A Model-based Architecture for Autonomic and Heterogeneous Cloud Systems

Hugo Bruneliere[1], Zakarea Al-Shara[2,*], Frederico Alvares[3], Jonathan Lejeune[4] and Thomas Ledoux[5]
[1] *NaoMod Team (LS2N-CNRS), IMT Atlantique, Nantes, France*
[2] *Berger-Levrault, Montpellier, France*
[3] *EasyVirt, Nantes, France*
[4] *REGAL Team (Inria & LIP6-CNRS), Sorbonne Universities - UPMC, Paris, France*
[5] *STACK Team (Inria & LS2N-CNRS), IMT Atlantique, Nantes, France*

Keywords: Cloud Computing, Autonomic Computing, Modeling, Heterogeneity, Interoperability, Constraints.

Abstract: Over the last few years, Autonomic Computing has been a key enabler for Cloud system's dynamic adaptation. However, autonomously managing complex systems (such as in the Cloud context) is not trivial and may quickly become fastidious and error-prone. We advocate that Cloud artifacts, regardless of the layer carrying them, share many common characteristics. Thus, this makes it possible to specify, (re)configure and monitor them in an homogeneous way. To this end, we propose a generic model-based architecture for allowing the autonomic management of any Cloud system. From a "XaaS" model describing a given Cloud system, possibly over multiple layers of the Cloud stack, Cloud administrators can derive an autonomic manager for this system. This paper introduces the designed model-based architecture, and notably its core generic XaaS modeling language. It also describes the integration with a constraint solver to be used by the autonomic manager, as well as the interoperability with a Cloud standard (TOSCA). It presents an implementation (with its application on a multi-layer Cloud system) and compares the proposed approach with other existing solutions.

1 INTRODUCTION

Cloud Computing is becoming widely considered by companies when building their systems. The number of applications developed/deployed for/in the Cloud is constantly increasing, even where software was traditionally not seen as central (*e.g.*, cf. the quite recent trend on Cloud Manufacturing (Xu, 2012)). The Cloud market also provides many varied services, platforms and infrastructures for customers to support such Cloud-based systems (Narasimhan and Nichols, 2011). Thus, it becomes more complex for Cloud providers and users to efficiently design, (re)configure and monitor their solutions.

There are already several initiatives intending to provide a more homogeneous Cloud management support. OASIS TOSCA (OASIS, 2017) is a promising Cloud standard used in practice by IBM, Cloudify or HP (for instance). It focuses on allowing an interoperable representation of Cloud applications and services, as well as their underlying infrastructures. There is also CloudML (Ferry et al., 2014) that has been developed, refined and used in different European projects notably (cf. ARTIST (Menychtas et al., 2014), MODAClouds (Ardagna et al., 2012) or PaaSage (Rossini, 2015)). Its objective is to allow modeling the provisioning, deployment, monitoring and migration of Cloud systems. From a technology perspective, some open source environments are gaining momentum. For example, tools from the OpenStack (OpenStack Foundation, 2017) ecosystem aim at providing more integrated compute, storage and networking resources via commonly shared services.

However, these solutions are still facing some challenges. Firstly, the Cloud heterogeneity makes it difficult for these approaches to be applied systematically in all possible contexts. Indeed, Cloud systems may involve many resources having various and varied natures (software and/or physical). Solutions

[*] Zakarea Al-Shara had a postdoc position in the STACK team, funded by Atlanstic2020 (CoMe4ACloud project).

Bruneliere, H., Al-Shara, Z., Alvares, F., Lejeune, J. and Ledoux, T.
A Model-based Architecture for Autonomic and Heterogeneous Cloud Systems.
DOI: 10.5220/0006773002010212
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 201-212
ISBN: 978-989-758-295-0

to support in a similar way resources coming from all the Cloud layers (*e.g.*, IaaS, PaaS, SaaS) are thus required. Secondly, Cloud systems are highly dynamic: clients can book/release “elastic” virtual resources at any moment, according to given Service Level Agreement (SLA) contracts. Solutions need to reflect and support transparently this dynamicity/elasticity, which is not trivial for systems involving many different services. Thirdly, Cloud systems are becoming so complex that they cannot be handled manually and efficiently. This concerns their configuration and monitoring, but also their runtime behavior to guarantee QoS levels and SLA contracts. This notably involves decision-making and re-configuration to translate taken decisions into actual actions on the systems. As a consequence, solutions coming with an automated support for dealing with these activities can provide interesting benefits (Krupitzer et al., 2015).

We propose in this paper a generic model-based architecture named CoMe4ACloud (Constraints and Model Engineering for Autonomic Clouds) [2]. The goal is to provide a generic and extensible solution for the autonomic management of Cloud services, independently from the Cloud layer(s) they belong to (*i.e.*, XaaS, cf. Section 2). An initial version of a supporting constraint model has already been proposed by (Lejeune et al., 2017). However, it does not come with a proper model-based architecture and a reusable modeling language directly applicable to all Cloud layers. Within this paper, we intend to address this issue via the following contributions:

1. A model-based architecture for XaaS modeling in order to support generic autonomic management;
   - Including the connection with a constraint solver (Choco (Jussien et al., 2008)) and a partial mapping to/from a Cloud standard (TOSCA) for interoperability with external solutions;
2. A related XaaS core modeling language, possibly supporting any of the possible Cloud layers;
3. An Eclipse-based tooling support, that has been applied on a realistic multi-layer Cloud system.

The paper is structured as follows. In Section 2, we introduce the general context and objectives of our approach. In Section 3, we illustrate and motivate further our work via a practical use case. Then, in Section 4, we present the overall model-based architecture we propose, including its relation with the Choco constraint solver and mapping to/from the TOSCA Cloud standard. In Section 5, we detail our XaaS generic modeling language as the core element of our architecture. In section 6, we describe the corresponding tooling support we built in Eclipse and how we applied it in the context of our use case. In Section 7, we give more insights on the current status of our approach. We discuss the related work in Section 8 before we conclude in Section 9.

