Towards a Domain-Specific Language to Deploy Applications in the Clouds

Eirik Brandtzæg
University of Oslo
SINTEF IKT
Oslo, Norway
eirik.brandtzaeg@sintef.no

Parastoo Mohagheghi
Networked Systems and Services
SINTEF IKT
Oslo, Norway
parastoo.mohagheghi@sintef.no

Sébastien Mosser
Networked Systems and Services
SINTEF IKT
Oslo, Norway
sebastien.mosser@sintef.no

***Abstract*—The cloud-computing paradigm advocates the use of virtualised resources, available "in the clouds". Application are now developed in order to be cloud-aware. Unfortunately, the deployment of such applications is still manually done, or relies on home-made shell script. In this paper, we propose to model cloud applications using a component-based approach. It leverage the existing deployment descriptors into a high-level domain-specific language. The language is then illustrated through the modelling of a prototypical application used to teach distributed programming at the University of Oslo.**

***Keywords*-Cloud-computing; Modelling; Deployment;**

I. Introduction

Cloud–Computing [1] was considered as a *revolution*. Taking its root in distributed systems design, this paradigm advocates the share of distributed computing resources designated as "*the cloud*". The main advantage of using a cloud-based infrastructure is the associated scalability property (called *elasticity*). Since a cloud works on a *pay–as–you–go* basis, companies can rent computing resources in an elastic way. A typical example is to temporary increase the server–side capacity of an e–commerce website to avoid service breakdowns during a load peak (*e.g.*, Christmas period). However, there is still a huge gap between the commercial point of view and the technical reality that one has to face in front of "*the cloud*".

A company that wants to deploy its own systems to the cloud (*i.e.*, be part of the cloud *revolution*) has to cope with existing standards. The Cloud–Standard initiative[1] lists dozens of overlapping standards related to Cloud–Computing. They focus on infrastructure modeling or business modeling. These standards does not provide any support for software modelling or deployment. Thus, the *deployment* of a cloud–based system is a difficult task, as it relies on handcrafted scripts. It is not possible to reason on the deployment, not to assess it *w.r.t.* to cloud business policies.

The Cloud-computing paradigm emphasizes the need for automated deployment mechanisms, abstracted from the underlying technical layer. As cloud-computing consider that the number of resources available in the cloud is not limited (cloud "elasticity"), it triggers new challenges from a deployment point of view. Even if several approaches consider the deployment target as "open" (*i.e.*, new host machines can be added in the environment), the "virtually unlimited" dimension provided by the cloud-approach is not taken into account.

Our contribution in this paper is to propose a component-based approach [2] to model software deployment in the clouds. This approach is operationalised as a Domain-Specific Language (DSL), which is given to the software designer. The language is based on a reduced component meat-model, and support the modelling of the deployment relationship between components. For the sake of concision, we only focus in here on the description of the cloud deployment language usage, and we do not address in this paper the run-time enactment. This work is done in the framework of REMICS[2], an European project dedicated to the migration of legacy application into cloud-based applications.

II. Related Works

We propose here to analyse the state of the art about software deployment, identifying good practices to be reused in our own solution, dedicated to cloud-computing. The cloud model always assume that the software to be deployed will be running on an host machine, virtualised in the cloud. Thus, its deployment depends on a lot of characteristics provided by the host, *e.g.*, IP address, operating system, available remote protocols. The deployment might also depends on the software to be deployed, *e.g.*, implementation language, configuration capabilities.

A. Deployment Models

Several approaches were proposed to abstract the user from the underlying platform *w.r.t.* the deployment point of view. These approaches propose to model the deployment of a software in a generic way, using the concepts described in a meta–model. In this domain, the two main approaches are *(i)* the UML Deployment Diagrams [3] and *(ii)* the OMG D&C meta-model [4]. These approaches are complemented by academic approaches like ORYA [5] and GADE [6].

This work is partially funded by the EU Commission through the REMICS project (FP7-ICT, Call 7, contract number 257793)

[1] http://cloud-standards.org/

[2] http://www.remics.eu

UML Deployment Diagrams: using the UML Deployment Diagram approach, one can use *artifacts* to model the physical element involved in the deployment (*e.g.*, a compiled executable to be copied on the host machine). Artifacts follows a composite pattern (*i.e.*, an artifact can be composed by others), and are expressive enough to model complex software dependencies. These elements are bound to physical *devices* to model which software artifact must be deployed on which machine. The infrastructure is modelled thanks to the definition of communication path between different devices.

