: Avg. solving time per placement constraint number

5 and placement constraint number $\rightarrow$ 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 Fig. 1. 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. Please note that we have posed a limit of 3 minutes to the solving time so as to be acceptable by the 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). Fig. 2 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 high-level constraints.

V. Related Work

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 [11] or path-based approach [12] leading to an over-simplification or a pessimistic view of the problem. Some approaches employ a heuristic [13] or a QoS decomposition [14] 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 [4]. Some key aspects of this approach were exploited in our cloud service composition work.

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 the other level. 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 end-user requirement types.

Concerning SaaS composition, the respective approaches can be separated into those which: (a) consider semantics [15], (b) use heuristics to solve the respective optimization problem [16], (c) address multi-tenant SaaS [17], (d) exploit feature models and multi-criteria decision making [18] 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 [19]. 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 [20] 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 [21], an hierarchical quality model is proposed going from end-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.

VI. Conclusions

In this paper, we have presented and analyzed a constraint problem modelling approach aiming at completely capturing and solving the cloud service composition problem in cloud application design by considering all types of requirements that may be posed by a specific end-user as well as alternative design choices. Due to the features of the constraint problem specification, we also justified the need of combining the constraint solving forces of two main techniques: CSOP and Constraint Programming with interval arithmetics. Such a need was realized by exploiting a particular well-supported constraint solving engine called Choco which exploits in the background another engine, called Ibex, for solving constraint problems with real variables.

A particular use case validated our approach indicating that is not only feasible but provides better results than those offered by the state-of-the-art. The results of our approach experimental evaluated also show that it has better performance than state-of-the-art IaaS composition work.

Our work can be extended as follows. First, we need to more thoroughly evaluate our approach in terms of performance and scalability. Second, in case when performance needs additional improvements, we have to explore other alternatives, such as the use of heuristics to reduce the complexity of the problem to be solved or techniques to decompose the global quality and cost constraints of the problem. Third, we have to explore other cost models which might require either changing some of the problem constraints or moving towards proposing different problem variants catering for addressing different provider cost models. Fourth, we will explore integrating our solution with a cloud service discovery solution in order to produce a complete design framework for multi-cloud applications.

Acknowledgement

We acknowledge the support of PaaSage (FP7-317715) & CloudSocket (Horizon2020-644690) EU projects.

References

[1] G. Baryannis, P. Garefalakis, K. Kritikos, K. Magoutis, A. Papaioannou, D. Plexousakis, and C. Zeginis, “Lifecycle Management of Service-based Applications on Multi-Clouds: A Research Roadmap,” in *MultiCloud*, 2013.

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

[3] A. Pannetrat, “D2.1: Security-aware SLA specification language and cloud security dependency model,” Cumulus Project Deliverable, 2013.

[4] A. M. Ferreira, K. Kritikos, and B. Pernici, “Energy-Aware Design of Service-Based Applications,” in *ICSOC*, ser. LNCS. Springer, 2009.

[5] T. Saati, *The Analytic Hierarchy Process*. McGraw-Hill, 1980.

[6] C. Hwang and K. Yoon, “Multiple Criteria Decision Making,” *Lect. Notes Econ. Math.*, 1981.

[7] P. Xiong, C. Pu, X. Zhu, and R. Griffith, “vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments,” in *ICPE*. New York, NY, USA: ACM, 2013, pp. 271–282.

[8] A. Jula, E. Sundararajan, and Z. Othman, “Review: Cloud computing service composition: A systematic literature review,” *Expert Syst. Appl.*, vol. 41, no. 8, pp. 3809–3824, 2014.

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

[10] C. Timpe, “Solving planning and scheduling problems with combined integer and constraint programming,” *OR Spectrum*, vol. 24, no. 4, pp. 431–448, 2002.

[11] G. Canfora, M. D. Penta, R. Esposito, and M. Villani, “QoS-Aware Replanning of Composite Web Services,” in *ICWS*, 2005, pp. 121–129.

[12] D. Ardagna and B. Pernici, “Adaptive Service Composition in Flexible Processes,” *IEEE Transactions on Software Engineering*, vol. 3, no. 6, pp. 369–384, 2007.

[13] T. Yu, Y. Zhang, and K.-J. Lin, “Efficient algorithms for web services selection with end-to-end qos constraints,” *ACM Trans. Web*, vol. 1, no. 1, May 2007.

[14] M. Alrifai and T. Risse, “Combining global optimization with local selection for efficient qos-aware service composition,” in *WWW*. ACM, 2009, pp. 881–890.

[15] C. Zeng, X. Guo, W. Ou, and D. Han, “Cloud computing service composition and search based on semantic,” in *CloudCom*. Springer-Verlag, 2009, pp. 290–300.

[16] K. Kofler, I. u. Haq, and E. Schikuta, “User-centric, heuristic optimization of service composition in clouds,” in *EuroPar*. Springer-Verlag, 2010, pp. 405–417.

[17] Q. He, J. Han, Y. Yang, J. Grundy, and H. Jin, “Qos-driven service selection for multi-tenant saas,” in *Cloud*. IEEE Computer Society, 2012, pp. 566–573.

[18] E. Wittern, J. Kuhlenkamp, and M. Menzel, “Cloud service selection based on variability modeling,” in *ICSOC*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 127–141.

[19] A. Klein, F. Ishikawa, and S. Honiden, “Towards network-aware service composition in the cloud,” in *WWW*, 2012.

[20] J. O. Gutierrez-Garcia and K. Sim, “Agent-based cloud service composition,” *Applied Intelligence*, vol. 38, pp. 436–464, 2013.

[21] R. Karim, C. Ding, and A. Miri, “An end-to-end qos mapping approach for cloud service selection,” in *SERVICES*. IEEE Computer Society, 2013, pp. 341–348.