ulm university universität
uulm

A Provider-agnostic Approach to Multi-cloud Orchestration using a Constraint Language

Daniel Baur, Daniel Seybold, Frank Griesinger, Hynek Masata and Jörg Domaschka

10.1109/CCGRID.2018.00032

A Provider-agnostic Approach to Multi-cloud Orchestration using a Constraint Language

Daniel Baur, Daniel Seybold, Frank Griesinger
Inst. for Information Resource Management
Ulm University, Germany
{firstname.lastname}@uni-ulm.de

Hynek Masata
CE-Traffic, a.s.
Praha, Czech Republic
hynek.masata@ce-traffic.com

Jörg Domaschka
Inst. for Information Resource Management
Ulm University, Germany
joerg.domaschka@uni-ulm.de

***Abstract*—Cloud computing and its *computing as an utility* paradigm provides on-demand resources allowing the seamless adaptation of applications to fluctuating demands. While the Cloud's ongoing commercialisation has lead to a vast provider landscape, vendor lock-in is still a major hindrance. Recent outages demonstrate that relying exclusively on one provider is not sufficient. While existing cloud orchestration tools promise to solve the problems by supporting deployments across multiple cloud providers, they typically rely on provider dependent models forcing prior knowledge of offers and obstructing flexibility in case of errors. We propose a cloud provider-agnostic application and resource description using a constraint language. It allows users to express resource requirements of an application without prior knowledge of existing offers. Additionally, we propose a discovery service automatically collecting available offers. We combine this with a matchmaking algorithm representing the discovery model and the user-given constraints in a constraint satisfaction problem (CSP) that is then solved. Finally, we manipulate this discovery model during runtime to react on errors. Our evaluation shows that using a constraint-based language is a feasible approach to the provider selection problem, and that it helps to overcome vendor lock-in.**

I. Introduction

Cloud computing and its paradigm of computing as an utility [1] provides on-demand compute, storage, and network resources to its users. This enables users to tailor the consumed resources specifically to the current application requirements and thus, to horizontally (or vertically) scale the application in times of higher or lower demand respectively. Making full use of these advantages requires automation, and hence an adaptive system.

The ongoing commercialisation of cloud computing has led to a vast offer landscape making it increasingly difficult to select a matching offer for an application. The reluctance of cloud providers to implement existing standards like OCCI [2] further aggravates this fact, as it causes a massive vendor lock-in once a particular provider has been chosen. This situation is worsened as current failures of cloud providers [3] have shown that relying exclusively on a single cloud provider may harm reliability and availability. It is therefore desirable to achieve architectures allowing the use of multiple cloud providers at the same time increasing application resilience and avoiding vendor lock-in [4].

These needs led to the raise of cloud orchestration tools (COTs) [5], that promise to overcome above problems by allowing the parallel use of multiple cloud providers. Moreover, they promise to realise an adaptive system automatically managing the deployment and the runtime adaptation of the application. Yet, existing tools require the user to define cloud provider-specific identifiers when modelling (*i.e.* describing) the application resource requirements, *(i)* forcing the user to have detailed knowledge about the provider landscape and *(ii)* locking the application to the initially selected provider preventing re-selection in case of failures.

To overcome these problems we present Cloudiator [6] that comes with a cloud-agnostic modelling language. Based on our agnostic model, this paper provides the following contributions:

- we present a constraint language based approach to achieve a provider-agnostic requirement language
- we provide the necessary features, architecture and implementation to deploy a provider-agnostic model
- we provide an evaluation, showing that our approach is feasible

The remainder of this paper is structured as follows: Section II discusses the concept of a provider-agnostic model in detail by giving the problem statement and introducing our approach. Sections III - VI then describe our approach in more detail. Afterwards Section VII shows how we implemented the approach in our framework, before Section VIII evaluates our approach. Finally, we discuss related work in Section IX before Section X concludes the paper.

II. Provider-Agnostic Application Model

For the remainder of the paper we assume the following use case. A (cloud) user wants to deploy an application that consists of several independent components using on-demand (compute) resources. Those resources are offered in the form of nodes (*e.g.* virtual machines) by (cloud) providers. Each component is deployable on a set of nodes, while the number of nodes may vary during the application's runtime. The deployment will start with a minimum set of nodes for each component, that will be increased during runtime by horizontal scaling.

A description of such an application with respect to COTs is typically separated into two different domains: one describing the steps necessary to deploy the individual components of the application on arbitrary nodes (*e.g.* by providing scripts)

and one describing the resource demands and/or constraints that apply for this node. However, to achieve a provider-agnostic resource description one needs to use a declarative approach to express resource demands on common properties (*e.g.* number of cores, amount of RAM) offered by multiple cloud providers, instead of using an imperative approach that explicitly references provider specific identifiers. In contrast to a provider dependent model, a provider-agnostic resource description offers two main advantages: *i)* it eases the description itself, as the modeller does not need to know cloud specific details and existing offers, but *ii)* also allows changing the provider during runtime if *e.g.* a provider suffers an outage.

The importance of both aspects becomes clear, when taking a closer look at the related problems. Data available at CloudHarmony[1] suggests that over 20000 different configurations for nodes exists (*cf.* Section VIII). As we assume that a user not only wants to select one node per component but a set of nodes across multiple cloud providers, the number of possible configurations is the permutation (unordered, repeatable) of all available resource offers that is given by $\frac{(i+n-1)!}{i!(n-1)}$ with i being the number of resources to choose from, and n being the number of configurations offered by providers. Even with a low number of nodes $i = 3$ this leads to $\approx 1.3 \times 10^{12}$ possible configurations for the CloudHarmony data, rendering a manual selection impossible. With respect to error handling, recent outages of AWS US-EAST-1 region [7] and Google Compute Engine [8] showed that relying on a single region or cloud provider may have serious impacts on an application's availability. Provider-dependent application models, however, rely on provider specific identifiers for the resource description. If one of these offers is no longer available, *e.g.* due to a failure of the provider, the description is no longer deployable. Using a provider-agnostic model, only the current resource selection becomes invalid, but can be replaced by a new selection no longer taking the failed provider into consideration.

