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
NTNU
SINTEF IKT
Trondheim, Norway
parastoo@idi.ntnu.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". Applications 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 leverages the existing deployment descriptors into a high-level domain-specific language. The language is then illustrated through the modeling of a prototypical application used to teach distributed programming at the University of Oslo.**

Keywords-Cloud-computing; Modeling; 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 (*e.g.*, *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 wiki [2] lists dozens of overlapping standards related to Cloud Computing. They focus on infrastructure modeling or business modeling. These standards do not provide any support for software modeling 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, nor to assess it with respect to (*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 considers that the number of resources available in the cloud is not limited, 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 [3] to model software deployment in the clouds. This approach is provided 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 modeling 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 [4], an European project dedicated to the migration of legacy application into cloud-based applications. Section II discusses related works, and Section III illustrates the challenges on a running example. Section IV describes the language meta-model, Section V describes its usage, and finally, Section VI concludes this paper.

II. RELATED WORKS

We propose here to analyze 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 [5] and *(ii)* the OMG D&C meta-model [6]. These approaches are complemented by academic approaches like ORYA [7] and GADE [8].

 ISBN: 978-1-61208-216-5

UML Deployment Diagrams: using the UML Deployment Diagram approach, one can use *artifacts* to model the physical elements 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 modeled 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 [9], [10].

Academic approaches: We consider here two prototypical examples. ORYA is similar to the UML deployment diagram approach, as it provides a purely descriptive meta-model to describe a deployed system. 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 modeling 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 provide mechanisms to support the user during the deployment of applications. This support can be textual (*e.g.*, Amazon Cloud Formation [11]), graphical (*e.g.*, Applogic [12]). But it immediately suffers from the "vendor lock-in" syndrome. Thus, several libraries can be found (*e.g.*, libcloud [13], jclouds [14], δ-cloud [15]) 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, 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 multi-cloud 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. They also provide an *Architecture Deployment Language* (ADL) to enforce properties on the modeled 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 aim to support the largest cloud providers through a common API. Libcloud have solved the multi-cloud 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 (introducing a bottleneck).

C. Conclusions

The deployment models available in the state of the art demonstrate that a descriptive modeling 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 modeling. 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 simply 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

 ISBN: 978-1-61208-216-5

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 infrastructure 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 artifacts on the same virtual machine, to reduce costs during development. Another alternative is to deploy these two artifacts on two different virtual machines. In concrete case, the variability of deployment possibilities is humongous.
- Clouds emphasizes 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 artifacts.

Our goal is to provide a meta-model that supports the application designer while deploying a cloud application.

IV. A DSL TO SUPPORT DEPLOYMENT IN CLOUDS

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 Figure 1. Using the DSL, the application designer models the software to be deployed. In parallel, the infrastructure provider describes 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 fulfill the requirements expressed by the application designer. The interpreter then do the provisioning, and actually deploys the modeled 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).

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

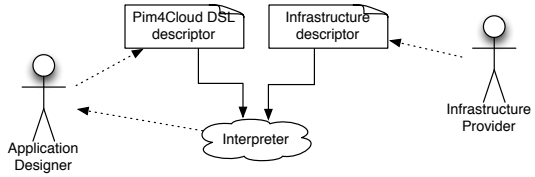


Figure 1. Pim4Cloud DSL overview

component system [16], [17], and we propose to transpose its core idea to cloud deployment.

To achieve this goal, we use a reduced component meta-model, described in Figure 2. This meta-model is expressive enough to support the modeling of both infrastructure and applicative artifacts in an endogenous way. **`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 specialized 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

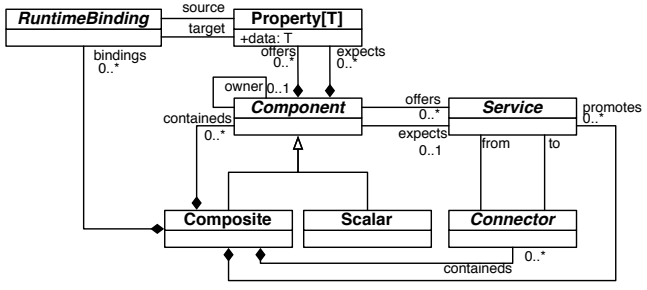


Figure 2. Modeling cloud components: a generic meta-model

languages. The DSL is designed in a modular way, and 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.