2 BACKGROUND

Cloud services can be carried out by several different layers of the Cloud stack. However, independently from the layer(s) they rely on, they always share some common characteristics (cf. Figure 1).

All Cloud architectures inherently expose and use services hosted by resources within a multi-layer stack. Each one of these services can play the role of 1) consumer of other services in the Cloud stack and/or 2) provider to other services in the Cloud stack or eventually to end-users. Client applications can consume services provided by a given SaaS resource. This SaaS resource can consume services provided by a given IaaS resource, which in turn can consume some lower-level services offered by an energy provider. In all cases, the objectives are very similar: 1) Find an optimal balance between costs and revenues by minimizing the cost of purchased services and SLA violation penalties while maximizing revenues from provided services; 2) Comply with all SLA and layer(s) internal constraints by having a manager that can find optimal layer configurations based on these objectives.

We consider the generic notion of XaaS (Anything-as-a-Service or Everything-as-a-Service (Duan et al., 2015)) as a way to represent all possible resources and layers in a Cloud stack as well as the service-oriented relationships between them. Any XaaS resource can both provide and consume services to/from other resources/layers. It also comes with constraints expressed in joint SLA contracts.

Moreover, frequent on-demand provisioning makes Cloud environments susceptible to short-term variations, often preventing them to be managed

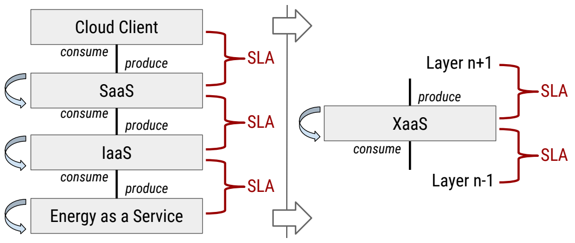


Figure 1: Specific Cloud layers vs. generic XaaS.

[2]This work has been funded by Atlanstic2020, cf. https://come4acloud.github.io for details.

manually. This is why Autonomic Computing (AC) (Kephart and Chess, 2003) is very popular in the Cloud community. AC provides architecture references and guidelines intended to design Autonomic Managers (AMs) that make Cloud systems self-manageable. The main objective is to free Cloud administrators of the burden of manually managing them. Interestingly for us, a generic AM can also be associated to the notion of XaaS in order to deal homogeneously with corresponding (re)configurations (*i.e.*, representations of real systems).

To realize this, it appears fundamental to be able to model in a generic way such multi-layer Cloud architectures. Thus, we first need to have a XaaS modeling language that allows describing all possible Cloud topologies as well as corresponding actual system configurations. Their core structure can be modeled as directed acyclic graphs (DAGs): the number of existing nodes/resources is always finite, edges/services have a stable orientation (consumer vs. producer), and the usual top-down structure of the Cloud stack prevents from having cycles. The language also has to support the proper modeling of some constraints on the defined topologies (*e.g.*, to reflect SLA contracts). Modeling Cloud systems, their service-oriented interactions and SLA contracts in an abstract way brings a significant advantage. Indeed, we can use such an abstract model within a generic decision-making framework to be part of our generic AM. In our case, we used the help of a constraint solver (see Section 4.1).

As mentioned in Section 1, there are already some existing Cloud modeling languages and approaches. However, their intended scopes are slightly different and their coverage may be limited regarding some aspects, notably concerning the modeling of the required constraints. Thus, we made the choice of designing our generic XaaS modeling architecture and language covering the expression of constraints, as needed for the associated AM. We compare our approach with the current related work in Section 8.

3 MOTIVATING EXAMPLE

To further motivate our architecture, and illustrate later its application, we need a realistic cross-layer Cloud system. The proposed use case encompasses two related E-Learning systems from two distinct academic institutions. The end-users are students, faculty members and/or administrative staff who actually consume the deployed E-learning services through a web browser and/or mobile devices (phones, tablets, etc.). Hence, the income load may drastically vary within a business day, according to each client (in-

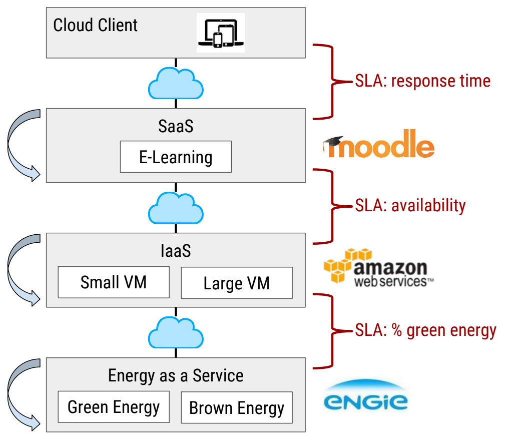


Figure 2: E-Learning Cloud system - Case study.

stitution) activity. Let us consider a practical session scheduled for a large group of students, possibly located in different campuses. If this session requires that students frequently interact with the E-learning application, the service will suffer some overload situations during the session. As a consequence, the application should be able to adapt itself by adjusting the required compute capacity. The objective is to keep the service response time at acceptable levels as defined in related SLA contracts. An overview of this use case is depicted in Figure 2.

Students, as the main end-users of the offered E-learning facilities, are the clients of a Software-as-a-Service (SaaS) provider. At the upper level, this SaaS provides a Cloud-based version of the Moodle learning management system [3], *i.e.*, an elastic Moodle. The SaaS provider is a Infrastructure-as-a-Service (IaaS) client, *i.e.*, it needs compute resources in order to run the E-learning services. Thus, at the lower level, a IaaS provides the required services by means of available VMs. The datacenters of the IaaS provider need electrical power in order to be able to operate. This power can be obtained: 1) via equipments deployed *in-situ* (*i.e.*, on the campus) allowing for local production of green energy (*e.g.*, solar panels, windmills) or 2) by electricity providers (brown energy) such as Engie [4] in the form of Energy-as-a-Service (EaaS). Hence, the IaaS should be capable of adapting itself in response to clients arrivals/departures or requests/releases of compute resources. Moreover, it should also adapt to the local energy production and/or to the fluctuating prices of energy applied by EaaS providers.

