Managing multi-cloud systems with CloudMF

Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, Arnor Solberg
Department of Networked Systems and Services, SINTEF, Oslo, Norway
{name.surname}@sintef.no

ABSTRACT

Dynamically adaptive systems (DAS) enable the continuous design and adaptation of complex software systems, but their main focus is limited to the application itself rather than the underlying platform and infrastructure. Cloud computing, in contrast, enables the management of the complete software stack, but it lacks integration with software engineering approaches, techniques, and methods from DAS. Model-based approaches have been successfully adopted for modelling DAS at design-time and facilitate their adaptation at run-time. Therefore, a natural next step is to adopt model-based approaches to enable cloud-based DAS. In this paper, we present the Cloud Modelling Framework (CLOUDMF), a model-based framework that addresses this issue. It consists of (*i*) a tool-supported domain-specific modelling language to model the provisioning and deployment of multi-cloud systems, and (*ii*) a models@run-time environment for enacting the provisioning, deployment and adaptation of these systems.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Configuration Management; H.1.0 [**Information Systems**]: Models and Principles

Keywords

Cloud computing, provisioning, deployment, dynamic adaptation, model-driven engineering, domain-specific modelling language, models@run-time

1. INTRODUCTION

Nowadays, software systems are leveraging upon an aggregation of dedicated solutions, which leads to the design of large scale, distributed, dynamic systems. However, the complexity of managing such systems challenges current software engineering approaches.

Dynamically adaptive systems (DAS) have recently emerged to cope with this challenge by enabling the continuous design and adaptation of complex software systems. DAS facilitates handling short-term changes in the execution environment as well as long-term changes in the system requirements [17]. However, the focus of DAS is typically limited to the application itself rather than the underlying platform and infrastructure.

NordiCloud 2013, 2-3 September, Oslo, Norway.

Cloud computing provides an ubiquitous networked access to a shared and virtualised pool of computing capabilities (*e.g.*, network, storage, processing, and memory) that can be provisioned with minimal management effort [16]. In contrast to DAS, cloud computing enables the management of the complete software stack, *i.e.* infrastructure, platform and application, where each layer is exposed as a service. In particular, cloud computing offers two types of scalability: horizontal (*e.g.*, add or remove new virtual machines) and vertical (*e.g.*, increase or decrease resources allocated to a virtual machine).

A key challenge is then to enable cloud-based DAS, *i.e.* to integrate the management capabilities of recent cloud solutions with software engineering approaches, techniques, and methods of DAS.

Our proposed solution is a model-based framework called Cloud Modelling Framework (CLOUDMF) [8, 13], which leverages upon models@run-time and combines it with recent cloud solutions. It consists of *(i)* the Cloud Modelling Language (CLOUDML), a tool-supported domain-specific modelling language (DSML) to model the provisioning and deployment of multi-cloud systems [13] and *(ii)* a models@run-time environment for enacting the provisioning, deployment and adaptation of these systems. The run-time environment provides a model-based representation of the underlying running system, which facilitates reasoning, simulation, and enactment of adaptation actions.

The remainder of the paper is organised as follows. In Section 2, we outline SENSAPP, an application that is adopted as a use case throughout the paper. In Section 3, we present CLOUDMF by first introducing its architecture and then detailing the *modelling-* as well as the *models@run-time* environments. Section 4, compares the proposed approach with the state-of-the-art. Finally, Section 5 draws some conclusions and offers some ideas for future work.

2. MOTIVATING EXAMPLE: SENSAPP

SENSAPP[1] is an open-source, service-oriented application for storing and exploiting large data sets collected from sensors and devices. It is designed to seamlessly bridge the gap between the Internet of Things (IoT) and the cloud [18]. The SENSAPP application can register sensors, store their data, and notify clients when new data are pushed. The third-party application SENSAPP ADMIN uses the public REST API of SENSAPP to manage sensors and visualise data using a graphical user interface.

