CloudMF: Model-Driven Management of Multi-Cloud Applications

NICOLAS FERRY, FRANCK CHAUVEL, HUI SONG, ALESSANDRO ROSSINI, MAKSYM LUSHPENKO, ARNOR SOLBERG

Citation:
NICOLAS FERRY, FRANCK CHAUVEL, HUI SONG, ALESSANDRO ROSSINI, MAKSYM LUSHPENKO, ARNOR SOLBERG
(2018) CloudMF: Model-Driven Management of Multi-Cloud Applications, ACM Transactions on Internet Technology, 2018, 18:2, 16:1-16:24 , DOI: 10.1145/3125621

CloudMF: Model-Driven Management of Multi-Cloud Applications

NICOLAS FERRY, SINTEF, Department of Software and Service Innovation
FRANCK CHAUVEL, SINTEF, Department of Software and Service Innovation
HUI SONG, SINTEF, Department of Software and Service Innovation
ALESSANDRO ROSSINI, EVRY Cloud Services
MAKSYM LUSHPENKO, SINTEF, Department of Software and Service Innovation
ARNOR SOLBERG, SINTEF, Department of Software and Service Innovation

While the number of cloud solutions is continuously increasing, the development and operation of large-scale and distributed cloud applications are still challenging. A major challenge is the lack of interoperability between the existing cloud solutions, which increases the complexity of maintaining and evolving complex applications potentially deployed across multiple cloud infrastructures and platforms. In this paper, we show how the Cloud Modelling Framework (CLOUDMF) leverages model-driven engineering (MDE) and supports the DevOps ideas to tame this complexity by providing: *(i)* a domain-specific language for specifying the provisioning and deployment of multi-cloud applications, and *(ii)* a models@run-time environment for their continuous provisioning, deployment, and adaptation.

CCS Concepts: •**Networks** → **Cloud computing;** •**Software and its engineering** → **Model-driven software engineering; System modeling languages;**

Additional Key Words and Phrases: Cloud computing, Multi-cloud, DevOps, Model-driven engineering, Models@run-time

ACM Reference Format:
Nicolas Ferry, Franck Chauvel, Hui Song, Alessandro Rossini, Maksym Lushpenko, and Arnor Solberg, 2018. CloudMF: Model-Driven Management of Multi-Cloud Applications. *ACM Trans. Internet Technol.* 18, 2, Article 16 (January 2018), 21 pages.
DOI: https://doi.org/10.1145/3125621

1. INTRODUCTION

Cloud computing provides ubiquitous access to a shared and virtualised pool of computing resources including processing, memory, network and storage [Mell and Grance 2011]. As it becomes possible to automate the management of these resources, cloud computing should significantly reduce operational expenses. However, in practice the development and operation of large-scale and distributed cloud applications typically face two key obstacles: *(i)* the lack of interoperability between cloud solutions and *(ii)* complex maintenance and evolution management.

The lack of interoperability among different cloud solutions leads to vendor lock-in and prevents the development of multi-cloud applications (*i.e.*, applications that can be deployed across multiple clouds) [Petcu 2014]. As stated in the CORDIS reports [SSAI Expert Group 2010; SSAI Expert Group 2012], *“whilst a distributed data environment (IaaS) cannot be easily moved to any platform provider (PaaS) [...], it is also almost impossible to move a service/image/environment between providers on the same level”*. This prevents cloud application providers to exploit the peculiarities of existing cloud solutions, *e.g.*, to optimise performance, availability, and cost.

This work is supported by the European Community’s Seventh Framework Programme (FP7/2007-2013), under grant 318484 (MODAClouds), 317715 (PaaSage) and the Horizon 2020 programme under grant 680478 (MC-Suite), 645372 (ARCADIA).
Author’s addresses: Nicolas Ferry, Franck Chauvel, Hui Song, Maksym Lushpenko, Arnor Solberg, SINTEF, Department of Software and Service Innovation, Norway, Oslo, Forskningsveien 1; Alessandro Rossini, EVRY Cloud Services, Norway, Oslo, Snarøyveien 30

The second challenge is to maintain and evolve such complex cloud applications, especially in a multi-cloud DevOps context. To shorten delivery time and foster the continuous evolution of multi-cloud applications, we should reconcile development and operation activities [Hüttermann 2012]. The DevOps movement [Humble and Farley 2010] thus calls for better collaborations between developers and operators. It advocates to further automate resource provisioning and deployment to improve the flexibility and efficiency of the delivery process.

This paper describes the Cloud Modelling Framework (CLOUDMF) [Ferry et al. 2014], which fosters the design, deployment, and maintenance of multi-cloud applications. As opposed to our previous publications [Ferry et al. 2013; Ferry et al. 2013; Ferry et al. 2014], this paper reports on its evaluation and includes two significant extensions: *(i)* the support for scalability mechanisms (*i.e.*, resource pool and load balancer) and *(ii)* a new models@run-time environment that enables parallel deployments and provides a language together with the mechanisms to optimise the deployment and adaptation process.

To address the interoperability challenge, CLOUDMF relies on a model-driven approach and the principle of "model once, generate anywhere". CLOUDMF simplifies the design and management of applications across multiple clouds, and their migration from one cloud to another.

CLOUDMF facilitates the management of multi-cloud applications in several ways. It supports the DevOps ideas by providing a representation of the applications that is causally connected to the underlying running systems using models@run-time techniques [Blair et al. 2009]. We thus offer the same concepts to developers and operators. In addition, by supporting both IaaS and PaaS, CLOUDMF enables several levels of control of multi-cloud applications: *(i)* in case of execution on IaaS or white box PaaS solutions, it offers full control with automatic resource provisioning and deployment of the entire cloud stack from infrastructure to application; *(ii)* in case of execution on black box PaaS solutions, it offers shared control of the application. Note that if parts of the multi-cloud application execute on IaaS or white box PaaS, CLOUDMF offers full control of these parts.

The remainder of the paper is organised as follows. Section 2 introduces SENSAPP, the multi-cloud application used as motivating example throughout the paper. Section 3 presents our cloud modelling language. Section 4 describes the supporting models@run-time environment before Section 5 details the interface it offers. Section 6 reports the evaluation of CLOUDMF in various projects as well as the status of its current reference implementation. Section 7 presents a selection of related work. Finally, Section 8 discusses some specificities of our DSML and highlights future research directions before Section 9 concludes.

2. MOTIVATING EXAMPLE

SENSAPP[1] is a service-oriented application for storing and exploiting large data sets collected from sensors and devices. SENSAPP offers capabilities to register sensors, store their data, and notify clients when new data are pushed. It consists of four main components. The *Registry* component stores metadata about the sensors (*e.g.*, description and creation date). The *Database* component stores raw data from the sensors in a MongoDB database. The *Notifier* component sends notifications to third-party applications when relevant data are pushed (*e.g.*, when new data collected by air quality sensors become available). Finally, the *Dispatcher* component orchestrates the other components: it receives data from the sensors, stores them in the Database according to the metadata from the Registry, and then sends notifications when necessary. SENSAPP ADMIN uses the public REST API of SENSAPP and provides capabilities to manage sensors and visualise data using a graphical user interface. To be deployed, SENSAPP requires a Servlet container and a database, while SENSAPP ADMIN requires a Servlet container only.

Hereafter, we use SENSAPP to illustrate different scenarios of provisioning and deployment. For testing, we deploy all components on our private OpenStack IaaS, with both SENSAPP compon-

[1] http://sensapp.org. SENSAPP is open-source software; the source code is available at https://github.com/SINTEF-9012/sensapp.

ents and MongoDB on the same virtual machine (VM). In contrast, when moving to production, we deploy the SENSAPP ADMIN components on Amazon Elastic Beanstalk, migrate the SENSAPP components along with a load balancer to Amazon EC2, while keeping MongoDB in the private cloud since it may store sensitive data. This example motivates for the following CLOUDMF requirements:

— **Cloud provider independence (R_1):** CLOUDMF shall support a cloud provider-agnostic specification of the provisioning and deployment. This simplifies the design of multi-cloud applications and prevents vendor lock-in.
— **Separation of concerns (R_2):** CLOUDMF shall support a modular, loosely-coupled specification of the provisioning and deployment so that the modules (*i.e.*, sub-parts of the deployment model) can seamlessly be substituted. This facilitates the maintenance as well as the dynamic adaptation of the deployment topology.
— **Reusability (R_3):** CLOUDMF shall support the specification of types (or patterns) that can be seamlessly reused. Those ease the evolution as well as the rapid development of different variants of a system.
— **Abstraction (R_4):** CLOUDMF shall provide a single domain-specific language to describe deployments on both IaaS and PaaS in a cloud provider-independent and -specific way. In addition, CLOUDMF shall provide a continuously up-to-date, abstract representation of the running system. This facilitates the reasoning, simulation, and validation of operation activities.
— **White- and black-box infrastructure (R_5):** CLOUDMF shall support IaaS and PaaS solutions. This requires coping with various degrees of delegation of control over underlying infrastructures and platforms of multi-cloud applications.
— **Application technology independence (R_6):** CLOUDMF shall be agnostic to development paradigm and technology. This enables developers to design application with their favourite programming languages and frameworks.
— **Reconcile design- and run-time activities (R_7):** CLOUDMF shall offer the same language for both the design-time specification of the deployment of multi-cloud applications and their run-time adaptation and reconfiguration. This reduces the gap between development and operation phases.
— **Fully controlled automated deployment (R_8):** CLOUDMF shall support the automatic deployment of multi-cloud applications on the basis of a CLOUDML deployment model only. This helps to reduce cycle-time. In addition, CLOUDMF shall provide the ability to fully control and adapt the deployment process, if and only if desired.