In all cases, the goal is to autonomically reconfigure the system in order to have the best balance between 1) costs, which are related to services consumed/bought from providers (*e.g.*, energy, compute

[3]https://moodle.org
[4]http://www.engie.com/en

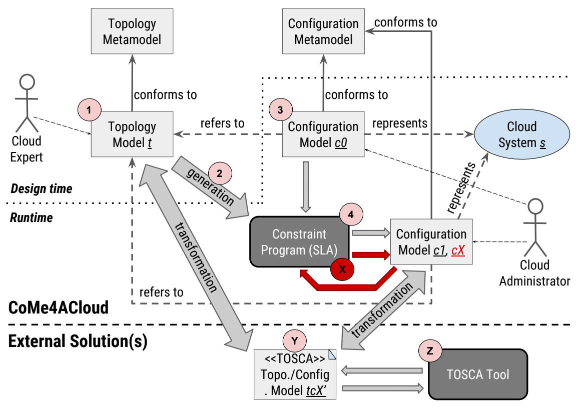


Figure 3: A model-based architecture.

resources) and 2) revenues, which refer to services offered/sold to clients (*e.g.*, E-learning service). A couple of XaaS model excerpts from our example (for the IaaS level) are shown in Section 5, complementary resources (e.g. models of the SaaS level) are also shown and available in Section 6.2.

4 PROPOSED ARCHITECTURE

The proposed architecture heavily relies on the joint use of complementary (meta)models as part of an iterative process (Atkinson and Kuhne, 2003). On the one hand, a Topology Metamodel is dedicated to the specification of the different topologies (*i.e.*, types) of Cloud systems. This can be realized generically for systems concerning any of the possible Cloud layers. On the other hand, a Configuration Metamodel is intended to the representation of actual configurations (*i.e.*, instances) of such systems. This is realized by referring to a corresponding (previously specified) topology. Once again, this metamodel is independent from any particular Cloud layer and topology/type of Cloud system. As shown in Figure 3, these two metamodels are the cornerstones of the proposed architecture and its core language (cf. Section 5).

In step (1) a topology model *t* is defined manually by a Cloud expert at design time. It specifies a particular topology of system to be modeled and then handled at runtime (*e.g.*, a given type of IaaS). This topology model notably includes the expression of the corresponding SLAs. In step (2) this topology model is used as the input of a code generator that translates it into a Constraint Program (CP), thus encoding these SLAs as constraints. The goal of this CP and related constraint solver is to automatically compute a new suitable system configuration from an original one according to the expressed constraints. In step (3) a first configuration model *c0* is initialized, either manually by a Cloud administrator or automatically by the CP. In step (4) the CP execution is performed according to the current system state, represented as configuration model *c0*, and to the related set of constraints/SLAs encoded in the CP. As a result, a new configuration model *c1* is produced. Re-configurations can then occur whenever required (step (X)), via the re-execution of the CP taking as input the current configuration model and producing as output a new updated one. Thus, the different configuration models (*e.g.*, *c0*, *c1* & *cX*) are representations of consecutive states of the modeled Cloud system *s* at given points in time (see more details in Section 7).

In parallel, the topology and configuration models can be transformed at any time into a partial equivalent TOSCA models (step (Y)). Any TOSCA supporting tool can then be used to handle such models (step (Z)). The available tooling support for this whole architecture (including the features detailed in the two next subsections) is presented in Section 6.1.

4.1 A Constraint Solver for an Automomic Loop

As introduced before, the proposed architecture integrates the use of a constraint solver (Choco (Jussien et al., 2008) in our case) in order to realize the decision-making of the autonomic loop allowing to automatically compute system re-configurations. A corresponding CP requires three different elements to find a solution (*i.e.*, in our case a new configuration): a fixed set of problem variables, a domain function (associating each variable to a domain) and a set of constraints. Our XaaS architecture and its two metamodels allow expressing these elements as required by the constraint model we currently rely on (provided to us by (Lejeune et al., 2017)).

From a Topology model, the implementation code is automatically generated for the various node and relationship types. This code also materializes the associated constraints (expressing the related SLAs) from this same model. For the sake of genericity, it actually calls only base pre-defined classes extended by the produced topology-specific classes. All this code is required to actually instantiate the constraint model, and for the solver to perform its analysis.

The instantiation is realized based on a given Configuration model. To this intent, a Configuration model is translated into a format that the generated CP can process. This is then taken as input by the CP that produces as output a new result providing an optimal solution according to the considered constraints. Finally, this result is translated back into a Configuration model [5].

[5]We plan to replace this intermediate representation di-

4.2 A TOSCA Mapping for Interoperability

For interoperability reasons, we have quite naturally chosen to rely on the TOSCA standard from OASIS (OASIS, 2017). However, the expressions/constraints we support in our architecture are not part of the TOSCA scope and so cannot be natively modeled with this standard. Because of that (cf. also Section 5.2 for complementary reasons), we decided to design our own XaaS language and to establish a direct (partial) mapping between our metamodels and TOSCA. Note that a extension of TOSCA could also be envisioned in the future to integrate of our constraint support into TOSCA.

The proposed mapping covers a large majority of the structural aspects of our Topology and Configuration metamodels. This way, we are able to automatically initiate corresponding models from existing TOSCA specifications. From an end-user perspective, this allows saving time and reusing parts of the already available data. In the opposite way, we are also able to export XaaS models as TOSCA specifications. It is then possible to rely on external relevant TOSCA-based solutions for dealing with various Cloud management activities (*e.g.*, system monitoring).

5 CORE MODELING LANGUAGE

As explained before, our proposed architecture relies on two metamodels forming a core XaaS modeling language. In what follows, we detail the abstract syntax of this language (*i.e.*, the Topology and Configuration metamodels). We also introduce the concrete syntax we propose to facilitate its usage by Cloud experts and system engineers. An implementation of this language, including the metamodels and full grammar, can be found from Section 6.1.