OMG D&C meta-model: D&C means "Deployment and Configuration". It was built to tackle the challenges encountered while standardizing the deployment of CORBA components. This meta-model defines *(i)* meta–data to be used during the deployment process (*e.g.*, configuration information for a given package) and *(ii)* a target model relying on these meta-data to describe the deployment process. The approach is extremely verbose, and suffers from the number of concepts to be used to model a deployment, even in front of a simple case. Another weaknesses is its close relationship with CORBA: the meta–model is too close to the one defined by CORBA, and existing work based on OMG D&C focus on the deployment of CORBA components [7], [8].

Academic approaches: We consider here two prototypical examples *w.r.t.* the cloud-computing domain. ORYA is similar to the UML deployment diagram approach, as it provides a purely descriptive meta-model to describe a deployed system. Contrarily to the UML approach, ORYA also provides concepts to model administrative and legal issues in the deployed system. But it suffers from the same drawback, *i.e.*, its lack of a clear semantics (or a least a reference implementation) to properly support the deployment in an automated and reproducible way. GADE is the complete opposite, as it concretely targets the deployment of software components in grid-computing environment. It focus on the capture of the grid domain, supporting the user in the deployment of processes to be executed on the grid. This approach emphasizes the need for a deep understanding of the domain while modelling a deployment meta-model.

B. State of Practice: Cloud–based solution

Cloud providers have already understood that deployment is crucial while talking about clouds. Thus, they provides mechanisms to support the user during the deployment of its application. This support can be textual (*e.g.*, Amazon Cloud Formation [9]), graphical (*e.g.*, Applogic [10]). But it immediately suffers from the "vendor lock-in" syndrome. Thus, several libraries can be found (*e.g.*, libcloud [11], jclouds [12], δ-cloud [13]) to abstract these providers.

Amazon Cloud Formation: it is a service provided by Amazon from their popular *Amazon Web Services* (AWS). It give users the ability to create template files in form of template files, which they can load into AWS to create stacks of resources. This is mainly meant for users that want to replicate a certain stack, with the ability to provide custom parameters. Once a stack is deployed it is only maintainable through the AWS Console, and not through template files. The structure and semantics of the template itself is not used by any other providers or cloud management tooling, so it can not be considered a multicloud solution and enforce a vendor lock-in syndrome.

Applogic: it is a proprietary model-based application for management of private clouds. This interface let users configure their deployments through a diagram with familiarities to component diagrams with interfaces and assembly connectors. This is one of the solutions that use and benefit from a model based approach. They also provide an *Architecture Deployement Language* (ADL) to enforce properties on the modelled deployment. But this solution is only made for private clouds running their own controller, this can prove troublesome for migration, both in to and out of the infrastructure.

Application Programming Interface (API): Libcloud and *jcloud* are APIs that aims to support the largest cloud providers through a common API. Libcloud have solved the multicloud problem in a very detailed manner, but the complexity is therefore even larger. The *δ-cloud* approach has a similar procedure as *jclouds* and *Libcloud*, but with a web-service approach. This can work with the benefits that many middleware software have, such as cahing, queues, redundancy and transformations, but it also has the disadvantages such as single point of failure and version inconsistencies.

C. Conclusions

The deployment models available in the state of the art demonstrate that a descriptive modelling of deployment is elegant and well understood by the end user. Such an approach must stay simple and focused, to avoid the multiplication of concepts. The approach must also be tailored to address its target domain, *i.e.*, cloud-computing in our case. The available tools analyzed from the state of practice demonstrate that the heterogeneity of the different underlying platforms needs to be abstracted. Anyhow, the current approaches are available at the code level, and does not provide an abstraction layer to be used by the application designer to properly model a cloud-based application to be deployed.

III. Running Example & Challenges

We consider here a simple application, sufficient to underline the intrinsic complexity of cloud–application deployment modelling. This application is called **`BankManager`**, and is used at the University of Oslo to teach distributed systems, based on the very classical *"bank account management"* case study. It consists of the two following parts:

- A back-end that contains a **`Database`**, used to store information about customers and accounts,

- A front-end that implements a web-based application, used to access to the different accounts and transfer money between accounts.