In the next section, we present how the two main components of CLOUDMF address these requirements. These two components are: *(i)* the Cloud Modelling Language (CLOUDML), a domain-specific modelling language (DSML) to model the provisioning and deployment of multi-cloud applications covering both IaaS and PaaS, and *(ii)* a models@run-time environment for automatically enacting the provisioning, deployment, and adaptation of these systems.

3. CLOUDML

CLOUDML is a DSML that captures the topology of cloud applications including its software components, their connections and the underlying computing resources.

CLOUDML relies on model-driven engineering (MDE), a branch of software engineering that focuses on models rather than source code and improves the productivity, quality, and cost-effectiveness of software development. Model transformation automatically generates (parts of) software systems and therefore frees developers from repetitive and error-prone tasks.

In particular, the CLOUDML architecture is inspired by the OMG Model-Driven Architecture (MDA)[2]. The MDA relies on three types of models representing three layers of abstractions, *i.e.*, the Computational-Independent Model (CIM), the Platform-Independent Model (PIM), and the

[2]http://www.omg.org/mda/

Platform-Specific Model (PSM). From the cloud perspective, the introduction of a new layer of abstraction improves the portability and reusability of cloud-related concerns amongst several clouds. Indeed, even if the system is designed for a specific platform including framework, middleware, or cloud services, these entities often rely on similar concepts, which can be abstracted from the specificities of each cloud provider. Typically, the topology of the system in the cloud, as well as the minimum hardware resources required to run it (*e.g.*, CPU and RAM), can be defined in a cloud-agnostic way.

CLOUDML refines the PSM abstract layer and allows developers to model resource provisioning and deployment of multi-cloud applications in both a Cloud Provider-Independent Model (CPIM) and a Cloud Provider-Specific Model (CPSM). The CPIM specifies the provisioning and deployment in a cloud provider-agnostic way (addressing the requirement R_1). The CPSM refines the CPIM with cloud provider-specific information. This two-level approach is agnostic to any development paradigm and technology, *i.e.*, developers are free to design and implement their applications using their preferred programming languages and frameworks.

CLOUDML is also inspired by component-based approaches, which isolate concerns (R_2) and ease reuse (R_3). Hence, we regard deployment models as assemblies of components exposing ports (or interfaces), with bindings between these ports.

In addition, CLOUDML implements the *type-object* pattern [Atkinson and Kühne 2002], which also facilitates reusability (R_3) and abstraction (R_4). This pattern exploits two flavours of typing, namely *ontological* and *linguistic*, respectively [Kühne 2006]. These typing mechanisms are relevant from a language engineering perspective as the ontological typing offers support for extending a language without modifying its definition.

CLOUDML has both a graphical and a textual notation. An Eclipse-based editor[3] helps create and edit CLOUDML models using the textual syntax and provides auto-completion and validation. This editor is based on the Xtext framework [Eysholdt and Behrens 2010] and leverages an Ecore representation of the CLOUDML metamodel. A web-based editor[4] supports a graphical syntax and interacts with the models@run-time environment. The models@run-time engine is a plain Java application, which reifies internally the CLOUDML metamodel as plain Java objects, and helps developers modify topologies programmatically. These models can be serialised in both XMI and JSON. In addition, during the MODAClouds project[5], our partner Softeam extended their modelling tool Modelio[6] to provide an alternative graphical modelling editor for CLOUDML.

3.1. CPIM

In the following, we provide a description of the most important classes and corresponding properties in the CLOUDML metamodel as well as sample models in the associated textual syntax. The textual syntax better illustrates the various concepts and properties that can be involved in a deployment model, and that can be hidden in the graphical syntax. Both the textual and the graphical syntax offer an abstraction over multi-cloud and cloud provider-specific concepts as well as over the deployment process. Figure 1 shows the type portion of the CLOUDML metamodel in Ecore format[7].

A `CloudMLModel` consists of `CloudMLElements`, which can be associated with `Properties` and `Resources`. A `Resource` represents an artefact (*e.g.*, scripts, binaries, or configuration files) adopted to manage the deployment life-cycle (*i.e.*, download, configure, install, start, and stop). The three main types of `CloudMLElements` are `Component`, `Communication`, and `Hosting`. Note that for readability purposes, we only depict the inheritance relation between `CloudMLElement` and `Component` in Figure 1.

[3]https://github.com/SINTEF-9012/cloudml-dsl
[4]https://github.com/SINTEF-9012/cloudml/tree/master/ui/webeditor/src/main/webapp
[5]http://www.modaclouds.eu
[6]https://forge.modelio.org/projects/creator-4clouds/wiki/Creator4Clouds-Tutorial-CML4C
[7]http://www.eclipse.org/modeling/emf/

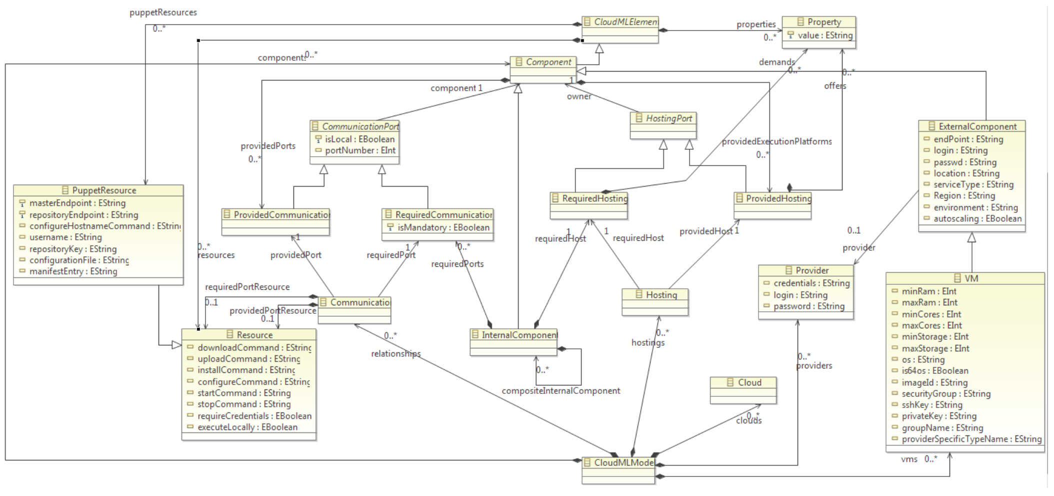

Fig. 1. Type part of the CLOUDML metamodel

A `Component` represents a reusable type of component of a cloud application. A `Component` can be an `ExternalComponent` managed by an external `Provider` or an `InternalComponent` managed by CLOUDMF (*e.g.*, a Servlet container or SENSAPP). This mechanism enables supporting both IaaS and PaaS solutions through the single abstract concept of component (R_5). Listing 1 illustrates the definition of an external PaaS provider. The property `location` represents the region of the data centre hosting it (*e.g.*, `location="eu-west-1"`, short for West Europe). The properties `username` and `password` represent the authentication information needed to access to this specific service. The properties `serviceType` and `environment` represent the type (*e.g.*, database, application container) and the environment (*e.g.*, Tomcat) of the service, respectively. Finally, the property `autoscaling` indicates that the auto-scaling mechanism of the cloud platform should be used.

Listing 1. An example of an External Component type from a CPIM

```
external component BeanstalkContainer {
  provider: Beanstalk, location: "eu-west-1"
  environment: "Tomcat", autoscaling: false
  provided host SCProvided {("language": "Java")}
}
```

An `ExternalComponent` can also be a `VM` (see Listing 2). The properties `minCores`, `maxCores`, `minRam`, `maxRam`, `minStorage`, and `maxStorage` represent the lower and upper bounds of virtual compute cores, RAM, and storage, respectively, of the required VM (*e.g.*, `minCores=1`, `minRam=1024`). The property `OS` represents the operating system to be run by the VM (*e.g.*, `OS="ubuntu"`). All these properties, which act as constraints during the refinement of the model toward a CPSM, are optional and do not have to be defined in the CPIM. Indeed, in case it is not possible to fulfil these constraints, default values will automatically be selected by the deployment engine.

