infrastructure (IaaS), platform (PaaS) and software (SaaS) (Konstantinou et al., 2009; Mietzner, Leymann, & Papazoglou, 2008). We define a conceptual model and operator calculus for a service description template that would allow both dimensions to be covered. Structural aspects of that template will be identified, formalized in an ontology and aligned with the Cloud development and deployment process.

The need to cover horizontal and vertical integration in a specific service description language has been recognized (Nguyen et al., 2011; Papazoglou & van den Heuvel, 2011; Bernstein et al., 2009). Specifically, the need to deal with *federated Clouds* while still considering all Cloud stack layers is acknowledged. We go beyond current solutions with a rich *ontology-based template manipulation framework*. The term template shall here subsume service descriptions known as recipes, manifest or blueprints in other contexts. The term shall also indicate that a template is an abstract, pattern-like artifact that can be instantiated.

This chapter is organized as follows. The next section provides background information on Cloud brokerage. The following section defines the reference architecture and introduces the service template. As this concept plays a crucial role, it will be investigated in more depth in the other sections. Further, we outline solution architecture for an implementation based on open-source Cloud platforms and discuss some trends in this space.

BACKGROUND

Both Gartner and NIST define Cloud Service Brokerage (Gartner, 2012; NIST, 2011). They follow a similar three-pronged classification. They define a Cloud service broker as an entity that manages the use, performance, and delivery of Cloud services and negotiates relationships between Cloud service providers and Cloud service consumers. We now detail the Gartner types, which we adopt in our reference architecture. Gartner's customization focuses on enhancing existing service. Gartner integration highlights flexible mediation and integration of different systems. For each type, we name the typical broker function (agent), the application types (application), and standard functionality examples of brokered (or mediated) services (function).

- Aggregation is about delivering two or more services to possibly many consumers, not necessarily providing new functionality, integration or customization, but offering centralized management of SLAs and security.
    - Agent: distributor
    - Application: marketplace, Cloud provisioning
    - Functions: discovery, billing, marketplaces

- Customization is about altering or adding capabilities, to change or improve and enhance the service function, possibly combined with analytics.
    - Agent: independent software vendor (ISV)
    - Application: analytics, monitoring, user interface
    - Functions: wrapper, adaptivity

- Integration addresses the challenges of making independent services work together as a combined offering, which is often integration of a vertical Cloud stack or data/process integration within a layer. Classical techniques such as transformation, mediation and orchestration are the solutions.
    - Agent: systems integrator (SI)
    - Application: integrated PaaS, horizontal and vertical
    - Functions: orchestration, Mashup, mediation

A number of Cloud service description notations exist, often aligned with a Cloud tool solution. In order to systematically review a number of solutions, we use a categorization framework. In Figures 1 and 2, we summarize language definition challenges and requirements for a number of solutions (Cloud Foundry, 2013; Cloudify, 2013; DeltaCloud, 2013; JClouds, 2013; LibCloud, 2013; Mosaic, 2013; OpenShift, 2012; simpleAPI, 2013). We categorize languages in terms of the Cloud layer support and specific features or functions each of them provides. We have defined a comparison framework to categorize solutions along the following concerns:

- *System Type:* Multi Cloud API Library, IaaS Fabric Controller, Open PaaS Solution, Open PaaS Provider.
- *Distribution Model:* Open Source (true for all solutions considered).
- *Core Capabilities:* Multi IaaS Support, Multi Language/Framework Support, Multi Stack Support.
- *Core Features/Components* (development and deployment time): Service Description Language, Native Data Store, Native Message Queue, Programming Model, Elasticity & Scalability, QoS/SLA Monitoring.
- *Advanced Features/Components*: Service Discovery/Composition, Broker, Marketplace - towards broker and marketplace features.

Figure 1. Comparison of Language Definition Concerns - Categories and Types

These concerns allow us to categorize the Cloud solution in terms of is main function, which is considered as the system type that indicates its target layer and central function in that layer. We also note whether the tool is proprietary or open-source. Furthermore, a number of standard properties and individual components are captured. The properties we have chosen here (called the Core Capabilities) refer to necessary capabilities for brokers to integrate service offerings. There are also two feature categories that organize system components into common and advanced ones.

Figure 2. Comparison of Language Definition Concerns - Capabilities and Features

From the figures above, we can see a trend towards the broker and marketplace solutions (Mietzner, Leymann, & Papazoglou, 2008; Rodero-Merino et al., 2010). These demonstrate to some extent a multi-faceted description with functional and quality aspects. However, a comprehensive coverage is not available that addresses the complexity due to the required multi-layer support and vertical integration. A richer and formally defined description notation is required. We select two specific template notations that represent the trend mentioned here for a more in-depth description of the state-of-the-art in Cloud service description.

The 4CaaSt Blueprint approach (Papazoglou & van den Heuvel, 2011; Nguyen et al., 2011) supports through its different languages a range of Cloud brokerage activities. Blueprints are initially abstract specifications, which can be resolved, i.e., mapped to lower Cloud stack layers until the specifications are finally deployed using a configuration and deployment manager. Services are described in a Blueprint (BP), which is an abstract description of what needs to be resolved into infrastructure entities. BPs are stored and managed in a repository. A BP is resolved when all requirements are fulfilled by another BP, via a Resolution Engine. This is a service orchestration feature. A number of XML-based blueprint languages are supported: BDL BP Description Language (for Cloud operations and capabilities), BCL BP Constraint Language (for definition of SLA parameters), BML BP Manipulation Language (for methods and operations, such as, service match, merge, compose, delete) and BRL BP Request Language (for user

and developer request definitions). Blueprints are suitable to support simple marketplaces offering individual services as a product, but lack the range of template manipulation support in their toolkit.

The CompatibleOne manifests suffer from similar limitations. The units of a CompatibleOne Service Manifest are Image and Infrastructure. An Image consists of a System (the base OS) and a Package (reflecting the software stack configuration). Infrastructure covers the classical IaaS concerns, such as, Storage, Compute, and Network. An Image is a description of a manual build of an application, used by development and configuration tools (like puppet or chef). An Image has an agent that is embedded in a VM and runs on start-up. An Agent is a script to run required configurations, set up monitoring probes, or to download required components.

Cordscript is proprietary CompatibleOne scripting language for expressing configuration actions. These support metadata handling within VMs, managing required linkages between VMs, and setting up monitoring probes in VMs. Cordscript is a statement-oriented language. A feature under development in CompatibleOne is a Cloud-implementable graph of OCCI category instances. Particularly well solved is the standards-compliancy at the lower layers (making it a good testbed), but the manifests only cover core concerns.

CONCEPTUAL ARCHITECTURE FRAMEWORK

We propose a conceptual framework for an automated Cloud service broker architecture based on two parts: a reference architecture for Cloud service brokers compliant with the NIST proposal and a service description template model that describes mediated Cloud services. The reference architecture provides a structure in which artifacts and their processing and distribution through components can be defined. The NIST proposal for Cloud service broker architecture shall be adapted to Gartner terminology and extended by a template mechanism that guides activities such as integration, customization, and aggregation.