Using this declarative approach leads to a resource selection problem. Typically, a node at a cloud provider is defined by *i)* a hardware flavour defining its computational resources, *ii)* a (virtual) location defining the datacenter the node will be placed in and *iii)* an image defining its basic setup like operating system. Each of these node building blocks provides several properties that will later define the started node. Let $H = \{a_{H_1}, ..., a_{H_i}\}$ be the attributes defining a hardware flavour that *e.g.* include the amount of cores or RAM, $I = \{a_{I_1}, ..., a_{I_i}\}$ the attributes that define an image (*e.g.* the operating system) and $L = \{a_{L_1}, ..., a_{L_i}\}$ be the attributes that define a location (*e.g.* the city the data centre is placed in). One offer given by a cloud provider is then defined by the tuple $< H, I, L, p >$ where H, I, L defines the selected hardware, image and location and p defines the price of this combination. A node n is then a selection of one of these offers, depicted by the union of the attributes $H \cup I \cup L$. As one may select multiple nodes, a placement decision is given by a node configuration representing a set of nodes $N = n_1, ..., n_i$.

[1]https://cloudharmony.com

The resource demand of the user is given by a set of constraints $C = c_1, ..., c_i$ that target the different attributes of a node. A valid placement decision is therefore given by a selected set of nodes N where each node $n \in N$ fulfils the constraints imposed by the user.

Achieving a provider-agnostic resource description thus needs to solve four main challenges: *i)* a discovery mechanism collecting possible configurations from providers , *ii)* a declarative language, allowing the user to express his resource demand independent from provider specific details, *iii)* a resource selection mechanism matching the user's demands to the available offers and *iv)* a mechanism reacting on changes in the provider landscape.

We therefore propose a discovery mechanism (*cf.* Section III) that automatically populates a discovery model with provider information retrieved from the APIs of different providers. Additionally, we provide a constraint-language based declarative resource description (*cf.* Section IV), allowing the user to express demands for a set of resources by referencing attributes of the discovery model. Our matchmaking mechanism (*cf.* Section V) then transforms the discovery model and the user expressed constraints into a CSP and solves the generated problem. Finally, we show how we can implement error handling by modifying our discovery model during runtime (*cf.* Section VI).

III. Discovery

The task of the discovery process is to incarnate the discovery model, represented by the discovery package of our class model in Figure 1. The package features three main entities defining a computing resource: Location, Image and Hardware. In addition, it holds a Cloud entity representing a cloud provider and a Price entity for storing the prices of the offers.

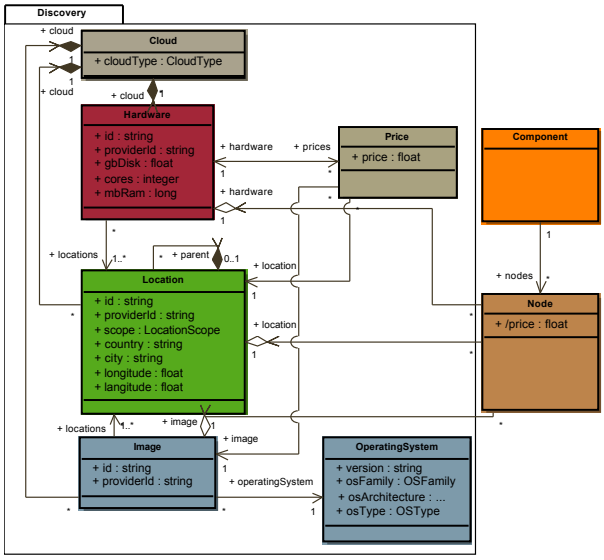


Fig. 1: Cloudiator Discovery Model

A *Location* represents (virtual) locations offered by a provider. They are stored in a hierarchical relationship with the scopes host, (availability) zone and region. *Images* represent the basic setup of a node, *e.g.* the operating system. *Hardware* (flavours) define the computational resources of a node by number of cores, disk space and RAM.

The compositions between *Cloud*, *Location*, *Image* and *Hardware* represent an *is offered by* relation, meaning that those offers are provided by this provider. The associations between *Location*, *Image* and *Hardware* represent an *is available in* relation depicting that the *Image* respectively the *Hardware* are solely available in those locations. This caters for the fact that especially in private clouds some hardware or images may be restricted to a specific location.

The discovery package also contains a notion of price. Our model defines the price as a function of Hardware, Image and Location, as the price for a specific hardware configuration varies based on location and image (*i.e.* license fees apply). We currently normalise the prices for an one hour interval in USD. More complex pricing models are left for future work.

While our application model is provider agnostic, there may be users that still want to explicitly reference a specific offer. For this purpose, the model has two identifier fields: *id* and *providerId*. The *providerId* is the original identifier issued by the provider. As this identifier might not be unique across multiple providers or even regions the *id* field contains a stable, globally unique identifier.

Collecting the necessary information imposes some challenges: *i)* the information is subject to change as cloud providers may update their offers, *ii)* while the information is publicly available for public clouds, it is not for private clouds, *iii)* the eligible offers may differ based on the user. To cope with those challenges, we automatically collect the information directly from the providers' APIs. This ensures up-to-dateness and correctness of data, while ensuring that the offers are usable by the authenticated user. As the information provided by providers is subject to semantic and syntactic differences we use our abstraction layer to harmonise the different data formats (cf. Section VII).

As the information provided by APIs may not be sufficient and lack important meta-data, we use other sources to enrich the data. For public cloud providers the service CloudHarmony provides data like geographical locations and prices. As such information is not available for private cloud providers, we either offer the user to provide the meta-data as tags directly within the cloud middleware or to manually edit the discovered offers.

The general collection flow consists of the following steps: *(1)* user creates a new cloud by providing his username, credentials and optionally an endpoint and a configuration, *(2)* the discovery agent collects all offers visible by the user and *(3)* if necessary the data is enriched by meta-data. Steps *(2)* and *(3)* are repeated periodically.

Collection Function	Description
Collection.sum(a)	Represents the sum of all values of the given attribute a. Only valid for numerical properties.
Collection.avg(a)	Represents the average of all values of the given attribute a. Only valid for numerical properties.
Collection.max(a)	Represents the maximum value of the given attribute a. Only valid for numerical properties.
Collection.min(a)	Represents the minimum value of the given attribute a. Only valid for numerical properties.
Collection.count(c)	Represents the number of elements that satisfy the constraint c.

