View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by INRIA a CCSD electronic archive server

Rapport de prospective sur l'interopérabilité dans le monde du Cloud et du SaaS

Clément Quinton, Daniel Romero, Laurence Duchien, Lionel Seinturier

To cite this version:

Clément Quinton, Daniel Romero, Laurence Duchien, Lionel Seinturier. Rapport de prospective sur l'interopérabilité dans le monde du Cloud et du SaaS. [Rapport de recherche] Inria. 2014. hal-01256654

HAL Id: hal-01256654
https://hal.inria.fr/hal-01256654

Submitted on 15 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Rapport de prospective sur l'interopérabilité dans le monde du Cloud et du SaaS

Clément Quinton, Daniel Romero, Laurence Duchien, Lionel Seinturier

Inria - Équipe Spirals

14 Février 2014

Abstract

Projet Hermès
Lot 5 : Module d'interaction entre hub et systèmes opérationnels
Livrable 5.1 : Rapport de prospective sur l'interopérabilité dans le monde du Cloud et du SaaS

Ce document présente une solution pour la configuration et le déploiement d'applications dans un environnement de cloud computing. La solution permet de : (1) découpler l'applicatif à déployer de l'environnement dans lequel il sera déployer, (2) spécifier les besoins de l'applicatif nécessaires à son bon déploiement, (3) spécifier les caractéristiques des offres d'hébergement, (4) permettre le calcul de la correspondance entre les besoins et les offres d'hébergement, (5) générer le script qui permet de déployer un applicatif sur une offre d'hébergement.

Cette solution est mise en œuvre dans l'outil Saloon dont les fonctionnalités sont présentées dans ce livrable. Saloon utilise des techniques de lignes de produits logiciels, d'ontologies et de modèles de caractéristiques pour atteindre les cinq objectifs énoncés ci-dessus.

1 Background

1.1 Cloud Computing: configuration side

In the cloud computing paradigm, computing resources are delivered as services. Such a model is usually described as *Anything as a Service* (XaaS or *aaS), where *anything* is divided into layers from *Infrastructure* to *Software* including *Platform*.

Figure 1: Cloud Computing layers[1]

This model in layers offers many configuration and dimension choices [16], for the application to be deployed as well as the configurable runtime environments. Indeed, lots of cloud providers offer services at various layers of the software stack as depicted in FIG. 1. At IaaS level, configuring the cloud environment such as those provided by Amazon or GoGrid means configuring the whole software stack running inside the virtual machine as well as the infrastructure aspects: number of VMs, bandwidth, Input/Output activities, number of nodes, of hard drives, database configuration, etc. Regarding platforms provided by PaaS clouds, *e.g.*, Heroku, the configuration part only focuses on software that compose this platform: which database(s), application server(s), compilation tool, libraries, etc. The software stack configuration process is entirely managed by the PaaS provider and the end-user directly interacts

1.2 Ontologies

In computer science, ontologies were first introduced by [12] back in 1993.

An ontology is "*a formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use, are explicitly defined.*

Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group" [7].

Ontologies are used to describe the concepts and the relation between these concepts of a given domain, thus providing a vocabulary [11]. Different knowledge representation formalisms exist for formalization of ontologies, but share a minimal set of components [7] that are described in the ontology metamodel MM_{Onto} depicted in FIG. 2. An ontology is a set of `Concepts` and `Relations` between `Concepts` of the domain. A `Relation` has a `range` and a `domain`. A `Concept` can also have a `Property`. A `Property` has a `domain`, a `range` that is a DataType (*e.g.*, String, Boolean) and defines a `value`. FIG. 3 depicts an extract of the $Onto_{PaaS}$ ontology, that is, an ontology that describes the cloud PaaS domain. A cloud PaaS provides, among others, an `ApplicationServer` such as `Tomcat` or `Jetty` and support several kinds of `Database`, *e.g.*, `MySQL` or `MongoDB`.

[1] `http://blog.gravitant.com/wp-content/uploads/2012/07/CloudTechSpectrum_Vendors_v21.png`

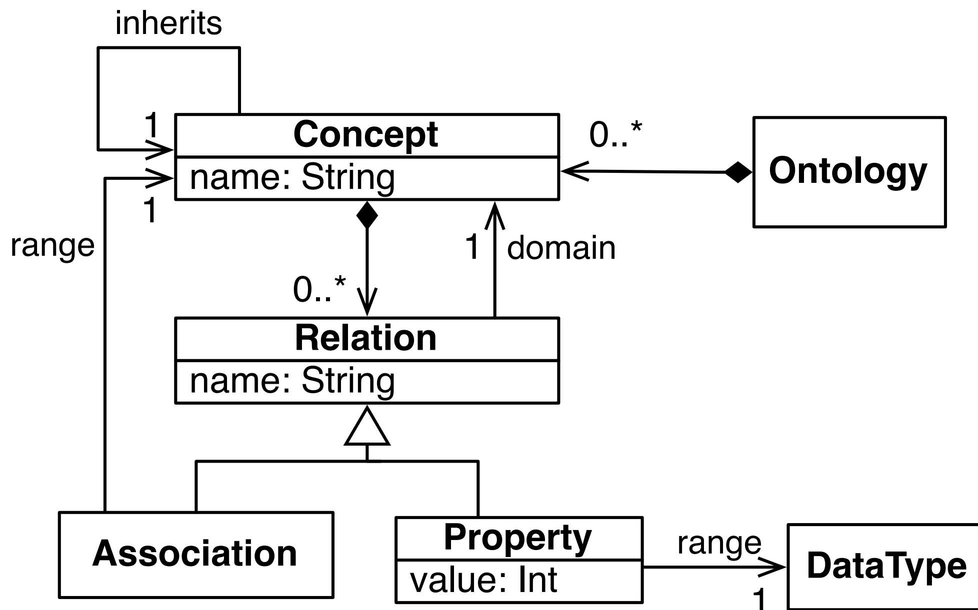

Figure 2: Ontology metamodel MM_{Onto} (extract)