5.1 Abstract Syntax: Two Metamodels

As shown in Figure 4, the Topology metamodel covers 1) the general description of the structure of a given topology and 2) the constraint expressions that can be attached to the specified types of nodes and relationships. Starting by the structural aspects, each Cloud system's *Topology* is named and composed of a set of *NodeType*s and corresponding *Relationship-Type*s that specify how to interconnect them. It can also have some global constraints attached to it.

rectly by our Configuration model in the future.

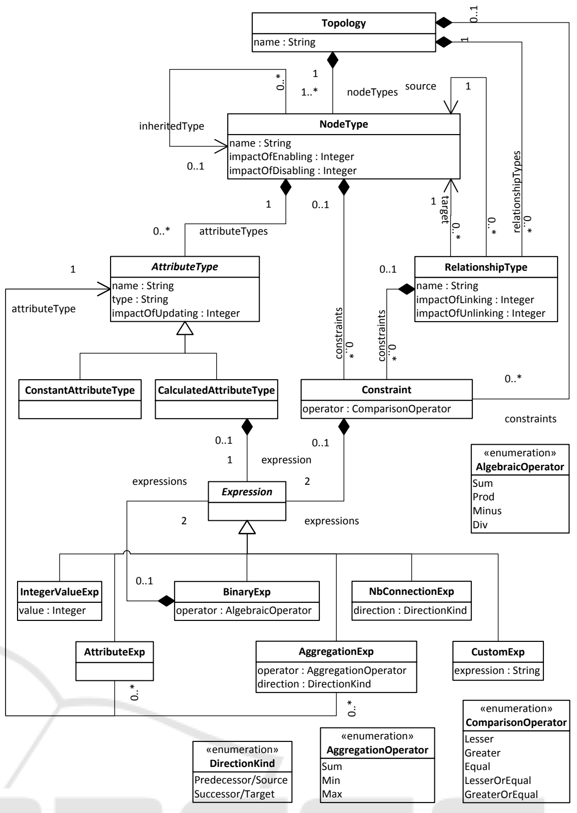


Figure 4: Topology metamodel - Design time.

Each *NodeType* has a name, a set of *Attribute-Type*s and can inherit from another *NodeType*. It can also have one or several specific *Constraint*s attached to it. Cloud experts can declare the impact (in terms of time, memory, price, etc.) of enabling/disabling nodes at runtime (*e.g.*, a given type of Physical Machine/PM node takes X seconds to be switched on/off).

Each *AttributeType* has a name and value type. It allows indicating the impact of updating related attribute values at runtime. A *ConstantAttributeType* stores a constant value at runtime, a *CalculatedAttributeType* allows setting an *Expression* automatically computing its value at runtime.

Any given *RelationshipType* has a name and defines a source and target *NodeType*. It also allows specifying the impact of linking/unlinking corresponding nodes via relationships at runtime (*e.g.*, migrating a given type of Virtual Machine/VM node from a type of PM node to another one can take several minutes). One or several specific *Constraint*s can be attached to a *RelationshipType*.

A *Constraint* relates two *Expressions* according to a predefined set of comparison operators. An *Expression* can be a single static *IntegerValueExpression* or an *AttributeExpression* pointing to an *AttributeType*.

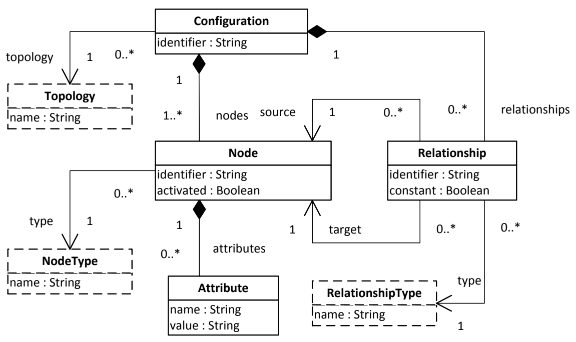


Figure 5: Configuration metamodel - Runtime.

It can be a *NbConnectionExpression* representing the number of *NodeType*s connected to a given *NodeType* or *RelationshipType* (at runtime) as *predecessor/successor* or *source/target* respectively. It can also be an *AggregationExpression* aggregating the values of an *AttributeType* from the predecessors/successors of a given *NodeType*, according to a predefined set of aggregation operators. It can be a *BinaryExpression* between two (sub)*Expression*s, according to a predefined set of algebraic operators. Finally, it can be a *CustomExpression* using any available constraint/-query language (*e.g.*, OCL, XPath, etc.), the full expression simply stored as a string. Tools exploiting corresponding models are then in charge of processing such expressions.

As shown in Figure 5, the Configuration part of the language is simpler and directly refers to the Topology one (cf. concepts with dashed lines). An actually running Cloud system *Configuration* is composed of a set of *Nodes* and *Relationships* between them.

Each *Node* has an identifier and is of a given *NodeType*, as specified by the corresponding topology. It also comes with a boolean value indicating whether it is actually activated or not in the configuration. This activation can be reflected differently in the real system according to the concerned type of node (*e.g.*, a given Virtual Machine (VM) is already launched or not). A node contains a set of *Attribute*s providing *name/value* pairs, still following the specifications of the related topology.

Each *Relationship* also has an identifier and is of a given *RelationshipType*, as specified again by the corresponding topology. It simply interconnects two allowed *Node*s together and indicates if the relationship can be possibly changed (*i.e.*, removed) over time, *i.e.*, if it is *constant* or not.

5.2 A YAML-like Concrete Syntax

We propose a notation for Cloud experts to quickly specify their topologies and initialize related configurations. It also permits sharing such models in a simple syntax to be directly read and understood by Cloud administrators. We first built an XML dialect and prototyped an initial version. But we observed that it was too verbose and complex, especially for newcomers. We also thought about providing a graphical syntax via simple diagrams. While this seems appropriate for visualizing configurations, this makes more time-consuming the topology creation/edition (writing is usually faster than diagramming for Cloud technical experts). Finally, we designed a lightweight textual syntax covering both topology and configuration specifications.