TABLE I: Collection Functions

IV. Resource Description

We use a declarative approach to define the resource description, using a constraint language to specify the requirements as *e.g.* proposed by [9] on a per component basis. The constraint language is translated into a CSP that is solved in the matchmaking process (*cf.* Section V).

The CSP is defined by a set of constraints $C_1, C_2, ..., C_n$ and a set of variables $X_1, X_2, ..., X_m$ each having a Domain D_i defining possible values for the respective variable. A solution is an assignment of values v_i to all variables $X_i := v_i, ...$ under $v_i \in D_i$ that satisfies all constraints. A constraint consists of a variable and an expression defining rules the variable needs to fulfil.

We represent all possible variables that can be used within constraints by the class' attributes of our discovery model depicted in Figure 1. For example the attribute core of the Hardware class will be represented by variable allowing the user to express constraints with respect to required cores. The types of the variables are given by the type of their corresponding attribute. They can be enumerations, strings, numeric values, boolean values, collections and collection functions (cf. Table I).

As expression we support *i)* arithmetic expressions using the operators $+, -, *, /$, *ii)* logical expressions using the operators $=, \neq, >, <, \leq, \geq, \wedge, \vee, \implies$ and *iii)* custom expressions to handle collection variables (*cf.* Table II). Those expression also allow the user to express an optimisation criteria. We currently allow only one optimisation criteria and leave multi-objective optimisation for future work.

For example the requirement *a node with at least 4 cores and 2048 MB RAM* can be expressed with the constraints $node.cores \geq 4$ and $node.mbRam \geq 2048$.

To allow the user to express constraints targeting the interplay of multiple nodes we explicitly model the *Component* to *Node* relationship as 1:n. As discussed earlier, we introduce collection specific functions and expressions allowing to express constraints for this relationship. Listing 1 provides example constraints targeting a component.

Listing 1: Constraint Example

```
{
 "name": "StreamingComponent",
 ...
 "requirements": [
  // one node needs to be in Germany
  "nodes.exists(location.country = 'de')",
```

Collection Expression	Description
Collection.forAll(c)	All elements in the collection are subject to the expressed constraint c.
Collection.exists(c)	At least one element exists in the collection that satisfies the constraint c.
Collection.unique(a)	Enforces uniqueness of all value assignments to the attribute a.
Collection.allEqual(a)	Enforces all value assignments to attribute a to be equal.
Maximise(e)	Maximises the expression e.
Minimise(e)	Minimise the expression e.

TABLE II: Collection Expressions

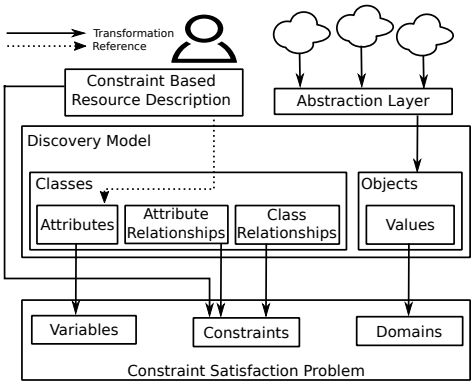


Fig. 2: Matchmaking Process

```
// at least two nodes need to have four or more cores
"nodes.count(hardware.cores >= 4) >= 2",
// all nodes need to be in a different location
"nodes.unique(location)",
// all nodes need to have at least two cores
"nodes.forAll(hardware.cores >= 2)",
// all nodes need to have at least 1024 MB of RAM
"nodes.forAll(hardware.ram >= 1024)",
// if a node has four or more cores its RAM needs to be
    larger or equal 4096
"nodes.forAll(hardware.cores ≥ 4 ⟹ hardware.ram ≥
    4096)",
// image needs to be Ubuntu 16.04 64-bit
"nodes.forAll(image.os.osFamily = OSFamily::UBUNTU)",
"nodes.forAll(image.os.version = "16.04")",
"nodes.forAll(image.os.osArchitecture = OSArch::AMD64)" ]
// we need a total amount of at least 15 cores
"nodes.sum(hardware.cores) >= 15"
// minimise the total costs of all nodes
"minimise(nodes.sum(price))"
}
```

V. Matchmaking

The task of the matchmaking is to find a valid node configuration, whenever *i)* a component is deployed or *ii)* our autoscaling engine or the user triggers a scale-in or out adding/removing a node or *iii)* we need to restore a node failure (*cf.* Section VI).

As described in Section II our problem is to find a node configuration N representing a set of nodes $N = n_1, ..., n_i$. A valid node configuration and therefore a solution to the matchmaking problem is a node configuration that fulfils the CSP C consisting of multiple constraints c, $c_1 \wedge ... \wedge c_i$. The optimal solution to the CSP is the solution that minimises resp. maximises the objective function given by the user.

To be able to solve the CSP, we translate the user given constraints (*cf.* Section IV) and the discovery model (*cf.* Section III) to a CSP that is then solved. For creating the CSP we follow the transformation process depicted in Figure 2: *i)* we represent the class attributes of the discovery model as variables in the CSP and use the *ii)* values assigned to the attributes in the object model as domains for those variables. Furthermore, we *iii)* represent the class and attribute relationships as additional constraints before *iv)* translating the user defined constraints. The following sections, will depict this transformation process in more detail.

A. Class Attributes

As described in Section IV we model every class attribute as a variable of the CSP. As a node configuration contains i node(s) where i is a natural number larger than 0 ($i \in \mathbb{N}^+$) the number of variables depends on the size of i.

With A being the set of all attributes a, we use the variable $v_{a,i}$ to represent attribute a of the i-th node. As an example the variable $v_{cores,1}$ will represent the attribute $cores$ of the first node. A node n_i is represented by the union of all variables for each attribute $n_i = \bigcup_{a \in A} v_{a,i}$.

As we do not know how many nodes are needed to fulfil the constraints imposed by the user, i is unknown. We therefore model it as parameter to the CSP generation logic.

B. Attribute Domain and Class structure

Let D_a be the domain of possible values for the attribute a. Let $d_a \in D_a$ be a possible value for a. Then the domain for each attribute D_a is given by all possible values for the attribute a that were discovered during the discovery process (*cf.* Section III). If the discovery model *e.g.* consists of two hardware objects (some attributes are left out for brevity) Hardware{id: t2.xlarge, cores: 4, ram: 16000} and Hardware{id: t2.micro, cores: 1, ram: 1000} the domain for the cores attribute is $D_{cores} = \{1, 4\}$ and the domain for the ram attribute is $D_{ram} = \{1000, 16000\}$.