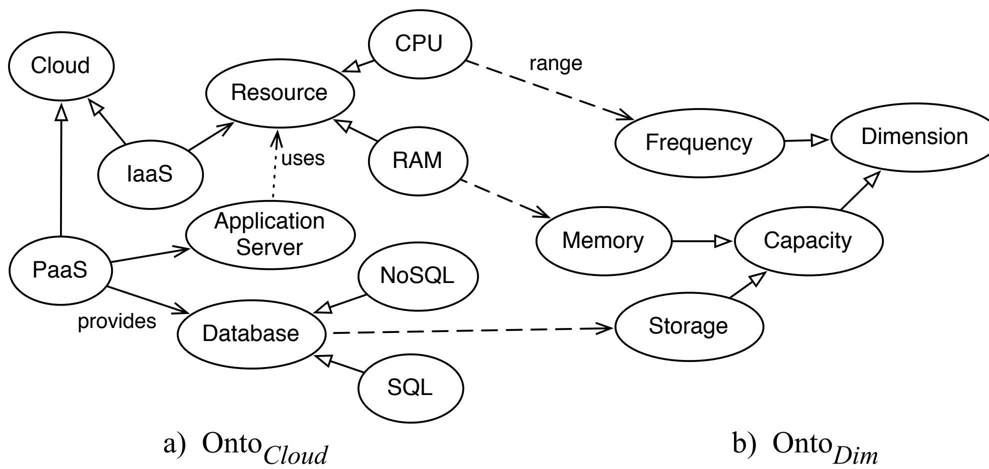

Figure 3: Cloud ontology $Onto_{PaaS}$ (extract)

2 Motivations & Challenges

In this section, we analyze the difficulties of selecting one or several clouds from an application's requirements, we discuss the reasons that lead to select a multi-cloud configuration and we identify several related challenges. We then present ApiSense, a motivating example for our approach.

2.1 Selecting Among Cloud Providers

With the cloud computing paradigm, computing resources are delivered as services. Such a model is usually described as *Anything as a Service* (XaaS or *aaS), where *anything* is divided into layers from *Infrastructure* to *Software* including *Platform*. This model in layers offers many configuration and dimension choices [16], for the application to be deployed as well as the configurable runtime environments. Indeed, lots of cloud providers offer services at various layers of the software stack. At IaaS level, configuring the cloud environment means configuring the whole software stack running inside the virtual machine as well as the infrastructure aspects: number of VMs, bandwidth, input/output activities, number of nodes, of hard drives, database configuration, etc. Regarding platforms provided by PaaS clouds, the configuration part only focuses on software that compose this platform: which database(s), application server(s), compilation tool, libraries, etc. In a multi-cloud configuration perspective, parts of the application can be deployed either at PaaS, IaaS or both level. The wide range of cloud providers [6] likely to host the application makes the choice difficult, and there is a lack of visibility among them to select one that matches the application technical requirements. Thus, some users tend to deploy their application on a cloud that has already been chosen for a previous application to meet more or less similar expectations, *e.g.*, a cloud with a different kind of database support. They do not take the risk of configuring a new cloud or migrating a service to another cloud even if it could be more appropriate. To fit application's requirements and dimensions as depicted by Fig. 4, we find several possibilities. First, in the case `a)`, the whole application is deployed on one given cloud. Second, a multi-cloud configuration can be used in case of *b)* privacy reasons, *c)* dimension reasons, in this case it's less expensive to store data on `Cloud A` or *d)* when required elements is not provided.

To the best of our knowledge, no approach based on configuration making tool has been proposed up to now enabling fine-grained selection of the ideal cloud or multi-cloud configuration, or at least the most suitable with the application's technical and non-functional environment. Nowadays, this choice relies on cloud computing experts' knowledge and raises issues about reliability and exhaustiveness of such a knowledge [20]. The approach described in this paper is *one* solution to help the user to identify and select among cloud providers. It aims at handle three challenges all stakeholders involved in cloud deployment have to face when looking for a cloud solution to host a customer's application.

2.2 Challenges

We propose in this contribution to combine ontologies and FMs into a single solutionthat gives strong support to all stakeholders involved in cloud deployment. This contribution aims at selecting a cloud or multi-cloud configuration for the application to be deployed and faces the following challenges:

$\mathcal{C}_1$: *Supporting application's requirements and variability.* The essential point when selecting a cloud configuration is to ensure the compatibility between the application's architecture

