Cloud Integrator: Building Value-Added Services on the Cloud

Everton Cavalcante, Frederico Lopes,
Thais Batista, Nelio Cacho
DIMAp – Department of Informatics and Applied Mathematics
UFRN – Federal University of Rio Grande do Norte
Natal, Brazil
evertonranielly@gmail.com, fred.lopes@gmail.com,
thais@ufrnet.br, neliocacho@dimap.ufrn.br

Flavia C. Delicato, Paulo F. Pires
Department of Computer Science, Institute of Mathematics
UFRJ – Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
fdelicato@gmail.com, paulo.f.pires@gmail.com

***Abstract*—With the advance of the Cloud Computing paradigm, a single service offered by a cloud platform may not be enough to meet all the application requirements. To fulfill such requirements, it may be necessary, instead of a single service, a composition of services that aggregates services provided by different cloud platforms. In this perspective, this work presents *Cloud Integrator*, a middleware platform for composing services provided by different Cloud Computing platforms. Composition is performed by considering metadata about the services such as QoS, prices etc. By integrating the complementary paradigms of Service-Oriented Computing and Cloud Computing, *Cloud Integrator* enables the use of services provided by providers in a transparent way for the user, being necessary solutions for publishing, discovering, and composing these services. In terms of cloud infrastructure, *Cloud Integrator* lies in the PaaS layer, allowing HaaS, SaaS and DaaS resources to be available for the users so that they can build applications consisting of the composition of services provided by these platforms. In this paper, the application of *Cloud Integrator* is illustrated using a simple cloud-based e-commerce application.**

***Keywords-Cloud Computing, Semantic Web, composition of semantic Web services, semantic workflows, Cloud Computing platforms integration, service selection*.**

I. INTRODUCTION

With the advances of the modern human society, essential basic services are easily available. Nowadays, public utility services, such as water, electricity, gas, and telephony are considered essential in daily routines, being necessary that they are available whenever consumers require them [1]. These services are provided by specific companies in a transparent way to residences, businesses, institutions, etc. that consume these services and pay only the amount used. Applying this business logic to Computing, *Cloud Computing* [2][3] enables that computational resources and Information Technology (IT) services are offered on demand to users. As a consequence, the payment is based only on the usage of the services. Thus, some specialized companies are responsible for providing these services and for their management and commercialization.

Nowadays, the computation power of personal computers seems not be enough to solve complex questions that require large computation and memory resources and occur in different scientific fields. In this perspective, *Cloud Computing* comes to solve these and other issues in the current IT context, such as the need of building more complex infrastructures. Moreover, users must deal with several software installations, setups and updates. In addition, computational and hardware resources are by nature prone to be out-to-date in a very short time space [4]. Thus, the essential idea of Cloud Computing is enabling the transition from the traditional computing to a new model where the consumption of computational resources, such as storage, processing, bandwidth, data input and output, is performed through *services*, which can be easily and pervasively accessed [4]. These resources can be made available and used in an on demand basis, practically with no limitations, through the Internet and according to a pay-per-use model, providing elasticity, customization, QoS (*Quality of Service*) warranties and low-cost infrastructures. Thus, clouds are large pools of easily access and virtualized resources (hardware, software, services, etc.). These resources can be dynamically configured and reconfigured to adjust to variable loads, optimizing the use of these resources [5].

Although Cloud Computing provides the aforementioned benefits to current technologies, this new paradigm is still in its infancy [6] and there are several new challenges that must be handled. One of these challenges is related to *composition of services*, which becomes a relevant issue in the context of Cloud Computing since, with the advance of this new paradigm, a single service offered by a Cloud Computing platform may not be enough to meet all the application requirements. To fulfill such requirements, it may be necessary, instead of a single service, a *composition of services* that aggregates services provided by different Cloud Computing platforms, which requires a solution in terms of *platform integration*.

Usually Cloud Computing platforms promise a high availability for their clients, and the only plausible solution to provide a really very high availability is to use multiple Cloud Computing providers [2]. Nevertheless, current Cloud Computing platforms are not implemented using common standards, each one having its own APIs, development tools, virtualization mechanisms and governance characteristics [7]. This makes hard for users to perform component integration among services provided by different platforms. For instance, it

This work was partially supported by the Brazilian Academic and Research Network (RNP), through the *AltoStratus* CTIC Project, and by the Brazilian Institute for Web Science Research (INCT Web Science Brazil).

is usually awkward to use a data storage service provided by a platform *X* and to transport this data to a platform *Y* where they are processed to finally be used by an application deployed in another cloud platform *Z*. This situation happens because Cloud Computing is still an emergent area and does not have a common standardized technological model. Each provider offers its own development, deployment and resource management technologies. Consequently, users of these resources must implement their applications relying on a single cloud provider, a problem known as *cloud lock-in* [2].

In this process of composition of services not only service functionality must be considered when choosing services to include in this composition. Besides the fulfillment of SLAs (*Service-Level Agreements*) [8], it is necessary to consider *metadata* about the services, such as QoS, prices, etc., because there may be many equivalent services offered by the set of cloud platforms that are being integrated. For example, consider a situation in which an application needs to use a storage service and, among the available cloud platforms, two of them, *A* and *B*, offer this service but with different response time, which is the time between the sending of a service request by the client and the receiving of the response. If the response time of the service provided by platform *A* smaller than the response time of the service provided by platform *B*, then it is preferable, in the service selection process, that the service provided by *A* be chosen to be included in the composition.