[1] http://sensapp.org

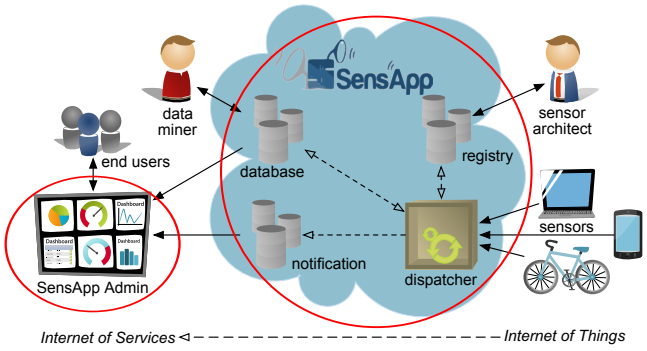

Figure 1: The SENSAPP architecture [18]

SENSAPP provides four essential services to support the definition of IoT applications (see Figure 1). The *registry* service stores metadata about the sensors (*e.g.*, description and creation date). The *database* service stores raw data from the sensors using a MongoDB database. The *notification* service 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* service orchestrates the other services: it receives data from the sensors, stores these data in the database according to the metadata from the registry, and finally triggers the notification mechanisms for the new data. In order to be deployed, SENSAPP requires a database and a servlet container, while SENSAPP ADMIN requires a Web server.

In this paper, we adopt a motivating example illustrating different scenarios of provisioning and deployment of SENSAPP. First, both SENSAPP and SENSAPP ADMIN are deployed on the same virtual machine. Then, this topology is adapted so that SENSAPP is redeployed on another virtual machine. These scenarios motivate for the following requirements for CLOUDMF:

Separation of concerns (R_1): CLOUDMF should support a modular, loosely-coupled specification of the provisioning and deployment so that the modules can be seamlessly substituted. This will facilitate the dynamic adaptation of the provisioning and deployment topology.

Cloud provider-independence (R_2): CLOUDMF should support a cloud provider-agnostic specification of the provisioning and deployment. This will simplify the design of multi-cloud systems and prevent vendor lock-in.

Reusability (R_3): CLOUDMF should support the specification of reusable types or reusable patterns composing the system. This will ease the evolution as well as the rapid development of different variants of a system in time and in space (*i.e.*, a product line).

Abstraction (R_4): CLOUDMF should provide an up-to-date, abstract representation of the running system. This will facilitate reasoning, simulation, and validation of adaptation actions before their actual enactments.

In the following section, we present CLOUDMF and how it addresses these requirements.

3. CLOUDMF

CLOUDMF includes a set of tools that aims at facilitating the provisioning, deployment, and adaptation of multi-cloud systems by leveraging upon model-driven engineering (MDE) techniques and methods.

MDE is a branch of software engineering that aims at improving the productivity, quality and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. Models and modelling languages, as the main artefacts of the development process, enable developers to work at a high level of abstraction by focusing on cloud concerns rather than implementation details. Model transformation, as the primary technique to generate (parts of) software systems, restrains developers from repetitive and error-prone tasks such as coding. This approach, which is commonly summarised as "model once, generate anywhere", is particularly relevant when it comes to provisioning and deployment of applications across multiple clouds, as well as migrating them from one cloud to another.

The model-driven approach adopted in CLOUDMF allows developers to model the provisioning and deployment of a multi-cloud system at various levels of abstraction. The two proposed levels are: *(i)* the Cloud Provider-Independent Model (CPIM), which can be regarded as a provisioning and deployment template: it describes the provisioning and deployment topology of an application in a cloud-agnostic way; *(ii)* the Cloud Provider-Specific Model (CPSM), which can be regarded as a provisioning and deployment model: it refines the CPIM and describes provisioning and deployment topology in a cloud-specific way. This two-levels approach is agnostic to any development paradigm and technology, meaning that the developers can design and implement their applications based on their preferred paradigms and technologies.

The architecture of CLOUDMF consists of two main components: the modelling environment and the models@run-time environment (see Figure 2).

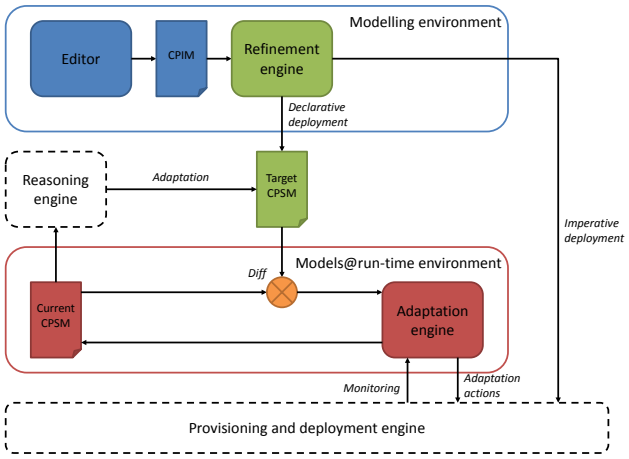

Figure 2: Architecture of CLOUDMF

The modelling environment is used to specify the provisioning and deployment of multi-cloud systems with CLOUDML. In particular, the *editor* is used to specify one or more CPIMs (*e.g.*, one where SENSAPP and SENSAPP ADMIN are deployed on a same virtual machine and another where they are deployed on two different vir-

tual machines). These CPIMs are then provided as inputs to the *refinement engine*, which supports the user in the selection of the right CPIM related to the actual context and needs (*e.g.*, select the template where SENSAPP and SENSAPP ADMIN are deployed on different virtual machines). The selected CPIM is semi-automatically transformed into CPSMs (*e.g.*, Amazon EC2 is selected as provider and two *t1.micro* instances running Ubuntu are selected as virtual machines).