With a_{cl} being an attribute of the class' sets of attributes Cl, domains for the attributes of the same class are interdependent. An assignment of a value to one of the attributes automatically constraints the domain of other class attributes, as the discovery model dictates possible value combinations. In the above example, an assignment to the cores attribute $cores := 4$ will restrict the domain of the ram attribute to $D_{ram} = 16000$ as the discovery model only allows this combination of the two attributes. The same restrictions apply vice versa.

To avoid having to model constraints for the cross product of all attributes of a class, we either use the globally unique *id* attribute of the class or an artificially generated object identifier to model the constraints with respect to each other attribute of the same class. This is done by expressing an *if then* relationship for every possible value of the id attribute with every other attribute of the class: $\forall d_{id} \in D_{id}\ \forall a_{cl} \in Cl\colon id = d_{id} \implies a_{cl} = d_{a_{cl}}$ where $d_{a_{cl}}$ is the only value of the domain $D_{a_{cl}}$ that occurs with the given id. As the id is unique the domain is guaranteed to hold only one value.

In the above example the Hardware{id: t2.xlarge, cores: 4, ram: 16000} object would be represented by two constraints: $id = t2.xlarge \implies cores = 4$ and $id = t2.xlarge \implies ram = 16000$

As a_{cl} is represented by multiple variables $\{v_{a_{cl},1}, ..., v_{a_{cl},i}\}$ for i nodes (*cf.* Section V-A), this constraint is created for every variable representing the attribute.

C. Relationship between classes

The same restrictions that are given by the structure inside a class also apply to the relationships between classes. In our case, we have to differentiate between 1:N (*e.g.* Cloud to Hardware) and N:M relationships (*e.g.* Hardware to Location).

For 1:N relationships we can apply the same ruleset as for attributes within a class, as a 1:N relationship is generally depicted by referencing the id of the 1-side as an attribute on the N-side. Taking Hardware{id: t2.xlarge, cloud: aws.ec2} referencing the Cloud{id: aws-ec2, type: PUBLIC} as an example, we will model the constraints $id_{hardware} =$ t2xlarge $\implies id_{cloud} =$ aws.ec2.

To represent N:M relationships we again use the *id*-attributes of both classes to represent the relationship. In contrast to the attributes of a class, selecting the value of one *id* attribute, will yield multiple possible values for the *id* attribute of the relating class. For a relation between classes A, B we therefore need to calculate the dependency domains $D_{id_B, d_{id_A}} \subset D_{id_B}$ that depicts the domain of the id variable of class B (id_B) under the condition that the id variable of class A is set to value $d_{id_A} \in D_{id_A}$ or vice versa. We then express the constraints $\forall d_{id_A} \in D_{id_A} : id_A = d_{id_A} \implies id_B \in D_{id_B, d_{id_A}}$

Taking the example Hardware{id: t2.xlarge, location: {eu-west-1, eu-central-1}} selecting Hardware as class A and Location as class B we would calculate the dependency domain $D_{id_{Location}, t2.xlarge} = \{$eu-west-1, eu-central-1$\}$. Using this domain we would express the constraint: $id_{Hardware} = t2.xlarge \implies id_{Location} \in \{eu{-}west{-}1, eu{-}central{-}1\}$.

Again, we apply the constraint to all variables representing the id attribute (*cf.* Section V-A).

D. User given constraints

For expressing the user given input we have to differentiate between *i)* collection functions that will be represented as additional variables in the CSP and *ii)* collection expressions that will be expressed as constraints applying to the variables of the CSP. The translation is depicted in detail in Table III and Table IV, with n_i being the i-th node.

E. Optimising the generated CSP

As a final step before solving the CSP we apply two optimisation rules. We *i)* filter variables and constraints based on their usage in the constraints given by the user and *ii)* we use the forAll constraints given by the user to proactively filter the domains. For step *i)* we check for each attribute of the classes if it is contained in the constraints imposed by the user. If an attribute is not contained, we remove all variables and constraints applying to this attribute. The only exception being the *id* attribute as it is required for depicting structure and relationship requirements and to later allocate the node from the given provider. Within step *ii)* we filter the object incarnation of each class (given by the discovery model) by applying the *forAll* constraints individually, as discovery objects not fulfilling them can never be part of a solution.

Collection Function	**Description**
Collection.sum(a)	$v_{sum,a} = \sum_{x=1}^{i} v_{a,x}$
Collection.avg(a)	$v_{avg,a} = \frac{1}{i} v_{sum,a}$
Collection.max(a)	$v_{max,a} = max(v_{a,1}, ..., v_{a,i})$
Collection.min(a)	$v_{min,a} = min(v_{a,1}, ..., v_{a,i})$
Collection.count(c)	$v_{count,c} = \sum_{x=1}^{i} c(n_x)$ $c(n_x) = \begin{cases} 0, \text{ if } n_x \text{ does not fulfil constraint } c \\ 1, \text{ if } n_x \text{ fulfils constraint } c \end{cases}$

TABLE III: Collection functions in the generated CSP

Collection Expression	**Description**
Collection.forAll(c)	$c_{all} = c_{n_1} \wedge ... \wedge c_{n_i}$ $c_{n_i} = \begin{cases} true, \text{ if } n_i \text{ fulfils } c \\ false, \text{ if } n_i \text{ does not fulfil constraint } c \end{cases}$
Collection.exists(c)	$c_{exists} = c_{n_1} \vee ... \vee c_{n_i}$
Collection.unique(a)	$c_u = \begin{cases} true, i = 1 \\ v_{a,1} \neq v_{a,2}, ..., v_{a,i-1} \neq v_{a,i}, i > 1 \end{cases}$
Collection.allEqual(a)	$c_e = \begin{cases} true, i = 1 \\ v_{a,1} = v_{a,2}, ..., v_{a,i-1} = v_{a,i}, i > 1 \end{cases}$
Maximise(e)	$max(e)$
Minimise(e)	$min(e)$

TABLE IV: Collection expressions in the generated CSP

F. Solving the generated CSP

While the resulting linear, integer optimisation problem (ILP) is in general NP-hard, the complexity of the resulting problem heavily depends on *i)* the discovery model (number of different cloud providers, number of offers each cloud provider has), *ii)* the number of nodes i that are required and *iii)* the constraints and optimisation function given by the user. A problem that *e.g.* only uses *forAll*-constraints can be easily solved by filtering offers and selecting the most optimal one, while constraints that target the interplay of multiple nodes (*e.g.* SUM) are harder to solve.