To provide a syntax that looks familiar to Cloud users, we considered YAML and its TOSCA version (OASIS, 2017) featuring most of the structural constructs we needed (for topologies and configurations). We started from this syntax and complemented it with our language-specific elements, notably concerning expressions and constraints as not supported in YAML (cf. Section 5.1). We also ignored some constructs from TOSCA YAML that are not required in our language (*e.g.*, related to interfaces, requirements or capabilities). We can still rely on other existing notations via our XaaS-TOSCA bridge (cf. Section 4.2). For instance, by translating a configuration definition from our language to TOSCA, users can benefit from the GUI offered by external tooling such as Eclipse Winery (Eclipse Foundation, 2017).

We show below how the Cloud experts and administrators can write the IaaS layer of the motivating example from Section 3.

Listing 1: Topology excerpt from our motivating example (IaaS level).

```
Topology: ELearning-IaaS

node_types:
InternalComponent:
PM:
    derived_from: InternalComponent
    properties:
      impactOfEnabling: 40
      impactOfDisabling: 30
      ...
VM:
    derived_from: InternalComponent
    ...
Cluster:
    derived_from: InternalComponent
    properties:
      constant ClusterConsOneCPU:
        type: integer
      constant ClusterConsOneRAM:
        type: integer
      constant ClusterConsMinOnePM:
        type: integer
      variable ClusterNbCPUActive:
        type: integer
        equal: Sum(Pred, PM.PmNbCPUAllocated)
      variable ClusterCurConsumption:
        type: integer
        equal: ClusterConsMinOnePM * NbLink(Pred) + ClusterNbCPUActive * ClusterConsOneCPU + ClusterConsOneRAM * Sum(Pred, PM.PmSizeRAMAllocated)
Power:
    derived_from: ServiceProvider
    properties:
      constant PowerCapacity:
        type: integer
      variable PowerCurConsumption:
        type: integer
        equal: Sum(Pred, Cluster.ClusterCurConsumption)
    constraints:
      PowerCurConsumption less_or_equal: PowerCapacity
...

relationship_types:
VM_To_PM:
  valid_source_types: VM
  valid_target_types: PM
PM_To_Cluster:
...
Cluster_To_Power:
...
```

As shown on Listing 1, each node type comes with its name and the node type it inherits from (if any). Then, the Cloud expert describes its various attribute types via the *properties* field, following the TOSCA YAML terminology. Similarly, for each relationship type the Cloud expert gives its name and then indicates its source and target node types.

Listing 2: Configuration excerpt from our motivating example (IaaS level).

```
Configuration:
  identifier: ElarningSystem_0
  topology: ELearning-IaaS
...
Node Power0:
    type: ELearning-IaaS.Power
    activated: 'true'
    properties:
      PowerCapacity: 1500
      PowerCurConsumption: 0
...
Node Cluster0:
    type: ELearning-IaaS.Cluster
    activated: 'true'
    properties:
      ClusterCurConsumption: 0
      ClusterNbCPUActive: 0
      ClusterConsOneCPU: 1
      ClusterConsOneRAM: 0
      ClusterConsMinOnePM: 5
...
Node PM0:
    type: ELearning-IaaS.PM
    activated: 'true'
    properties:
      ...
...
Relationship PM_To_Cluster0:
  type: ELearning-IaaS.PM_To_Cluster
  constant: true
  source: PM0
  target: Cluster0
...
```

As explained before, expressions can be used to indicate how to compute the initial value of an attribute type. For instance, the variable *ClusterCurConsumption* of the *Cluster* node type is initialized at configuration level by making a product between the value of other variables. Expressions can also be used to attach constraints to a given node/relationship type. For example, in the node type *Power*, the value of the variable *PowerCurConsumption* has to be lesser or equal to the value of the constant *PowerCapacity* (at configuration level).

As shown on Listing 2, for each configuration the Cloud administrator provides a unique identifier and indicates which topology it is relying on. Then, for each actual node/relationship, its particular type is explicitly specified by directly referring to the corresponding node/relationship type from a defined topology. Each node describes the values of its different attributes (calculated or set manually), while each relationship describes its source and target nodes.

6 TOOLING SUPPORT & APPLICATION

In this section, we briefly describe the current implementation of our approach as well as its application on a concrete scenario of adaptation.

6.1 Implementation

The proposed model-based architecture has been designed to be deployed by Cloud experts and administrators in any context requiring some Cloud modeling (*e.g.*, independently from the autonomic aspects). To this intent, they come with a corresponding tooling support. Based on our own experience and the rich available ecosystem, we decided to work on developing tooling based on Eclipse/EMF (Steinberg et al., 2008) for our architecture and corresponding XaaS language. This choice was reinforced by the fact that the Choco constraint solver (Jussien et al., 2008), a reference solver in the Constraint community (and for which we have access to a solid expert), has a compatible Java API. All the corresponding source code is available from a Git repository [6].