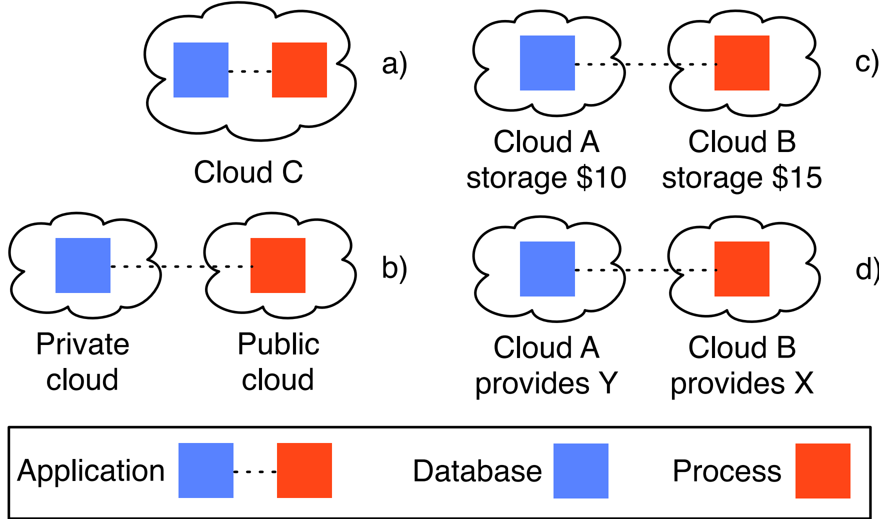

Figure 4: Cases of Multi-Cloud Configurations

and functionalities, and the cloud provider's ones *before* the application deployment to avoid a costly trial-and-error process.

$\mathcal{C}_2$: *Supporting configuration dimensions.* Among potential cloud providers identified with $\mathcal{C}_1$, a more accurate choice can be done, based on specific resource dimensions related to the cloud configuration (e.g., database size) and user-defined priorities between such dimensions.

$\mathcal{C}_3$: *Supporting multi-cloud configurations.* Challenges $\mathcal{C}_1$ and $\mathcal{C}_2$ can be applied on multi-cloud based applications to find a valid configuration distributed over several cloud providers.

2.3 Case Study

To illustrate our proposition, we introduce ApiSense, a software platform for developing crowd-sourcing applications [13]. The platform targets multiple research communities that need to collect realistic datasets for their studies, by providing a lightweight means of building and deploying sensing experiments over a large population of mobile users.

The ApiSense platform provides two mains components: a server-side infrastructure and a mobile phone application. The platform distinguishes between two roles. The former, called *scientist*, is a researcher who wants to define and deploy a sensing experiment over mobile users. The scientist interacts with a web environment on server-side to exploit all the services provided by the platform such as *(i)* describe experiment requirements in a domain-specific language, *(ii)* deploy experiment over a subset of participants and *(iii)* connect other services to the platform to extract and reuse collected data (*e.g.*, visualize, analysis). The latter is the mobile phone user, identified as a *participant*. A participant uses the mobile application to download experiments, execute them in a dedicated sandbox and automatically upload the collected datasets on the ApiSense server-side infrastructure. The main objective of ApiSense is to provide to scientists an open, easily extensible and configurable platform to be reused in various contexts.The whole ApiSense fm represents up to 130 different configurations. The ApiSense server part is a Java-based application built as an assembly of services that can be entirely customized before deployment, according to scientist requirements. Three of them are mandatory for each experiment

to be deployed on mobile phones. The `Collector` is in charge of retrieving datasets uploaded by participants. The `Publisher` makes the experiment available to participants once defined by the scientist. Finally, an `Export` functionality is provided to extract data in a computable format. The whole collected datasets are stored in a `Database`, either `BaseX` or `MongoDB`. ApiSense platform proposes a set of services for data analysis using different kinds of algorithms and implementation languages. Configuring it to be deployed in the cloud is a good way to avoid the server crashing when a peak load arises. Indeed, scientists cannot foresee how many participants are going to install their experiment on their mobile phones and send data to the ApiSense server. In a multi-cloud configuration scenario, the ApiSense database and the services could be deployed on different clouds. Another scenario could be the deployment of different services on several clouds (nodes).

3 Proposal

To face the challenges identified in Sec. 2.2, we define a model-based approach used to *(i)* capture cloud providers' offer knowledge and *(ii)* bridge the gap between an application requirements and cloud providers available configurations. Based on ontologies and fms, the approach allows its user to *(i)* define a technical requirements configuration for the cloud providers likely to host the application considering the application configuration and *(ii)* add resource dimensions to this configuration. Ontologies and fms (Fig. 5) form the base architecture of the approach.

We propose in a first step to tackle challenge $\mathcal{C}_1$ by *(i)* defining cloud providers fms, more precisely one fm per cloud provider and *(ii)* mapping cloud ontologie ($Onto_{Cloud}$) concepts to cloud providers fms' features. In a second step, we handle challenge $\mathcal{C}_2$ by proposing a dimension ontology ($Onto_{Dim}$) whose concepts represent resources dimensions and are mapped to cloud providers fms' attributes. At the end, the application configuration is mapped with each cloud provider fm, and each fm configuration validity is checked. Finally, challenge $\mathcal{C}_3$ can be tackled by combining several valid cloud configurations to fit the application's requirements. The architecture distinguishes between two roles, the *domain experts* and the *user*.

Domain Experts Cloud computing experts are involved in the domain description. They are able to describe their cloud variability and commonality points, thus providing the corresponding fm to the architecture. They interact with other cloud experts to formalize the domain semantics and model the ontologies and they establish mapping between ontologies and fms.

Users The users are all stakeholders involved in cloud deployment, such as an application developer, a software architect or even a cloud provider, *e.g.* to test its own SaaS. It can be used to configure a new service to be deployed as well as to migrate an existing application to the cloud. Using such an approach only requires to have necessary knowledge to properly configure the application's requirements.

3.1 Cloud Systems Variability Modelling

The architecture relies on two distinct parts, fms on one hand and ontologies on the other hand. fms define the commonalities and variabilities of cloud providers while ontologies represents the *scope*, *i.e.*, the set of cloud providers. The definition of the commonality, the variability and the scope is part of the *Domain Engineering* process in a spl approach [19].