As in Cloud Computing payment is based on the use of services, their cost may be very relevant for the user so that he/she considers a maximum cost as limiting factor for the composition. Thus, if the user specifies that the composition must not exceed a given value, then the composition mechanism must take this aspect into account at composition time in order to include in the composition services whose total costs do not exceed the established value. In addition, if cost and QoS parameters (e.g. availability, response time, performance, etc.) are considered, thus this mechanism can make a trade-off between these metadata enabling a better choice taking into account these aspects. Moreover, it is necessary to increase the flexibility of service search and composition, providing results according to user requirements.

In order to address such issue, the literature presents few proposals for service composition in the Cloud Computing context [7][9][10]. The main drawback of those approaches is to consider only service functionality to compose services. They do not provide composition and selection of services based on quality metadata (QoS, prices, and so on). In addition, they do not deal with the considerably heterogeneity present in Cloud Computing environments, so that they do not enable the access of different cloud platforms in a transparent and uniform way.

In this perspective, this paper introduces *Cloud Integrator*, a middleware platform for composing services provided by different Cloud Computing platforms. Some ideas employed in *Cloud Integrator* are shared with OpenCOPI (*Open COntext Platform Integration*) [14][15], a platform for integration of context service providers, that offers an environment that allows a quick and easy development of context-aware applications in Ubiquitous Computing environments. Similarly to OpenCOPI, the middleware proposed here is based on SOA (*Service-Oriented Architecture*) [11] and it works as a mediator, enabling to build fault-tolerant applications by composing services. Moreover, in *Cloud Integrator*, the composition of services is specified through a *workflow* that abstractly describes the order in which a set of tasks must be performed by several cloud services to complete a given action, represented by the application goals. Workflows are automations of business processes and are useful in environments in which several services provided by different providers are available, where some of these services have similar functionality. In addition, workflows can handle environmental changes at runtime according to resources availability, service quality, etc. specifying ways of undoing previous operations and going back to a legal state from where another path can be taken to reach the stated goal [12]. The composition model adopted by *Cloud Integrator* is based on the functionality of each service as well as on their respective metadata, enabling a better choice between the available services.

This paper is structured as follows. Section II presents the background of this work: basic concepts necessary to fully understand *Cloud Integrator* (Section II.A), and a cloud-based e-commerce application, the running example used along this paper (Section II.B). In Section III it is presented *Cloud Integrator,* its architecture, and the service selection algorithm. Section IV discusses related works. Finally, Section V contains final remarks.

II. BACKGROUND

A. Basic concepts

In the context of Cloud Computing, a point to call attention concerns to the concept of *service*. A *cloud service* differs from what is observed in other contexts such as Web services and grid computing, in which the composition of services is well-defined using standards such as WSDL (*Web Services Description Language*). Cloud services reach far beyond simple operations once they can encompass software applications, platforms or almost entire computational infrastructures. In this perspective, it is important to distinguish the types of services provided by the platforms which are integrated by the *Cloud Integrator* in order to be used by the applications. The first type is *resources as services* (or simply *cloud resources*), which are related to resources as services provided by cloud platforms, like processing, storage, database services, etc. The second type is *application services*, which are characterized as third-party services deployed on cloud platforms and use underlying resources provided by these platforms (virtual machines, databases, storage services, etc.), such as credit card payment services, cloud storage applications, etc.

Another important element to understand *CloudIntegrator* operation is the *semantic workflow*. A *semantic workflow* is an abstract representation of a workflow described in terms of *activities*, representing the application's execution flow, i.e. a workflow defines the sequence in which these activities should be performed to meet the application's business goal. In the

workflow, these activities are specified by a tuple <*task*, *object*> (e.g. <*store*, *file*>, <*send, message*>, etc.), where a *task* represents an operation implemented by one or more services, and an *object* can be used to describe inputs, outputs, and pre-conditions related to a *task*. Thus, the services that perform each activity can be dynamically found at runtime just based on the tuples <*task*, *object*>.

The services that perform the specified *activities* are described through *semantic Web services* [19] descriptions. Semantic Web services can improve the power of service representation, whose ontology languages such as OWL-S (*Web Ontology Language for Web Services*) [13] used in *Cloud Integrator*, allow describing Web services in a machine-interpretable and unambiguous way, increasing the degree of automation of service composition. Moreover, inference machines can be employed to automatically discover and compose services through their inputs, outputs, preconditions and effects (IOPEs). They are documented by *ontologies*, which offer formal expressiveness, prevent different semantic interpretations of the same information and allow software interoperability and transparent integration of the services. Thus, the application developer does not need to directly choose the services to be used by the application at development time, giving a greater flexibility to promote on-demand service access. So, when executing the workflow, *CloudIntegrator* makes inferences based on the workflow specification and the semantic Web services descriptions, performing the composition of services and identifying which activities must be selected to meet the workflow's business goal.

In order to execute a semantic workflow, it is necessary to create at least a concrete specification for the workflow, i.e. an *execution plan* (EP), which contains a set of concrete Web services that are orchestrated through their execution in a particular order. The EPs are built through an on-the-fly process of service discovery and composition according to the semantic interface of the selected services and the semantic workflow specification. Usually there is more than one EP for each workflow and the number of these plans depends on the amount of available services with the same functionality in the environment (at a given moment). Hence, it is necessary a service selection algorithm to choose what EP among the available ones should be executed. To perform this choice, the selection algorithm uses the metadata about the services. This algorithm is presented at Section II.B.