Figure 3. High-level Reference Architecture for Cloud Service Broker

Service Broker Reference Architecture

Figure 3 describes the high-level architectural setting with a separation of integration, customization, and aggregation functions of a broker. The broker sits between the consumer and a range of different, independent providers. The broker might be an independent third-party service provider. The role of the service templates is indicated. Firstly, it forms the basis of requests formulated by the consumers. Secondly, templates collected from the providers are used for customization, integration, and aggregation by the broker. The detailed structure of the templates will be presented later.

We identify the following as the main description categories in the templates:

- *Operations* as a functional perspective.
- *Quality* to address non-functional SLA and contract concerns.
- *Resources* that define the execution environment and context.
- *Policies* that define governance and technical constraints.

While a reference architecture might be easy to agree upon - since it brings expected features into a common framework - the requirements for the service templates are more difficult to establish as they not only have to describe “what” is in the architecture, but also “how” the architectural framework can be converted into an effective solution. Cross-layer support in a federated context (i.e., vertical and

horizontal integration support) is the critical challenge (Barrett et al., 2006; Buyya, Ranjan, & Calheiros, 2010). The reference architecture thus aims to provide integration in two dimensions: horizontal integration across federated Cloud offerings and vertical integration across the Cloud stack layers (IaaS, PaaS, SaaS).

Cloud Service Broker Interfaces and Stages

We envisage the Cloud consumer to interact with the Cloud service broker in two ways. Firstly, *request submission and Cloud service construction*: this is pre-deployment and pre-provisioning. Secondly, *service monitoring and management*: this is post-deployment during the provisioning of the service.

Figure 4 illustrates the two stages:

Figure 4. Cloud Broker Interfaces and Access to Functions

A *Brokerage Stage* allows the construction (e.g., aggregation) of a possibly complex or customized Cloud services based on individual, independent services. Migration into the Cloud could be facilitated through customization and integration (Jamshidi, Ahmad, & Pahl, 2013). Matching user requirements with descriptions of provided services can be expressed as a refinement of templates where the provider (over)satisfies the user requirements. Data and information that is communicated needs to be integrated through appropriate mediation and transformation techniques. In addition, the integration of wider processes is often necessary, similar to work on document-based service process integration in SOA frameworks such as ebXML.

A *Management Stage* allows the configuration, provisioning, monitoring, and analysis of broker-provided services. The broker is often the direct contract partner, resulting in the need to deal with service-level agreements (SLAs) and billing/payment issues at that level. Monitoring provides the quality and usage data for both quality and payments concerns.

Broker Features and Capability Requirements

In order to clarify the requirements for a service description notation for Cloud service brokerage, we detail the functions we expect to be performed by the broker across the Cloud layers IaaS, PaaS, and SaaS (see Figure 5).

Figure 5. Broker Features and Capabilities

For pre-deployment (Brokerage Stage), the following aspects are relevant:

- *Virtual Machine Portability* can be supported and automated at lower layers by adopting OVF (Open Virtualization Format, which is an open standard for packaging and distributing virtual appliances) (DMTF, 2010) or TOSCA (Topology and Orchestration Specification for Cloud Applications, a standard which specifically supports portability between public and private Clouds) (OASIS, 2013).

- *Expected workload* is a concern that should be considered at design/build-time, as service selection and later provisioning need to take this into account. This requires workload to be made explicit at all layers.

- *Horizontal interoperability* refers to the integration of services within the same Cloud layer, requiring common interfaces. Again, standards compatibility is a solution. OCCI is a lifecycle management standard describing APIs for service management.

- *Vertical interoperability* requires the integration of Infrastructure, Platform and Software application needs. For instance, key performance indicators (KPIs) differ across the layers - while at lower infrastructure layers, network bandwidth, and latency are concerns, further up it is the response time or availability per service. However, higher-level ones need to be mapped down seamlessly to lower layers, and vice versa. Thus, formalized mappings need to be defined.

Integration, interoperability and portability are the key concerns to support the consumer in creating multi-vendor applications and to move or migrate between vendors (Bernstein et al., 2009).

For post-deployment management (Management Stage), the lifecycle support is crucial, covering the configuration and provisioning as well as on-going monitoring and analytics, as follows:

- *Intelligent selection, matching, and configuration support for resources.* Resources need to be configured, based usually on quality requirements from a consumer such that the resources match the needs of the consumer. Intelligent selection, matching and configuration support is necessary.

- *Correlation of user requirements and the resulting abstract configurations* with the available resources provided at the time of provisioning the services. In addition to the configuration concerns, this requires lower-level input from resource managers and workload balancers.

- *Integration of monitored data from different sources* is an essential part of a provisioning platform. Monitoring solutions are available, both commercial and open-source. However, due to the requirement of supporting federated Clouds, the need to collect and integrate monitored data from different sources arises.

- *Interoperability between monitoring sources and the analysis tool.* The requirements are similar with respect to standardization and compatibility, e.g., SLA-related data.

The above discussion provided the requirements list for the service template notation and the operator calculus, which will be discussed in detail in the next section. The operators will be required to support mappings between layers or the matching between required and provided features.

SERVICE DESCRIPTION TEMPLATES – STRUCTURE AND FORMALIZATION

In this section, we introduce and define the service description template for Cloud service brokerage in an ontology format based on the identified requirements. An operator calculus to manipulate templates is introduced afterwards.

A service template is an abstract description of a service's capabilities, in terms of functional and non-functional aspects. Different types of templates to describe the capabilities of Cloud resources have been proposed, often under different names such as Recipes [Chef, Cloudify] (Cloudify, 2013), Blueprints [4CaaSt] (4Caast, 2013) or Manifests [OVF, CompatibleOne] (CompatibleOne, 2013). Based on a review of these in (Fowley, Pahl, & Zhang, 2013), we define a model for Cloud service description for brokerage solutions.

Template Structure

We propose a 4-part structure for the service description template: operation, quality, resources, and policies. The first two cover *service-internal* aspects - the functional and non-functional aspects of the service. The remaining two cover more of the *service-external* perspective with the resources required and also the security and compliance requirements. We define a *4-part hierarchy of concepts*, which we

will later formalize as the core taxonomy of a service description ontology (Bandara, Wang, & Pahl, 2009; Papazoglou & van den Heuvel, 2011).

- *Operation* - the functional service interface. Its subconcepts follow WSDL in its structure:
  - Service type/interface defines the interface as an abstract data type.
  - Operations that provide the functionality of the service.
  - Messages and Types needed to communicate data to and from the service for operation invocations.

- *Quality* - the non-functional service properties. Its subconcepts are common performance-related concerns:
  - Availability of the service in question.
  - Latency/Response Time of the service and its operations.
  - Bandwidth as another example of a quality concern.

- *Resources* - deployment requirements. Its subconcepts are aspects specific to the actual provisioning of a service in a PaaS or IaaS environment:
  - Average/peak workload requirements is a typical example for a PaaS configuration requirement.

- *Policies* - the business agreements and constraints. Its subconcepts are:
  - Security and Privacy concerns in terms of confidentiality, integrity and authenticity as examples.
  - Compliance and Governance as further rules to describe to correct alignment of a service to its execution environment.