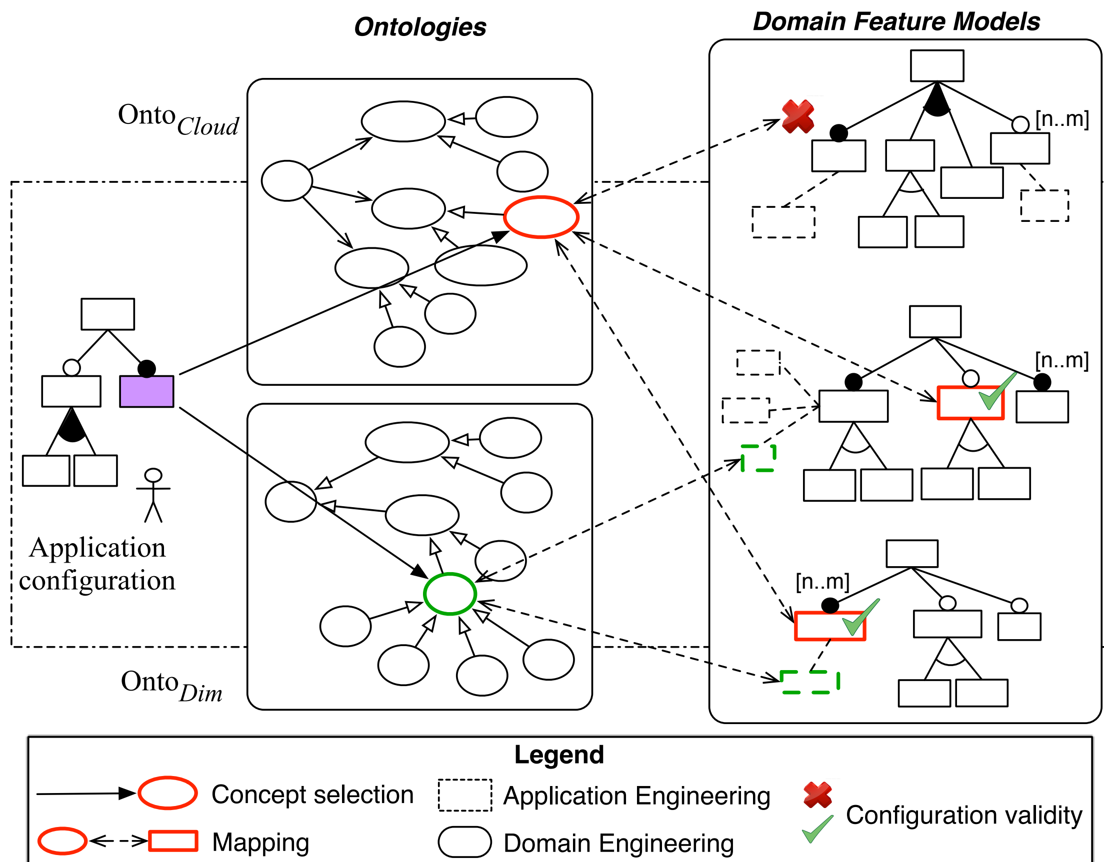

Figure 5: Approach Overview

3.1.1 Feature Models

FMs form the reasoning part of the architecture. They are used to specify the functionalities provided by a given cloud by describing its commonalities and variabilities and their valid combinations. Each FM represents a cloud provider and the set of configurations related to this cloud.

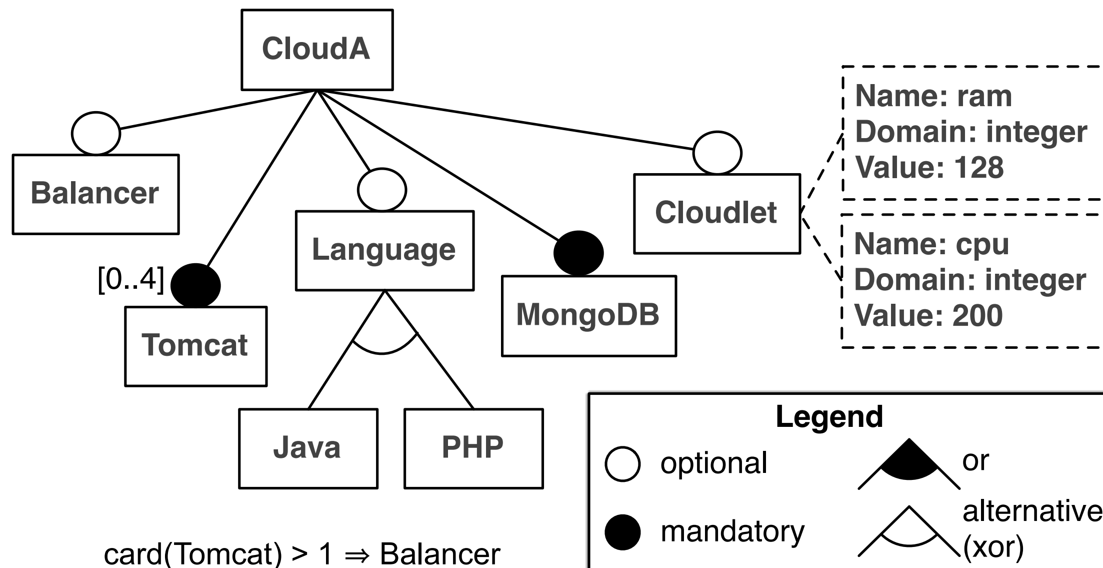

Figure 6: Excerpt of the CloudA FM (FM_{CloudA})

A *Feature Diagram* (FD) (see Fig. 2 that depicts an excerpt FM of a cloud `CloudA`) consists of a hierarchy of features (typically a tree), which may be mandatory (commonality) or optional (variability) and may form Xor or Or-groups. Constraints, *e.g.*, implies or excludes, can also be specified using propositional logic to express inter-feature dependencies. In the above example, the `AppServer` is mandatory and can be either `Tomcat` or `Jetty`, while the `Balancer` feature is optional. Configuring the `CloudA` to have more than one `AppServer` implies such a cloud configuration to have a load `Balancer`. Such a relation is described as a constraint between features and is associated to the FD. We consider that a FM consists of a FD and the associated set of constraints.