Listing 2. An example of a VM type from a CPIM

```
vm SL {
  provider: OpenStackNova, os: "ubuntu", os64
  ram: 1024.., cores: 1.., storage: 50..
  securityGroup: "SensApp", sshkey: "cloudml", groupName: "sensapp"
  provided host slProvided
}
```

`Components` are connected through two kinds of ports. A `CommunicationPort` represents a communication interface of a component. A `ProvidedCommunication` provides a feature to another component (*e.g.*, SENSAPP provides a REST interface, see Listing 3), while a RequiredCommunication consumes a feature from another component (*e.g.*, SENSAPP requires a MongoDB interface, see Listing 3). Only internal components can have a RequiredCommunication since they are managed by CLOUDMF. The property isLocal indicates that the component providing the feature and the component consuming the feature must be deployed on the same external component (*e.g.*, in the initial deployment SENSAPP and MongoDB must be deployed on the same VM, see Listing 3). The property isMandatory of RequiredCommunication represents that the InternalComponent depends on this feature (*e.g.*, SENSAPP depends on MongoDB and hence MongoDB has to be deployed before SENSAPP, see Listing 3).

A HostingPort represents a hosting interface of a component. Similarly, a HostingPort represents that the component provides hosting facilities (*e.g.*, a VM running GNU/Linux provides hosting to a Servlet container, see Listing 2), while a RequiredHost represents that the internal component requires hosting from another component (*e.g.*, SENSAPP requires hosting from a Servlet container, see Listing 3).

Listing 3. An example of an Internal Component type from a CPIM

```
internal component SensApp {
  resource SensAppResource {
    download: "wget -P ~ http://github.com/downloads/SINTEF-9012/sensapp/sensapp.war;
     wget -P ~ http://cloudml.org/scripts/linux/ubuntu/sensapp/sensapp.sh",
    install: "sudo bash sensapp.sh"
  }
  provided communication restProvided {local, port: 80}
  required communication mongoDBRequired {local, port: 0, mandatory}
  required host SCRequired { ("language" : "Java") }
}
```

A Communication represents a reusable type of communication binding between a Required- and a ProvidedCommunication (*e.g.*, SENSAPP communicates with SENSAPP ADMIN through HTTP on port 80, see Listing 4). A Communication can be associated with Resources specifying how to configure the components so that they can communicate with each other.

Listing 4. An example of a Communication type from a CPIM

```
communication SensAppAdmin2SensApp {
  from SensAppAdmin.restRequired
  to SensApp.restProvided,
  resource SensappAdmin2SensAppResource: {
    download: "wget -P ~ http://cloudml.org/scripts/linux/ubuntu/sensappAdmin/
     install_sensappadmin.sh",
    configure : "sudo bash install_sensappadmin.sh"
  }
}
```

A Hosting represents a reusable type of hosting binding between Required- and a ProvidedHost (*e.g.*, a Servlet container is contained by a VM running GNU/Linux, see Listing 5). A Hosting can be associated with Resources specifying how to configure the components so that the contained component can be deployed on the container component.

Listing 5. An example of an Hosting type from a CPIM

```
execution JettySC2SL {
  from JettySC.slRequired to SL.slProvided
}
```

These types can be instantiated to form an assembly that can effectively be deployed. Each instance is identified by a unique identifier and refers to a type (see Listing 6).

Listing 6. Example of instances from a CPIM

```
instances {
  external component bc1 typed BeanstalkContainer
  internal component sensApp1 typed SensApp
  internal component sensAppAdmin1 typed SensAppAdmin
  internal component jettySC1 typed JettySC
  vm sl1 typed SL
  connect sensAppAdmin1.restRequired to sensApp1.restProvided typed SensAppAdmin2SensApp
  host jetty1.slRequired on sl1.slProvided typed JettySC2SL
}
```

3.1.1. Exposing Deployment Data to the Application. `Properties` can be exploited to export environment variables on specific VMs or to PaaS application containers. In particular, such mechanism can be used by application developers and operators to export data or deployment information (*e.g.*, IP address of a VM provisioned during deployment) that can in turn be exploited to properly configure internal components. The export of deployment related information can be specified via a predefined set of substitutable expressions (*e.g.*, the id of the VM hosting the component, see Listing 7) or a cross-reference within the model in the format of XPath[8] (*e.g.*, the id of the SENSAPP component, see Listing 7).

Listing 7. Example of environment variable declaration

```
properties {
  env var HOST_ID: {this.host.id}
  env var SENSAPP_ID: /componentInstances[name='sensapp1']/id
}
```

3.1.2. Support for Scalability. At the instance level, a `ResourcesPoolInstance` represents a set of components to be scaled in or out (*e.g.*, SensApp might be scaled out depending on the workload, see Listing 8). The properties `minReplicats` and `maxReplicats` represent the minimum and maximum number of replicated instances, respectively, embedded into the pool. The property `BaseInstance` represents the list of VMs that constitutes the `ResourcesPoolInstance`. By default, all the `InternalComponents` hosted on these VMs belong the pool. The property `excludedInstances` permits to exclude some.

Listing 8. Example of resource pool

```
resource pool sensappPool {
  minReplicats: 1, maxReplicats: 10
  base instances: sl1
}
```

`ResourcePoolInstances` are typically behind a load balancer. Load balancers may be internal components fully installed and configured by CLOUDML or external components as shown in Listing 9.

Listing 9. Load Balancer as an external component

```
external component LB {
  provider: aws-ec2, service type: loadbalancer
  endPoint: http://ip:5000
  provided communication lbPrv (local, port: 0)
}
```

3.1.3. Integration with Configuration Management Tools. CLOUDML can be extended to support third-party cloud configuration management tools. To exploit Puppet[9] for installing and configuring part of an application on a specific VM, the metamodel includes the concept of `PuppetResource`, which extends `Resource` as shown in Listing 10. The properties `masterEndpoint` and `repositoryEndpoint` represent the endpoints of (i) the Puppet master node responsible for configuring a Puppet client and (ii) the repository containing the Puppet manifests (describing the configuration), respectively. The properties `repositoryKey` and `username` hold the credentials to be

[8] http://www.w3.org/TR/xpath20/
[9] http://puppetlabs.com

used to access the manifest repository. The property `configurationFile` holds the name of the manifest that the Puppet master uses to perform the configuration. Finally, the `configureHostname` command specifies how to modify the hostname of a VM and enables its identification by the Puppet master.

Listing 10. Example of Puppet resource

```
puppet resource puppet_mongodb {
  master endpoint: IP, username: cloudml
  configuration file: puppetMongo
  repository endpoint: ssh://cloudml@ip//etc/puppet/manifests/sensapp-nodes
}
```

3.2. From CPIM to CPSM

In the following, we present how a CPIM can be refined into a CPSM. A deployment model at the CPSM level consists of an enrichment of the instances of the corresponding CPIM with cloud provider-specific information. This enrichment mainly affects external components.

The transformation from CPIM to CPSM fills in additional information regarding both the external resources and how to manage them. External resources depend on each provider's offer and vary in number of cores, RAM size, storage size, etc. In addition, each provider has a different procedure to access and manage resources (IP, authentication protocol, endpoints, etc.), and these details are needed to properly configure the components and their bindings. To transform CPIM into CPSM, the CLOUDML's models@run-time engine directly retrieves these cloud provider-specific information from the cloud providers' APIs. In addition, users can manually refine all or part of the CPIM with provider-specific information such as the type of a resource to be allocated by the provider (*e.g.*, m1.micro on Amazon EC2).

The refinement of a CPIM into a CPSM is performed as follows. The application provider specifies the cloud provider on which the application shall be provisioned and deployed (*e.g.*, the `sensApp1` Servlet, `jettySC1` Servlet container, and the `sl1` VM running GNU/Linux will be provisioned and deployed in the private OpenStack IaaS). The models@run-time engine requests the cloud providers for a list of available VMs compatible with the constraints defined in the VM type (*e.g.*, the list of VMs with at least 1 core, at least 1024 MiB of RAM, and at least 50 GiB of storage available on the private OpenStack IaaS). The cloud provider responds the models@run-time engine with this list along with metadata associated with each VM (*e.g.*, a `small` VM instance located in the `EU`). Similar metadata can be requested for PaaS (*e.g.*, the public Amazon Elastic Beanstalk PaaS supports Java and auto-scaling). Finally, the models@run-time engine uses this metadata to refine the CPIM into a CPSM and enacts the actual provisioning and deployment of this CPSM.

In the following, we describe how the models@run-time environment exploits CPSMs to provision, deploy, and adapt multi-cloud applications.

4. MODELS@RUN-TIME TO SUPPORT CONTINUOUS DEPLOYMENT

The demand to evolve and update systems typically increase with multi-cloud applications, since the cloud enables to dynamically adjust and evolve infrastructures and platforms, while in traditional IT these are rigid or fixed. On the one hand, this indicates more opportunities and flexibility to better evolve and adjust the systems to various needs and requirements. On the other hand, the complexity of designing, delivering, managing, and maintaining such systems challenges current software engineering techniques.

In particular, as stated in [Hüttermann 2012] and promoted by the DevOps movement [Humble and Farley 2010], to reduce the delivery time and foster the continuous evolution of these systems, there is a need to close the gap between development and operation activities. In the following, we detail how we leverage models@run-time to address this issue and support the DevOps ideas.

Models@run-time [Blair et al. 2009] is an architectural pattern for dynamically adaptive systems. It provides an abstract representation of the underlying running system (a model) that facilitates reasoning, simulation, and enacting modifications. A change in the running system is automatically reflected in the model. Similarly, any modification to this model may be enacted on the running system, if need be. This causal connection enables the continuous evolution of the system with no strict boundaries between design-time and run-time activities (addressing R_7).