The modelling environment can either enact the initial deployment of the application imperatively by interacting with the provisioning and deployment engine directly, or declaratively by providing the CPSM to the models@run-time environment, which in turn interacts with the provisioning and deployment engine. In the latter case, the CPSM is causally connected to the running system; *i.e.*, a change in the CPSM is reflected on-demand in the running system, whereas a change in the running system is automatically reflected in the CPSM. In this way, the models@run-time environment provides a model-based interface to any reasoning systems and modelling tools for future adaptation.

3.1 CloudML modelling environment

In this section, we detail the two main components of the modelling environment.

3.1.1 Editor

The editor uses CLOUDML to specify CPIMs. CLOUDML is defined by a metamodel inspired by component-based approaches and implements the *type-instance* pattern [6]. Currently, the editor supports two formats for the concrete syntax, namely the JavaScript Object Notation (JSON) and the XML Metadata Interchange (XMI).

Type level. Figure 3 presents the portion of the DSML metamodel that covers the types of a CPIM. A CPIM encompasses the topology of the *nodes* of the cloud infrastructure, the topology of the software *artefacts* to be deployed on these nodes, and the possible *bindings* between software artefacts.

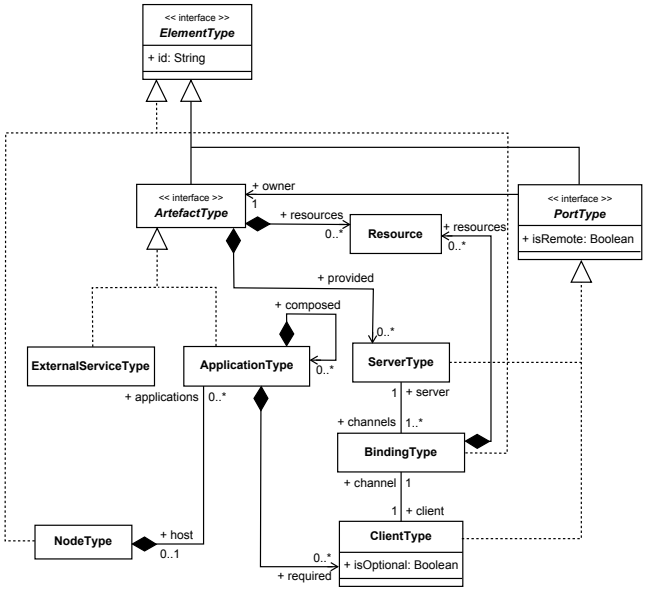

Figure 3: Type part of the DSML metamodel

Listings 1 - 3 present some excerpts of the types in JSON syntax specifying the provisioning and deployment of SENSAPP as described in our SENSAPP use case.

A *node type* represents a generic virtual machine (*e.g.*, a virtual machine running GNU/Linux, see Listing 1). This element can be parameterised by provisioning requirements (*e.g.*, 2 cores $\leq$ compute $\leq$ 4 cores, 2 GiB $\leq$ memory $\leq$ 4 GiB, storage $\geq$ 10 GiB, location = Europe). All these requirements are optional and do not have to be defined in the CPIM.

Listing 1: An example of a node type from a CPIM in JSON

```
"nodeTypes" : [
  {
    "id" : "SmallGNULinux",
    "os" : "GNULinux",
    "compute" : [2, 4],
    "memory" : [2048, 4096],
    "storage" : [10240],
    "location" : "eu",
    "sshKey" : "smallGNULinux",
    "securityGroup" : "SensApp",
    "groupName" : "smallGNULinux",
    "privateKey" : "YOUR KEY",
    "provides" : [
      { "id" : "SSH" }
    ]
  }
]
```

An *artefact type* represents a generic component of the system. In particular, an *application type* represents a component to be deployed on a node (*e.g.*, a SENSAPP servlet, a Jetty container, or a MongoDB database, see Listing 2), while an *external service type* represents a component managed by an external party. An artefact type can be associated to *resources* specifying how to manage the deployment's life-cycle of the associated artefact (*e.g.*, retrieve the servlet from `http://cloudml.org/`, configure it, and run it). Application types may be grouped together and reused in the form of composites.