In this paper, we extend the basic FM notation to include more information about features. First, we extend the FM with feature cardinalities [8]. This kind of FMs is said *cardinality-based feature models*. A feature cardinality is an interval $[n..m]$ with n as lower bound and m as upper bound of this interval. This interval determines the number of instances of the feature allowed in the product configuration. For example, one possible configuration of the FM_{CloudA} allows up to 4 `AppServer` instances (Fig. 6). The second extension is done by adding attributes related to features, as proposed in [2, 1]. These attributes are used to fill the lack of information in the basic FM notation. FMs with additional information are called *extended* FMs. As these proposals, we consider a feature attribute as a triplet $<$*name, domain, value*$>$. Thus, the `CPU` feature attribute in the FM_{CloudA} specifies the `CPU frequency` provided by this feature. Constraints related to FMs can also be cardinality-based, as described by the `card(AppServer) > 1` $\rightarrow$ `Balancer` constraint: if the number of `AppServer` instances is upper than *one*, then a load `Balancer` must be selected.

By means of constraints, we assume that the resulting configuration is fully functional.

3.1.2 Ontologies

The second part of the architecture are ontologies. An ontology is a formal definition of a domain knowledge, in this case the `cloud` providers' offer one. In computer science, ontologies were first introduced by [12] back in 1993. An ontology is *"a formal, explicit specification of a shared conceptualization"* [7].

Domain experts identify relevant concepts of a given domain and explicitly describe these concepts in a model, as well as constraints and properties linked to these concepts, in a machine-readable format. Ontologies are used to describe the concepts and the relations between these concepts of a given domain, thus providing a vocabulary [11]. Different knowledge representation formalisms exist for formalization of ontologies, but share a minimal set of components [7] that are described in the ontology metamodel MM_{Onto} depicted in Fig. 2.

An ontology is a set of `Concepts` and `Relations` between `Concepts` of the domain. A `Relation` has a `range` and a `domain`. A `Concept` can also have a `Property`. A `Property` has a `domain`, a `range` that is a DataType (*e.g.*, String, Boolean) and defines a `value`. We propose to use ontologies to semantically bridge the gap between the application's requirements and the architecture reasoning part, the FMs. In our approach, we define two ontologies. The first one, $Onto_{Cloud}$, models technical requirements supported by cloud providers, (*e.g.*, application server, database) and the relations between these concepts. The second ontology, $Onto_{Dim}$, describes the dimension properties the user can specify and associate to the technical requirements selected in $Onto_{Cloud}$, *e.g.*, database size, CPU frequency, etc. Fig. 3 depicts an excerpt of the $Onto_{Cloud}$ ontology, that is an ontology that describes the cloud domain. A cloud provides, among others, an `ApplicationServer` such as `Tomcat` or `Jetty` and supports several kinds of `Database`, *e.g.*, `MySQL` or `MongoDB`.

3.1.3 Mapping Ontologies with FMs

We define two kinds of mapping to link these two ontologies with the cloud providers' FMs. On the one hand, $Onto_{Cloud}$ `Concepts` are mapped to FMs `Features`. Two syntactically different `Features` in different `Feature Models` can be semantically equivalent and thus mapped with the same $Onto_{Cloud}$ `Concept`. On the other hand, regarding $Onto_{Dim}$, `Concepts` are mapped to feature `Attributes`. More precisely, `Concept`'s `properties` are mapped to feature `Attributes` the following way: *(i)* `Property's domain name` (*i.e.*, `Concept's name` attribute) with `Attribute's name`, *(ii)* `Property's value` with `Attribute's value` and *(iii)* `Property's range` with `Attribute's domain`. These two mappings bridge the gap between application's requirements and the range of cloud FMs.

3.2 Clouds Configuration & Selection

Previously described mappings allow the user to select its application technical requirements and add dimensions to these requirements only once in the configuration process, thus avoiding a tedious and error-prone task involving the selection of such requirements for each cloud provider's FM.

The selection of a valid combination of variability identified in the domain engineering process is part of the *Application Engineering* process [19]. The user selects by hand among ontologies concepts those necessary for her/his application requirements, as informally described by Algo. 1. She/He also specifies related dimension's value when required. Let us now consider the following ApiSense configuration: $conf_1$ = {`Database, MongoDB, Collector, Publisher,`

Algorithm 1 *selectConceptFromRequirements*$(conf, req)$

Require: a valid application configuration $conf$ and a set of requirements req

Ensure: a set of selected concepts in $Onto_{Cloud}$ and $Onto_{Dim}$ mapping req requirements

```
1: for all Feature f in conf do
2:   while f has requirement r in req do
3:     for all Concept c in Onto_Cloud ∪ Onto_Dim do
4:       if c satisfies r then
5:         if c ∈ Onto_Dim then
6:           select c in Onto_Dim
7:           c.defineProperty(r.domain, r.range, r.value)
8:         else
9:           select c in Onto_Cloud
10:        end if
11:      end if
12:    end for
13:  end while
14: end for
```

`Export`}. Based on the description given in Sec. 2.3, the ApiSense architect defines the set of $conf_1$ requirements as: req_1 = `[Java, MongoDB {capacity, real, 5}]`, the last number being the wished amount of GigaBytes, here given as an example. She/He then applies Algo. 1 with $conf_1$ and req_1 as parameters and gets as a result a set of selected concepts *concepts*, *i.e.*, `Java` and `MongoDB` in $Onto_{Cloud}$ and `Capacity` in $Onto_{Dim}$. The `Capacity` concept is selected and defines a property with the required information, `MongoDB` as *domain*, *real* as *range* and 5 as *value*. Once the concepts selection done, features are automatically selected in the different FMs thanks to the mapping between ontologies and FMs previously described: *(i)* $Onto_{Cloud}$ concepts are mapped with features and *(ii)* $Onto_{Dim}$ concepts are mapped with feature attributes (roughly).