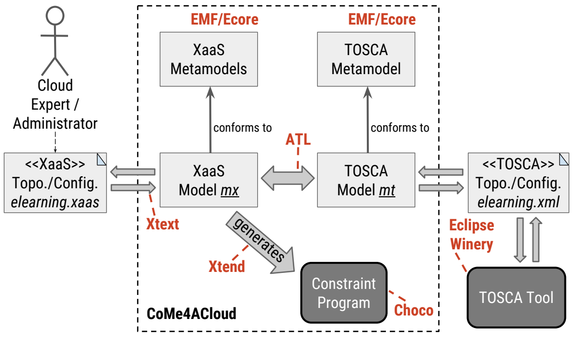
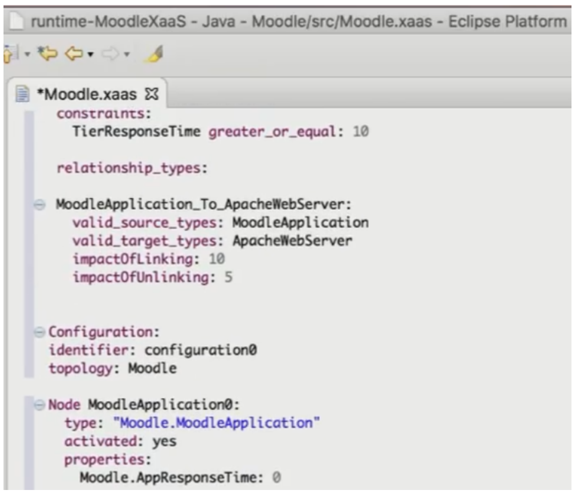


Figure 6: An Eclipse/EMF-based implementation.

As it can be seen from Figure 6, we made the choice of using our language as the core representation in our approach. The abstract syntax of our XaaS language has been defined via an Ecore (meta)model (Steinberg et al., 2008), while its concrete syntax has been defined via an Xtext grammar (Bettini, 2016). Thanks to Xtext, we have also been able to produce a dedicated editor coming with useful features such as syntax highlighting, code completion, static checks, etc (cf. Figure 7). The connection with the constraint solver has been implemented via corresponding code generator, in Xtend (Bettini, 2016), from our language to 1) the Java API provided by Choco and 2) the XML format currently expected by the CP program.


runtime-MoodleXaaS - Java - Moodle/src/Moodle.xaas - Eclipse Platform
*Moodle.xaas

```
  constraints:
    TierResponseTime greater_or_equal: 10

  relationship_types:

  MoodleApplication_To_ApacheWebServer:
    valid_source_types: MoodleApplication
    valid_target_types: ApacheWebServer
    impactOfLinking: 10
    impactOfUnlinking: 5


 Configuration:
 identifier: configuration0
 topology: Moodle

 Node MoodleApplication0:
   type: "Moodle.MoodleApplication"
   activated: yes
   properties:
     Moodle.AppResponseTime: 0
```

Figure 7: Screenshot of the developed Eclipse-based XaaS modeling environment.

In addition to this language and constraint support, we developed the required ATL (Jouault and Kurtev, 2005) model-to-model transformations so that our XaaS models can be partially transformed into TOSCA ones (and vice-versa). Thus, it is possible to benefit from the tooling already offered by TOSCA-based solutions (*e.g.*, Eclipse Winery (Eclipse Foundation, 2017) and its provided GUI for monitoring configurations).

6.2 Application on our Motivating Example

As a validation of the proposed technical solution, the presented Eclipse tooling has been deployed based on our motivating example (cf. Section 3). A complete video-demonstration of this application is available online [7]. It shows 1) the design of the SaaS layer with our XaaS modeling language, 2) the interoperability between the obtained XaaS models and a TOSCA-based tool (Eclipse Winery (Eclipse Foundation, 2017)) and 3) an illustration of a given adaptation, i.e. a reconfiguration, via our decision-making architecture (based on the Choco solver).

In the scenario from the video-demo, the modeled system (more precisely its SaaS-level) evolves from a current state to another one, both represented as configuration models in our language. In the initial configuration model, described graphically in Figure 8, a client WebApp is connected to a single instance of the Moodle application. This application relies on a Apache Web server and a MySQL server. Both are running on a same worker node that is deployed on a single Apache VM (from a unique provider).

In self-adaptative systems, there are two main types of trigger than can launch a re-configuration: 1) planned/periodical ones (*e.g.*, every 30 seconds) and 2) event-based ones (*e.g.*, a new client subscribe to a node, a monitored value changed in the system). In the scenario presented here, we are in the second case where a sudden increase of the Apache Web server's actual workload has been observed. This is going to augment the application response time and so consequently reduce the expected incomes and quality of service. As a result of this observation, the automated computation of a new configuration (according to the expressed SLAs) is triggered. The obtained reconfiguration is depicted in Figure 9.

In this new configuration model, the Apache Web server and the MySQL server are now running on two distinct worker nodes. These nodes are also deployed on two different Apache VMs, thus reducing the gen-

[6]https://gitlab.inria.fr/come4acloud/xaas

[7]http://hyperurl.co/come4acloud

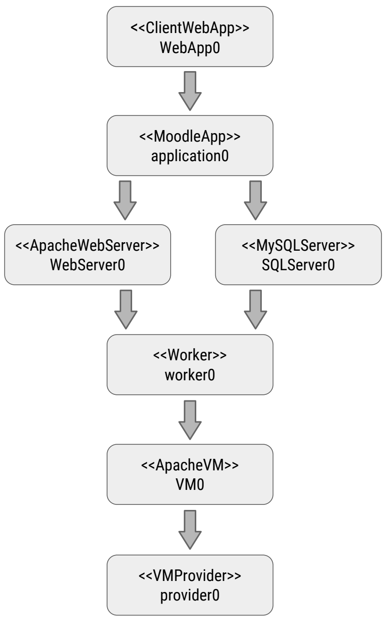


Figure 8: An initial configuration from our motivating example (SaaS level).

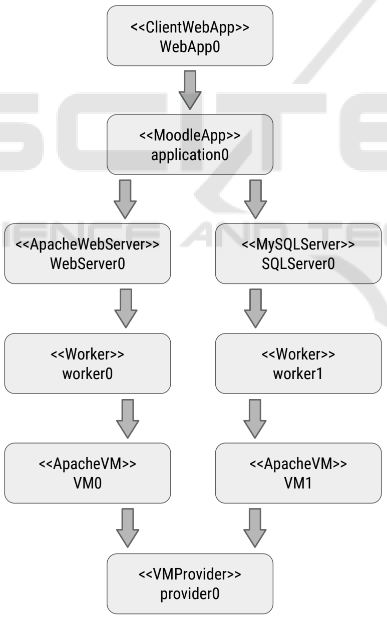


Figure 9: A reconfiguration from our motivating example (SaaS level).

eral workload and improving the response time of the whole system accordingly.