However, as discussed in the start of this section, our matchmaking algorithm is called at specific points of time, meaning we can make the assumptions that: *i)* during the initial deployment, the user wants to find a solution that has the least nodes possible, *ii)* when scaling, the user wants to add a single node, and keep existing ones and *iii)* when restoring failures, the user wants to replace the failed node while keeping others. Points *ii)* and *iii)* are especially important, as migrations of existing nodes are typically costly in multi-cloud environments. We therefore have two independent problems that need to be solved: *i)* we have to find a solution for the

initial node size and *ii)* we have to find a solution for node sizes i, that reuses the previous solution found for $i-1$.

For solving the problem for the initial node size we rely on two black-box CSP solving techniques. The first approach uses activity-based search (ABS) [10]. The second approach uses a best-fit approach (BF), that relies on `wdeg` [11] for variable selection. For value selection this approach uses a heuristic that calculates the average relation to the optimisation function. For this calculation we rely on the fact that our variables are related to each other either by belonging to the same class or via class relationships. If $D_{v_{opt}}$ is the domain of the variable v_{opt} to be optimised, and the variable selected by `wdeg` v_{wdeg} has the domain $D_{v_{wdeg}}$ we can first calculate the dependency domain $D_{v_{opt},d_{v_{wdeg}}}$ that only holds the values of $D_{v_{opt}}$ that occur when v_{wdeg} is assigned to value $d_{v_{wdeg}} \in D_{v_{wdeg}}$. Afterwards, we calculate the average of these values, and assign the maximum or minimum value (for maximisation resp. minimisation). As an example we assume the variable $v_{country}$ has the domain $D_{v_{country}} = DE, FR$, and we want to minimise the price. We would first collect all individual prices for $DE = 5, 3, 2, 6$ and $FR = 2, 1, 3, 2$ and afterwards calculate the average values $DE = 4$ and $FR = 2$. In this example, we would therefore assign the $v_{country}$ first to FR and then to DE. In our approach, both algorithms serve a different purpose. While the task of ABS is to find an optimal solution as fast as possible, the task of the BF approach is to find the best-possible solution within the given time limit. To find a solution we therefore execute both algorithms in parallel. If an optimal solution is found within a fixed time limit, we use this solution, otherwise we use the better solution found.

If we already have an existing solution (*e.g.* during scale-out or while restoring node failures), we reuse the existing solution during the generation of the variables (*cf.* Section V-A) by adding the possibility of making them constants if a solution for them already exists. Afterwards, we rely again on ABS to solve the remaining variables. This ensures, that already started nodes are reused in subsequent solutions, and reduces the execution time significantly (*cf.* Section VIII). This is important if we assume that the user typically wants fast-reaction during scale-out and node failures.

The resulting algorithm is depicted in Listing 2. The algorithm starts with a node size of one and an empty solution set. Then, until reaching the target node size, it firsts generates the CSP for the current node size and tries to solve it directly using ABS and BF, selecting the best solution found. If no solution is found, it will try this again with a node size increased by one. However, if the solver could find a solution for the given node size, subsequent calls will reuse the existing solution.

Listing 2: Matchmaking Algorithm

```
input: discoveryModel, constraints, targetNodeSize, timeout
output: solution
begin
  def csp
  def nodeSize ← 1
  def solution, abs, bf ← EMPTY

  while nodeSize <= targetNodeSize
    csp ← generateCSP(constraints, nodeSize)
    if solution == EMPTY
      parallel
        bf ← solveBF(csp, timeout)
        abs ← solveABS(csp, timeout)
      end
      solution ← best(bf, abs)
    else
      csp ← generateCSP(constraints, nodeSize, solution)
      solution ← solveABS(csp, timeout)
    end
  nodeSize++
  end
  return solution
end
```

VI. Error Handling / Recovery

Our error handling consists of three actions: *i)* a continuous error detection cycle detecting failures of nodes and providers, *ii)* a categorisation step that derives the necessary changes to the discovery model (*cf.* Section III) and *iii)* an error mitigation step that reschedules the failed nodes.

We assume that the application itself is modelled in a fault tolerant way. Fault tolerance means that it can tolerate failures of single nodes *e.g.* by specifying requirements (*cf.* Section IV) that force the usage of multiple providers, regions or availability zones.

A. Error Detection

To detect errors we rely on two points of interaction with the cloud provider. First, our discovery process (*cf.* Section III) will continuously query the APIs of the providers and report errors that occur during this communication. Additionally, whenever a new node is started at a provider we report errors based on responses we receive. In addition, we use a *Node watchdog* to detect unexpected state changes and unresponsiveness of allocated nodes. We use this additional watchdog to cater for the fact that the API of the provider may become unresponsive without affecting already running nodes.

B. Error Categorisation

We categorise the errors based on the error code and message received by the provider. As the APIs are typically web-based a categorisation based on typical http status codes is useful. The different error types that we distinguish are described in Table V. As syntax and semantics of provider errors may differ, we use an abstraction layer (*cf.* Section VII) to harmonise them.

C. Error Mitigation

For error mitigation we apply a baseline set of retry functionality. Each idempotent request will be retried a configurable amount of times to eliminate short-lived problems. In addition, a *clean-up agent* will delete orphaned nodes and component instances. For errors where detailed information exists, we apply the mitigation strategies based on the categorisation as depicted in Table V. The mitigation strategies

Error Type	Description	Mitigation
Authorisation	the user's credentials are invalid or privileges are insufficient.	remove provider, inform user
NotFound	desired offer is not available	remove the offer
InsufficientResources	the region can not provide the desired resource or the user's quota is reached	remove region
IllegalRequest	groups other request errors	remove region
Unavailable	the region is currently not available or timed out	remove region
ServerError	groups other server errors	remove region

TABLE V: Error Types and Mitigation

typically target provider regions, as in most cloud providers they reflect separate installations of their middleware. Removing a region, will automatically remove all related offers (*cf.* Section III). The offers will be removed for a configurable timespan. Whenever we notice the failure of a node, we reschedule the node, meaning that its resource description will again pass the matchmaking process. As the discovery model will no longer contain the failed provider and its offers, a new valid solution will be selected.