A *port type* represents a port of an artefact. In particular, a *server port* serves features provided by the artefact (*e.g.*, the Jetty server is a servlet container), while a *client port* requests features required by the artefact (*e.g.*, the SENSAPP servlet requires a servlet container). A client port is *optional* if the required feature is not mandatory to run the artefact (*e.g.*, the required features of a Jetty container and a MongoDB database for the SENSAPP servlet are mandatory). The visibility of a port can be *remote* or *local*. A local port has to be bound to another port from an artefact on the same node. A port type can be associated to *resources* specifying how to configure the artefact accordingly (*e.g.*, configure the Jetty container to listen on port 8080).

Listing 2: Examples of artefacts types from a CPIM in JSON

```
"artefactTypes" : [
  {
    "id" : "Jetty",
    "retrieval" : "wget http://cloudml.org/services/jetty
        .sh",
    "deployment" : "sudo jetty.sh",
    "provides" : [
      { "id" : "JettyCapability" }
    ]
  },
```

```
  {
    "id" : "SensApp",
    "retrieval" : "wget -P ~ http://cloudml.org/SensApp.
        war; wget -P ~ http://cloudml.org/startsensapp.
        sh ; wget -P ~ http://cloudml.org/deploysensapp.
        sh",
    "deployment" : "deploysensapp.sh",
    "start" : "startsensapp.sh",
    "requires" : [
      { "id" : "JettyCapability", "isOptional" : false },
      { "id" : "MongoDBCapability", "isOptional" : false
          }
    ],
    "inputs" : [
      {
        "id" : "RESTChannel",
        "portNumber" : "8080",
        "isRemote" : true
      }
    ],
    "provides" : [
      {
        "id" : "RESTServer",
        "portNumber" : "8080"
      }
    ]
  }
]
```

A *binding type* represents a binding between two *port types* with the same visibility. In particular, a *deployment dependency* represents a binding involving a mandatory client port (*e.g.*, the Jetty container and the MongoDB database have to be deployed before the SENSAPP servlet), while a *communication channel* represents the other bindings (*e.g.*, the SENSAPP servlet communicates with SENSAPP ADMIN servlet through Hypertext Transfer Protocol (HTTP) on port 8080 as depicted in Listing 3). A binding type can be associated to *resources* specifying how to configure the artefact types in order to communicate with each other.

Listing 3: An example of a binding type from a CPIM in JSON

```
"bindingTypes" : [
  {
    "id" : "RESTBinding",
    "client" : "RESTClient",
    "server" : "RESTServer",
    "clientResource" : {
      "id" : "client",
      "retrieval" : "wget -P ~ http://cloudml.org/
          configuresensappadmin.sh",
      "configuration" : "cd ~; sudo bash
          configuresensappadmin.sh"
    }
  }
]
```

Instance level. The portion of the DSML metamodel that covers the instances of the CPIM is akin to the one that covers the types. Hence, it encompasses three main concepts, namely *node instances*, *artefact instances*, and *bindings instances*. Please note that these instances will constitute the provisioning and deployment template.

Listings 4 and 5 present some excerpts of instances of the types described above in JSON syntax.

A node instance represents an instance of a virtual machine (*e.g.*, a virtual machine running GNU/Linux called `smallGNULinux1`).

Listing 4: An example of a node instance from a CPIM in JSON

```
"nodeInstances" : [
  {
    "id" : "smallGNULinux1",
    "type" : "SmallGNULinux",
    "provides" : [
      {
        "id" : "ssh1",
        "type" : "SSH"
      }
    ]
  }
]
```

An artefact instance represents an instance of a component of the application on a specific virtual machine (*e.g.*, an instance of the Jetty container and of the SENSAPP server deployed on the virtual machine above).

Listing 5: Examples of artefact instances from a CPIM in JSON

```
"artefactInstances" : [
  {
    "id" : "jetty1",
    "type" : "Jetty",
    "destination" : "smallGNULinux1",
    "provides" : [
      {
        "id" : "jettyCapability1",
        "type" : "JettyCapability"
      }
    ]
  },
  {
    "id" : "sensApp1",
    "type" : "SensApp",
    "destination" : "smallGNULinux1",
    "requires" : [
      { "id" : "jettyCapability1" },
      { "id" : "mongoDBCapability1" }
    ],
    "inputs" : [
      {
        "id" : "restChannel1",
        "type" : "RESTChannel"
      }
    ],
    "provides" : [
      {
        "id" : "restServer1",
        "type" : "RESTServer"
      }
    ]
  }
]
```

The CPIMs specified with the editor are provided as input to the refinement engine.