While Operation and Quality address standard service description elements, Resources are important as the Cloud deployment capability of the provider is catered for here. Nested reference templates internally refer to resources used. The Policies section allows expressing customer or broker customization needs.

Template Ontology Formalization

We propose to formalise the service template as an ontology. An ontology is a conceptual model formulated in a formal description language. In this context, we utilized the core of the Semantic Web technology, the *Web Ontology Language* OWL. This language is based on description logics, i.e., has a formal definition and is supported by various tools including reasoners that can even be used at runtime. Using Semantic Web technologies brings us closer to interoperability of meaningful descriptions in an Internet context.

An ontology is a representation of concepts (or classes) that are defined in terms of their properties, i.e., relationships to other concepts. Concepts can be instantiated into concrete values. Properties can be object or data valued, depending on whether they refer to another complex object (an example is the relationship of a service to another service through invocation or inheritance) or data (an example is the performance of a service as a numeric value). Concepts can be arranged into a hierarchy using a sub/superconcept relationship. This forms the taxonomy of the ontology.

Our ontology, see Figure 6, has a root concept, which is 'ServiceTemplate'. Using an 'isPartOf' relationship, the four aspects, namely Operation, Quality, Resources, and Policies, are connected to ServiceTemplate. The individual concerns, such as ServiceType or Message for Operation or Availability for Quality, are the subconcepts of the respective four core aspects. These properties define a basic hierarchy.

Figure 6. Service Template Ontology

However, we need a richer ontology that addresses the requirements outlined earlier in the previous section. In terms of interfaces and operations, we need to distinguish the different layers - as ontology subconcepts of the Operation concept. For IaaS-Operation, we define a service API with the corresponding operations that is OCCI-compliant. While VM image management of the lifecycle is relatively uniform (thus, OCCI is possible as a standard), at the PaaS layer, we need to provide for standard tools like databases or application server. For instance a Tomcat server could be modeled as an instance of a Web application server concept. Standardized solutions in terms of APIs/operations would then only be specific to a tool category. Quality concerns (KPIs for the different layers) need to be differentiated too. We assume each quality criterion to be qualified by an attribute (data-valued property) that indicates the relative Cloud layer to which it applies.

Required resources can be explicitly or abstractly specified by referring to other templates that capture the required resource. This is facilitated by a special resource relationship. The need for formality becomes clear here, as circular resource relationships cannot be tolerated and need to be detected. The Policies part needs to link to special kinds of entities - the rules. We propose utilizing the *SWRL language* (Semantic Web Rule Language). The specified rules can be evaluated at build time and dynamically verified.

We have singled out vertical integration as a concern. Here, we have explicitly separated the different Cloud stack layers (e.g., IaaS and PaaS in Figure 6) and have modeled their inter-linkages. Integration across the four template aspects is required. Essentially, compositionality is the solution. We will introduce a composition operator later in the chapter.

We briefly discuss the first two template aspects. For the Operation aspect, for instance, two platform components such as a database and an application server can be composed, provided as a bundle and integrated into a VM as the execution environment. For the Quality aspect, KPI mappings have already been mentioned, where IaaS-level KPIs are aggregated into an integrated PaaS-level KPI. For instance, a PaaS-level service response time depends on IaaS-level network throughput and CPU utilization, among others.

The ontology is meant to be extensible for particular circumstances. For concrete resources, instances are attached. This clarifies the definition of a template – an abstract template is concept-level in terms of the ontology and can be instantiated by providing instance-level values. Templates can be generated from other resources and provide input to further processing. Based on SLAs or other legal agreements, some operational, quality, and policy requirements can be derived that would define a partial template. Service integration plans need to be generated in many cases where a single service will not satisfy the need. A service process needs to be orchestrated in a classical Web service manner, associated to an integration process connecting abstract resources.

While we have proposed a comprehensive ontology capturing all conceptual aspects of the service template language framework, in order to tailor this to the needs of different actors and components of the reference architecture, facilitation through different mechanisms would be beneficial. As core technical aspects, we can adopt a structure proposed in (Nguyen et al., 2011):

- Service template *definition* (Operation and Quality) addressing the service internal aspects.
- Service template *configuration/instantiation* (Resources) addressing the first type of external aspects.
- Service template *constraints* (Policies) also addressing external aspects as the ones above, but as already explained, a rule language rather than an ontology language needs to be facilitated here.

TEMPLATE MANIPULATION OPERATORS – OPERATOR CALCULUS

These core template mechanisms (definition, configuration and constraints) that form a service template ontology can be complemented by two types of *template manipulation operators* - core composition and non-compositional manipulation (Benslimane, Dustdar & Sheth, 2008). Service Template Manipulation provides non-compositional template manipulation. This could include the refinement or specialization of a template, or a merger or restriction. On the other hand, the Service Template Composition (Integration Processes) deals with service aggregation, by supporting a range of standard composition operators, to allow services to be orchestrated into a process across different Cloud providers by combining lower stack layer components to higher-layer ones.

In the implementation, we follow work on architecture languages (Pahl, Giesecke, & Hasselbring, 2007) and adapt an ontology operator calculus for architecture manipulation in the Cloud service description context. We utilize ontology-based architecture and pattern languages that combine composition and abstraction mechanisms. The conceptual core, particularly for the manipulation and composition aspects is an operator calculus, including operations such as match/select, merge, restrict, refine, behaviorally compose, and abstract/instantiate, to support the following core manipulation and composition activities:

- Basic Operators such as *Rename, Restrict, Union, and Intersection* allow standard service specification changes to be carried out.

- *Refinement* allows the derivation of a related service description that preserves functional properties (the scope of the service can be tailored through restrict and union operations). Instantiation, i.e., providing concrete values for an abstract concept-level template in order to denote a concrete service.

- *Composition* uses aggregation operators such as sequence, selection, and iteration to aggregate and orchestrate services.

While some operators are generic and suitable for changing and adapting software specifications in general, the combination and particularly the more advanced ones like refinement and composition specifically address the Cloud brokerage needs for service composition (as already explained, the template structure is also Cloud-specific):

- The basic operators allow mappings between Cloud layers and Cloud service integration to be facilitated. For instance, lower-layer concerns such as bandwidth can be filtered out through restriction and replaced by other metrics.

- In the Cloud service broker context, refinement formalizes the requester-provider matching. It is also a tool to facilitate Cloud service customization.

- Composition based on a selection of Cloud services form a composite service process serving a more complex goal, specifically addressing aggregation brokerage, both horizontally and vertically.

Formalization

Each service template is defined by a separate ontology-based specification. In order to reuse service templates as specification artifacts, the templates need to be related to each other, e.g., to be compared to each other or derived from another. Different templates can be related based on ontology relationships. We introduce the central operators - renaming, restriction, union, intersection, and refinement. Instead of

general ontology mappings, we introduce a notion of a template specification and define template manipulation and composition operators on it.