From a software architecture point of view, this application wsimply consists of a relational database to support the back-end, and java-based *servlets* bundled in a **`war`** archive to support the front-end. The front-end must hold a reference to the back-end to address the proper database. But when confronted to the "cloud-computing" domain, the following points needs to be also considered:

- Clouds implement open environments. As a consequence, we do not know where the application will be deployed. Thus, establishing the link between the front–end and the back–end requires a particular attention.
- Clouds provides different mechanisms to support application deployment. Where instructure cloud (IaaS) mainly provides low-level (*e.g.*, SSH, FTP) connectivity to the virtual machines, platform clouds (PaaS) provides deployment protocols dedicated to the technology they implement (*e.g.*, WAR deployment).
- Clouds work on a pay-as-you-go basis. Thus, one can consider to deploy both back-end and front-end artefacts on the same virtual machine, to reduce costs during development. Another alternative is to deploy these two artefacts on two different virtual machines. In concrete case, the variability of deployment possibilities is humongous.
- Clouds emphasises reproducibility. Thus, a given deployment descriptor should be easily re-usable as–is, in the same context or in a new one.
- Clouds support scalability through replication and load-balancing. The deployment descriptor should be easily replicable to support the on-demand replication of computation-intensive artefacts.

Based on these challenges, our goal is to provide a meta-model that supports the application designer while designing his/her cloud application.

IV. A DSL to support Cloud-Application Deployment

We named the language *Pim4Cloud DSL*, as it is a *Platform Independent Model* dedicated to *Clouds*. The key idea of the Pim4Cloud DSL is to support the deployment of application in the cloud. An overview of the approach is depicted in Fig. 1. Using the DSL, the application designer models the software to be deployed. In parallel, the infrastructure provider describe the available infrastructure to be used by the application. From a coarse–grained point of view, it means that the designer requires "computation nodes" (*e.g.*, virtual machines) from the cloud, and the infrastructure provider describes such nodes (based on its own catalog). An interpreter is then used to identify which resources have to be used in the infrastructure to fulfil the requirements expressed by the application designer. The interpreter then do the provisioning, and actually deploy the modelled application. It returns as feedback to the designer a *living model* of its application, annotated with run-time property bound to each modeled artifact (*e.g.*, the public IP address associated to a given virtual machine).

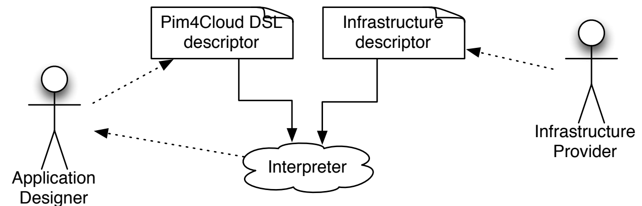


Figure 1. Pim4Cloud DSL overview

Based on the points previously described, we propose to use a component–based approach to model the deployment of cloud applications. This approach was successfully used by the DeployWare framework in the context of adaptive component system [14], [15], and we propose to transpose its core idea to cloud.

To achieve this goal, we use a reduced component meta-model, described in Fig. 2. This meta-model is expressive enough to support the modeling of both infrastructure and applicative artefacts in an endogenous way. According to the CBSE paradigm, everything is a **`Component`**. **`Component`**s can be scalars or composite, *i.e.*, containing sub-component inside their boundaries. A **`Component`** may offer one or more deployment **`Service`**s, *i.e.*, deployment protocols one can used to deploy other components onto this one (*e.g.*, a *servlet* container will offer a **`WAR`** service to support the deployment of java-based web applications). Obviously, it may require one **`Service`** if it aims to be deployed on another one (*e.g.*, a war artifact will require a **`WAR`** service). **`Components`** are connected among others through **`Connectors`**. A component can offer and expects **`Property`**, *e.g.*, a database component may expect both **`username`** and **`password`**, and provide an **`url`** to be remotely accessed. These elements are used at run-time (asked in a deployment descriptor, or filled using the feed-back obtained from the underlying cloud infrastructure). In a **`Composite`**, one can express bindings between properties, that is, a formal link between an expected and an offered property. These links (**`RuntimeBinding`**) are used at run-time to properly transfer the expected information.