Exploiting models@run-time for the continuous deployment of cloud applications results in the following process. Developers specify a model of the deployment for their application (*e.g.*, using CLOUDML) and thus automatically deploy it on a test environment. Therefore, they exploit this test environment to tune their application and redeploy it automatically. Any change to the deployment model is reflected (on demand) on the running application while its status is reflected in the model. Once a new release is mature, the developers hand-off the application and its deployment model to the operation team, which exploits the model to deploy the new release in the production environment. The operators can tune this model to maintain and manage the running system. Because the models shared by the developers and operators conform to the same metamodel, they can share and exchange information at any time.

Within CLOUDMF, the models@run-time engine provides a CPSM causally connected to the running system (R_4). On the one hand, any modification to the CPIM will be reflected in the CPSM and, in turn, automatically propagated to the running system. On the other hand, any change in the running system will be reflected in the CPSM, which, in turn, can be assessed with respect to the CPIM.

In the next section, we detail how the CLOUDMF models@run-time engine enables the continuous deployment and adaptation of multi-cloud applications.

5. THE CLOUDMF MODELS@RUN-TIME ENVIRONMENT

Automatic deployment typically comes in two flavours: imperative and declarative [Binz et al. 2013]. The *imperative* approach requires a "deployment plan" that details how to reach the desired application state, usually using a workflow language. This deployment plan defines a sequence of atomic tasks to execute—possibly in parallel. This imperative approach gives full control over the deployment process, but specifying and reusing such plans remains complex. In contrast, the *declarative* approach only requires a specification of the desired application state. This state usually captures the needed application topology using a DSL and a "deployment engine" then computes how to reach this state. This declarative approach better supports evolving and reusing topology models, although the deployment engine may not compute optimal plans.

Existing models@run-time engines typically follow the declarative approach. Projects such as DiVA [Morin et al. 2009], JavAdaptor [Cazzola et al. 2013], Genie [Bencomo et al. 2008] all implicitly define an adaptation plan that specifies the set of actions needed to enact the changes made on the run-time model. However, in complex systems where guarantees in the quality of services are major concerns, we must allow customisation of the adaptation plan.

The CLOUDMF models@run-time engine combines both the declarative and imperative approaches (addressing R_8). The deployment engine generates the provisioning and deployment plan from the defined deployment model, but the user has then the ability to change it before its execution. Moreover, in the case of continuous deployment, the same process takes place. For instance, when the user changes the topology of an application, an adaptation plan is generated, and the user can tune it before the plan is executed.

To this end, we evolved the classical models@run-time architecture [Morin et al. 2009] as depicted by the grey boxes in Figure 2. The reasoning engine reads the current CPSM (Step 1), which describes the running system and produces a target CPSM (Step 2). When a target model is ready (Step 3), it can be compared to the run-time model, if there is one. In contrast with the classical approach, the target model or the result of this comparison is then fed to an adaptation plan generator (Step 4), which produces an initial adaptation plan (Step 5). This plan is exposed by the models@run-time engine and can be modified by third parties. Once the appropriate plan is identified, it can be validated (Step 6) and finally executed (Step 7). The execution engine then triggers a sequence of atomic actions (Step 8) on the running system (Step 9).

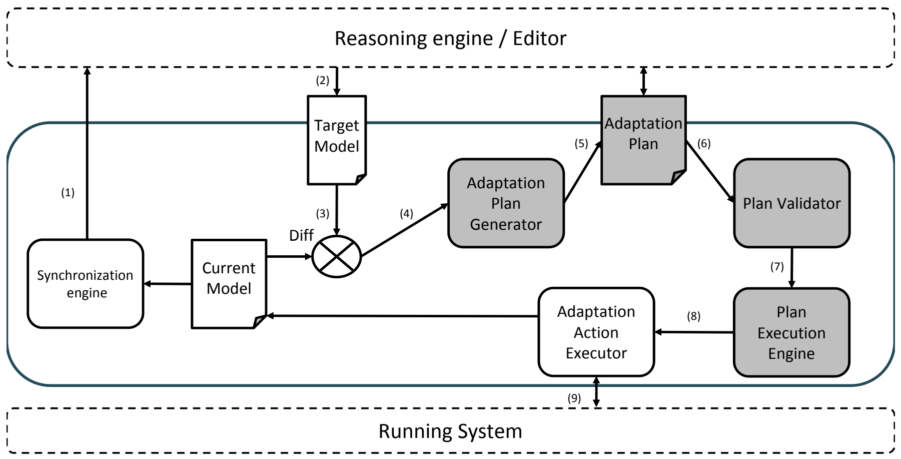

Fig. 2. The CLOUDMF models@run-time architecture

5.1. Comparison Engine

The inputs to the comparison engine (also called `Diff`) are the current and target deployment models. The output is a list of actions to transform the current model into the target model. These actions can result in: *(i)* modification of the deployment and resource provisioning topology, *(ii)* modifications of the components' properties, or *(iii)* modifications of their status on the basis of their life-cycle. In particular, the status of an external component (*i.e.*, representing a VM or a PaaS solution) can be: `running`, `stopped`, or in `error`. The status of an internal component (*i.e.*, representing the software to be deployed on an external component) can be: `uninstalled`, `installed`, `configured`, `running`, or in `error`.

The comparison engine processes the entities of the deployment models in the following order: `external components`, `internal components`, `execution binding`, and `relationships`, according to their dependencies. Any component is therefore deployed after its dependencies. For each of these concepts, the engine compares the instances from the current model with the instances from the target model. This comparison matches the properties of the instances, their types, as well as their dependencies (*e.g.*, if the host of a component has changed, this component should be redeployed). Each unmatched instance in the current model yields a `remove` action. Similarly, each unmatched instance from the target model yields an `add` action.

The target model has priority over the current model. For example, any VM in the target model that does not exist in the current model will be created, and, conversely, any VM in the current model that does not exist in the target model will be removed. Coping with changes occurring during reasoning could be handled in various ways, *e.g.*, as part of a third step of the adaptation process (model checking). Currently, the models@run-time engine does not handle such changes.

In the motivating example (*cf.* Section 2), the migration of SENSAPP to a new VM generates new instances of the VM type `SL` and the hosting type `JettySC2SL` but also removes the existing instance of the hosting type `JettySC2SL`. Because of the abstraction offered by CLOUDML, this adaptation is achieved without the need for new types and resources or for redefining the existing ones. In addition, thanks to the comparison process, the MongoDB remains available during the adaptation process.

5.2. From CLOUDML Models to Deployment Plans

Engines that generate adaptation plans are typically dedicated to the domain of the models@run-time environment. In our context, we created a transformation that generates an adaptation plan from a CLOUDML model or the comparison between two of them. We model adaptation plans as workflows. Although existing models@run-time environments usually derive adaptation plans as a sequence of actions, these actions may often be executed in parallel. For instance, in CLOUDMF, adaptation actions are executed in parallel as the associated running system is inherently distributed, and deployment actions are often time-consuming. More precisely, the plans are generated using the language presented in Section 5.3.

The transformation consists of the following four rules that are executed sequentially. During the execution of each rule, a set of `ActivityNodes` and `ActivityEdges` is created, together with the data required for the proper execution of the `Actions`.

(1) **Provision cloud resources:** This rule yields the actions responsible for the parallel provisioning of VMs or cloud services on a given cloud provider. A `Fork` node is created to enable parallel provisioning.
(2) **Install components:** This rule creates an action for each of the upload, download, and install commands associated with the life-cycle of each internal component. In addition, if a component depends on another, it ensures that the related actions are created in the proper order (*i.e.*, required components are installed first).
(3) **Connect components:** This rule creates an action for each command in the resources associated with each communication. Moreover, it creates a `Join` node to synchronise install actions of the components involved in the communication.
(4) **Start components:** This rule creates an action for each of the configure and start commands of each component.

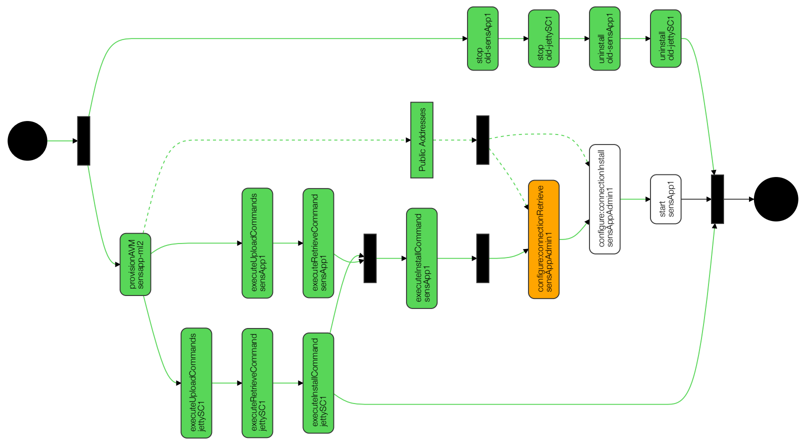
stop old-sensApp1
stop old-jettySC1
uninstall old-sensApp1
uninstall old-jettySC1
Public Addresses
provisionAVM sensapp-ml2
executeUploadCommands sensApp1
executeRetrieveCommand sensApp1
executeInstallCommand sensApp1
configure:connectionRetrieve sensAppAdmin1
configure:connectionInstall sensAppAdmin1
start sensApp1
executeUploadCommands jettySC1
executeRetrieveCommand jettySC1
executeInstallCommand jettySC1

Fig. 3. Example of adaptation plan

The comparison engine yields the adaptation plan depicted in Figure 3 when migrating SENSAPP on a new VM. In the following, we present our DSL for specifying adaptation plans.