In *Cloud Integrator*, the EPs are represented by a directed acyclic graph (DAG) in which each intermediate node represents a specific cloud service and the directed edges represent the sequence of services execution. Each graph begins with an *initial* node and ends with a *final* node; these nodes do not represent services; they are used only to indicate respectively the beginning and the ending of the graph. Accordingly, the DAG represents the workflow with all possible EPs. Figure 1 shows a directed acyclic graph with all possible EPs. For instance, some possible EPs are: (i) $S1 \rightarrow S2 \rightarrow S3 \rightarrow S4 \rightarrow S5$, (ii) $S1 \rightarrow S2' \rightarrow S3 \rightarrow S4'' \rightarrow S5$, (iii) $S1 \rightarrow S6 \rightarrow S4'' \rightarrow S5$, etc.

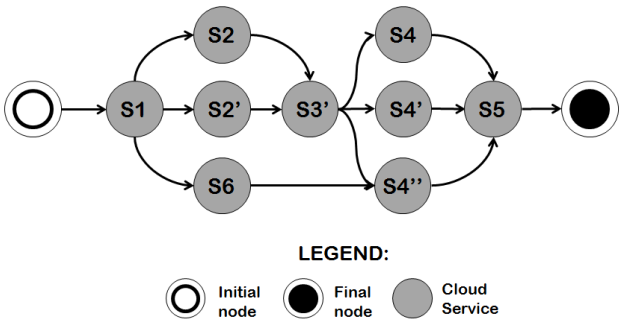

Figure 1. Example of directed acyclic graph representing a workflow with different execution plans options.

In some cases, services can be *dependent* on other services. In *Cloud Integrator*, this *dependency* between services denotes a relationship in which the dependent service can only be executed if and only if the service whose it depends has been previously executed. For instance, considering the EP $S1 \rightarrow S2' \rightarrow S3 \rightarrow S4 \rightarrow S5$ in Figure 1, if the service $S4$ is *dependent* on the service $S3$, $S4$ can only be executed if $S3$ has also been executed before, so if for some reason service $S3$ is unavailable, then the EP that contains the dependent service $S4$ cannot be executed.

B. *Running example: A cloud-based e-commerce application*

Consider a simple cloud-based e-commerce application used by small and medium companies through which a user can purchase products offered by a company. In this scenario, once informed what product the user wants to purchase, a sales automation service searches for the desired product and the company's inventory (a database) is consulted about its availability. If the desired product is available, then: (i) the purchase is made; (ii) the payment via credit card is executed; (iii) this transaction is logged into a file, and; (iv) the inventory of the company is updated. At the end, it is sent to the user a confirmation of the performed purchase. Figure 2 illustrates an UML (*Unified Modeling Language*) Activity Diagram that represents the flow of these activities.

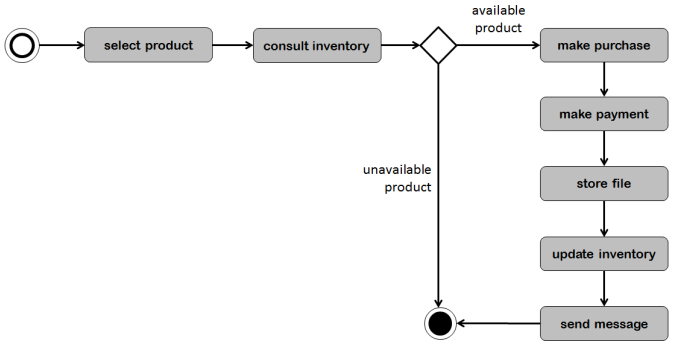

Figure 2. UML Activity Diagram for the running example.

To perform these activities, the application can use services provided by different Cloud Computing platforms. These services can be: (i) *resources as services*, such as storage services, relational databases and messages sending services (e.g. e-mails), and; (ii) *application services*, such as the sales automation and credit card payment services used in this

scenario. In the case of application services, if they are deployed in different cloud platforms (even they have the same implementation), then certainly they will have different metadata since these metadata also depend on the platforms in which these services are deployed.

As illustrated in Figure 3, the Cloud Computing platforms (named only for illustrative purposes) that provide resources as services as well as host application services are integrated to *Cloud Integrator* through specific *binders* (explained in Section III.A), which map specific APIs of each integrated cloud platform to the *Cloud Integrator* API.

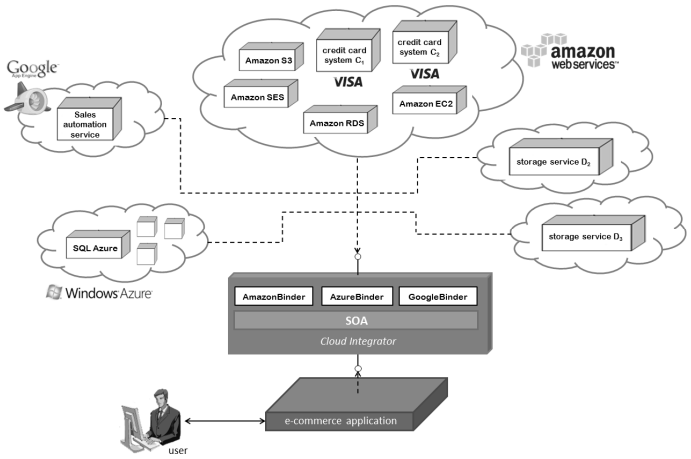

Figure 3. Environment of a cloud-based e-commerce application that uses several services provided by different platforms.