3.1.2 Refinement engine

The aim of the refinement engine is to produce from CPIMs a CPSM to be consumed by the run-time platform (see Figure 4). The inputs of this component are three: *(i)* CPIMs, *(ii)* deployment resources (*e.g.*, scripts, binaries, source code) and *(iii)* constraints and metadata from the cloud application vendor through a deployment wizard.

The core element of the refinement engine is an in-memory CPIM. A CPIM can be loaded through the codecs module. This module is responsible for serializing and unserializing a deployment model into or from an in-memory model.

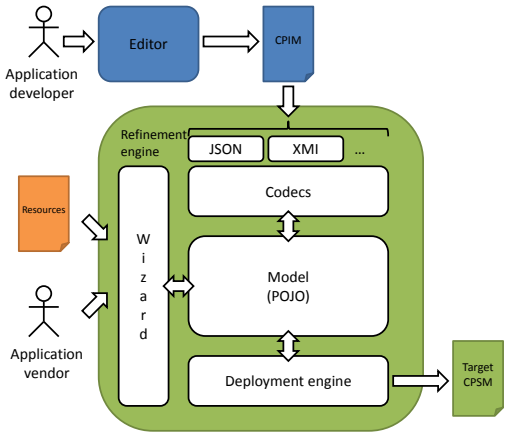

Figure 4: Overview of the architecture of the modelling environment

The deployment wizard aims at driving the user into the selection of the CPIM to be loaded. Thanks to this wizard, the cloud application vendor can also extend or tune the deployment model with constraints or metadata related to her business. Once this process is completed, the refinement engine will interact with the provisioning and deployment engine in order to transform semi-automatically the CPIM into a CPSM.

In order to transform a CPIM into a CPSM, the refinement engine interacts with the provisioning and deployment engine, which in turn interacts with the underlying provider. This way, the CPIM is enriched by provider-specific concepts.

Figure 5 presents a sequence diagram of this process applied in the context of our SENSAPP use case.

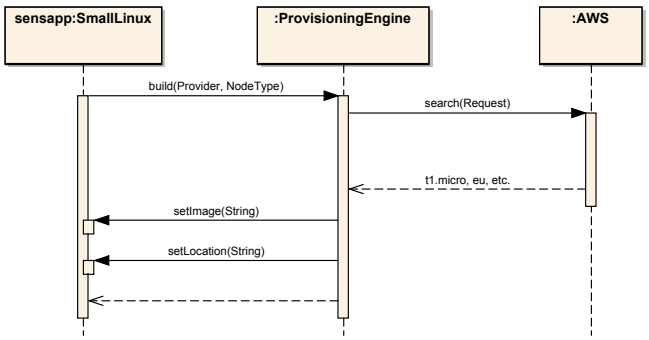

Figure 5: Example of CPSM derivation

The first step consists in the specification of the provider on which the CPIM instances will be deployed (*e.g.*, the virtual machine running GNU/Linux called smallGNULinux1 node will be provisioned on Amazon EC2). A request is then sent to the provisioning and deployment engine for details on virtual machines available from this provider according to the constraints defining its type (see Listing 4). As a result, this engine will interact with the underlying provider in order to retrieve this information before updating the metadata associated to the instance (*e.g.*, *t1.micro* instance in *eu-west-1* location).

Once completed, the initial provisioning and deployment process can be triggered either by interacting directly with the provisioning and deployment engine or through the models@run-time environment.

3.2 Models@run-time environment

Models@run-time [17, 7] is an architectural pattern for dynamic adaptive systems that proposes to leverage models during the execution of the system. In particular, models@run-time provides an abstract representation of the underlying running system, which facilitates reasoning, simulation, and enactment of adaptation actions. A change in the running system is automatically reflected in the model of the current system. Similarly, a modification to this model is enacted on the running system on demand. A classical architecture to achieve this is depicted in Figure 2 whose models@run-time part is inspired from [14]. As mentioned, the current model of the system is provided by the models@run-time environment. The current model can then be consumed by a reasoning system that produces a target model. The target model may undergo a validation process before enacting the adaptation. If passed, the current model and the target model of the system are compared. This way, it is possible to identify the parts of the system that require adaptation. The adaptation is then enacted by the adaptation engine and the target model becomes the current model.