5.3. A DSL for Adaptation Plans

Our language to model adaptation plans has evolved from a subset of UML activity diagrams. Figure 3 reflects the synchronisation needed to migrate SENSAPP on a new VM in a new cloud. It first provisions in parallel all the cloud resources, then stops and uninstalls the software components that are migrated. Once the VM provisioned, it *(i)* installs the Jetty application container while it *(ii)* retrieves in parallel the code of the application. Then, and before the migrated SENSAPP is started, the SENSAPP ADMIN is configured to interact with the new instance of SENSAPP.

5.3.1. Metamodel. Figure 4 shows the metamodel of our language. An adaptation plan consists of an `Activity` made of `ActivityNodes` and `ActivityEdges`. An `ActivityNode` can be an `Action` performed on the running system. For instance, in the context of CLOUDML, it represents deployment actions such as the provisioning of a VM. `ActivityNodes` are depicted in the diagram as rounded boxes. An `ObjectNode` is another type of `ActivityNode` that stores data for later retrieval by `Actions` (*e.g.*, the IP address of the provisioned VM). `ObjectNodes` are depicted in the diagram as rectangles with text inside. Finally, an `ActivityNode` can be an `ActivityInitial` or `ActivityFinal` node specifying the start and end of an adaptation plan, but also a `Join` or `Fork` node to parallelise and synchronise the execution of independent `Actions`. These nodes are depicted in the diagram as black circles and black rectangles, respectively. An `ExpansionRegion` specifies that tasks must be executed several times in parallel or sequentially. `ActivityNodes` are linked through `ActivityEdges`. The property `objectFlow` specifies that the edge represents either the control flow orchestrating the execution of `ActivityNodes` or the data flow to exchange objects between tasks. The `ActivityEdges` representing the control flow are depicted in the diagram as plain arrows while the data flow ones are depicted as dashed arrows.

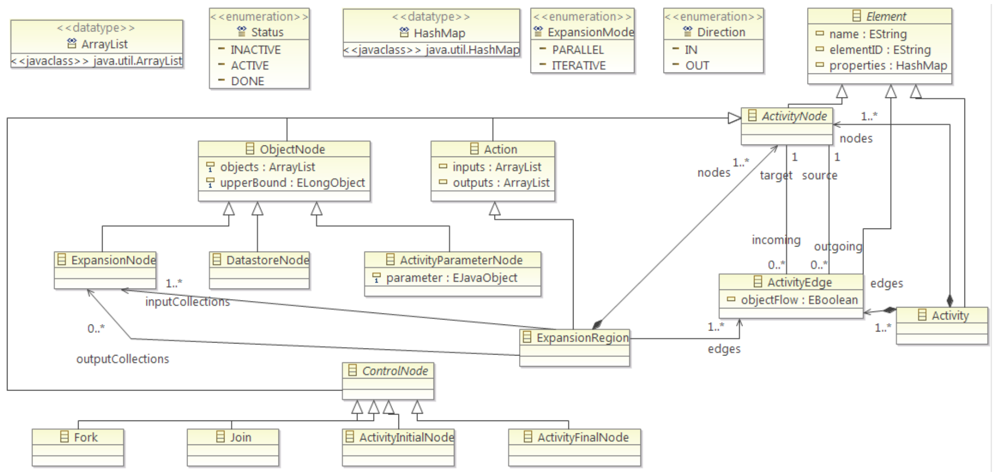

Fig. 4. Adaptation plans metamodel

5.3.2. Language Implementation. Our language is implemented as plain Java objects, and also exists as an internal DSL within the CLOUDMF API. As for our SENSAPP example, a whole deployment plan is represented as an `Activity` (see Listing 11).

Listing 11. Creating a deployment plan
```
Activity deploymentPlan = ActivityBuilder.getActivity();
```

The minimal set of nodes required for an executable plan consists of one `StartNode`, one or several `ActionNodes` and one `StopNode` (see Listing 12). The first task that has to be performed during a deployment is the provisioning of the VMs. For each VM to provision, an action is created in the plan. Two parameters must be defined when creating an action: *(i)* the name of the method that will be used to enact the action (*e.g.*, reflection will be used to call the "provision" method that exploits the cloud provider API to actually provision the VM); and *(ii)* the necessary information to perform the call.

Listing 12. Creating start, stop and action nodes
```
ActivityInitialNode start = ActivityBuilder.controlStart();
Action provision = ActivityBuilder.actionNode("Provision",VM);
```

```
ActivityFinalNode stop = ActivityBuilder.controlStop();
```

VMs can typically be provisioned in parallel. Thus, a `ForkNode` must connect the provisioning actions. A similar operation has to be performed to synchronise the tasks after provisioning (see Listing 13). The Boolean parameter used when creating these nodes indicates if the operators are applied to the control or data flows. In this case, they are applied to the control flow.

Listing 13. Creating synchronisation actions

```
Fork fork = ActivityBuilder.forkNode(false);
ActivityBuilder.connect(fork, provision, true);
Join join = ActivityBuilder.joinNode(false);
ActivityBuilder.connect(provision, join, true);
```

Finally, `ObjectNodes` can be created to store some data, such as the IPs of the provisioned VMs (see Listing 14). An `ObjectNode` can be specialised into two types: `DatastoreNode`, which does not allow to store duplicates or `ParameterNode`, which can be used to supply some data to the deployment plan at the beginning of its execution.

Listing 14. Creating an object node

```
ObjectNode IP = ActivityBuilder.objectNode("IP", Type);
```

5.4. Execution Engine

Once an adaptation plan has been generated and validated, the adaptation engine traverses the plan (*i.e.*, taking into account parallelisation and synchronisation points) and executes every `Action`.

This engine relies on two libraries: the Java Reflection API and the Java Fork/Join framework. The first one is used to ensure the independence of both the language and the execution engine from the domain on which the models@run-time approach is applied. Each action within an adaptation plan refers to a method that will enact the adaptation. Thus, the execution engine exploits the reflection mechanism to trigger the call to the specified method. The second library supports the concurrency and synchronisation, necessary to start multiple threads asynchronously and for joining them afterwards.

Thus, the engine concurrently executes each branch: there are no time dependencies between tasks in parallel branches. It creates new threads whenever it enters a `Fork` node (explicit fork) or an `Action` node with multiple outgoing edges (implicit fork), and waits for these threads whenever it reaches a `Join` node (explicit join) or an `Action` node with multiple incoming edges (implicit join).

The execution engine also tracks the status of every adaptation action and reflects their status in the adaptation plan. Each node and edge may be `Inactive`, `Active`, or `Done` (*i.e.*, before the execution of the node/edge, during, and after the execution respectively). The adaptation plan becomes a run-time model of the adaptation process that co-evolves with the run-time model of the running system.

5.5. Interaction with the Models@Run-Time Environment

In addition to the run-time model, third parties can interact with the CLOUDMF models@run-time environment using a synchronisation engine or a pre-defined set of high-level commands.

The models@run-time environment allows remote third parties (*e.g.*, reasoning engines) to adapt the system. This need has emerged from several use cases that apply self-adaptive mechanisms on top of CLOUDMF. These mechanisms typically involve decision-making engines, which reason on their views of the model of the running system. Because these views must remain synchronised with the running system, appropriate transformations must be automatically triggered at run-time.

Such model synchronisations are implemented by propagating changes in both directions, namely *notification* and *command*. A notification allows the models@run-time environment to propagate changes to third parties, while a command enables modifications on the current CPSM. Because two models used by two parties can be isolated from each other and may not be aware of the whole model state, only the sequence of modifications is propagated, without carrying the start state of each change. Therefore, both notification and command are a sequence of modifications.

The communication with third parties relies on the WebSocket protocol. Events are encoded as plain text and can be defined using a domain-specific language. This includes the text formatting, the query and criteria to locate the relevant model element, the modification or change on the element, and the combination of other events. We have defined the standard MOF-reflection modifications as the primitive events, and allow developers to define further high-level events as the composition of primitive ones. Using this language, it is also possible to define the changes on an abstract model as the composition of events on a concrete model and thus implement event-based transformation. After each adaptation, the engine wraps the modification events into one message and sends it to the WebSocket port.

To handle conflicting actions from several third parties, the models@run-time environment uses transactions. The WebSocket component creates a single transaction that contains all the modifications from a third party and passes it to a concurrency handler. The handler queues this transaction and executes them sequentially. Since all the modifications are simply assignment or object instantiation commands on the model in the form of Java objects, the time to finish a transaction of events is significantly shorter than the adaptation process.