Figure 4 presents an example of how OWL-S semantically describes the application service *CreditCardService*, which implements the credit card payment service. This service ontology has the following elements: (i) *CreditCardProfile* (*profile* element), which presents the service and its functionality; (ii) *CreditCardProcess* (*process* element), which describes how this service works and its input and output parameters, and; (iii) *CreditCardGrounding* (*grounding* element), which describes how to access the service through WSDL, by specifying the port type and the operation that must be invoked.

Figure 5 shows a DAG that represents the execution plans of the workflow for the running example, containing six EPs. For example, to perform the activity regarding to the payment via credit card, the application can use more than one service of this type, corresponding in Figure 5 to the application services C_1 and C_2, which may be deployed in the same platform or in different cloud platforms. As also shown in Figure 5, the data can be stored by using equivalent services provided by different cloud platforms, and can even be made available in terms of resources (e.g. Amazon S3, represented by service D_1) or application services (such as services D_2 and D_3). Thus, as these services have similar functionality, it can be generated six different EPs with different combinations of the services offered by the cloud platforms that are begin integrated by Cloud Integrator, as shown in Figure 5. Since services can provide more than one operation, they can be used more than once in the same EP (as is the case of services *A* and *B*), being invoked the operation related to the activity served by the service.

```
<!-- Service -->
<service:Service rdf:ID="CreditCardService">
   <service:presents rdf:resource="#CreditCardProfile" />
   <service:describedBy rdf:resource="#CreditCardProcess" />
   <service:supports rdf:resource="#CreditCardGrounding" />
</service:Service>

<!-- Profile -->
<profile:Profile rdf:ID="CreditCardProfile">
   <service:presentedBy rdf:resource="#CreditCardService" />
   <profile:serviceName>Credit Card Service</profile:serviceName>
   <profile:textDescription>
      Credit card payment service.
   </profile:textDescription>
   <profile:hasInput rdf:resource="#CreditCardNumber" />
   <profile:hasInput rdf:resource="#Value" />
</profile:Profile>

<!-- Grounding -->
<grounding:WsdlGrounding rdf:ID="CreditCardGrounding">
   <service:supportedBy rdf:resource="#CreditCardService" />
   <grounding:hasAtomicProcessGrounding rdf:resource="#CreditCardWsdl" />
</grounding:WsdlGrounding>
<grounding:WsdlAtomicProcessGrounding rdf:ID="CreditCardWsdl">
   <grounding:owlsProcess rdf:resource="#CreditCardProcess" />
   <grounding:wsdlDocument>CreditCardSystem?wsdl</grounding:wsdlDocument>
   <grounding:wsdlOperation>
      <grounding:WsdlOperationRef>
         <grounding:portType>CreditCardSystemPortType</grounding:portType>
         <grounding:operation>CreditCard</grounding:operation>
      </grounding:WsdlOperationRef>
   </grounding:wsdlOperation>
</grounding:WsdlAtomicProcessGrounding>
```

Figure 4. Excerpt of the OWL-S ontology referring to the application service *CreditCardSystem*.

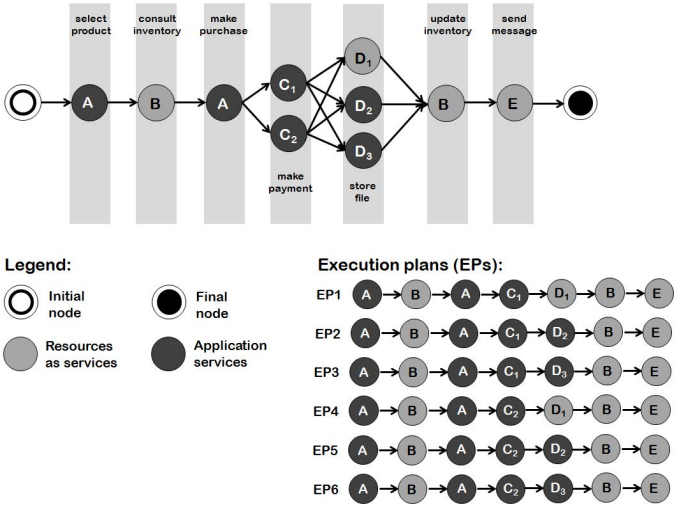

Figure 5. Workflow regarding to the running example with six execution plans.

III. CLOUD INTEGRATOR

A. Architecture

Cloud Integrator architecture is composed of two layers, named *Service Layer* and *Integration Layer*, as illustrated in Figure 6. The *Service Layer* is responsible for managing the abstractions of services (OWL-S descriptions) provided by service providers. The components of the *Service Layer* use these abstractions to create and execute workflows, select and compose services, perform an adaptation in case of failure, among other functionalities. The *ApplicationInterface* interface links applications and the *Cloud Integrator*'s *Service Layer*.

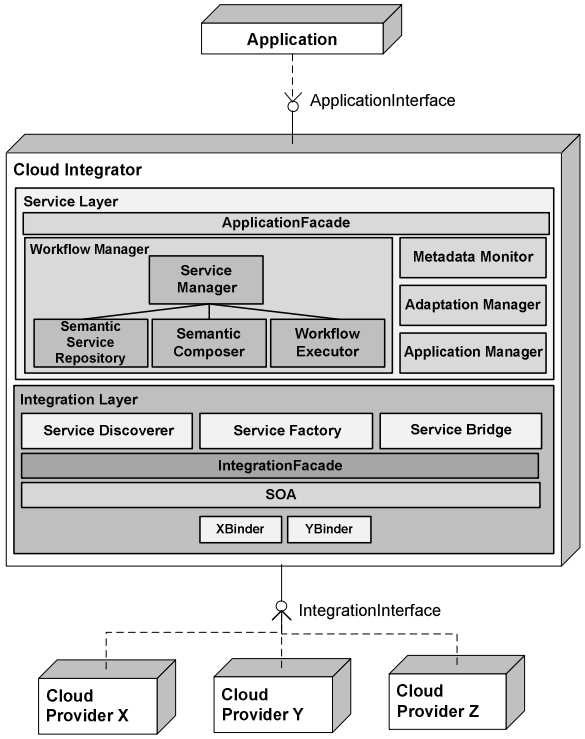

Figure 6. *Cloud Integrator* architecture.