Template Syntax and Semantics

Before defining the operators, the notions of template specification and their semantics need to be made more precise. We assume a template to be a specification $Templ = <\sum, \Phi>$ based on the template ontology with:

- A signature $\sum = <C,R>$ consisting of template concepts C like *Operation* or *Quality* and roles R (i.e., concept properties as relations on concepts).
- A set of concept descriptions (conjunction, disjunction, negation, quantification) $\varphi \in \Phi$ based on $\sum$.

A template Templ is interpreted by a set of models *M*. A model is an algebraic structure that satisfies all concept descriptions $\varphi \in \Phi$. The set *M* contains algebraic structures (or models) $m \in M$ with sets of objects *CI* for each concept *C* and relations $RI \subseteq C_iI \subseteq C_jI$ (where $\subseteq$ refers *to a* subsumption, i.e., inclusion of respective set interpretations) for the roles $R : C_i \rightarrow C_j$, such that *m* satisfies the concept description. This satisfaction relation is defined as usual, inductively over the connectors of the description logic ALC (Attributive Concept Language with Complements). The combination of two templates should be conflict-free, i.e., no semantic contradictions should occur. A consistency condition can be verified by ensuring that the set-theoretic interpretations of two templates S1 and S2 are not disjoint, $S1^I \cap S2^I \neq \varnothing$, i.e., their combination is satisfiable and no contradictions occur.

Renaming

Cloud service template development might require elements to be renamed, e.g., during customization. A renaming operator can be defined element-wise for a given signature $\sum$. By providing mappings for the elements to be modified, a new signature $\sum'$ is defined:

$$\sum' = \sum [\, n_1 \rightarrow n_1', \ldots, n_m \rightarrow n_m' \,]$$

for all names of concepts or roles n_i ($i=1, \ldots, m$) of $\sum$ that need to be modified. Generally, renaming is used at instance level or other levels deeper in the ontology hierarchy as the upper layers are part of the pre-defined framework and should generally not be changed.

Restriction

While often service templates are used as-is in combinations and relationships, it is sometimes required to focus on selected parts, for instance, before refining a template. Restriction is an operator that allows template combinations to be customised and undesired elements (and their properties) to be removed. A restriction is a projection that can be expressed using the restriction operator $<\sum, \Phi>|\sum'$ for a specification, defined by:

$$<\sum, \Phi>|\sum' := <\sum \cap \sum', \{\, \varphi \in \Phi \mid rls(\varphi) \in rls(\sum \cap \sum') \wedge cpts(\varphi) \in cpts(\sum \cap \sum') \,\} >$$

assuming usual definitions of role projections $rls(\sum) = R$ and concept projections $cpts(\sum) = C$ on a signature $\sum = <C,R>$.

Restriction preserves consistency as constraints are, if necessary, removed. Restriction would allow specific Cloud stack layers to be excluded or quality aspects to be left unspecified if that is required.

Intersection and Union

Adding elements of one template to another or removing specific properties from a template is often required, e.g., if quality metrics of a different Cloud stack layer need to be added or filtered out, or if two services need to be joined in their capabilities. Union and intersection deal with these situations, allowing adding or removing, respectively. Two Cloud service templates S1 = < ∑1, Φ1 > and S2 = < ∑2, Φ2 > shall be assumed.

- The *intersection* of S1 and S2, expressed by S1 ∩ S2, is defined by S1 * S2 := < ∑1 ∩ ∑2 , (Φ1 ∪ Φ2)| {∑1 ∩ ∑2} >. Intersection is semantically defined based on an intersection of template interpretations, achieved through projection onto common signature elements.

- The *union* of S1 and S2, expressed by S1 ∪ S2, is defined by S1 + S2 := < ∑1 ∪ ∑2 , Φ1 ∪ Φ2 > . Union is semantically defined based on a union of template interpretations.

In the case of fully different templates, their intersection results in the elementary aspects and their properties. Both operations can result in consistency conflicts.

Refinement and Instantiation

Consistency is a generic requirement in automated environments such as the Cloud that should apply to all combinations of service ontologies. A typical situation is the derivation of a new template from an existing one. The refinement operator that we are going to introduce is a consistent type of template derivation. Refinement can be linked to the subsumption relation and semantically constrained by an inclusion of interpretations, i.e., the models that interpret a template. Refinement carries the connotation of preserving existing properties, for instance, the satisfiability of the original template specification for template-based Cloud service matching.

A consistency-preserving refinement operator provides a constructive subsumption variant that allows new subconcepts and new subrelationships to be added and new constraints to be added if these apply consistently to the new elements. Assume a template S = < ∑ , Φ >. For any other template < ∑', Φ' > with ∑ ∩ ∑' = ∅, we define a *refinement* (⊕) of S by < ∑', Φ' > as follows:

$$S \oplus < \sum', \Phi' > := < \sum + \sum', \Phi + \Phi' >$$

The precondition ∑ ∩ ∑' = ∅ implies that consistency is preserved: Φ ⊓ Φ' = < (⊓ refers to a conjunction, i.e., intersection of respective set interpretations; and < refers to conjunction of all sets, i.e., empty set). In this situation, existing properties of template specification S = < ∑, Φ > would be inherited by S ⊕ < ∑', Φ' >. Existing relationships can in principle be refined as long as consistency is maintained - which might require manual proof in specific situations that go beyond the operator-based application. Instantiation is essentially a form of refinement, adding an instance-level concept.

Composite Concepts in Service Templates

Explicit composition support is an important element of conceptual modeling languages. Composition is also central for Cloud service brokerage, specifically for aggregation brokering. Subsumption (⊑) is usually the central relationship in ontology languages, allowing concept taxonomies to be defined in terms of subtype or specialization relationships. In conceptual modeling, composition is also a fundamental relationship that focuses on the part-whole relationship between concepts or objects. In ontology languages, subsumption is well understood and supported, but composition is not.

The notion of composition is applied in the context of software descriptions in two different ways:

- *Structural composition*: Structural hierarchies of some aspects define an important aspect of services. Structural composition can be applied to service data or quality descriptions.

- *Behavioral and sequential composition*: Dynamic elements can be composed to represent sequential behavior. Connectors are usually seen as dynamically oriented architectural elements. Extending the idea of sequential composition, a number of behavioral composition operators including choice and iteration can describe interaction behavior, e.g., to service operations.

Structural composition can be applied to templates in general. Sequential/behavioral compositions can only be applied to concepts that can be associated with behavioral semantics, such as aspects of the Operations ontology facet. Composition is syntactically used in the same way as subsumption to relate concept descriptions.

Service compositions shall consist of unordered subcomponents, expressed using the composition operator .. An example is *Service* . *Operation*, meaning that a Service consists of Operations as parts. In order to provide this with an adequate semantics, these concepts are interpreted by unordered multisets. Operations can be ordered into sequences or can have complex behaviors that consist of ordered process elements, again expressed using the composition operator .. An example is *Process* . [*Operation1, Operation2*], saying that Process is actually a composite process, which contains, for instance, two *Operation* elements. We semantically define composite elements as ordered tuples providing a notion of sequence. For more complex behavioral compositions, graphs serve as models to interpret the behavior.