Algorithm 2 *configureFM*$(concepts, fm)$

Require: a set of $Onto_{Cloud}$ and $Onto_{Dim}$ concepts

Ensure: a fm configuration $config$

```
1: for all Concept c in concepts do
2:   for all Feature f in fm do
3:     if c mapped with f then
4:       select f
5:       if c.properties ≠ ∅ then
6:         for all Property p in c.properties do
7:           f.attribute ← p
8:         end for
9:       end if
10:    end if
11:  end for
12: end for
```

The Algo. 2 informally describes the process of feature selection for a given cloud provider FM. Considering the *concepts* obtained from the previous algorithm, the application of Algo. 2

with *concepts* and FM_{CloudA} as parameters result in the following FM_{CloudA} configuration: `[Database, MongoDB {capacity, real, 5}, Language, Java]`. This feature selection process is applied on each cloud FM. The validity of a cloud configuration is then checked using generic and specific constraints. The former describes what a valid configuration is, *e.g.*, if a feature is selected, its parent must also be selected or only one feature can be selected in an alternative group. The latter are FM specific, *e.g.*, a *requires* constraint between two features in a given FM is not necessary the same in other FMs. Applying successively ALGO. 1 and ALGO. 2 with user's application requirements as input tackles the two first challenges described in SEC. 2.2. For $\mathcal{C}_1$, *Supporting application's requirements and variability*, we define a mapping from $Onto_{Cloud}$ concepts to cloud FMs features. For $\mathcal{C}_2$, *Supporting configuration dimensions*, we define another mapping at concept's properties and feature's attributes level. Each FM configuration validity is then checked to determine wether the corresponding cloud fits user's technical requirements and resources dimensions choices or not.

3.3 Multi-Cloud Configurations

The previously described approach can be used in a multi-cloud configuration perspective and tackles challenge $\mathcal{C}_3$. Indeed, a FM configuration can be valid even if not fully covering application's requirements. In this case, it can be associated with another cloud FM configuration for a multi-cloud application to be deployed. Let us now consider the following `[Java, MySQL]` set of requirements req_2. $conf_2$ = `[Language, Java]` is a valid configuration for FM_{CloudA} where `CloudA` supports `Java`-based application but does not provide the `MySQL` database. $conf_2$ does not fulfill req_2. Regarding FIG. 4 case d), there could be another `Cloud B` providing the `MySQL` support. Thus, the multi-cloud configuration required to deploy the application is $conf_{multi}$ = {`[Language, Java]`$_{CloudA}$, `[Database, MySQL]`$_{CloudB}$} with `[Language, Java]` and `[Database, MySQL]` valid configurations for $_{CloudA}$ and $_{CloudB}$ respectively.

4 Preliminary Validation

In this section, we present the details of the SALOON framework we developed to implement the approach described in SEC. 3, as well as the results obtained from first experimentations we led to handle the challenges we have identified in SEC. 2.2.

4.1 Implementation

Regarding the implementation, the SALOON framework relies on *Eclipse Modeling Framework* (EMF) [23] metamodels, which is one of the most widely accepted metamodeling technologies. The EMF provides code generation facilities to produce a set of Java classes for the metamodels used in the SALOON framework. Each metamodel is described as an *ecore* file and allows us to create a dynamic instance as XMI model. These models represent either an ontology, either a FM or even a mapping model. The XMI format is used to support model persistence. During the PaaS FMs configuration process, ontology models are loaded and the user selects the required concepts. Then, mapping and FMs models are loaded. The former is used while looping on the latter ones to select the corresponding features. This choice brings high flexibility to the SALOON framework. Indeed, one can target a new PaaS provider by adding corresponding FM model that conforms to the MM_{FM} metamodel. Once features are selected, FMs are translated to propositional logic and constraints are loaded and checked against each XMI model, *i.e.*, each PaaS FM. The translation of FMs to propositional logic is well known [17] and off-the-shelf SAT

solvers such as Sat4j [3] can be used to check FMs configuration validity. For each PaaS FM, SALOON check if the FM is satisfiable, *i.e.*, if there is at least one valid configuration.

4.2 Experimentation

To validate our approach, we used two different APISENSE configurations as SALOON inputs: $conf_1$, described in the previous section and $conf_2$ = `[Database, MongoDB, Collector, Publisher, Export, DataAnalysis]`, where `DataAnalysis` is a PHP-based algorithm used to compute collected datasets. Regarding these configurations, the associated sets of requirements are req_1 = `[Java, MongoDB]` and req_2 = `[Java, MongoDB, PHP]` respectively. We applied ALGO. 1 with these configurations and requirements and ran ALGO. 2 with four different PaaS FM describing Cloudbees[2], Cloud Foundry[3], dotCloud[4] and Heroku[5] PaaS providers. SALOON gave us as output the results reported in TABLE. 1.