7 CURRENT STATUS

Within this paper, we presented in details the context of our approach (cf. Sections 2 & 3) as well as its underlying architecture (cf. Section 4) and core modeling language (cf. Section 5). Model-based principles and techniques acted as the required framework for bridging together the involved domains (*i.e.*, Cloud and Constraint) in a common integrated architecture. The use of models, as core pivot representations to be homogeneously used and shared both inside and outside the proposed architecture, illustrates this in practice. Modeling also acted as an enabler/facilitator for designing, building and handling the architecture's core language. Indeed, having a cleaner language definition and easier-to-maintain implementation is also an important aspect in our cross-domain context. Thanks to this architecture and language, we are able to provide a generic support for the automated reconfiguration of multi-layer Cloud systems. In coming Section 8, we compare our approach with the state-of-the-art in this area.

The implementation we propose (cf. Section 6) comes with other interesting challenges to tackle. We list below the most relevant ones and give insights on how we plan to address them in the future (or how we are already working on it for some of them).

- **Runtime.** The required synchronization between the XaaS models and the actual system they represent must be developed for each specific XaaS target. To preserve genericity as much as possible, we propose to implement a common adaptor interface for each target running system. Thus, we are currently working on building such a connector for the OpenStack (OpenStack Foundation, 2017) popular open source IaaS platform. At the SaaS level, we are also developing an Amazon integration for benchmarking our motivating example.
- **Scalability.** We provide a generic system-independent approach, so it has a certain price in terms of the scalability of the related constraint model/problem to be solved. However, we are already able to find solutions (*i.e.*, new configurations) quite efficiently (*e.g.*, in 10 seconds) for models of relatively important size (*e.g.*, several hundreds of nodes simulating virtual/physical machines). Other alternatives could be studied further in order to improve performances even more: for instance, we could consider hierarchizing the constraints (modeling the SLAs) to avoid combinatorial explosion.
- **Language V&V.** The current version of our XaaS modeling language comes with support for basic syntactical validation. However, we currently do not verify the correctness of the topologies and/or configuration described in our language. For example, we do not provide features allowing to verify a priori that the Cloud expert is not expressing

conflicting constraints in a given topology model. Relying on some existing verification solutions, such a support could be added to our approach in order to improve its general user experience.

- **Integration with Cloud Standards.** To go further than the current interoperability with TOSCA, we could propose the original features of our core XaaS modeling language (*e.g.*, the support for expressions/constraints) to the TOSCA standardization group. This way, we could collect their practical feedback which could eventually lead to a deeper integration of our approach with the well-known and used TOSCA standard.

8 RELATED WORK

To discuss our approach, we identified common characteristics we believe important for autonomic Cloud (modeling) solutions. Table 1 compares our approach with other existing work regarding different criteria:

- *Genericity* - The solution can support all Cloud system layers (*e.g.*, XaaS), or is specific to some particular and well-identified layers;
- *UI/Language* - It can provide a proper user interface and/or a modeling language intended to the different Cloud actors;
- *Interoperability* - It can interoperate with other existing/external solutions, and/or is compatible with a Cloud standard (*e.g.*, TOSCA);
- *Runtime support* - It can deal with runtime aspects of Cloud systems, *e.g.*, provide support for autonomic loops and/or synchronization.

In the industrial Cloud community, there are many existing multi-cloud APIs/libraries [8] [9] and DevOps tools [10] [11]. APIs enable IaaS provider abstraction, therefore easing the control of many different Cloud services, and generally focus on the IaaS client side. DevOps tools, in turn, provide scripting language and execution platforms for configuration management. They rather provide support for the automation of the configuration, deployment and installation of Cloud systems in a programmatical/imperative manner.

The Cloudify[12] platform overcomes some of these limitations. It relies on a variant of the TOSCA standard (OASIS, 2017) to facilitate the definition of Cloud system topologies and configurations, as well as to automate their deployment and monitoring. In the same vein, Apache Brooklyn[13] leverages Autonomic Computing (Kephart and Chess, 2003) to provide support for runtime management (via sensors/actuators allowing for dynamically monitoring and changing the application when needed). However, both Cloudify and Brooklyn focus on the application/client layer and are not easily applicable to all XaaS layers. Moreover, while Brooklyn is very handy for particular types of adaptation (*e.g.*, imperative event-condition-action ones), it may be limited to handle adaptation within larger architectures (*i.e.*, considering many components/services and more complex constraints). Our approach, instead, follows a declarative and holistic approach which is more appropriated for this kind of context.

The European project 4CaaSt proposed the *Blueprint Templates* abstract language (Nguyen et al., 2011) to describe Cloud services over multiple PaaS/IaaS providers. The Cloud Application Modeling Language (Bergmayr et al., 2014) studied in the ARTIST EU project (Menychtas et al., 2014) suggests using profiled UML to model (and later deploy) Cloud applications regardless of their underlying infrastructure. Similarly, the mOSAIC EU project proposes an open-source and Cloud vendor-agnostic platform (Sandru et al., 2012). StratusML (Hamdaqa and Tahvildari, 2015) provides another language for Cloud applications dealing with different layers to address the various Cloud stakeholders concerns. All these approaches focus on enabling the deployment of applications (SaaS or PaaS) in different IaaS providers. Thus they are quite layer-specific and do not provide support for autonomic adaptation.