This approach enables the continuous evolution of the system with no strict boundaries between design-time and run-time activities. Thanks to the use of models, it provides a well-defined interface to monitor and adapt the system.

In order to use models@run-time, the following entities are required:

- A metamodel that defines the concepts to specify valid models of the system.
- A model of the system that conforms to the metamodel and that is processed by the models@run-time environment.
- Adequate sensors to detect changes in the running system in order to reflect them in the current model.
- Adequate actuators to enact the modification to the current model into the running system.
- A synchronisation engine that uses sensors and actuators to causally connect the current model with the running system.

Within CLOUDMF, the models@run-time environment provides a CPSM causally connected to the running system. On the one hand, any modification to the CPIM will be reflected in the CPSM and, in turn, automatically propagated onto the running system. On the other hand, any change in the running system will be reflected in the CPSM, which, in turn, can be compared to the CPIM. The current CPSM of the system can be manipulated in both imperative and declarative ways, *i.e.*, it can be modified through a set of instructions, or a new CPSM can be provided to replace the existing one. In both cases the adaptation can result in: *(i)* modification of the deployment and provisioning topology or *(ii)* modifications of the status of the artefacts composing the system (see Figures 6 and 7).

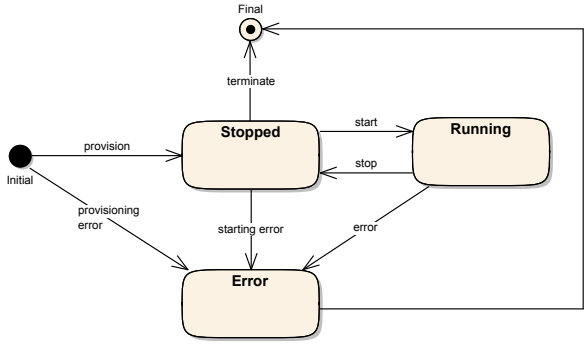

Figure 6: Life-cycle of a node

Changing the status of an artefact encompasses adapting all its dependencies accordingly. The resources that can be associated to an artefact type can be annotated with commands describing how to move from one state to another.

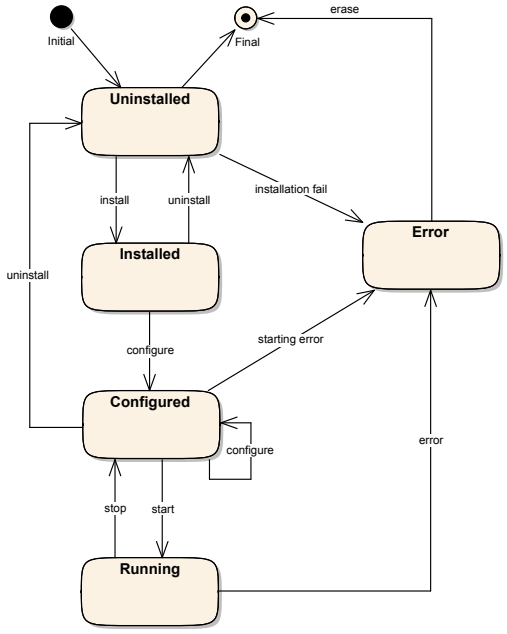

Figure 7: Life-cycle of an application

In our SENSAPP use case, the adaptation will result in the following. First, the SENSAPP artefact running on the same node of the SENSAPP ADMIN artefact is stopped. Then, a new node is provisioned and the SENSAPP artefact is deployed on it with all its dependencies.

In order to deploy SENSAPP, a first command is triggered in order to retrieve the deployable artefact (*e.g.*, the SENSAPP servlet). Once completed, this artefact falls in the *uninstalled* state. Then commands are called to: (1) install the artefact (*e.g.*, deploy the servlet in the Jetty container), (2) configure it (*e.g.*, configure the Jetty for the web-app) and (3) start it (*e.g.*, restart Jetty). During this process, the CPSM is enriched with run-time information (see Figure 8).