The deployment engine also provides operators with high-level commands that avoid direct manipulation of the models. In particular, the `scale` command enables scaling out a VM and the `burst` command enables scaling out a VM in another cloud.

6. SYNTHESIS

In this section, we discuss how our approach addresses the requirements defined in Section 2 and describe the CLOUDMF reference implementation.

6.1. Evaluation

CLOUDMF has been applied in a set of use cases in various research projects and in particular the FP7 EU projects: MODAClouds, PaaSage, and Diversify. Due to size limitation, we will focus on the evaluation performed within the MODAClouds project as it is the most advanced and complete, while we will briefly discuss the results of the PaaSage and Diversify projects.

6.1.1. MODAClouds. MODAClouds delivers an advanced model-driven approach along with an integrated software development environment to support engineers in building, deploying, and managing multi-cloud applications, together with related data [Ardagna et al. 2012]. CLOUDMF is exploited here both at design-time to describe the deployment of multi-cloud applications at the CPIM and CPSM levels and at run-time to manage the deployed applications. In particular, CLOUDMF has been applied in the context of four case studies:

- **Business Process Modelling System (BOC):** ADOxx is a metamodelling desktop application. The objective of this case study was to leverage the MODAclouds technology to provide this application as SaaS. From the CLOUDMF perspective, this case study evaluated *(i)* the deployment of ADOxx on an IaaS infrastructure (*i.e.*, CloudSigma[10]), *(ii)* the integration of CLOUDMF with Puppet, and *(iii)* the cloud bursting and multi-cloud load balancing features.
- **Smart City Urban Safety Planner (Siemens):** the objective of this system is to collect and analyse data from various sources to validate and react to emergency events that might occur in a city. From the CLOUDMF perspective, this case study evaluated its ability *(i)* to deploy and manage application running a complex software stack (*i.e.*, Apache Storm and Cassandra

[10]http://cloudsigma.com

clusters) on an IaaS infrastructure and *(ii)* to partially replicate the application over two clouds (*i.e.*, Flexiant[11] and Amazon EC2[12]).

- **Health-care application (Atos):** is an e-Health solution offering an integrated and online clinical, educational, and social network for patient suffering from dementia. From the CLOUDMF perspective, this case study evaluated *(i)* the deployment of an application on a private PaaS (*i.e.*, application container and database) running Cloud Foundry[13], and *(ii)* the bursting of the client part of the application on a public PaaS (*i.e.*, Pivotal[14]).
- **Constellation (Softeam):** Modelio[15] is a desktop modelling tool. The objective of this case study was to extend Modelio with new features hosted in the cloud (*i.e.*, model versioning and sharing). From the CLOUDMF perspective, this case study evaluated *(i)* the deployment of an application on both IaaS and PaaS solutions from multiple clouds providers (*i.e.*, Amazon RDS, Amazon EC2, Amazon SQS, Flexiant), and *(ii)* the automatic scale-in and -out of part of the application depending on the load.

The evaluation of CLOUDMF has been conducted as part of the evaluation of the overall project results, which consisted in the following steps: *(i)* elicitation of the requirements following the Goal-Question-Metric (GQM) methodology [Solingen and Berghout 1999] (see [Solberg et al. 2014]); *(ii)* initial evaluation by the technical partners as well as the case study providers (see [Pop et al. 2014]); *(iii)* addition of new requirements (see [Ferry et al. 2015]); and *(iv)* final evaluation by the technical partners as well as the case study providers (see [Ferry et al. 2015]).

Following the GQM methodology, for each artefact of the project, goals were defined as well as questions were used to characterise how the assessment of a specific goal will be performed. Finally, metrics were associated with each question to answer it quantitatively. The evaluators were not only asked to assess these questions and metrics but also to provide recommendations and to spot deviations. Table I, extracted from [Ferry et al. 2015], summarizes the evaluation of CLOUDMF.

6.1.2. PaaSage.

Context and ecosystem. PaaSage delivers an integrated platform to support modelling and execution of cloud applications on multiple clouds, together with an accompanying methodology to allow model-based provisioning, deployment, and adaptation of these applications independently of the existing underlying cloud infrastructures and platforms [Jeffery et al. 2013]. To cover the necessary aspects of the modelling and execution of multi-cloud applications, PaaSage adopts the Cloud Application Modelling and Execution Language (CAMEL). CAMEL integrates and extends existing DSLs, namely CLOUDML, Saloon [Quinton et al. 2013], and the organisation part of CERIF [Jeffery et al. 2014]. In addition, CAMEL integrates new DSLs developed within the project, such as the Scalability Rule Language (SRL) [Kritikos et al. 2014].

Evaluators. The use case providers of PaaSage performed an evaluation of CAMEL, within which the deployment package is based on CLOUDML. The use cases involved the provisioning, deployment, and adaptation of multi-cloud applications from the heterogeneous domains of eScience, industrial sector, public sector, and financial sector.

Overall evaluation. In summary, the participants were satisfied with the expressive power provided by CLOUDML for specifying components as well as VMs (78% being satisfied or very satisfied and the remaining 22% moderately satisfied). Similarly, participants were satisfied with the expressive power for specifying hosting and communication bindings. In this respect, they only requested the capability to distinguish between TCP and UDP ports. The interested reader may consult [Rossini et al. 2017] for the complete evaluation of CAMEL.

[11]http://flexiant.com
[12]http://aws.amazon.com
[13]http://cloudfoundry.com
[14]https://pivotal.io
[15]https://www.modeliosoft.com

Table I. Summary of the evaluation of CloudMF by the MODAClouds EU project

CloudML modelling capabilities	**Multi-cloud IaaS and PaaS deployment and adaptation**
CloudML has been used and evaluated by all the partners. They were all able to model the deployment of their application on multiple clouds at both the IaaS and PaaS level. This also includes the modelling of *(i)* Puppet resources and *(ii)* multi-cloud load balancers. It is also reported in the evaluation that the language has also been used to model different deployment alternatives for the same application. At design-time, it is worth noting that Modelio was the preferred editor for specifying deployment models while the web-based graphical user interface was preferred for the run-time monitoring of the status of the deployment. Finally, the JSON textual syntax was preferred by the reasoning engines.	All partners reported successful continuous deployment of their applications on multiple clouds on both IaaS and PaaS solutions. This also includes the deployment of the application on top of Windows-based VMs. In particular, it is reported that: "The fact that the models@run-time engine builds on the CloudML deployment technology already tested during the deployment use case was very beneficial. We could simply incrementally update the models and the deployments worked as expected." One of the recommendations made after the first evaluation was to provide mechanisms to synchronise and control deployment actions happening in different cloud instances during deployment (*e.g.*, specify how and when a software component should be initialized). CloudMF has been successfully extended and evaluated in this respect. In particular, in the context of the Constellation use case, it is reported that the models@run-time engine "allows us to automate the deployment of the Constellation server on cloud environment and monitor the deployment process. Very useful to observe the current state of the deployment and interact with it." The scaling and bursting commands of CloudMF have been successfully exploited in the case studies. As a limitation of the CloudMF models@run-time engine, it has been suggested that, when feasible, security groups could be automatically generated on the cloud provider site based on the ports specified in the deployment model.

6.1.3. Diversify.

***Context and ecosystem*.** Diversify investigates whether the diversity of a system architecture improve its quality. The project focused mainly on two quality properties: the robustness to internal failures and the flexibility to context changes. Within Diversify, CloudMF is exploited to assess these two quality metrics on cloud applications. In particular, CloudMF is used as the basis for automatically planning valid architectures with different diversity levels [Song et al. 2015]. In this context, the CloudMF models@run-time environment is combined with different reasoning engines. These engines use model transformations and constraint solving to generate valid CloudML instance models, and then exploit the CloudMF models@run-time environment to automatically deploy the systems according to these models.

***Evaluators*.** CloudMF is used by two partners inside the Diversify project, *i.e.*, SINTEF and Trinity College Dublin. The experiments were conducted in the smart city domain with a system named SmartGH and developed within the Diversify project [Nallur et al. 2015].

***Overall evaluation*.** CloudMF has been successfully exploited [Chauvel et al. 2015] and while implementing these experiments, the partners observed that CloudMF offers adequate support for modelling the deployment of cloud applications and their automatic configuration and deployment. Moreover, it appears that the proposed models@run-time approach facilitates its integration with external reasoning engine [Chauvel et al. 2013].

6.2. Requirements Fulfilment

The following list summarises how CloudMF fulfils the requirements presented in Section 2.

— **Cloud provider-independence** (R_1)**:** The layering of the modelling stack into CPIMs and CPSMs ensures that the provisioning and deployment templates are cloud provider-independent.
— **Separation of concerns** (R_2)**:** The component-based design of the CloudML metamodel ensures that the provisioning and deployment components within a model are modular and loosely

coupled. Furthermore, the CPIM and CPSM abstraction levels effectively ensure the separation of concerns between the cloud provider-specific and the cloud provider-independent provisioning and deployment design.