A. Modeling a Simple Component

We represent in Figure 3 a graphical representation of a **`WarContainer`** model, using standard graphical notation for component assemblies. This container is used to host **`WAR`**-based artifacts. 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`** artifacts:

- The virtual machine is modeled 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 modeled 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 modeled 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.

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. Internal DSLs immediately benefit from the mechanisms of the hosting language, *e.g.*, variable visibility and scoping mechanisms. We describe in Listing 1 the code necessary to model this system with the DSL.

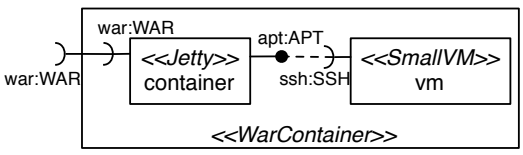


Figure 3. **`WarContainer`**: component diagram representation

```
class WarContainer
    extends CompositeComponent with WarOffering {
  private[this] val container = instantiates[Jetty]
  private[this] val vm = instantiates[SmallVM]
  override val war = promotes(container.war)
  this deploys container.apt on vm.ssh
}
```

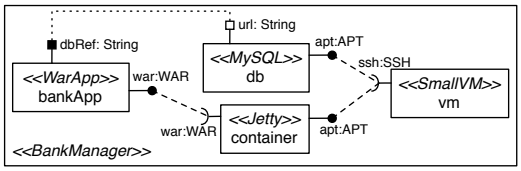
Listing 1. **`WarContainer`** code

The **`WarComponent`** class extends the **`CompositeComponent`** concept (it is able to contains other components), and mixes the **`WarOffering`** trait (statically informing other components that it offers a **`war`** port). It instantiates two internal sub-components: *(i)* a **`Jetty`** component named **`container`** to host the servlets applications and *(ii)* a virtual machine of type **`SmallVM`**. It promotes the **`war`** service offered by the **`container`** sub-component, and finally deploys the servlet container on the virtual machine.

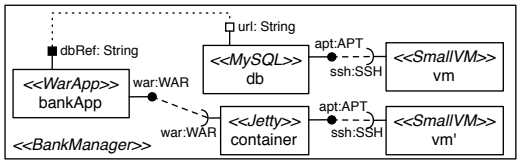
B. Multiple Topologies for *BankManager*

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 Figure 4 different deployment configuration for such a system.

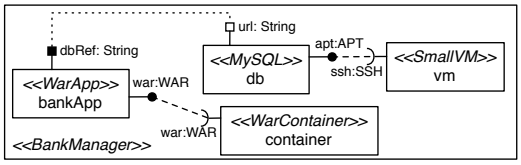
- Figure 4(a). In this version, the front-end and the back-end are deployed on the same virtual machine. This is



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



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



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

Figure 4. **`BankManager`** deployment variability

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.

- Figure 4(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.
- Figure 4(c). This versions demonstrates the strength of the component 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.

We give in Listing 2 the DSL code that models these different topologies. First, we define our application (**MyCloudApp**) as an abstract class: it factorizes shared elements, and each concrete topology will extends this class to refine its content. The top-level class instantiate a **BankManager** component (the WAR file that contains the application), as well as a **MySQL** database. It defines an abstract **container**, with the assumption that this subcomponent will offer WAR deployment (it is typed as **WarOffering**). The bank manager application is then deployed on this container. The database property required by the application is filled with the url provided by the database.