PaaS	Req. set	Req. covering	Missing
Cloudbees	req_1	✓	-
	req_2	✓	-
Cloud Foundry	req_1	✓	-
	req_2	×	PHP
dotCloud	req_1	✓	-
	req_2	✓	-
Heroku	req_1	×	MongoDB
	req_2	×	MongoDB

Table 1: PaaS FM configuration using SALOON

Cloudbees and dotCloud are the only PaaS to provide both `Java` and `PHP` support. As they also provide a configured environment to deploy a `MongoDB` instance, they cover entirely the application's requirements and are able to host the two APISENSE configurations corresponding to req_1 and req_2 requirements. Cloud Foundry does not provide a `PHP` support, and cannot be configured to host APISENSE in its $conf_2$ configuration. Finally, Heroku can be configured to host `Java`-based applications, but only PostgreSQL database instances are provided. We then checked these results by deploying the two APISENSE configurations on the four real cloud PaaS named before. As described by TABLE. 1, we were able to deploy $conf_1$ on Cloudbees, Cloud Foundry, dotCloud and $conf_2$ on Cloudbees and dotCloud. Although successful, we achieved these deployments not without difficulty. Indeed, once libraries and tools, *e.g.*, MongoDB, selected on PaaS side and configured to host the application, this application is not yet *cloud ready*. Some modifications can be required before the application upload, *e.g.*, setting a correct database connection URL. Moreover, these modifications are often PaaS specific, that is, a cloud ready configuration for a given PaaS is not necessarily ready to be uploaded on another PaaS. Let us now consider that for any reason, using `Heroku` to deploy the APISENSE services is required. In a multi-cloud configuration perspective, one possibility is to configure the `MongoDB` database on another cloud *e.g.*, on `Cloudbees` that provides fully managed Databases-As-A-Service.

[2]`http://www.cloudbees.com`
[3]`http://www.cloudfoundry.com`
[4]`https://www.dotcloud.com`
[5]`http://www.heroku.com`

5 Related Work

Multi-Cloud & Configuration Some recent works were proposed to deal with the problem of multi-cloud and configuration. In [21], the authors propose a model-based approach that helps to model, deploy and configure complex applications in multiple IaaS. The application to be deployed is modeled and configured as an OVF appliance to be run in VMs while we configure the cloud(s) likely to host the application considering its requirements. In [4], a DSL is used to model the application to be deployed in the clouds while an interpreter is provided to identify which resources have to be used in the infrastructure to fulfill application's requirements. These authors pursue the same goal than us but there is no semantic mapping between requirements and cloud platforms. Moreover, they can not check the validity of the obtained cloud configuration. Leusse *et al.* [10] propose in their vision paper an architecture to facilitate the deployment of different components of a same application onto different clouds. They point out the lack of visibility among cloud providers that is one of the challenges we face in this paper. In [18], the authors present a federated multi-cloud PaaS infrastructure deployed on top of several existing IaaS/PaaS. This infrastructure is based on an open service model used to design both the multi-cloud PaaS and the SaaS applications running on top of it. Contrarily to our approach, they don't need to configure the multi-cloud platform since both SaaS and PaaS are implemented using the same service model.

Ontologies & Cloud Several related work propose an ontology-based approach to discover a cloud provider. Dastjerdi *et al.* present an ontology-based architecture aiming at deploying software stacks on IaaS providers [9]. They propose an ontology describing user's functional and non-functional requirements considering quality of service aspects as well as an ontology that describes several services provided at IaaS level. They then establish relationships between both ontologies to determine which IaaS service fits best user's requirements. The approach we propose in this paper goes in the same direction but deals with PaaS provider instead of IaaS ones. Moreover, we take into consideration the configuration of several services, *e.g.*, an application server, a database and a language support while Dastjerdi *et al.* approach only deals with one service at a time (*e.g.*, which IaaS provides this amount of CPU?). In [14], the authors present an agent-based system to discover cloud services. In addition, they propose *taxonomies* (kind of ontology with hierarchical structure of concepts) for PaaS and IaaS clouds. System's agents loop on cloud services and establish similarities between user requirements and cloud services. They argue that these similarities help to find the most relevant service and provide a service ranking. These authors pursue the same goal than us but once again, the user can only search for one service at a time. Moreover, proposed taxonomies are not sufficient enough for our needs since they do not define relationships between concepts except the inheritance one.

Software Product Lines & Cloud SPL-based approaches as well as Feature Models ones have also been proposed to deal with cloud computing variability. Thus, Cavalcante *et al.* propose an adaptation of the SPL-based development process to deploy their Health Watcher system [5]. Regarding the application to be deployed, they include in its FM "cloud features", *e.g.*, `Amazon S3` for the storage feature, `Google Authentication` for the login one. These cloud features have been collected by studying applications already deployed in the cloud. Contrarily to our approach, they modify the original application FM and, in a way, they influence the cloud provider final choice. They also use extended FMs but feature attributes usage is not clearly explained. In [22], the authors propose an approach based on SPL engineering to configure multi-tenant cloud applications. They rely on an extended FM to describe functionalities and quality of services for the application to be deployed and plan to use in their future work an adaptive

staged configuration process for reconfiguration of FM variants. A stakeholder for each cloud level (IaaS, PaaS, SaaS) selects features for its level. Cloud provider choice is thus limited to the IaaS/PaaS selected by the corresponding stakeholder. Other authors such as [20], [15] present a feature model based approach to select and manage software configurations and deploy virtual appliances on IaaS providers. They consider virtual appliances as SPL products and rely on FMs to describe and select configurations. Each configuration's feature holds as asset a software to be installed in the virtual appliances. Unlike our approach, they do not choose among different deployment PaaS clouds since they configure the whole software stack that host the application. This software stack can only be deployed at IaaS level.