— **Reusability (R_3):** The type-object pattern in the CLOUDML metamodel ensures that types can be reused across several models.
— **Abstraction (R_4):** Our framework offers a single domain-specific language and abstraction that enables the management of application on both IaaS and PaaS solutions. Independently of their delivery model, these solutions can be represented in a homogeneous way as components. In addition, the models@run-time engine provides an abstract and up-to-date representation of the running system that can be dynamically manipulated, and the CPIM provides abstraction over cloud provider-specific concerns.
— **White- and Black-box infrastructure (R_5):** CLOUDMF includes concepts and mechanisms that support both IaaS and PaaS solutions, enabling management of components where CLOUDMF has full control of their underlying infrastructure and platforms (IaaS/white-box) and exploitation of advanced and rigid PaaS solutions that offer little control from the outside (black-box).
— **Application technology independence (R_6):** The concept of component within CLOUDML is independent of the implementation of the software it represents. A component is related to the software via the set of commands describing how to manage its deployment life-cycle.
— **Reduction of the gap between design and run-time activities (R_7):** CLOUDMF allows exploiting the same model and language (*i.e.*, CLOUDML) for both design- and run-time activities. This facilitates interactions between operation and development teams.
— **Fully controlled automated deployment (R_8):** The CLOUDMF run-time environment includes a DSL for specifying adaptation plans together with an execution engine that provides control over the deployment process. The plans can be automatically generated providing useful insights on how the deployment will be performed and can be tuned adequately if desired.

6.3. Reference Implementation

The CLOUDMF graphical editor is a Javascript application, the textual syntax editor[16] is based on the Xtext framework [Eysholdt and Behrens 2010] and leverages an Ecore representation of the CLOUDML metamodel.

The models@run-time engine is a plain Java application, available as an open-source project.[17] It reifies internally the CLOUDML metamodel as plain Java objects, and offers an internal DSL that helps developers modify topologies programmatically. We serialise these internal graphs of Java objects in JSON and XMI using the Kevoree Modeling Framework (KMF) [Fouquet et al. 2012], which in turn leverages our Ecore representation of our CLOUDML metamodel. Our models@run-time engine also exploits the Java reflection and the join/fork frameworks. In addition, an engine has been created to automatically generate a DOT file from an adaptation plan together with a web page for their run-time graphical visualisation. The adaptation plan language is implemented as plain Java objects, and also exists as an internal DSL within the CLOUDMF API. For the IaaS level management, the provisioning and deployment engine relies on jclouds[18] plus some Iaas and PaaS ad-hoc adaptors. This engine has been successfully tested against the following cloud solutions: Cloud Foundry, CloudBees, Pivotal, Amazon SQS, Beanstalk, SQS and RDS, CloudSigma IaaS, Amazon EC2, Microsoft Azure, Flexiant, and OpenStack.

7. RELATED WORK

In the literature, several efforts aimed to support the design, optimisation and management of multi-cloud applications.

[16]https://github.com/SINTEF-9012/cloudml-dsl
[17]https://github.com/SINTEF-9012/cloudml
[18]http://jclouds.incubator.apache.org/

Cloud Modelling Languages. Several domain-specific languages exist in the literature to specify the deployment and orchestration of cloud services. The Cloud Application Modelling Language (CAML) [Bergmayr et al. 2014] developed during the ARTIST EU project supports the specification of deployment topologies as well as their refinement towards concrete cloud solutions. CAML is a UML internal language that consists of a set of UML profiles capturing cloud solutions from different perspectives, *e.g.*, offerings and cost. In contrast with CLOUDMF, environment-specific information is captured by UML profiles concerning well-known cloud environments (*e.g.*, Amazon AWS, Google Cloud Platform). CAML fosters reuse of deployments through the concept of templates. As opposed to CLOUDMF, CAML does not include an execution engine to enact the deployment of multi-cloud applications.

CloudMIG [Frey and Hasselbring 2011] focuses on the migration of legacy applications to cloud environments. In particular, CloudMIG focuses on the reverse engineering applications into models conforming to the Knowledge Discovery Metamodel (KDM) as well as their cloud deployment. Cloud environments are specified using Cloud Environment Models (CEM), which offer support for the infrastructure and platform levels. The CloudMIG Xpress tool enables the automatic computation of optimal cloud deployments [Frey et al. 2013].

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [Palma and Spatzier 2013] is a specification developed by the OASIS consortium, which provides a language for specifying the components comprising the topology of cloud applications along with the processes for their orchestration. TOSCA supports the specification of types and templates, but not instances necessary to support the run-time. A TOSCA "Ad Hoc" group has been created to examine how TOSCA could be extended to support instance modelling and run-time aspects.

In contrast with CLOUDMF, the aforementioned approaches are conceived for design-time only, and models are not causally connected to the run-time system.

Cloud Application Management. In the cloud community, libraries such as jclouds[19] or Libcloud[20] provide generic APIs abstracting over the heterogeneous APIs of IaaS providers, thus reducing cost and effort of deploying multi-cloud applications. While these libraries effectively foster the deployment of cloud applications across multiple cloud infrastructures, they remain code-level solutions, which makes design changes difficult and error-prone. More advanced frameworks such as Cloudify[21], Puppet[22], or Chef[23] provide DSLs that facilitate the specification and enactment of provisioning, deployment, monitoring, and adaptation of cloud applications, without being language-dependent. As for the research community, the mOSAIC [Sandru et al. 2012] project tackles the vendor lock-in problem by providing an API for provisioning and deployment of multi-cloud applications. This solution is also limited to the code level.

The literature encompasses several approaches to the management of cloud applications deployed on PaaS environments. Sellami *et al.* [Sellami et al. 2013] propose a model-driven approach to PaaS-independent provisioning and management of cloud applications. This approach includes a language for modelling provisioning and deployment, as well as a REST API for enacting them. The Cloud4SOA EU project[24] provides a framework for facilitating the matchmaking, management, monitoring, and migration of cloud applications on PaaS environments. In contrast with CLOUDMF, these approaches focus on one cloud delivery model only (*i.e.*, either IaaS or PaaS, but not both). In addition, their models are not causally connected to the running system and may become irrelevant as soon as the running system is changed. The approaches proposed in the CloudScale [Brataas et al. 2013] and Reservoir [Rochwerger et al. 2009] projects suffer similar limitations.

[19]http://jclouds.apache.org
[20]https://libcloud.apache.org
[21]http://www.cloudifysource.org/
[22]https://puppetlabs.com/
[23]http://www.opscode.com/chef/
[24]http://www.cloud4soa.eu

The work of Shao *et al.* [Shao et al. 2010] was a first attempt to build a models@run-time platform for the cloud, but remains restricted to monitoring, without providing support for enactment of provisioning and deployment. To the best of our knowledge, CLOUDMF is thus the first attempt to reconcile cloud management solutions with modelling practices through the use of models@run-time.

8. DISCUSSION AND FUTURE WORK

The development of CLOUDMF brought new insights about the design and engineering of DSMLs. In this section, we will briefly discuss these aspects before we present future works.

DSMLs capture information that can be observable, controllable or both. The *controllability*, which reflects the extent to which we can observe and control, is inherent to the application domain and cannot be changed. However, CLOUDMF offers a means to increase (or decrease) controllability by migrating from a PaaS to an IaaS solution. Choosing an IaaS solution provides operations with a higher visibility and control over the underlying software stack. In contrast, choosing a PaaS solution delegates the management of the software stack to the PaaS provider. CloudMF allows to combine and dynamically migrate parts of the system across the IaaS and PaaS space, which essentially offers a slider mechanisms with respect to controllability.

From the software engineering standpoint, the design of CLOUDMF entailed the definition of a domain-specific (object-oriented) model tailored for cloud applications. This model is implemented as plain Java objects (POJO), which are made available through multiple “adapters”. Some adapters convert this data model into other formats such as ECore or JSON, while other adapters connect with specific cloud APIs. The focus on domain-specific concepts beyond any textual or graphical syntax led to an architecture similar to those realised according to *domain-driven design* (DDD) [Evans 2004], which is proven to be especially successful for enterprise systems. The focus on domain concepts brings stability, as domain concepts evolve much more slowly than the underlying technologies.

Cloud computing still evolves quickly. Thus, it remains imperative for CLOUDMF to be easily extendible to new APIs, services, and concepts. As for cloud APIs, our architecture helps to reduce the cost of integration with new cloud providers: new adapters must be developed to synchronise our in-memory model with these new cloud APIs. By contrast, new concepts such as “container”, which emerged as we developed CLOUDMF, required changing the domain model itself.

Moreover, as the cloud solutions evolve, the enactment of a deployment must be tailored to best fit with applications requirements. In complex systems such as cloud applications, where QoS is a major concern, the deployment engine must allow for customisation of the provisioning, configuration, and deployment plan [Binz et al. 2013]. Typical models@run-time systems do not permit correction of the behaviour of the models@run-time engine (*i.e.*, to influence which actions are triggered and in which order). In specific cases, this may lead to a non-optimal adaptation plan and even to erroneous states during the execution of the adaptation. To overcome this limitation, we extended the classical models@run-time pattern as follows. First, the models@run-time engine derives a tentative adaptation plan from a target model describing the desired system state. Next, the user or a reasoning engine can adjust it. Finally, the consolidated adaptation plan is fed to the adaptation engine, which is responsible for its execution. From the language perspective, such feature provides the ability to partially and dynamically adapt the operational semantic of the deployment engine (*i.e.*, the language interpreter).