VII. Implementation in Cloudiator

We have implemented our approach in the cross-cloud orchestration toolset Cloudiator[2].

The application model of Cloudiator is built on two concepts: the concept of components depicting self-contained deployable and executable artefacts and the concept of applications grouping components forming a possibly distributed application. Each individual component defines a set of lifecycle actions, that describe interface actions for each step in the lifecycle of the component. These executable scripts install, start and stop the component and additionally detect failure. In addition, each component defines ports, depicting capabilities it can provide to or it consumes from other components. An application groups the components using the notion of communication, representing a directed channel between a provided port and a required port. We attach our provider-agnostic resource description on the component level. This means for every component the user modelled in his application he can attach constraints using our constraint language.

Cloudiator already provides an abstraction layer that hides the syntactic and semantic differences of different cloud providers and their APIs. It is built using jclouds[3], but addresses shortcomings of the library [12]. We use this abstraction layer to retrieve the provider's offers and transform them into our common discovery model.

Cloudiator features a deployment engine able to allocate the required nodes using the abstraction layer and installing our deployment and monitoring agents on the nodes. When a node is provisioned, the lifecycle agent is instructed to deploy the component. When needed, Cloudiator derives a deployment workflow from the inter-component communication dependencies, ensuring that components start in a valid order.

In order to provide adaptation capabilities Cloudiator follows the MAPE-K [13] cycle. It implements the cycle by using a rule language [14]. Using this language, the user can define event patterns triggering scaling actions. These include vertical and horizontal scaling in both directions. The scalability rule language also allows the user to define the monitoring demand by specifying sensors and aggregating the collected data using mathematical functions [15].

We implement our matchmaking algorithm in the resource broker component of Cloudiator. It is the resource broker's responsibility to compute matching offers, whenever a new instance of a component needs to be deployed, be it for an initial deployment or during auto-scaling. We rely on the Choco solver [16] Java library for describing and solving the CSP.

VIII. Evaluation

During the evaluation, we want to explore how the different CSP solving techniques activity-based search (ABS), best-fit search (BF) and the iterative approach presented in Section V perform with respect to runtime and solution quality on different discovery data-sets.

A. Methodology

We evaluate our matchmaking using three different discovery data-sets: *i)* an artificial, small data set (`S`), *ii)* an artificial, large data set (`L`) and *iii)* a real-world data set with data collected from CloudHarmony (`C`). For the small dataset we use a single provider, 15 locations within distinct countries, four hardware offers (Cores,RAM) = $\{(1, 512), (2, 1024), (4, 2048), (8, 4096)\}$ and one image. For the large dataset we use nine providers each having 15 locations within distinct countries, 60 hardware offers (Cores,RAM) = $\{2, 4, 8, 16, 32, 64\} \times \{512, 1024, 2048, 4096, 8192, 16384\}$ and 3 images. For both artificial datasets we calculate the price as a function $p = (cores + ram/1000) * l * c$ where l is a location price factor from 1.0 - 1.14 and c is a provider price factor from 1.0 - 1.08. An offer from provider 0 with 4 cores and 4096 RAM in location 5 would therefore cost $(4 + 4096/1000) * 1.4 * 1.0 = 11.3344$. For the CloudHarmony dataset we crawl their provided API. We only use providers and offers where CloudHarmony provides detailed pricing information. As CloudHarmony typically lists multiple prices for one offer based on the pricing model (e.g. reserved, on-demand and spot) we select the cheapest one. This leads to a total offer size of 60 possible nodes for S, 14580 for L and 21132 for C. We use the example constraints depicted in Listing 1 assuming that the location 'DE' has a price factor of 1.00.

Using the datasets described we execute seven experiments for each data set. For both the ABS and the BF search we limit

[2]http://cloudiator.org

[3]https://jclouds.apache.org/

the execution time to 1, 5 and 10 minutes to depict the solution quality they can achieve during this time limit. In addition, we execute the iterative approach, that reuses existing solutions of previous steps. We repeat each experiment ten times depicting the average values for costs and time. We omit cost datasets if the solver did not find any solution. We execute our evaluation on a workstation with an Intel Core i5-4570 CPU with 4 cores (each having 3.20GHz) and 16384 MB of RAM.

B. Results

Figure 3 depicts our evaluation results. The graphs in Figures 3a, 3b, 3c depict the execution times for the respective experiments, while the Figures 3d, 3e, 3f show the best solution found in the given time. We use $1 < i <= 15$ as node size, as a solution to the CSP requires at least two nodes (count constraint) and at most 15 (unique country constraint).

For the S dataset we can see that ABS can find an optimal solution for up to seven nodes for all time limits. For eight nodes ABS can no longer find an optimal solution, even when using a 10 minute time limit. BF can solve the problem optimally only for six nodes, while already requiring longer than ABS. The iterative approach reusing existing solutions significantly outperforms both ABS and BF time-wise. This holds especially true, when considering that the solving time is the time difference between the existing solution node size and the target solution node size. With respect to costs, all approaches perform well, even with a small time-limit of one minute. However, the solution quality of ABS and BF slightly decreases as soon as they are no longer finding an optimal solution and they are then outperformed by the iterative approach.

For the L dataset both ABS and BF can only find an optimal solution for the minimum node size of two, where again ABS outperforms BF. Again, the iterative approach performs best with respect to time. With respect to costs, the solution quality of ABS becomes bad as soon as it can no longer find optimal solutions. The solution quality of BF stays stable, but it can not find any solutions within the one minute time-limit. Again, the iterative approach performs good, but gets outperformed by BF for node sizes ≥ 6. This is due to the *nodes.sum(hardware.cores >= 15)* that for higher node sizes can be fulfilled with smaller, thus cheaper, nodes. For the same reason, the solution for a node size of 6 is cheaper than a node size of 5.