Implementation: This meta-model is intended to be specialised according to user's needs, as its intrinsic simplicity makes it easy to introduce in user's code. We provide a reference implementation of this approach using the Scala language, exposed as an internal domain–specific language to support the usage of this meta–model in JVM–based languages. The DSL is designed in a modular way, and

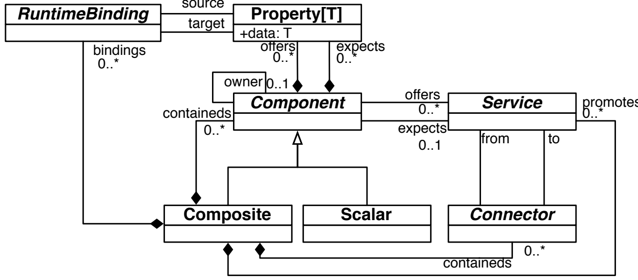


Figure 2. Modelling Cloud Components: a generic meta–model

implements several constructions (*e.g.*, "offering a service", "containing a component") as independent modules, implemented as *traits*. This design support the evolution of the DSL, as adding a new syntactic construct is assimilated as the mix of a new *trait*.

V. Using the language

Based on this internal DSL, one can model a cloud–based software to be deployed. We represent in Fig. 3 a graphical representation of a *WAR container* model, using standard graphical notation for component assemblies. This container is used to host **`WAR`**–based artefacts. It is made as the composition of *(i)* a virtual machine obtained from a IaaS provider and *(ii)* a *Jetty* server used to actually support the hosting of **`WAR`** artefacts:

- The virtual machine is modelled as a component named **`vm`**, typed as a **`SmallVM`**. This component does not require any other, and is therefore considered in the models as an element obtained from an external provider (outside of the scope of the modelled system). It offers a **`ssh`** service, and one can use this protocol to interact with the component at run-time. This can be considered as the *IaaS* layer of this example
- The **`WAR`** hosting artifact is modelled as a component named **`container`**, typed as a **`Jetty`** server. It offers a **`war`** service, and one can use it to deploy WAR–based application. This component relies on the **`APT`** package system to be properly deployed. Replacing the hosting server (*e.g.*, from **`Jetty`** to **`Tomcat`**) only means to replace this component by another one.
- The final component (**`WarContainer`**) composes the ones previously described as the following: it *(i)* promotes the **`war`** port of the **`container`** component, and *(ii)* binds the **`apt`** requirement of the **`container`** to the **`ssh`** offering provided by the **`vm`** one.

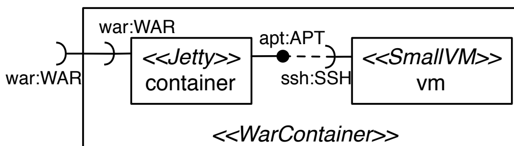


Figure 3. *WarContainer*: Component diagram representation

We describe in Fig. 4 the code necessary to model this system with the DSL. As the language relies on Scala, the declaration of a scalar component is assimilated to the declaration of a class, that extends the concepts previously described. Thus, the user is completely free in such a class to write all the code he/she thinks necessary. The DSL is only used to support the user when dealing with its system from a deployment point of view.

- **`SmallVM`**. This class is the simplest example. It extends the **`ScalarComponent`** concept, and defines an instance variable named **`ssh`** used to model the offering of the **`SSH`** service.
- **`AptComponent`**. This *trait* shows how one can specialize the generic meta-model previously described. As our future models will use software components available through the **`APT`** package system[3], we define here a trait that capture this concept. It reifies an **`Aptcomponent`** as an extension of the **`ScalarComponent`** concept, that offers an **`apt`** service. It also contains a list of packages (the ones to be installed thanks to the **`APT`** system). This example demonstrates the extensibility of the Pim4Cloud DSL: it is possible for the end-user to specialise it for his/her own needs, using common mechanisms like inheritance.
- **`Jetty`**. This component use the **`AptComponent`** described in the previous point. It defines an instance variable named **`war`**, that offers a **`War`** service for deployment. It also states that this component relies on several packages provided through **`APT`** (*i.e.*, packages named **`jetty`** and **`jetty-extra`**).
- **`WarContainer`**. This component extends the **`CompositeComponent`** concept. Thus, it is able to contains other components, here *(i)* a **`Jetty`** component named **`container`** and *(ii)* a **`SmallVM`** component named **`vm`**. The composite also uses the DSL to promotes the **`war`** service offered by one of its sub-component (*i.e.*, **`container`**). Finally, it connects the **`apt`** service required by the **`container`** sub-component to the **`ssh`** service provided by the **`vm`** one. As the DSL is internal, it immediately benefits from the mechanisms of the hosting language, *e.g.*, variable visibility and scoping mechanisms. It also benefits from the IDE mechanisms available for the hosting language, like code completion and error handling: as shown in the figure, the **`ssh`** service is automatically proposed, and the unfinished instruction is underlined in red.