The core of the *Service Layer*, the *Workflow Manager*, manages the abstraction of the services available provided by service providers, i.e. cloud platforms that are interacting with *Cloud Integrator*. This component is composed of four subcomponents (*Service Manager*, *Semantic Composer*, *Semantic Services Repository*, and *Workflow Executor*), which provide support to the specification of semantic workflows, generation and execution of EPs and workflow execution, as well as being responsible for discovering and composing Web services according to the specifications of the semantic workflow, i.e. to map the workflow activities into Web services. The functionality of *Metadata Monitor* component is acquiring metadata about the services provided by the cloud platforms. *Cloud Integrator* adopts a SLA [8] approach in which service providers publish the quality metadata of their services and these metadata are used to select the services to be provided to the applications, service consumers.

The *Adaptation Manager* component is responsible for the selection and adaptation processes in *Cloud Integrator*, where an *adaptation* in an application means to replace an EP in execution by another EP that performs the same activities. This component works directly with the *Metadata Monitor* and *Workflow Manager* components in order to identify a failure or any other condition that triggers an adaptation, e.g. unavailability of a service with the best quality, and automatically to change the execution flow to use other EP. This paper focuses on the metadata-based selection process performed by the *Adaptation Manager* component. Fault-tolerance and the adaptation process will be addressed in a future work. Finally, the *Application Manager* component is responsible for storing the application configuration (the user-requested quality and the workflow) and monitoring its execution. This component uses the metadata stored in the *Metadata Monitor* component regarding to the available services for triggering an adaptation to the *Adaptation Manager* component, if an adaptation is necessary.

The *Integration Layer* is responsible for integrating the underlying Cloud Computing platforms (service providers), promoting, if necessary, conversion between communication protocols. To allow the communication between these platforms and *Cloud Integrator*, it is necessary to build, at development time, *binders* implemented for each platform. These binders map specific APIs of each integrated cloud platform to the *Cloud Integrator* API, in order to have compatibility (*e.g.* to use the API of a cloud platform to access the resources provided by it, such as automatically and/or programmatically start and instantiate virtual machines). Moreover, as represented in Figure 6, the *Cloud Integrator* architecture provides a SOA abstraction level that allows the access to application services as traditional Web services and thus not requiring binders.

The *Service Discoverer* component is responsible for discovering cloud services in the environment and registering them in *Cloud Integrator*, the *Service Factory* component is responsible for creating abstractions (services) that encapsulate the specific cloud platforms APIs, and the *Service Bridge* component links these services to the *Workflow Manager* component. Thus, each service provided by a cloud platform API is represented by a Web service created by the *Service Factory* component to represent the respective service API, so that each one of these services Web that are created uses a binder developed specifically for the underlying cloud platform. Finally, the *IntegrationInterface* interface links cloud platforms and *Cloud Integrator*'s *Integration Layer*.

B. Service Selection

Even if there are different EPs options, only one of them should be selected to be executed thus meeting the application's business goals. In *Cloud Integrator*, this *service selection* is performed based on metadata about the services involved in the composition, namely (i) QoS parameters, e.g. response time, availability, performance, etc.; (ii) QoC (*Quality of Context*) parameters, if available, e.g. precision, information correctness, etc., and; (iii) price (cost) of the services. These quality and cost parameters are considered separately since selecting an EP with the lowest cost does not necessarily imply that such plan has the best quality, which may not be desired by the user.

In terms of cost, it is necessary perform *transformations* on pricing models. In Cloud Computing, different service providers may adopt different pricing models [16], e.g. price per hour, price per month, etc. being necessary to standardize this model, which can be done simply making time conversions (for example, converting a given amount to be paid per hour to an equivalent amount to be paid per month). Afterwards, it is computed the *aggregated cost value c*(*EP*) for each EP, making the sum of the costs of the *m* services that compose this plan *EP*.

In some situations, the user may establish that the total cost of a composition of cloud services does not exceed a given value θ of his choice. Since this is a global constraint for the

whole workflow, so EPs whose aggregated cost value exceeds the specified value θ should not be considered; however, it is possible that many EPs are not be considered, including plans that may have a high quality. To reverse this, the user can previously configure some exceptions, for example, allow that, instead of just not consider EPs, the selection process can accept EPs that extrapolate the limit θ at most *x*%.

Once calculated the aggregated cost value for the available EPs, this value is *normalized* to fit it in the range of values between 0 and 1, which is done by Equation 1

$$c_N = \frac{c_{max} - c_i}{c_{max} - c_{min}}, \quad c_{max} \neq c_{min} \qquad (1)$$

where c_N is the normalized cost, c_{max} e c_{min} are the maximum and the minimum aggregated value among the considered EP (if $c_{max} = c_{min}$ then $c_N = 1$). If the user specifies the maximum cost θ, then $c_{max} = \theta$.