We introduce two basic syntactic forms, before looking at behavioral composition as an extension of sequential composition:

- The *structural composition* between template concepts *C* and *D* is defined through *C* . { *D* }, i.e., *C* is structurally composed of *D*. Structural composition into multiple elements *C* . { *D1, … , Dn* } is defined by:

  $$C \ . \{ D1 \} \sqcap \ldots \sqcap C \ . \{ Dn \}$$

  where the parts *Di, i=1,..,n* are not assumed to be ordered.

- The *sequential composition* between template concepts *C* and *D* is defined through *C* . [*D*], i.e., *C* is sequentially composed of *D* if *type(C) = type(D) = Operator*. The sequential composition operator is specific to behavior-oriented template concepts, i.e., operation. Sequential composition into multiple elements *C* . [*D1, … , Dn*] is defined by:

  $$C \ . [D1] \sqcap \ldots \sqcap C \ . [Dn]$$

  where the parts *Di* with *i=1,..,n* are assumed to be ordered with *D1* < … < *Di* < … *Dn*.

We can now formalize the semantics of these composition operators. So far, we assumed models $m \in M$ as algebraic structures consisting of sets of objects C^I for each concept *C* in the template signature and relations R^I for roles *R*. Now, we assume objects to be composite:

- Structurally composite concepts *C* . { *D1, … , Dn* } are interpreted as multisets, i.e., $C^I = \{\{ D1^{I1}, \ldots, D1^{Ik}\}, \ldots, \{ Dn^{I1}, \ldots, Dn^{Ik} \}\}$. We allow multiple occurrences for each concept *Di, i=1,..,n* which is a part of concept *C*. With $c \in C^I$ we denote set membership.

- Sequentially composite concepts C . [$D1, \ldots, Dn$] are interpreted as tuples, i.e., $C^I = [\ D1^I, \ldots, Dn^I\ ]$. Tuples are ordered collections of sequenced elements. In addition to membership, we assume here an index-based access to these tuples in the form $C^I(i) = Di^I,\ i=1,..,n$ that selects the i-th element in a tuple.

While subsumption as a relationship is defined through subset inclusion, the composition relationships introduced here are defined through membership in collections (multisets for structural composition and tuples for behavioral composition).

The introduction of behavioral specification is specific to composition of functional template elements like operations. This operator allows us to refine a service and specify detailed behavior in the form of a process constraint on the execution order of operations. While basic behavior in the form of sequencing has been defined above, we now introduce a more expressive operator that requires a more complex semantic model, graphs in this case. We can define a service process P through a *behavioral specification* P . [B] where B is a behavioral expression consisting of:

- A basic operation B or
- A unary operator '!' applied to a behavioral expression ! B, expressing *iteration*, or
- A binary operator '+' applied to two behavioral expressions $B1 + B2$, expressing *non-deterministic choice*, or
- A binary operator ';' applied to two behavioral expressions $B1$; $B2$, expressing the previously introduced *sequence*.

This behavioral composition syntax can now be embedded into the basic forms of composition:

- The iteration P . [! B] is defined by P . [B , …, B]
- The choice P . [$B1 + B2$] is defined by P . [$B1$] $\sqcup$ P . [$B2$] ($\sqcup$ refers to disjunction, i.e., union of respective set interpretations)
- The sequence P . [$B1$; $B2$] is defined as introduced in the sequential composition.

We extend the semantics by interpreting behaviorally composite operations (processes) through graphs (N,E), where operations are represented by edges $e \in E$ and nodes $n \in N$ represent connection points for sequence, choice and iteration.

The three operators are defined through simple graphs: ({$n1$,$n2$},{($n1$,$n2$)}) for a sequence, ({n},{(n,n)}) for an iteration, and for choice we define ({$n1$,$n2$,$n3$,$n4$},{($n1$,$n2$),($n2$,$n4$),($n1$,$n3$),($n3$, $n1$)}) for a simple example.

VALIDATION IN A SOLUTION ARCHITECTURE

In order to demonstrate the validity of the conceptual ontology-based template description and manipulation framework in terms of the completeness and necessity of the modeling constructs, we look at different aspects. We first describe a sample Cloud architecture in which our solution could be utilized to address the need for service templates for brokerage. We then introduce a use case and apply the template description and operators to address the validity the conceptual framework. A full implementation including semantic technologies or an empirical evaluation is however beyond the scope of this chapter, since our focus was the conceptual framework, not on a finalized language and supporting tools.

Architecture

An implementation architecture for the service templates is a brokerage test-bed currently developed at IC4 (a Cloud research centre at Dublin City University), where a combination of OpenStack as the IaaS solution and CompatibleOne as the core PaaS broker solution is being developed. This serves us as an implementation example to investigate the vertical integration of IaaS, core PaaS and broker layers. While CompatibleOne is a PaaS broker, the support provided does not satisfy the requirements of our needs here.

OpenStack (OpenStack, 2013) is a private Cloud enabling solution that transforms a physical hardware infrastructure into an Infrastructure as a Service (IaaS). This Cloud platform can be used to run Cloud compute and storage infrastructure. There are five main services exposed by the OpenStack platform: Compute Infrastructure (Nova), Storage Infrastructure (Swift), Imaging Service (Glance), Identity management (Keystone) and Network management (Quantum). The OCCI lifecycle operation API based on resources and capabilities is encompassed by the template ontology here.

CompatibleOne (CompatibleOne, 2013) is a French-based academic and industry consortium that set out to provide a framework to allow for portability and interoperability of Cloud software. The project focuses on producing a broker platform to provide intermediation, aggregation and arbitration of Cloud services. This allows Cloud service consumer requirements to be mapped to Cloud provider resources. The project follows the OCCI standard, which enabled us to link to OpenStack. It consists of the CORDS information model and the ACCORDS software platform. The CORDS model is as an open, object-oriented, OCCI-compliant representation of Cloud computing entities that can be used to build complex Cloud middleware.

The ACCORDS platform is a suite of Cloud middleware programs that provide Cloud broker services. Using ACCORDS, a customer can describe its service requirements in a CORDS-compliant format. These can be matched to available Cloud provider offerings and then ultimately provisioned at run-time by interacting with the actual Cloud vendor resources. The model includes representations for service level agreements, pricing structures and transaction management for usage billing, but the descriptions are less comprehensive than our template framework.

Figure 7 describes the architectural setup of OpenStack and Compatible One (which we run on top of VirtualBox as the virtualization environment) that we use as the experimentation environment – we focus on the Cloud settings and ignore the semantic technology support.

Figure 7. Architecture for OpenStack and CompatibleOne Brokerage Environment

Use Cases

The adequacy of the proposed service template model shall be discussed using use cases derived from the broker definition as implementations in the above described testbed architecture.

First, we look at the broker functions - Customization, Aggregation, and Integration. These form use cases, which need to be supported by adequate operators. We have proposed an ontology-based operator calculus that provides suitable support. Restriction and renaming allow syntactic integration. Refinement allows customization at a semantic level, e.g., changing functionality. The composition operators that we propose allow aggregation. Integration is facilitated by allowing semantic integration through a rich ontology and a rule-based approach to control the integration into specific environments. Thus, specifically, the manipulation and composition operators are necessary to fully support service brokerage. An ontology-based solution helps with integrating multi-faceted aspects into a coherent manipulation and reasoning framework.