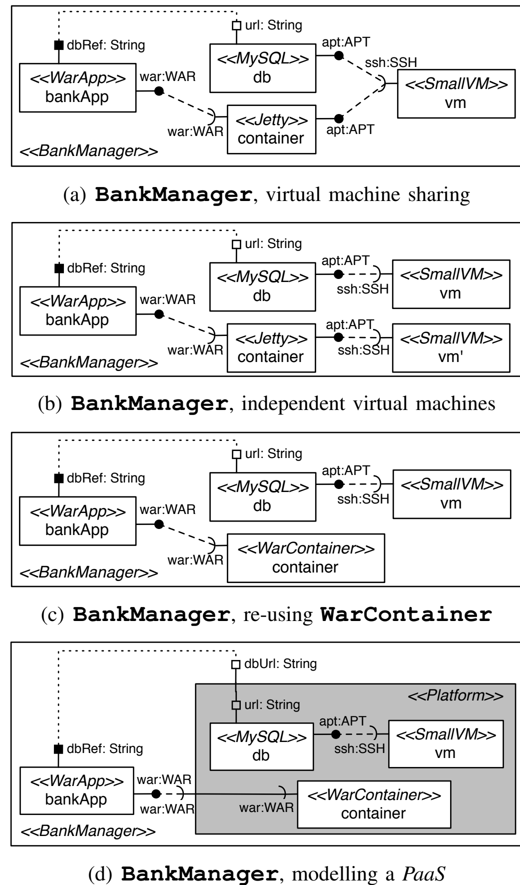
[3] APT is the standard software package manager widely used to support the installation and set-up of common software on Linux-based computers.

Figure 4. *WarContainer* modelled with Pim4Cloud DSL

Based on the previously described mechanisms, we can now model several version of our initial example, the *BankManager*. This software is implemented in Java, and requires the two following elements: *(i)* a database for its back-end and *(ii)* a web server able to host WAR–based software. We represent in FIG. 5 different deployment configuration for such a system. For the sake of concision, we do not provide the Pim4Cloud DSL code associated to each model. This code is available on the public repository[4].

- FIG. 5(a). In this version, the front-end and the back-end are deployed on the same virtual machine. This is typical for test purpose, where the idea is to minimize the cost of the rented infrastructure during development. The database component exposes a property named **`url`**. This property will be filled at run-time by the deployment engine associated to the Pim4Cloud DSL (out of the scope of this paper). The **`bankApp`** component expects a property named **`dbRef`**, and a binding is expressed at the composite level to specify that this property will be set based on the value obtained from **`db`** at run-time.
- FIG. 5(b). In this version, two virtual machines are used. This is the main difference when compared to the previous one. This separation allows the replication of the **`container`** component, ensuring elasticity through horizontal scalability.
- FIG. 5(c). This versions demonstrates the strength of the CBSE approach when applied to this domain. It is immediately possible to re-use the previously described **`WarContainer`**. As a component is considered as a black-box, the end-user will not care about hos it works internally from an infrastructure point of view. It will simply re-use a given component that provides the needed deployment services.

[4]https://github.com/eirikb/cloudml-client



(a) **`BankManager`**, virtual machine sharing

(b) **`BankManager`**, independent virtual machines

(c) **`BankManager`**, re-using **`WarContainer`**

(d) **`BankManager`**, modelling a *PaaS*

Figure 5. **`BankManager`** deployment

- FIG. 5(d). This version emphasize the idea described in the previous version. Actually, the DSL allows us to model in an endogenous way *IaaS* and *PaaS*. Building a *PaaS* becomes as simple as modeling a software stack on top of virtual machines. In this case, we modelled

a **Platform**, which exposes a **war** port for service hosting and a **dbUrl** property for persistence. This *platform* is then used to deploy the bank application, but can also be used to host any application implemented as a **War** and requiring a database.