For quality parameters (QoS/QoC), the *global value* of a given parameter of an execution plan *EP* is nothing more the value of this parameter considering the execution plan as a whole, i.e. all of the *n* services that compose it. The computation of this global value is performed by *aggregation functions* [20], as shown in Table 1. For example, *response time* is the QoS parameter used to measure the response time to execute each service, so that the global value of this parameter $q_r(EP)$ is given by the sum of $q_r(s)$ values of this parameter for each service *s* that composes the execution plan *EP*. For another QoS parameter, *availability*, its global value $q_a(EP)$ can be aggregated through a multiplication of the $q_a(s)$ availability value of each service *s* which composes *EP*. Finally, *performance* is a QoS parameter that describes the amount of requests $q_p(EP)$ served by the provider of a service *s* in a given period of time, so that the performance $q_p(EP)$ for an execution plan *EP* is limited by the service *s* which has the smallest value for this parameter.

TABLE 1. AGGREGATION FUNCTIONS EXAMPLES.

Type	Example of parameter	Expression
Summation	*response time*	$q_r(EP) = \sum_{s=1}^{n} q_r(s)$
Multiplication	*availability*	$q_a(EP) = \prod_{s=1}^{n} q_a(s)$
Minimum	*performance*	$q_d(EP) = \min_{s \in EP} q_d(s)$

Once calculated the global values of the parameters for the available EPs, only these values are considered and not more the individual values of the parameters for the services. However, just calculate the global values is not enough since each parameter deals with different sizes and therefore it is necessary to make a *normalization* to fit them into the same range of values, allowing an uniform measurement of the quality of EPs, as done in some works [17][18]. Depending on their nature, some quality parameters can be *positive*, i.e. the quality is better if the value is greater (e.g. the *availability* parameter); other parameters are *negative*, i.e. the quality is better if the value is smaller (e.g. the *response time* parameter).

For each EP it is necessary to calculate the normalized value for all parameters. Equations 2 and 3 respectively show formulae for normalization of positive and negative parameters. In these equations, q_{Ni} is the normalized value of the parameter *i*, q_i is the global value of this parameter for the considered execution plan, and q_{max} and q_{min} are the maximum and the minimum values of this parameters for all considered EPs (if $q_{max} = q_{min}$ then $q_{Ni} = 1$). In this process, q_{Ni} results in a value between 0 and 1.

$$q_{Ni} = \frac{q_i - q_{min}}{q_{max} - q_{min}}, \quad q_{max} \neq q_{min} \qquad (2)$$

$$q_{Ni} = \frac{q_{max} - q_i}{q_{max} - q_{min}}, \quad q_{max} \neq q_{min} \qquad (3)$$

Finally, with the normalized values of cost and quality, the *utility* $u(EP)$ of each execution plan *EP* is calculated. This utility is computed as a weighted sum (Equation 4) between these normalized values and *weights* assigned by the user, intending to capture user preferences for these values according his/her needs. Thus, at the end of the selection process, the EP with maximal quality is selected; if there is more than one EP with maximal quality, one of them is randomly selected.

$$u(EP) = \sum_{i=1}^{m} q_{Ni} * w_i + (c_N * w_c) \qquad (4)$$

In Equation 4, q_{Ni} e c_N are the normalized values for the quality parameter *i* and the cost, considering the execution plan *EP*, and w_i e w_c are the weights assigned by the user for these parameters.

C. Case Study

In order to enable a better understanding of the selection process detailed above, consider the services and EPs for the running example presented in Figure 5, whose cost and quality parameters values are presented in Table 2. For simplicity, only the quality parameters *availability* (percentage), *response time* (in milliseconds), and cost (in dollars per hour) are considered here.

TABLE 2. COST AND QUALITY PARAMETERS VALUES FOR THE SERVICES IN THE RUNNING EXAMPLE.

Services / Parameters	Cost	Quality parameters	
		Availability	***Response time***
A	0.20	0.950 (95.0%)	10
B	0.10	0,990 (99.0%)	45
C_1	1.00	0,900 (90.0%)	65
C_2	0.80	0,985 (98.5%)	80
D_1	1.00	0,999 (99.9%)	50
D_2	0.80	0,995 (99.5%)	30
D_3	0.60	0,990 (99.0%)	75
E	0.40	0,989 (98.9%)	25

According to the selection process, the first aspect to be considered is the cost of the services. As for this example the costs are all in the same unit, i.e. adopt the same pricing model (dollars per hour of use of the service), it is not necessary to

perform transformations on these values. Next, the aggregated cost value is calculated for each EP. The column "Aggregated value/Cost" in Table 3 shows the aggregated cost values for each of the six EPs (*EP*1 to *EP*6) of Figure 5, calculated by the sum of costs of the seven services that compose such plans. In a hypothetical situation, if the user had established that the total cost of a composition of services should not exceed $\theta = 2.70$ dollars per hour, then the execution plans *EP*1, *EP*2 and *EP*4 would not be considered, because they have aggregated cost value greater than 2.70 dollars per hour.

TABLE 3. AGGREGATED COST VALUE AND GLOBAL VALUES OF THE QUALITY PARAMETERS.