The CORDS and ACCORDS integration is a second aspect. The CORDS model with its lower, OCCI-based description layer provides a standards-based platform to enable its utilization in a concrete OCCI-compliant IaaS solution, i.e., vertical integration. The CORDS model relating to the ACCORDS platform features can be fully represented in terms of the service template notation. ACCORDS services are instantiated service templates. The OCCI-compliant OpenStack forms the execution platform. With CompatibleOne, we have used an advanced broker platform (see our comparison earlier). Thus, we can be confident that the vertical layering problem between IaaS and PaaS can be addressed for a rich, multi-faceted description template. Assuming that OCCI-based IaaS services and ACCORDS platform service are specified as templates, the refinement and composition allow semantic mapping where a broker-level PaaS metric like performance can be mapped (refined or decomposed) into lower-level IaaS ones based on bandwidth, latency, or throughput as network performance metrics:

$$Performance \; . \{ Bandwidth \sqcup Latency \sqcup Throughput \}.$$

While we have not fully implemented the proposed service template model as an extension of the CORDS matching, our experiments with CompatibleOne support the feasibility of a full integration of the two Cloud solutions. These experiments validate the scope that a service description template has to cover in terms of vertical integration. Horizontal integration has been addressed through composition and integration.

Template Specification

Now, we illustrate template specification and manipulation mechanisms based on the Cloud service broker requirements in the wider context of the desktop virtualization architecture and scenario. The need for these operators arises from several reasons.

- Firstly, even if syntactic interoperability in terms of formats is given, the requirements and specifications of provided service still need to be matched. Renaming, restriction, and refinement allow this to happen.

- Secondly, Union and Intersection, but also Refinement and Structural Composition, allow templates to be further developed and changed.

- Thirdly, behavioral aspects of composition can be formulated using the Sequential and Behavioral Composition. Although not applicable to all ontology elements, behavioral composition is important in order to define processes (which are of importance in the context of business process-as-a-service aspects in the Cloud).

We have already motivated these constructs using Cloud brokerage examples in previous sections. The template approach is adapted to the Cloud needs as described earlier. The template structure combines common features from service description approaches on functional and non-functional aspects, but adds Cloud-specific concerns such as using resource abstractions and a wider frame to capture compliancy and other extended SLA concerns.

General service properties relevant to Cloud service specifications can be expressed in a template using the ontology language, e.g., for quality aspects in the template category “Quality” we can express that security and performance are difficult to achieve together:

$$Security \sqcup Performance = <$$

Furthermore, you can state that security is a combination of different sub-aspects:

$$Security \sqsupseteq Integrity \sqcap Authentication \sqcap Non\text{-}repudiation \sqcap Confidentiality.$$

Roles in this context can be defined as triples linking two concept instances through a role:

$$hasPerformance(Service, 10ms)$$

With an extension to a rule language, derivations of quality aspects can also be formulated as:

$$Service(?s) \wedge HasSecurity(?s; ?x) \wedge stringEqual(?x, \text{"}high\text{"}) \wedge HasResponseTime(?s; ?y) \rightarrow GreaterThan(?y, 100ms).$$

Now we look at the template manipulation operators that can be applied to templates. For instance, different aspect concepts and roles can be renamed, e.g., for the '*Resource*' category of the template, which is important for the desktop virtualization context:

$$\sum' = \sum [\{ Platform \rightarrow OS \} ; \{ hasPlatform \rightarrow hasOperatingSystem \}]$$

These types of mappings would support the virtual desktop scenario, where different OS platforms with different naming conventions can be hosted. A restriction is a projection that can be specified through

$$rls(\varphi) = \{ hasIntegrity \} \text{ and } cpts(\varphi) = \{ Integrity \}$$

A refinement adds specific properties, e.g., if specific features of a virtual environment depend on the device:

$$<\sum, \Phi> \oplus < \{ Device, Feature \} ; \{ hasDevice, hasFeature \} >$$

Two composition examples are given below. A structural composition could be applied if security is specified. This might be useful, if security mechanisms are implemented through different components for integrity checks or authentication.

$$Security \ . \{ Confidentiality, Integrity \}$$

A behavioral composition can be used to detail a sequence in which a number of service operations need to be executed (here for a desktop initialisation service):

$$InitializationProcess \ . [ConfigService, ProvisioningService, StartUpService]$$

FUTURE RESEARCH DIRECTIONS

Cloud brokers and Cloud marketplaces are different components that emerge within the context of Cloud service brokerage. Both create their own challenges for the future.

Brokers need to automate the process of matching service requirements with resource capacity and capabilities as far as possible (Rodero-Merino et al., 2010). The ideal solution would be a step towards utility computing, i.e., a commoditization of Cloud resources such that any compute resource from a private Cloud or a public instance could be plugged into a user's compute capacity. Therefore, interoperability will remain of key importance. In this regard, we need to look at new areas of

compatibility to be considered in matching services that are not handled by brokers currently. For example, none of the current open-source solutions consider data integrity as a matching criterion; however, they all include performance as part of their concerns. Security is another aspect that has a technical dimension, but is also abstract insofar as it can be implemented by a Cloud provider.

The marketplace will need in addition to focus on the architecture of both the applications and the Cloud. The appstore model appears to be the de-facto model of choice for marketplaces, but this appears more an admission of the success of the dominating initiatives rather than research. There may be a potential to explore other forms of the online marketplaces suitable to Cloud apps and their composition (Fehling & Mietzner, 2011; Mietzner, Leymann, & Papazoglou, 2008). This could also be pushed to an even more commodity-based scenario where services could be registered on a multi-marketplace scale.

Service and resource commoditization for brokers and marketplaces can be facilitated through a uniform representation in the form of description templates such as recipes, manifests and blueprints. These need to cover the architecture stack and meet language and quality concerns. Commoditization is an enabler of marketplace functions that sits on top of a broker. Thus, additional challenges and requirements for marketplaces are (4CaaSt, 2013; Papazoglou & van den Heuvel, 2011):

- Data integration and security enforcement as non-functional requirements.
- Social network functions to allow service ratings by the communities.
- SLA management to be integrated, e.g., in terms of monitoring results.

Commoditization needs to be facilitated through an operational development and deployment model. It therefore acts as an enabler. Trust is an equally important concern that is more difficult to facilitate technically than commoditization. A mechanism is needed for not only vetting individual providers, but also to allow this to happen in layered, federated and brokered Cloud solutions.

In another extension of this idea, there has not been a proliferation of Cloud capacity clearing-houses that would operate similar to a spot market to allow Clouds to buy and sell spare Cloud capacity on a very short-term basis. It is not clear what new areas of research would be needed to facilitate such a movement in the Cloud. It seems reasonable that, with the continued commoditization of the Cloud by brokers and marketplaces, such a trend could be seen eventually.