VI. Conclusions & Perspectives

We described how the Pim4Cloud DSL can be used to support the application designer while modelling an application to be deployed in the clouds. We also describe how the DSL is implemented, using Scala as a hosting language. We showed on a prototypical example how the DSL is used to properly model the deployment. The key points are first the Independence of cloud provider, *i.e.*, the fact that the Pim4Cloud DSL does not enforce any vendor lock-in. Application deployment can be modelled in an agnostic way *w.r.t.* the targeted cloud provider. Secondly, the component based approach support the definition of static analysis (*e.g.*, type consistency), as well as the reuse of components from a deployment to another one (*i.e.*, architectural patterns can be reified as cloud components). This approach also support the endogenous modelling of both Paas and Iaas.

This work is currently pursued and hort terms perspectives of this work includes the two following axis: *(i)* "models@run.time" and *(ii)* verification. The feed-back returned to the user is for now reduced to its minimum, that is, the IP of virtual machines provisioned in the cloud. With regard to the humongous amount of data available from cloud providers (*e.g.*, load average, cost), one of our objective to enhance this feed-back to take into account more information. We plan to achieve this goal with a *"Models@run.time"* approach. Instead of returning a set of IP addresses, the Pim4Cloud interpreter will return a model of the running system, available at run-time. It will maintain the link between the running system and the models, providing a model-driven way of querying the cloud–based application (*e.g.*, about its status, its load). From the verification point of view, the current mechanisms included in the DSL are static for now, and intensively rely on the type system.: the engine assumes that a static model (*i.e.*, a model that can be compiled) will always be properly deployed in the cloud. One of our objective is to make this mechanism more robust, and break this assumption. We plan to use a transactional approach coupled to the action-based mechanism previously described. Thanks to the *acidity* of the transactional model, the action interpreter will be able to recover when an error will be encountered during the deployment process.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] O. Object Management Group, "UML 2.0 Superstructure Spec." Object Management Group, Tech. Rep., Aug. 2005.

[4] ——, "Deployment and Conf. of Component-based Distributed App. Spec., Version 4.0," Tech. Rep., Apr. 2006.

[5] P.-Y. Cunin, V. Lestideau, and N. Merle, "Orya: A strategy oriented deployment framework," in *Component Deployment*, ser. Lecture Notes in Computer Science, A. Dearle and S. Eisenbach, Eds., vol. 3798. Springer, 2005, pp. 177–180.

[6] S. Lacour, C. Pérez, and T. Priol, "Generic application description model: toward automatic deployment of applications on computational grids," in *GRID*. IEEE, 2005, pp. 284–287.

[7] G. Deng, D. C. Schmidt, and A. S. Gokhale, "Cadance: A criticality-aware deployment and configuration engine," in *ISORC*. IEEE Computer Society, 2008, pp. 317–321.

[8] J. Dubus and P. Merle, "Applying omg d&c specification and eca rules for autonomous distributed component-based systems," in *MoDELS Workshops*, ser. Lecture Notes in Computer Science, T. Kühne, Ed., vol. 4364. Springer, 2006, pp. 242–251.

[9] A. AWS, "Amazon Cloud Formation Language," http://aws.amazon.com/en/cloudformation/, February, 1st, 2012.

[10] CA, "Applogic CA," http://www.ca.com/us/products/detail/CA-AppLogic.aspx, February, 1st, 2012.

[11] Apache, "Apache Libcloud, a Unified Interface to the Cloud," http://libcloud.apache.org, February, 1st, 2012.

[12] JClouds, "JClouds," http://www.jclouds.org/, February, 1st, 2012.

[13] Apache, "δ-cloud: Many Clouds. One API. No problems." http://deltacloud.apache.org/, February, 1st, 2012.

[14] A. Flissi, J. Dubus, N. Dolet, and P. Merle, "Deploying on the Grid with DeployWare," in *CCGRID*. IEEE Computer Society, 2008, pp. 177–184.

[15] J. Dubus, "Une démarche orientée modèle pour le déploiement de systèmes en environement ouverts distribués," Ph.D. dissertation, Université des Sciences et Technologies de Lille, Lille, France, Oct. 2008. [Online]. Available: https://iris.univ-lille1.fr/dspace/bitstream/1908/1507/1/50376-2008-Dubus.pdf

View publication stats