For the C dataset we can see different results. In contrast to both the S and the L dataset, the BF approach clearly outperforms ABS with respect to time and it can find optimal solutions for up to ten nodes while ABS finds the last optimal solution for a node size of nine. The same holds true for costs, where again the solution quality of ABS strongly decreases as soon as it can not find an optimal solution. Again, the iterative approach performs good with respect to time and performance, but as in dataset L gets outperformed by BF for node sizes ≥ 5. Interestingly, all algorithms perform better on C than on S with respect to finding optimal solutions. This can be explained when considering the structure of the data sets. While our artificial S and L datasets are very homogenous as every cloud has the same offers the CloudHarmony data used in C has a high heterogeneity. For example, while the cost range for offers with two cores in S data set is $[3.024 - 3,447]$ the range in C is $[0.009 - 41.7]$. This helps the algorithms to quickly refute unfeasible selections. However, it also leads to worse solution quality, if the optimal solution can not be found.

We can conclude, that the combination of both algorithms ABS and BF can solve an initial placement for up to 15 nodes in reasonable time, considering the three different data sets. While ABS finds a optimal solution faster for the data sets S and L, BF provides better solutions for C and if the time limit is not sufficient. We can also conclude, that the iterative approach presented, while sometimes outperformed in terms of costs, can quickly provide solutions for auto-scaling and node failures.

IX. Related Work

Commercial and open-source cloud orchestration tools that support multi-cloud or even cross-cloud deployment comprise Cloudify[4], Terraform[5] and Scalr[6]. However, their model is not cloud provider-agnostic, as the user has to explicitly reference a specific provider offer. This forces the user to *(i)* know all provider offers during design time and *(ii)* renders different placements due to *e.g.* failures impossible.

Typically, a cloud provider-agnostic language is achieved by specifying requirements that target infrastructure properties. TOSCA's [17] node_filter or node capabilities and requirements specification allow the user to describe constraints by referring to properties of the underlying host, *e.g.* defining an allowed range for the number of cores. Yet, such descriptions lack expressiveness as they automatically target all hosts of this component and do not allow to express interdependencies between attributes. Constraints such as *I need a node in Germany, US and Asia* or *If a node has 4 cores this implies that it needs 4096 MB RAM* are not expressible.

The following approaches represent and collect provider information: Saloon [18] uses feature models from software product lines to describe offers from cloud providers. While this results in a generic description of available offers, Saloon relies on an expert user to create these models. DrACO [19] automatically discovers available offers and represents them using TOSCA. They, however, crawl available offers from public websites which does not work for private clouds.

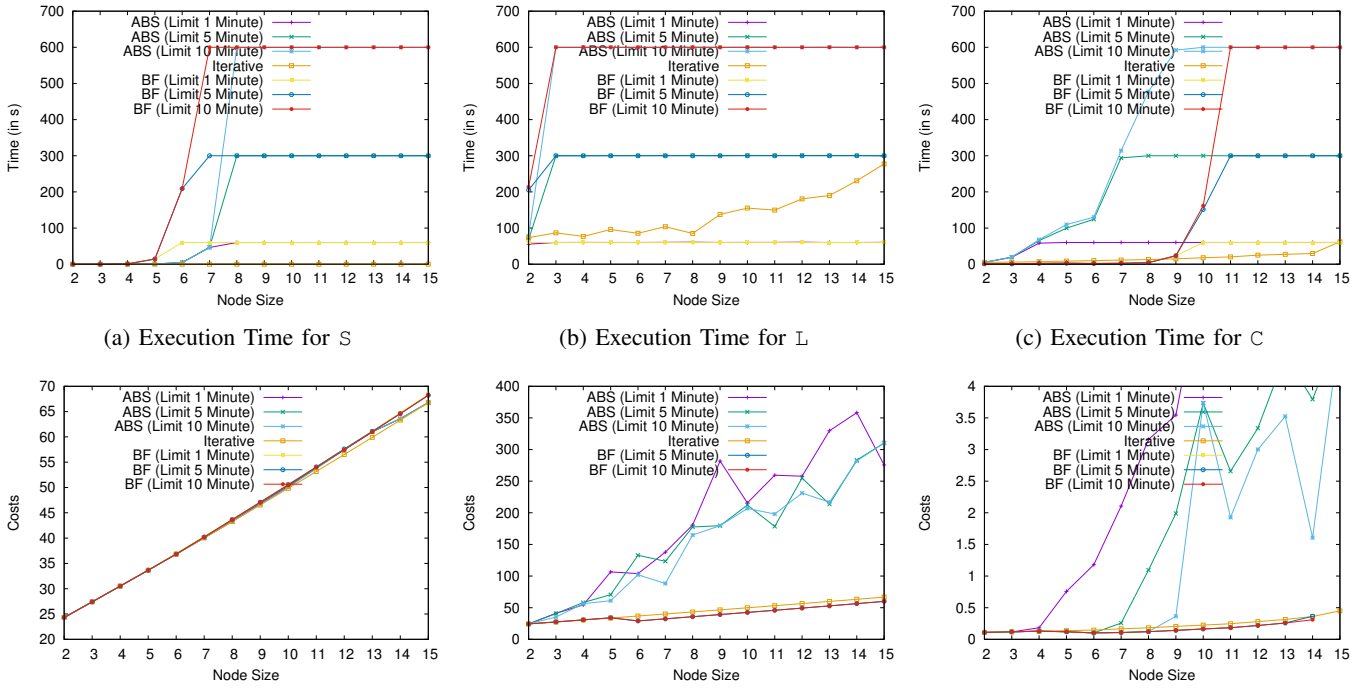
[20] transform an UML diagram and OCL constraints to a CSP. In contrast to our approach they do not use an object model, but check the satisfiability of the OCL constraints

[21] use mixed integer linear programming (MILP) for the resource selection and application placement under SLA constraints also considering reconfiguration. They however only consider four hardware-based dimensions (CPU, Memory, bandwidth and storage) where the user has to explicitly state the requirements similar to TOSCA and evaluate their

[4]http://cloudify.co/

[5]https://www.terraform.io/

[6]https://www.scalr.com/



(a) Execution Time for S

(b) Execution Time for L

(c) Execution Time for C

(d) Costs for S

(e) Costs for L

(f) Costs for C

Fig. 3: Evaluation Results

approach with at most 20 VM types. [22] considers the same dimensions, but also considers uncertainty of future demands. [23] propose a cloud brokering mechanism that uses integer programming to maximise performance while considering user given constraints targeting total number of nodes, node configuration and distribution across clouds. However, only hardware related constraints are considered and the evaluation is executed with 12 different VM types. [24] use particle swarm to place scientific workflows in the cloud environment. While their work also considers task dependencies, the heterogeneity of underlying resources and deadline constraints, the resource selection is only based on two dimensions (cost and processing capacity) and only 6 different types are considered during the evaluation. [25] use a genetic algorithm for solving the resource selection problem, by considering three basic dimensions (processing power, memory and storage) and communication cost between different provider offers. [26], model a bin-packing problem were a set of components is assigned to multiple VM types and propose heuristics to solve the given problem. Similar to our approach, they use a higher number of dimensions (8) and evaluate their approach using up to 10000 different VM types. While they also allow the user to express constraints, their approach is similar to TOSCA, while also targeting initial placement only.