Distribution and federation are other requirements for brokerage (Buyya, Ranjan, & Calheiros, 2010), i.e., to work across independently managed and provided heterogeneous Cloud offerings. Reference architectures, e.g., the NIST Cloud brokerage reference architecture needs to be extended. The management of configuration and deployment based on integrated and/or standards-based techniques needs to be addressed (Konstantinou et al., 2009). Distribution mechanisms such as federation and organization and coordination techniques for clustering need to be investigated as forms of distributed Cloud architectures.

CONCLUSION

Standards and compatibility are effectively the only way to make the vision of a truly interoperable Cloud service broker work. In (Fowley, Pahl, & Zhang, 2013), we noted a trend towards Cloud marketplaces utilizing brokerage facilities. The proposed solution in this chapter is a first step in this direction: partly, through the language support in terms of request specification languages and powerful template manipulation operators that will support central marketplace functions (like matching and selection); and the reference architecture which is the other core solution that defines how the architectural components interact and how the service templates are processed in this context.

Abstract descriptions of Cloud services (in the form of templates) are the key to a service brokerage solution in an interoperable environment. In this report, we presented a conceptual model for the service templates. We defined the core properties as an ontological model and suggested a range of operators to support this feature. The motivation and promotion of an ontology-based solution was one of our objectives here.

More utility and the commoditization of Cloud services is an emerging need from the discussion of brokers and markets. A trend is to move from lower IaaS layers to PaaS and onwards to encompass SaaS, aiming to integrate lower layers. To make this work, services at all layers need to be provided in a uniform way to allow processing in terms of selection, adaptation, integration and aggregation. Commoditization is the concept to capture this need. Some concrete observations related to the reviewed Cloud platforms, which are addressed in our conceptual framework, are: fully functional vertical stack integration capabilities, operational support of Cloud service composition, and rich manipulation support of service abstractions.

The work presented here only forms the conceptual framework for a template notation that can be used in brokerage and marketplace context. More work needs to be done in order to integrate the framework with concrete API formats and service descriptions and evaluate the approach empirically.

KEY TERMS & DEFINITIONS

Broker Reference Architecture: A Cloud service broker assumes some common components, connections and interactions to be present to fulfil its tasks. A broker reference architecture defines some common components, connectors, and interactions along these connectors for a service broker.

Cloud Broker: A Cloud broker is a service in a Cloud environment that mediates between service requestors and service providers. A Cloud broker facilitates initial match-making based on requirements and capability descriptions and also manages the dynamic integration.

Cloud Service Brokerage: Cloud service brokerage is the process of integrating, customizing, or aggregating Cloud services for service consumers. Widely accepted Cloud service brokerage definitions are provided by US standardization organisation NIST and the analyst Gartner.

Cloud Service Template: A Cloud service template is a service description that captures Cloud-specific characteristics in an abstract and instantiable format.

Federated Cloud Architecture: A federated Cloud architecture assumes that individual components of the traditional Cloud stack are distributed across possibly independently controlled nodes.

Service Description: A service description captures the functional and non-functional characteristics of a service in a processable format.

Service Ontology: A service ontology is a knowledge representation format capturing service information in a rich conceptual modeling framework that enables description, manipulation, and reasoning.

Template Manipulation Calculus: Templates are specifications that need to be manipulated to allow their adaptation to specific situations. The manipulation operators include single template changes such as

renaming, restriction, and refinement, and combination operators such as union, intersection, and composition.

REFERENCES

4Caast. (2013). 4CaaSt PaaS Cloud Platform. Retrieved November 12, 2013, from http://4caast.morfeo-project.org/.

Barrett, R., Patcas, L.M., Pahl, C., Murphy, J. (2006). Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. In *Proceedings of the International Conference on Web Engineering*, pp.129-136.

Benslimane, D., Dustdar, S., & Sheth, A. (2008). Services Mashups: The New Generation of Web Applications. *Internet Computing*, *12*(5), 13-15.

Bernstein, D., Ludvigson, E., Sankar, K., Diamond, S., & Morrow, M. (2009). Blueprint for the interCloud-protocols and formats for Cloud computing interoperability. In *Fourth International Conference on Internet and Web Applications and Services*, pp. 328-336.

Buyya, R., Ranjan, R., & Calheiros, R.N. (2010). InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing.*

Cloud Foundry. (2013). Open Source PaaS Cloud Provider Interface. Retrieved November 12, 2013, from http://www.Cloudfoundry.org/.

Cloudify. (2012). Cloudify Open PaaS Stack. Retrieved November 12, 2013, from http://www.Cloudifysource.org/.

CompatibleOne. (2013). Open Source Cloud Broker. Retrieved November 12, 2013, from http://www.compatibleone.org/.

DeltaCloud. (2013). DeltaCloud REST Cloud abstraction API. Retrieved November 12, 2013, from http://deltaCloud.apache.org/.

Fehling, C., & Mietzner, R. (2011). Composite as a Service: Cloud Application Structures, Provisioning, and Management. *IT - Information Technology*, *53*(4), 188-194.

Fowley, F., Pahl, C., & Zhang, L. (2013). A Look at Cloud Architecture Interoperability through Standards. In *Proceedings of the Fourth International Conference on Cloud Computing, Grids and Virtualization*.

Gartner. (2012). Cloud Services Brokerage. Gartner Research. Retrieved from http://www.gartner.com/technology/research/Cloud-computing/Cloud-services-brokerage.jsp.

Jamshidi, P., Ahmad, A., & Pahl, C. (2013). Cloud Migration Research: A Systematic Review *IEEE Transactions on Cloud Computing.* Retrieved November 12, 2013, from http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6624108&punumber%3D6245519.

JClouds. (2013). jClouds Java and Clojure Cloud API. Retrieved November 12, 2013, from http://www.jClouds.org/.

Konstantinou, A.V., Eilam, T., Kalantar, M., Totok, A.A., Arnold, W., & Snible, E. (2009). An architecture for virtual solution composition and deployment in infrastructure Clouds. In *Proceedings of the international workshop on Virtualization technologies in distributed computing*, pp. 9-18.

LibCloud. (2013). Apache LibCloud Python library. Retrieved November 12, 2013, from http://libCloud.apache.org/.

Mietzner, R., Leymann, F., & Papazoglou, M. (2008). Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. In *Proceedings of the 3rd Internet and Web Applications and Services Conference.*

Mosaic. (2013). mOSAIC Multiple Cloud API. Retrieved November 12, 2013, from http://www.mosaic-Cloud.eu/.

NIST. (2011). Cloud Computing Reference Architecture. NIST. Retrieved November 12, 2013, from http://www.nist.gov/customcf/get_pdf.cfm?pub_id=909505.

Nguyen, D.K., Lelli, F., Taher, Y., Parkin, M., Papazoglou, M.P., & van den Heuvel, W.-J. (2011). Blueprint Template Support for Cloud-Based Service Engineering. In *Proceedings of Serviceware Towards a Service-Based Internet.*

OpenShift. (2013). Cloud computing platform as a service. Retrieved November 12, 2013, from https://openshift.redhat.com/.