Execution plans (EPs)	Aggregated value	Global value	
	***Cost* ($c(EP)$)**	***Availability* ($q_a(EP)$)**	***Response time* ($q_r(EP)$)**
EP1	3.00	0.787 (78.7%)	250
EP2	2.80	0.783 (78.3%)	230
EP3	2.60	0.779 (77.9%)	275
EP4	2.80	0.861 (86.1%)	265
EP5	2.60	0.857 (85.7%)	245
EP6	2.40	0.853 (85.3%)	290

Once calculated the aggregated cost values for EPs, these values must be *normalized*. From Table 3, the maximum aggregated cost value among the EPs is $c_{max} = 3.00$ and $c_{min} = 2.40$. Using Equation 1 and each EP's aggregated value c_i, it is obtained the normalized cost values c_N shown in Table 4, fitted in the range of values between 0 and 1. It is noteworthy that a high aggregated cost value for an EP implies in a c_N value closer to 0; vice versa, a low aggregated value for an EP implies in a c_N value closer to 1.

TABLE 4. NORMALIZED VALUES FOR COST AND QUALITY PARAMETERS.

Execution plans (EPs)	Normalized value		
	***Cost* (c_N)**	***Availability* (q_{Na})**	***Response time* (q_{Nr})**
EP1	0.000	0.087	0.667
EP2	0.333	0.048	1.000
EP3	0.667	0.000	0.250
EP4	0.333	1.000	0.417
EP5	0.667	0.958	0.750
EP6	1.000	0.905	0.000

For the considered quality parameters, *availability* and *response time*, their *global values* are respectively calculated using summation and multiplication aggregation functions (see Table 1) of the quality values for each of the seven services that compose the EPs. Table 3 also shows the global values for these parameters for each of the six EPs in Figure 5.

Once calculated the global values for *availability* and *response time* for the EPs, such values must be *normalized*. For the *availability* parameter, from Table 3, the minimum global value is $q_{min} = 0.779$ and the maximum global value is $q_{max} = 0.861$. Since this parameter is *positive* (i.e. the quality is better if the value is greater), Equation 2 is used to calculate the normalized value q_{Na} for this parameter by considering each EP's global value q_a. Similarly, for the *response time* parameter the minimum global value is $q_{min} = 230$ and the maximum global value is $q_{max} = 290$. Since this parameter is *negative* (i.e. the quality is better if the value is smaller), Equation 3 is used to calculate the normalized value q_{Nr} for this parameter by considering each EP's global value q_r. The normalized values q_{Na} and q_{Nr} of each EP are shown in Table 4, also worth to emphasize that values closer to 1 correspond to the best global values for the parameters.

Finally, once normalized the cost and quality values, the *utility* $u(EP)$ of each execution plan *EP* is calculated using the Equation 4. Taking as weights for each of the parameters the values $w_c = 0.40$, $w_a = 0.30$ and $w_r = 0.30$ (respectively, *cost*, *availability* and *response time*), it is obtained the utility values for each execution plan, presented in Table 5. Thus, at the end of the selection process, the execution plan *EP*5: $A \rightarrow B \rightarrow A \rightarrow C_2 \rightarrow D_2 \rightarrow B \rightarrow E$ is selected since it has the greatest utility value (highlighted in Table 5) considering the user preferences expressed through the weights assigned to each of the parameters.

TABLE 5. UTILITY VALUES FOR EXECUTION PLANS *EP*1 TO *EP*6.

Execution plans (EPs)	Utility ($u(EP)$)
*EP*1	0.226
*EP*2	0.448
*EP*3	0.342
*EP*4	0.558
*EP*5	0.779
*EP*6	0.671

IV. RELATED WORK

This Section presents some proposals existent in literature in terms of composition of Cloud Computing services, aiming to give an overview of them and to discuss aspects that differentiates them from *Cloud Integrator*.

The authors in [7] propose a strategy for Web services composition that, according to them, can be adapted to Cloud Computing services. First, a service description in WSDL language is processed and its main elements (service name, function, description, operators and input/output parameters) are stored in different tables of relational databases. Afterwards, a Web Service Matching Algorithm (SMA) uses this information to match two services, where output parameters of a service can match input parameters of another service, what is called *semantic similarity*. In sequence, service matching is represented as a weighted directed graph and thus the service composition problem is reduced to find all reachable paths between two nodes (Web services) in the graph by using another algorithm named Fast-EP. At last the QoS information is used to rank the search results. Although this proposal seems promising, the authors do not have yet applied it in any Cloud Computing context. Moreover, it lacks for mechanisms to allow dynamic service composition and adaptation in case of service failure or quality degradation, nor it considers an abstract model (e.g. workflows), which has

several advantages, neither SLAs. Finally, the authors argue that the precondition of Cloud Computing service composition is to have a description with unified standard to introduce its functionality and interface and states, buy they consider only services described by WSDL, which they state that could not fully meet the requirement of Cloud Computing services composition.

The work in [9] proposes an approach for workflow engine that is capable of integrating different Cloud Computing platforms and services, offering OaaS – *Orchestration as a Service*. OaaS can be seen as a specialized form of PaaS from the workflow developer's view and generates additional value for the users without revealing any internals of the workflow. The prototype implementation of the architecture lacks some features like: (i) automatic service selection, which is helpful in scenarios where multiples services from different providers offer same functions and the user can define his/her preferences; (ii) mechanisms for failure/fault handling and adaptability at runtime since the workflow engine integrates services from different vendors and service providers and a crash or unattainability of some services is occasionally possible; (iii) use of quality metadata since invoked services can be distributed over several cloud platforms all offering different QoS properties, and; (iv) SLAs support. These features are covered by *Cloud Integrator*.