6 Conclusion

Selecting a cloud provider to host its application leads to complex choices to deal with a wide range of resources at different levels of functionality among available cloud solutions. Configuration and customization choices arise due to the heterogeneous and scalable aspect of the Cloud Computing paradigm and the selection of a cloud among others remains challenging, due to the amount of providers and their intrinsic variability. Moreover, developers today do not only deploy applications on a specific cloud, but consider migrating services from one cloud to another and manage distributed applications spanning multiple clouds. In this paper, we used feature models to represent a cloud variability, as well as ontologies to describe the heterogeneous aspect of the cloud ecosystem. $Onto_{Cloud}$ is dedicated to application's technical requirements while $Onto_{Dim}$ describes the *dimensions* that can be specified for the cloud or multi-cloud configuration. We proposed a model-driven approach implemented in the SALOON framework that bridges the gap between an application configuration and a cloud provider FM, thus facing the challenges identified in SEC. 2.2. For the first challenge regarding the application's requirements, $Onto_{Cloud}$ concepts are mapped with FMs features. For the second challenge regarding the cloud configuration dimensions, $Onto_{Cloud}$ concept's properties are mapped with FMs feature's attributes. This mapping is automated and avoids the SALOON user to select features and define feature's attributes for each cloud provider FM manually. For third challenge, the combination of several valid cloud configurations obtained by tackling C_1 and C_2 can be used to deploy a multi-cloud configuration. As a preliminary validation, we used our framework on four cloud providers and showed that SALOON can be used to *(i)* check wether a cloud configuration is valid or not and *(ii)* if invalid, wether another cloud could be used in a multi-cloud configuration or not.

For future work, our approach can be extended to help the user in its configuration process. Some previous configuration decisions could be used as a feedback for quite equivalent application's requirements. Moreover, the user could be assisted by the framework. Relying on constraints result, he/she could be advertised of an available configuration. Finally, we hope to apply our approach as the entry point of a whole software product line, adding assets composition and product generations steps to the SALOON framework.

References

[1] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th international conference on Software Product Lines*, SPLC'05, pages 7–20, 2005.

[2] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Reasoning on Feature Models. In *Proceedings of the 17th international conference on Advanced Information Systems Engineering*, CAiSE'05, pages 491–503, 2005.

[3] D. L. Berre and A. Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[4] E. Brandtzæg, M. Parastoo, and S. Mosser. Towards a Domain-Specific Language to Deploy Applications in the Clouds. In *3rd International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 213–218, 2012.

[5] E. Cavalcante, A. Almeida, T. Batista, N. Cacho, F. Lopes, F. C. Delicato, T. Sena, and P. F. Pires. Exploiting Software Product Lines to Develop Cloud Computing Applications. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 179–187, 2012.

[6] CloudTimes. Cloud Computing Ecosystem. `http://cloudtimes.org/wp-content/uploads/2011/11/Clouds.cloudtimes.png`, 2012. Accessed 31.10.12.

[7] O. Corcho, M. Fernández-López, and A. Gómez-Pérez. Ontological Engineering: Principles, Methods, Tools and Languages. In *Ontologies for Software Engineering and Software Technology*, pages 1–48. 2006.

[8] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[9] A. V. Dastjerdi, S. G. H. Tabatabaei, and R. Buyya. An Effective Architecture for Automated Appliance Management System Applying Ontology-Based Cloud Discovery. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 104–112, 2010.

[10] P. de Leusse and K. Zielinski. Towards governance of rule and policy driven components in distributed systems. In *ServiceWave*, volume 6994 of *Lecture Notes in Computer Science*, pages 317–318, 2011.

[11] D. Gasevic, D. Djuric, and V. Devedzic. *Model Driven Engineering and Ontology Development (2. ed.)*. 2009.

[12] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[13] N. Haderer, R. Rouvoy, and L. Seinturier. A Preliminary Investigation of User Incentives to Leverage Crowdsensing Activities. In *2nd International IEEE PerCom Workshop on Hot Topics in Pervasive Computing (PerHot)*, San Diego, États-Unis, Mar. 2013. IEEE Computer Society.

[14] J. Kang and K. M. Sim. Cloudle: An Ontology Enhanced Cloud Service Search Engine. In *Proceedings of the 2010 international conference on Web information systems engineering*, WISS'10, pages 416–427, 2011.

[15] T. Le Nhan, G. Sunyéet, and J.-M. Jézéquel. A Model-Driven Approach for Virtual Machine Image Provisioning in Cloud Computing. In *Service-Oriented and Cloud Computing*, volume 7592 of *Lecture Notes in Computer Science*, pages 107–121. 2012.

[16] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, 2009.

[17] M. Mendonca, A. Wąsowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 231–240, 2009.

[18] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier. A Federated Multi-cloud PaaS Infrastructure. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 392 –399, june 2012.

[19] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* 2005.

[20] C. Quinton, R. Rouvoy, and L. Duchien. Leveraging Feature Models to Configure Virtual Appliances. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, CloudCP '12, pages 2:1–2:6, 2012.

[21] A. Sampaio and N. Mendonça. Uni4Cloud: An Approach based on Open Standards for Deployment and Management of Multi-cloud Applications. In *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages 15–21, 2011.

[22] J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau. Dynamic Configuration Management of Cloud-based Applications. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 171–178, 2012.

[23] D. Steinberg, et al. *EMF: Eclipse Modeling Framework (2nd Edition).* 2nd revised edition, 2009.