OpenStack. (2012). OpenStack Open Source Cloud Computing Software. Retrieved November 12, 2013, from http://www.openstack.org/.

Pahl, C., Giesecke, S., & Hasselbring, W. (2007). An Ontology-based Approach for Modelling Architectural Styles. In *Proceedings of the European Conference on Software Architecture*, pp. 60-75.

Papazoglou, M.P., & van den Heuvel, W.J. (2011). Blueprinting the Cloud. *IEEE Internet Computing* 15(6).

Rodero-Merino, L., Vaquero, L.M., Gil, V., Galán, F., Fontán, J., Montero, R.S. & Llorente, I.M. (2010). From Infrastructure Delivery to Service Management in Clouds. *Future Generation Computer Systems*, *26*, 226-240.

simpleAPI. (2013). Simple API for XML. Retrieved November 12, 2013, from http://en.wikipedia.org/wiki/Simple_API_for_XML.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A. … Stoica, I. (2010). A view of Cloud computing. *Communications of the ACM*, *53*(4), 50-58.

Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., & Holley, K. (2008). SOMA: A method for developing service-oriented solutions. *IBM System Journal, 47*(3).

Bell, M. (2008). *Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture*. Wiley.

Benson, T., Sahu, S., Akella, A., & Shaikh, A. (2010). Peeking into the Cloud: Toward User-Driven Cloud Management. In *Proceedings of Clouds Conference.*

Bruneliere, H., Cabot, J., & Frederic, J. (2010). Combining model-driven engineering and Cloud computing. In *Proceedings of Modelling, Design, and Analysis for the Service Cloud.*

Buyya, R., Broberg, J. & Goscinski, A. (2011). *Cloud Computing - Principles and Paradigms.* Wiley.

Cai, H., Zhang, K., Wang, M., Li, J., Sun, L., & Mao, X. (2009). Customer centric Cloud service model and a case study on commerce as a service. In *Proceedings of the IEEE International Conference on Cloud Computing.*

Celesti, A., Tusa, F., Villari, M., & Puliafito, A. (2010). Improving virtual machine migration in federated Cloud environments. In *Second International Conference on Evolving Internet (INTERNET),* pp. 61-67.

Chieu, T., Mohindra, A., Karve, A., & Segal, A. (2010). Solution-based deployment of complex application services on a Cloud. In *Proceedings of the IEEE International Conference on Service Operations and Logistics and Informatics*.

Cloud Standards Customer Council. (n.d.). Cloud Standards. Retrieved November 12, 2013, from http://Cloud-standards.org/wiki/index.php?title=Main_Page

Dowell, S., Barreto, A., Michael, J.B., & Shing, M.T. (2011). Cloud to Cloud interoperability. In *Proceedings of the 6th International Conference on System of Systems Engineering*, pp. 258-263.

Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., & Pohl, K. (2008). A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering, 15*(3), 313-341.

Endo, P.T., Gonçalves, G.E., Kelner, J., & Sadok, D. (2010). A survey on open-source Cloud computing solutions. In *Proceedings of the VIII Workshop on Clouds, Grids and Applications*, pp. 3-16.

Galan, F., Sampaio, A., Rodero-Merino, L., Loy, I., Gil, V., & Vaquero, L.M. (2009). Service specification in Cloud environments based on extensions to open standards. In *Proceedings of the International ICST Conference on COMmunication System softWAre and middlewaRE*, 19:1-19:12.

Greer, M., & Martin, L. (2012). Mission Resiliency via Interoperability and Data Portability in the Cloud. Retrieved from http://safegov.org/2012/4/23/mission-resiliency-via-interoperability-and-data-portability-in-the-Cloud.

Kourtesis, D., Bratanis, K., Friesen, A., Verginadis, Y., Simons, A.J.H., Rossini, A., Schwichtenberg, A., & Gouva, P. (in press). Brokerage for Quality Assurance and Optimisation of Cloud Services: an Analysis of Key Requirements. In *Proceedings of the Cloud Service Brokerage Workshop.* Springer.

La, H.J., & Kim, S.D. (2009). A Systematic Process for Developing High Quality SaaS Cloud Services. In *Proceedings of the International Conference on Cloud Computing*, pp. 278-289.

Maximilien, E.M., Ranabahu, A., Engehausen, R., & Anderson, L.C. (2009). Toward Cloud-agnostic middlewares. *OOPSLA Companion,* 619-626.

Nezhad, H.R.M., Benatallah, B., Casati, F., & Toumani, F. (2006). Web services interoperability specifications. *Computer, 39*(5), 24-32.

Nguyen, D.K., Lelli, F., Papazoglou, M.P., & Van Den Heuvel, W.J. (2012). Blueprinting approach in support of Cloud computing. *Future Internet*, *4*(1), 322-346.

OASIS. (2012). Topology and Orchestration Specification for Cloud Applications Version 1.0.

OCCI. (n.d.). Open Cloud Computing Interface. Retrieved November 12, 2013, from http://occi-wg.org/

OpenNebula. (2012). OpenNebula - Open Source Data Center Virtualization. Retrieved November 12, 2013, from http://opennebula.org/.

Optimis. (2013). Optimis - Optimized Infrastructure Services. Retrieved November 12, 2013, from http://www.optimis-project.eu/.

OVF. (n.d.). Open Virtualization Format. Retrieved November 12, 2013, from http://www.dmtf.org/sites/default/files/standards/documents/DSP0243_1.1.0.pdf.

Pahl, C. & Xiong, H. (2013). Migration to PaaS Clouds - Migration Process and Architectural Concerns. In *Proceedings of the IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pp. 86-91.

Pahl, C., Xiong, H., & Walshe, R. (2013). A Comparison of On-premise to Cloud Migration Approaches. In *Proceedings of the European Conference on Service-Oriented and Cloud Computing*, pp. 212-226. Springer.

Pahl, C., Giesecke, S., & Hasselbring, W. (2009). Ontology-based Modelling of Architectural Styles. *Information and Software Technology*, *1*(12), 1739-1749.

Petcu, D. (2011). Portability and interoperability between Clouds: challenges and case study. In *Proceedings of Serviceware Towards a Service-Based Internet*, pp. 62-74.

Petcu, D., Craciun, C., Neagul, M., Lazcanotegui, I., & Rak, M. (2011). Building an interoperability api for sky computing. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*, pp. 405-411.

Sotomayor, B., Montero, R.S., Llorente, I.M., & Foster, I. (2009). Virtual infrastructure management in private and hybrid Clouds. *IEEE Internet Computing*, *13*(5), 14-22.

Thrash, R. (2010). Building a Cloud computing specification: fundamental engineering for optimizing Cloud computing initiatives. *Computer Science Corporation Whitepaper.*

Tsai, W.-T., Sun, X., & Balasooriya, J. (2010). Service-Oriented Cloud Computing Architecture. In *Proceedings of the Seventh International Conference on Information Technology New Generations*, pp. 684-689.

Wang, M.X., Bandara, K.Y., & Pahl, C. (2010). Process as a service distributed multi-tenant policy-based process runtime governance. In *Proceedings of the IEEE International Conference on Services Computing*, pp. 578-585.