At last, [10] addresses workflow engines and its application to the Cloud Computing paradigm, presenting a Workflow Management System (WfMS) composed of a workflow engine responsible for managing the workflow execution, a resource broker and plug-ins for communication with various technological platforms, similar to the binders existent in *Cloud Integrator*. The WfMS schedules jobs (tasks) in workflow to remote resources based on user-specified QoS requirements and typical parameters that drive these scheduling decisions, such as deadline (time) and cost. Moreover, SLA-based negotiation with remote resources capable to meeting these demands is taken into account, being envisioned two scenarios, one where the workflow is entirely executed in a cloud platform and another where the WfMS schedules workflow tasks to resources that are located at a local cluster and in cloud platforms. However, like the proposal in [7], the proposed WfMS does not support dynamic service composition and adaptation in case of service failure or quality degeneration.

V. FINAL REMARKS

In this paper it was presented *Cloud Integrator*, a middleware platform for the composition of services offered by different Cloud Computing platforms. *Cloud Integrator* is based on SOA and the composition of services is specified using a semantic workflow. *Cloud Integrator* generates different execution plans for the workflow, containing a set of concrete Web services provided by the different underlying cloud platforms. The focus of this paper was on the composition mechanism of the platform. It was described the metadata-based service selection algorithm and its application in a simple case study. In a future work it is intended to specify the adaptation process.

REFERENCES

[1] Rajkumar Buyya; Chee Sin Yeo, and Srikumar Venugopal, "Market-oriented Cloud Computing: Vision, hype and reality for delivering IT services as computing utilities", in Proceedings of HPCC 2008 – 10th IEEE International Conference on High Performance Computing and Communications, 2008, pp.5–13.

[2] Michael Armbrust et al., Above the clouds: A Berkley view of Cloud Computing. Technical report, Reliable Adaptive Distributed Systems Laboratory, University of California at Berkley, USA, 2009.

[3] Qi Zhang, Li Cheng, Raouf Boutaba, "Cloud Computing: State-of-the art and research challenges", Journal of Internet Services and Applications, vol. 1, n. 1, 2010, pp.7–18.

[4] Lizhe Wang, Gregor Von Laszewski; Marcel Kunze; Jie Tao, "Cloud Computing: A perspective study", New Generation Computing, vol. 28, n. 2, 2010, pp.137–146.

[5] Luis M. Vaquero; Luis Rodero-Merino; Juan Caceres; Maik Lindnet, "A break in the clouds: Towards a cloud definition", ACM SIGCOMM Computer Communication Review, vol. 39, n. 1, 2009, pp.50–55.

[6] John W. Rittinghouse and James F. Randsome, Cloud Computing: Implementation, management and security. USA: CRC Press, 2010.

[7] Cheng Zeng, Xiao Guo, Weijie Ou, and Dong Han, "Cloud Computing service composition and search based on semantic", in Proceedings of CloudCom'09 – International Conference on Cloud Computing, Lecture Notes in Computer Science, v. 5931. Springer-Verlag Berlin/Heidelberg, 2009, pp.290–300.

[8] Sumit Bose et al., "SLA management in Cloud Computing: A service provider's perspective", in Cloud Computing: Principles and paradigms, Rajkumar Buyya, James Broberg, and Andrzej Goscinski, Eds. New Jersey, USA: John Wiley & Sons, 2011, pp. 413–436.

[9] André Höing, Orchestrating secure workflows for cloud and grid services, Ph.D. dissertation, Elektrotechnik und Informatik, Technischen Universität Berlin, Berlin, Germany, 2010.

[10] Suraj Pandey, Dileban Karunamoorthy, and Rajkumar Buyya, "Workflow engine for clouds", in Cloud Computing: Principles and paradigms, Rajkumar Buyya, James Broberg, and Andrzej Goscinski, Eds. New Jersey, USA: John Wiley & Sons, 2011, pp. 321–344.

[11] David S. Linthicum, Cloud Computing and SOA convergence to your enterprise: A step-by-step guide, USA: Pearson Education, Inc.

[12] Rob Allen, "Workflow: An introduction", in Workflow Handbook 2001, Layna Fischer, Ed. Florida: Future Strategies, Inc., 2000, pp.15–38.

[13] David Martin et al., "Bringing semantics to Web services: The OWL-S approach", in Proc. of SWSWPC 2004 – First International Workshop on Semantic Web Services and Web Process Composition, Lecture Notes in Computer Science, v. 3387. Springer-Verlag Berlin/Heidelberg, 2005, pp.26–42.

[14] Frederico Lopes et al. "On the integration of context-based heterogeneous middleware for Ubiquitous Computing", in Proceedings of MPAC'08 – 6th International Workshop Middleware for Pervasive and Ad-hoc Computing, 2008, pp.31–36.

[15] Frederico Lopes, Flavia C. Delicato, Thais Batista, and Paulo F. Pires, "Context-based heterogeneous middleware integration", in Proceedings of WMUPS'09 – Workshop on Middleware for Ubiquitous and Pervasive Systems, 2009, pp.13–18.

[16] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel, "The cost of a cloud: Research problems in data center networks", ACM SIGCOMM Computer Communication Review, vol. 39, n. 1, 2009.

[17] Liangzhao Zeng et al., "QoS-aware middleware for Web services composition", IEEE Transactions on Software Engineering, vol. 30, n. 5, 2004, pp.311–327.

[18] Frederico Lopes et al., "AdaptUbiFlow: Selection and adaptation in workflows for Ubiquitous Computing", in Proceedings of EUC 2011 – 9th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2011, in press.

[19] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng, "Semantic Web Services", Intelligent Systems, vol. 16, n. 2, 2001, pp.46–53, Special Issue on the Semantic Web.