In future work, we plan to extend CLOUDMF towards the Big Data domain. In particular, we are interested in extending both the language and the models@run-time engine with specific support for the deployment, configuration, and adaptation of complex big data frameworks such as Apache Storm and Kafka. Such automation is important as, after the initial prototyping, data engineers often need sophisticated refinement, optimisation, and management on the application. For example, depending on the volume of the data to be processed, they often need to fine-tune the parameters of the supportive data processing framework, add more nodes for parallel processing, or increase the computing resources.

We will also evolve the current CLOUDMF models@run-time architecture to facilitate the evolution of its monitoring part. The monitoring transformation within a models@run-time engine exploits the APIs of the system to build a model of its internal state, which may include its main components, their current status as well as the environment. While models@run-time help continuously adjust to a changing environment, not all changes can be foreseen at design-time, and some will eventually exceed the abilities of adaptations. For instance, in pervasive systems where mobile devices may join or leave at any time, we must dynamically extend the metamodel to capture new unforeseen types of devices. Besides, the API may offer different calls to retrieve data about these new devices. Therefore, the models@run-time engine must allow for both the customisation of its abstractions and the customisation of the monitoring of these abstractions.

9. CONCLUSION

We presented how the Cloud Modelling Framework (CLOUDMF) make the most of model-driven techniques and methods to reduce the vendor lock-in and support the provisioning and deployment of multi-cloud applications. On the one hand, the Cloud Modelling Language (CLOUDML) allows capturing the provisioning and deployment of these applications in cloud provider-independent models. On the other hand, the models@run-time environment brings life to these models and automates key operations such as provisioning, deployment, and adaptation. Therefore, CLOUDMF further unifies development and operation activities, with a single set of concepts integrated within tools supporting both activities.

References

Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Balligny, Francesco D'Andria, Cosmin-Septimiu Nechifor, and Craig Sheridan. 2012. MODACLOUDS, A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds. In *ICSE MiSE*. IEEE/ACM, 50–56.

Colin Atkinson and Thomas Kühne. 2002. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation* 12, 4 (2002), 290–321.

Nelly Bencomo, Paul Grace, Carlos A. Flores-Cortés, Danny Hughes, and Gordon S. Blair. 2008. Genie: supporting the model driven development of reflective, component-based adaptive systems. In *ICSE 2008*. ACM, 811–814.

Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. 2014. UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. In *CloudMDE 2014 (CEUR Workshop Proceedings)*, Vol. 1242. CEUR, 56–65.

Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. 2013. OpenTOSCA—A Runtime for TOSCA-Based Cloud Applications. In *ICSOC 2013 (LNCS)*, Vol. 8274. Springer, 692–695.

Gordon S. Blair, Nelly Bencomo, and Robert B. France. 2009. Models@run.time. *IEEE Computer* 42, 10 (2009), 22–27.

Gunnar Brataas, Erlend Stav, Sebastian Lehrig, Steffen Becker, Goran Kopčak, and Darko Huljenic. 2013. CloudScale: scalability management for cloud systems. In *ICPE 2013*. ACM, 335–338.

Walter Cazzola, Nicole Alicia Rossini, Mohammed Al-Refai, and Robert B. France. 2013. Fine-Grained Software Evolution Using UML Activity and Class Models. In *MODELS 2013 (LNCS)*, Vol. 8107. Springer, 271–286.

Franck Chauvel, Nicolas Ferry, Brice Morin, Alessandro Rossini, and Arnor Solberg. 2013. Models@Runtime to Support the Iterative and Continuous Design of Autonomous Reasoners. In *MRT 2013, co-located with MODELS 2013 (CEUR Workshop Proceedings)*, Vol. 1079. CEUR.

Franck Chauvel, Hui Song, Nicolas Ferry, and Franck Fleurey. 2015. Evaluating robustness of cloud-based systems. *Journal of Cloud Computing* 4, 1 (2015), 1–17.

Eric Evans. 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Moritz Eysholdt and Heiko Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *SPLASH/OOPSLA 2010*. ACM, 307–309.

Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. 2013. Managing multi-cloud systems with CloudMF. In *NordiCloud 2013*. ACM, 38–45.

Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. 2013. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *CLOUD 2013*. IEEE CS, 887–894.

Nicolas Ferry, Hui Song, Alessandro Rossini, Franck Chauvel, and Arnor Solberg. 2014. CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications. In *UCC 2014*. IEEE CS, 269–277.

Nicolas Ferry et al. 2015. *D3.7.2—MODAClouds evaluation report, final version*. MODAClouds project deliverable.

François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. 2012. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *MODELS 2012 (LNCS)*, Vol. 7590. Springer, 87–101.

Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. 2013. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *ICSE 2013*. IEEE/ACM, 512–521.

Sören Frey and Wilhelm Hasselbring. 2011. The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications. *Intl. J. Advances in Software* 4, 3&4 (2011), 342–353.

Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

Michael Hüttermann. 2012. *DevOps for developers*. Apress.

Keith Jeffery, Geir Horn, and Lutz Schubert. 2013. A Vision for Better Cloud Applications. In *MultiCloud 2013*. ACM, 7–12.

Keith Jeffery, Nikos Houssos, Brigitte Jörg, and Anne Asserson. 2014. Research information management: the CERIF approach. *IJMSO* 9, 1 (2014), 5–14.

Kyriakos Kritikos, Jörg Domaschka, and Alessandro Rossini. 2014. SRL: A Scalability Rule Language for Multi-Cloud Environments. In *CloudCom 2014*. IEEE CS, 1–9.

Thomas Kühne. 2006. Matters of (meta-)modeling. *Software and Systems Modeling* 5, 4 (2006), 369–385.

Peter Mell and Timothy Grance. 2011. *The NIST Definition of Cloud Computing*. Special Publication 800-145. National Institute of Standards and Technology.

Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. 2009. Models@Run.time to Support Dynamic Adaptation. *IEEE Computer* 42, 10 (2009), 44–51.

Vivek Nallur, Amal Elgammal, and Siobhán Clarke. 2015. Smart Route Planning Using Open Data and Participatory Sensing. In *OSS 2015 (IFIP Advances in Information and Communication Technology)*, Vol. 451. Springer, 91–100.

Derek Palma and Thomas Spatzier. 2013. *Topology and Orchestration Specification for Cloud Applications (TOSCA)*. Technical Report. Organization for the Advancement of Structured Information Standards (OASIS). http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.pdf

Dana Petcu. 2014. Consuming Resources and Services from Multiple Clouds—From Terminology to Cloudware Support. *Journal of Grid Computing* 12, 2 (2014), 321–345.

Daniel Pop et al. 2014. *D3.7.1—MODAClouds evaluation report, initial version*. MODAClouds project deliverable.

Clément Quinton, Daniel Romero, and Laurence Duchien. 2013. Cardinality-based feature models with constraints: a pragmatic approach. In *SPLC 2013*. ACM, 162–166.

B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán. 2009. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development* 53, 4 (July 2009), 535–545.

Alessandro Rossini, Kiriakos Kritikos, Nikolay Nikolov, Jörg Domaschka, Daniel Seybold, Frank Griesinger, Daniel Romero, Michal Orzechowski, Georgia Kapitsaki, and Achilleas Achilleos. 2017. The Cloud Application Modelling and Execution Language (CAMEL). *OPen Access Repository of Ulm University (OPARU)* (2017).

Calin Sandru, Dana Petcu, and Victor Ion Munteanu. 2012. Building an Open-Source Platform-as-a-Service with Intelligent Management of Multiple Cloud Resources. In *UCC 2012*. IEEE CS, 333–338.

Mohamed Sellami, Sami Yangui, Mohamed Mohamed, and Samir Tata. 2013. PaaS-Independent Provisioning and Management of Applications in the Cloud. In *CLOUD 2013*. IEEE CS, 693–700.

Jin Shao, Hao Wei, Qianxiang Wang, and Hong Mei. 2010. A Runtime Model Based Monitoring Approach for Cloud. In *CLOUD 2010*. IEEE CS, 313–320.

Arnor Solberg et al. 2014. *D3.6—Evaluation Plan*. MODAClouds project deliverable.

Rini Van Solingen and Egon Berghout. 1999. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill.

Hui Song, Amal Elgammal, Vivek Nallur, Franck Chauvel, Franck Fleurey, and Siobhán Clarke. 2015. On Architectural Diversity of Dynamic Adaptive Systems. In *ICSE 2015*. IEEE Computer Society, 595–598.

SSAI Expert Group. 2010. *The Future of Cloud Computing*. Technical Report. http://cordis.europa.eu/fp7/ict/ssai/docs/cloud-report-final.pdf

SSAI Expert Group. 2012. *A Roadmap for Advanced Cloud Technologies under H2020*. Technical Report. http://cordis.europa.eu/fp7/ict/ssai/docs/cloud-expert-group/roadmap-dec2012-vfinal.pdf