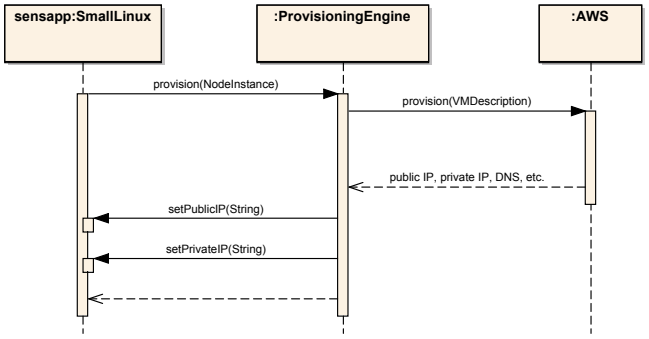

Figure 8: CPSM run-time enrichment

3.3 Synthesis

As illustrated through our SENSAPP use case, CLOUDMF can be used to provision, deploy, and adapt multi-cloud systems. The following list summarises how it fulfills the requirements presented in Section 2.

Separation of concerns (R_1): The component-based design of the CLOUDML metamodel ensures that the provisioning and deployment templates and models are modular and loosely-coupled.

Cloud provider-independence (R_2): The layering of the modelling stack into CPIMs and CPSMs ensures that the provisioning and deployment templates are cloud provider-independent.

Reusability (R_3): The type-instance pattern in the CLOUDML metamodel ensures that the types can be reused within several models.

Abstraction (R_4): The models@run-time environment provides an abstract and up-to-date representation of the running system which can be dynamically manipulated.

3.4 Reference implementation

CLOUDMF is available as an open-source project[2]. It is implemented with Java and Scala as programming languages and Maven as a build tool. The current codebase consists of around 5 000 lines of Java code and 1 000 lines of Scala code. The CLOUDML models and metamodels are represented as plain Java objects. These models can be serialised in either JSON or XMI. The JSON and XMI codecs are based on Kotlin [3] and the Kevoree Modeling Framework (KMF) [4] [15], respectively. The current provisioning and deployment engine is jclouds [5], but other engines such as Cloudify [6] are under consideration. CLOUDMF has been used with several use cases from EU projects such as REMICS [4], MODAClouds [1, 5] and PaaSage [3]. These use cases range from IoT applications to enterprise systems.

[2] https://github.com/SINTEF-9012/cloudml
[3] http://kotlin.jetbrains.org/
[4] https://github.com/dukeboard/kevoree-modeling-framework
[5] http://jclouds.incubator.apache.org/
[6] http://www.cloudifysource.org/

4. RELATED WORK

Deployment and monitoring of distributed systems have been extensively researched over the past decades by various communities including server management systems, distributed systems, and cloud-based systems. To the best of our knowledge, there is not yet any approach combining the strength of recent cloud solutions with the flexibility of the models@run-time architecture.

Server management solutions, such as IBM Tivoli [11], BCFG2 [12], or CFEngine 3 [10] initially tackled the issue of server (and network) configuration by providing consistent, reproducible and verifiable descriptions of servers configuration. However, by contrast with our approach, these solutions are not tailored to the cloud environment, and do not leverage infrastructure as a service facilities for instance.

In the cloud community, several libraries such as jclouds[7], Simple Cloud[8], or DeltaCloud[9] recently emerged to help in reducing cost and effort related to deployment and maintenance of cloud-based systems. While such libraries effectively foster their deployment and maintenance, they remain code-level tools, on which making redesign decisions is difficult and error-prone. Similarly, research projects such as mOSAIC [2, 20], which tackles the vendor lock-in problem by providing an API for provisioning and deployment, are also limited to the code level.

At a higher level of abstraction, more advanced frameworks such as Cloudify [10], Puppet [11] or Chef [12] provide capabilities for the automatic provisioning, deployment, monitoring, and adaptation of cloud systems without being language-dependent. Such solutions provide DSML to capture and enact cloud-based system management. However, by constrast with our approach, the resulting models are not causally connected to the running system, and may become irrelevant as manual maintenance operations are carried out. The approach proposed in the CloudScale [9] and Reservoir [19] projects suffer similar limitations.

In the models@run-time [7] community, several frameworks already provide causal connections between a running system and its representation as a model. Kevoree [14] provides a first complete models@run-time platform to manage distributed Java applications, but does not leverage infrastructure as a service to operate on cloud-based systems. By contrast, the work of Shao *et al* [21] was the first attempt to build a models@run-time platform for the cloud, but remained restricted to monitoring, without providing support for configuration enactment. To the best of our knowledge, CLOUDMF is thus the first attempt to reconcile cloud solutions with modelling practices through the use of models@run-time.