To address failures, existing tools apply retry mechanisms for failed requests or failed application components. Apache jclouds *e.g.* automatically retries idempotent requests providing a low-level protection against communication failures. Cloudify supports a heal workflow that redeploys a user defined subgraph of the application in case of errors. Due to its imperative modelling language Cloudify cannot deal with provider failures as it would always reuse the same provider.

X. Future Work and Conclusion

We have presented a provider-agnostic resource description language based on a constraint language to overcome vendor lock-in. We combine the language with an automatic discovery engine collecting the offers of different providers. We implement a matchmaking engine, representing both the constraints given by the user and the discovery model as a CSP and propose an iterative algorithm solving it. Additionally, we show that we can dynamically adapt the discovery model to react to provider failures. The study of related work shows that our language combined with a dynamic discovery model achieves higher expressiveness and flexibility than existing approaches typically targeting a very specific problem. Our evaluation shows that our approach is feasible, even for a large offer size.

There are several points for improvements in future work. The iterative approach does not consider changing already selected offers, which may result in suboptimal solutions (cf. Section VIII). While reconfiguration is typically costly in a multi-cloud environment, its costs may vary based on the difference of the found solutions and could be included in the selection process. While our constraint language is more expressive than basic approaches targeting only single properties of a host, we could achieve higher expressiveness by *e.g.* adopting the Object Constraint Language OCL [27]

to express constraints on our discovery model. The same holds true for the properties depicted in the diagram. We currently focus mainly on attributes that are provided by most providers APIs. However, we could further increase the expressiveness by adding additional attributes, *e.g.* depicting benchmarking and profiling results like computational power or availability. Furthermore, the results of this paper will be used in Gibbon [28], an availability evaluation framework for distributed databases.

Acknowledgment

The research leading to these results has received funding from the EC's Framework Programme HORIZON 2020 under grant agreement number 731664 (MELODIC) and 732258 (CloudPerfect).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," Tech. Rep. UCB/EECS-2009-28, University of California, Berkeley, Tech. Rep., 2009.

[2] Open Grid Forum, "Open Cloud Computing Interface - Core," 2011.

[3] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages." in *SoCC*, 2016.

[4] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience*, 2014.

[5] D. Baur, D. Seybold, F. Griesinger, A. Tsitsipas, C. B. Hauser, and J. Domaschka, "Cloud orchestration features: Are tools fit for purpose?" in *UCC*, 2015.

[6] J. Domaschka, D. Baur, D. Seybold, and F. Griesinger, "Cloudiator: a cross-cloud, multi-tenant deployment and runtime engine," in *9th Symposium and Summer School on Service-Oriented Computing*, 2015.

[7] (2017) Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region. [Online]. Available: https://aws.amazon.com/de/message/41926/

[8] (2016) Google Compute Engine Incident #16007. [Online]. Available: https://status.cloud.google.com/incident/compute/16007

[9] C. Liu and I. Foster, "A constraint language approach to matchmaking," in *Research Issues on Data Engineering*, 2004.

[10] L. Michel and P. Van Hentenryck, "Activity-based search for black-box constraint programming solvers," *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, 2012.

[11] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proceedings of the 16th European Conference on Artificial Intelligence*, 2004.

[12] D. Baur and J. Domaschka, "Experiences from building a cross-cloud orchestration tool," in *3rd Workshop on CrossCloud Infrastructures & Platforms*, 2016.

[13] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, 2003.

[14] K. Kritikos, J. Domaschka, and A. Rossini, "Srl: A scalability rule language for multi-cloud environments," in *6th International Conference on Cloud Computing Technology and Science*, 2014.

[15] J. Domaschka, D. Seybold, F. Griesinger, and D. Baur, "Axe: A novel approach for generic, flexible, and comprehensive monitoring and adaptation of cross-cloud applications," in *Advances in Service-Oriented and Cloud Computing*, 2016.

[16] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Documentation*. [Online]. Available: http://www.choco-solver.org

[17] D. Palma and T. Spatzier, "Topology and orchestration specification for cloud applications (tosca)," *OASIS, Tech. Rep*, 2013.

[18] C. Quinton, D. Romero, and L. Duchien, "Automated selection and configuration of cloud environments using software product lines principles," in *IEEE 7th International Conference on Cloud Computing*, 2014.

[19] A. Brogi, P. Cifariello, and J. Soldani, "Draco: Discovering available cloud offerings," *Computer Science - Research and Development*, 2017.

[20] J. Cabot, R. Claris, D. Riera *et al.*, "Verification of uml/ocl class diagrams using constraint programming," in *ICSTW'08*, 2008.

[21] C. Seçinti and T. Ovatman, "On optimizing resource allocation and application placement costs in cloud systems." in *CLOSER*, 2014.

[22] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimal virtual machine placement across multiple cloud providers," in *APSCC*, 2009.

[23] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Generation Computer Systems*, 2012.

[24] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Transactions on Cloud Computing*, 2014.

[25] L. Heilig, E. Lalla-Ruiz, and S. Voß, "A cloud brokerage approach for solving the resource management problem in multi-cloud environments," *Computers & Industrial Engineering*, 2016.

[26] P. Silva, C. Perez, and F. Desprez, "Efficient heuristics for placing large-scale distributed applications on multiple clouds," in *CCGrid*, 2016.

[27] Object Management Group (OMG), "Object constraint language specification, version 2.4," 2014.

[28] D. Seybold, C. B. Hauser, S. Volpert, and J. Domaschka, "Gibbon: An availability evaluation framework for distributed databases," in *OTM*, 2017.