We then present in Listing 3 the three different components that actually implements such deployment topologies. The first one (**VirtualMachineSharing**) instantiates a single virtual machine and deploys both the container and the database on it. The second component (**IndependentVirtualMachine**) deploys the servlet container and the database on different virtual machines (**vm1** and **vm2**). Finally, the last component (**UsingWarContainer**) reuse the

```
abstract class MyCloudApp extends CompositeComponent {
  private[this] val bankApp = instantiates[BankManager]
  protected val db = instantiates[MySQL]
  protected val container: WarOffering
  this deploys bankApp.war on container.war
  this sets bankApp.dbRef using db.url
}
```

Listing 2. **BankManager**: Abstract class to model **MyCloudApp**

```
class VirtualMachineSharing extends MyCloudApp {
  override val container = instantiates[Jetty]
  private[this] val vm = instantiates[SmallVM]
  this deploys container.apt on vm.ssh
  this deploys db.apt on vm.ssh
}

class IndependentVirtualMachine extends MyCloudApp {
  override val container = instantiates[Jetty]
  private[this] val vm1 = instantiates[SmallVM]
  private[this] val vm2 = instantiates[SmallVM]
  this deploys container.apt on vm1.ssh
  this deploys db.apt on vm2.ssh
}

class UsingWarContainer extends MyCloudApp {
  override val container = instantiates[WarContainer]
  private[this] val vm = instantiates[SmallVM]
  this deploys db.apt on vm.ssh
}
```

Listing 3. Multiple deployment topologies for **BankManager**

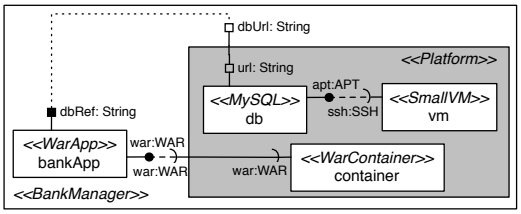


Figure 5. Deploying the **BankManager** on a *PaaS*

WarContainer component defined in Listing 1 to host the servlet container.

C. Modeling Platform as a Service Artifacts

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 (Figure 5). In this case, we modeled 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.

From a DSL point of view, one can imagine a library of available platforms. In Listing 4, we describe a platform named **AGivenPlatform**, provided by **AGivenProvider** (modeled as a package). Then, one can use this platform by simply importing it in its component, and using it like any other. The **UsingPaaS** component in Listing 4 shows how it can be done with the DSL.

```
package AGivenProvider {
  class AGivenPlatform extends CompositeComponent with
       WarOffering {
    private[this] val db = instantiates[MySQL]
    private[this] val vm = instantiates[SmallVM]
    private[this] val container =
         instantiates[WarContainer]
    override val war = promotes(container.war)
    val dbUrl = externalize(db.url)
    this deploys db.apt on vm.ssh
  }
}
```

```
class UsingPaaS extends CompositeComponent {
  import AGivenProvider.AGivenPlatform
  private[this] val bankApp = instantiates[BankApp]
  private[this] val platform =
       instantiates[AGivenPlatform]
  this deploys bankApp.war on platform.war
  this sets bankApp.dbRef using platform.dbUrl
}
```

Listing 4. Modeling a *Platform as a Service* using Pim4Cloud DSL

VI. Conclusion & Perspectives

We described how the Pim4Cloud DSL can be used to support the application designer while modeling 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. Application deployment can be modeled in an agnostic way *w.r.t.* the targeted cloud provider. The 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 modeling of both Paas and Iaas.

This work is currently pursued, including a standardization effort at the OMG in the context of the REMICS project. Short 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 large 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. 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.

Acknowledgments

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

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] CloudStandard, "The Cloud Standards Coordination Wiki," http://cloud-standards.org/, [retrieved: 05, 2012].

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

[4] REMICS, http://remics.eu/, [retrieved: 05, 2012].

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] A. AWS, "Amazon Cloud Formation Language," http://aws.amazon.com/en/cloudformation/, [retrieved: 05, 2012].

[12] CA, "Applogic CA," http://www.ca.com/us/products/detail/CA-AppLogic.aspx, [retrieved: 05, 2012].

[13] Apache, "Apache Libcloud, a Unified Interface to the Cloud," http://libcloud.apache.org, [retrieved: 05, 2012].

[14] JClouds, "JClouds," http://www.jclouds.org/, [retrieved: 05, 2012].

[15] Apache, "δ-cloud: Many Clouds. One API. No problems." http://deltacloud.apache.org/, [retrieved: 05, 2012].

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

[17] 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.

 ISBN: 978-1-61208-216-5