[7] `http://www.jclouds.org`
[8] `http://simplecloud.org/`
[9] `http://deltacloud.apache.org/`
[10] `http://www.cloudifysource.org/`
[11] `https://puppetlabs.com/`
[12] `http://www.opscode.com/chef/`

5. CONCLUSIONS AND PERSPECTIVES

In this paper, we presented CLOUDMF to enable cloud-based DAS. The framework is built upon MDE techniques and methods to manage multi-cloud systems. The DSML facilitates the specification of provisioning and deployment concerns of multi-cloud systems at design-time while the models@run-time environment provides a model-based abstract representation of the running system that facilitates reasoning, simulation, and enactment of adaptation actions.

Until now, our focus has been on managing platforms and infrastructures. In a future work, we will consider concerns such as cost or location of data. Moreover, we will introduce the adaptation features offered by cloud providers (*e.g.*, load balancing, auto-scaling) as concepts in CLOUDML. Furthermore, we will improve the synchronisation engine between the CPSM and the running system to tackle the scenario where if the enactment of an adaptation is too time-consuming, a change in the environment may require another adaptation while the system is still being adapted.

6. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number: 318484 (MODAClouds), 317715 (PaaSage) and 257793 (REMICS).

7. REFERENCES

[1] MODAClouds EU project.
[2] mOSAIC EU project.
[3] PaaSage EU project.
[4] REMICS EU project.
[5] D. Ardagna, E. Di Nitto, G. Casale, D. Pectu, P. Mohagheghi, S. Mosser, P. Matthews, A. Gericke, C. Balligny, F. D'Andria, C.-S. Nechifor, and C. Sheridan. MODACLOUDS, A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds. In *ICSE MiSE: International Workshop on Modelling in Software Engineering*, pages 50–56. IEEE/ACM, 2012.
[6] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12(4):290–321, 2002.
[7] G. Blair, N. Bencomo, and R. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.
[8] E. Brandtzæg, M. Parastoo, and S. Mosser. Towards a Domain-Specific Language to Deploy Applications in the Clouds. In *CLOUD COMPUTING 2012: 3rd International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 213–218. IARIA, 2012.
[9] G. Brataas, E. Stav, S. Lehrig, S. Becker, G. Kopčak, and D. Huljenic. CloudScale: scalability management for cloud systems. In *ICPE 2013: 4th ACM/SPEC International Conference on Performance Engineering*, pages 335–338. ACM, 2013.
[10] M. Burgess and R. Ralston. Distributed Resource Administration Using Cfengine. *Softw., Pract. Exper.*, 27(9):1083–1101, 1997.
[11] T. Delaet, W. Joosen, and B. Vanbrabant. A survey of system configuration tools. In *LISA 2010: 24th international conference on Large installation system administration*, pages 1–8. USENIX Association, 2010.
[12] N. Desai, R. Bradshaw, S. Matott, S. Bittner, S. Coghlan, R. Evard, C. Lueninghoener, T. Leggett, J.-P. Navarro,

G. Rackow, C. Stacey, and T. Stacey. A Case Study in Configuration Management Tool Deployment. In *LISA 2005: 19th Conference on Systems Administration*, pages 39–46. USENIX, 2005.

[13] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *CLOUD 2013: IEEE 6th International Conference on Cloud Computing*, pages 887–894. IEEE Computer Society, 2013.

[14] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, and J.-M. Jézéquel. Dissemination of Reconfiguration Policies on Mesh Networks. In K. M. Göschka and S. Haridi, editors, *DAIS 2012: 12th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, volume 7272 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2012.

[15] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *MODELS 2012: 15th International Conference on Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2012.

[16] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology, September 2001.

[17] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@Run.time to Support Dynamic Adaptation. *IEEE Computer*, 42(10):44–51, 2009.

[18] S. Mosser, F. Fleurey, B. Morin, F. Chauvel, A. Solberg, and I. Goutier. SENSAPP as a Reference Platform to Support Cloud Experiments: From the Internet of Things to the Internet of Services. In *SYNASC 2012: 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 400–406. IEEE Computer Society, 2012.

[19] 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. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):535–545, July 2009.

[20] C. Sandru, D. Pectu, and V. I. Munteanu. Building an Open-Source Platform-as-a-Service with Intelligent Management of Multiple Cloud Resources. In *UCC 2012: IEEE 5th International Conference on Utility and Cloud Computing*, pages 333–338. IEEE Computer Society, 2012.

[21] J. Shao, H. Wei, Q. Wang, and H. Mei. A Runtime Model Based Monitoring Approach for Cloud. In *CLOUD 2010: IEEE 3rd International Conference on Cloud Computing*, pages 313–320. IEEE Computer Society, 2010.