The MODAClouds EU project (Ardagna et al., 2012) introduced some support for runtime management of multiple Clouds, notably by proposing CloudML as part of the Cloud Modeling Framework (CloudMF) (Ferry et al., 2013; Ferry et al., 2014). As in our approach, CloudMF provides a generic provider-agnostic model that can be used to describe any Cloud provider as well as mechanisms for runtime management by relying on Models@Runtime techniques (Blair et al., 2009). In the PaaSage EU project (Rossini, 2015), CAMEL (Achilleos et al., 2015) extended CloudML and integrated other languages such as the Scalability Rule Language (SRL) (Domaschka et al., 2014). The framework Saloon (Quinton et al., 2016) was also developed in this same project, relying on feature models to provide support for automatic Cloud configuration and selection. However, contrary to our generic approach, in these cases the adaptation decisions are delegated to

[8]Apache jclouds: https://jclouds.apache.org

[9]Deltacloud: https://deltacloud.apache.org

[10]Puppet: https://puppet.com

[11]Chef: https://www.chef.io/chef/

[12]http://cloudify.co

[13]https://brooklyn.apache.org

Table 1: Comparing different Cloud (modeling) solutions (✓ for full support, ∼ for partial support).

	Genericity	UI / Language	Interop-erability	Runtime support
APIs/DevOps		✓	✓	
Cloudify		✓	✓	✓
Brooklyn		✓	✓	∼
4CaaSt (Nguyen et al., 2011)		✓		
ARTIST-CAML (Bergmayr et al., 2014)		✓	✓	
mOSAIC (Sandru et al., 2012)		✓		∼
Stratus ML (Hamdaqa and Tahvildari, 2015)	✓	✓		
CloudMF (Ferry et al., 2013; Ferry et al., 2014)		✓		∼
PaaSage-CAML (Achilleos et al., 2015)		✓		∼
SRL (Domaschka et al., 2014)	∼	✓		✓
Saloon (Quinton et al., 2016)	✓	✓		
ARCADIA (Gouvas et al., 2016)	✓	✓		∼
Descartes (Kounev et al., 2016)		✓		✓
MODAClouds (Pop et al., 2016)		✓		✓
(García-Galán et al., 2014)	✓	✓		✓
CoMe4ACloud	✓	✓	∼	∼

3rd-parties tools and tailored to specific problems/-constraints (da Silva et al., 2014).

Recently, the ARCADIA EU project proposed a framework to cope with highly adaptable distributed applications designed as micro-services (Gouvas et al., 2016). While in a very early stage and with a different scope than us, it may be interesting to follow this work in the future. Among other existing approaches, we can cite the Descartes modeling language (Kounev et al., 2016) based on high-level meta-models to describe resources, applications, adaptation policies, etc. A generic control loop is proposed on top of it to fulfill some requirements for quality-of-service and resource management. Quite similarly, Pop *et al.*, (Pop et al., 2016) propose an approach for the deployment and autonomic management at runtime on multiple IaaS. However both approaches are targeting only Cloud systems structured as a SaaS deployed in a IaaS, whereas our approach allows modeling Cloud systems at any layer. In (García-Galán et al., 2014), feature models are used to define the configuration space (along with user preferences) and game theory is considered as a decision-making tool. This work focuses on features that are selected in a multi-tenant context, whereas our approach targets the automated computation of SLA-compliant configurations in a cross-layer manner.

To the best of our knowledge, there is currently no work that features at the same time genericity w.r.t. the Cloud layers, interoperability with standards (such as TOSCA), high-level modeling language support and some autonomic runtime management capabilities. The proposed model-based architecture described in this paper is a first step in this direction.

9 CONCLUSION

The proposed architecture and related XaaS modeling language intend to provide a generic solution for the autonomous runtime management of heterogeneous Cloud systems. To realize this, we notably rely on Constraint Programming as a decision-making tool to automatically obtain system configurations respecting specified SLA contracts. A main objective of this paper was to provide a suitable interface with our architecture, via its core XaaS modeling language, to both Cloud experts and administrators. Another goal was to have generic XaaS models that can possibly interoperate with standards (*e.g.*, TOSCA).

In the future we intend to apply our approach to other contexts somehow related to Cloud Computing, such as in the domain of Fog Computing for instance. This is expected to come with particular challenges in terms of scalability notably. Thus, the defined architecture and modeling language will have to be re-evaluated in the light of this new Fog context. We foresee their needed evolution in order to be able to efficiently model and support Fog characteristics such as a higher geographic distribution, the diversity of the involved resources/services, their reliability, etc.

Finally, we are aware that specifying constraints for the considered systems and their SLAs can be a difficult and time-consuming activity. Moreover, the quality of the produced contraints highly depends on human knowledge and experience (*e.g.*, from the Cloud Experts/Administrators). To limit potential errors and improve the efficiency of our approach, it would be interesting to be able to exploit automatically the historical data of the modeled systems. To this end, we plan to explore the possible use of some Machine Learning techniques which could guide or assist the constraint specification process.

REFERENCES

Achilleos, A. P., Kapitsaki, G. M., Constantinou, E., Horn, G., and Papadopoulos, G. A. (2015). Business-Oriented Evaluation of the PaaSage Platform. In *IEEE/ACM UCC 2015*, pages 322–326.

Ardagna, D., Nitto, E. D., Casale, G., Petcu, D., Mohagheghi, P., Mosser, S., Matthews, P., Gericke, A., Ballagny, C., D'Andria, F., et al. (2012). MODA-Clouds: A Model-driven Approach for the Design and Execution of Applications on Multiple Clouds. In *MiSE@ICSE 2012*, pages 50–56.

Atkinson, C. and Kuhne, T. (2003). Model-Driven Development: a Metamodeling Foundation. *IEEE Software*, 20(5):36–41.

Bergmayr, A., Troya, J., Neubauer, P., Wimmer, M., and Kappel, G. (2014). UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. In *2nd CloudMDE@MODELS 2014*, pages 56–65.

Bettini, L. (2016). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd.

Blair, G., France, R. B., and Bencomo, N. (2009). Models@ run.time. *IEEE Computer*, 42:22–27.

da Silva, M. A. A., Ardagna, D., Ferry, N., and Prez, J. F. (2014). Model-Driven Design of Cloud Applications with Quality-of-Service Guarantees: The MODAClouds Approach. In *SYNACS 2014*, pages 3–10.

Domaschka, J., Kritikos, K., and Rossini, A. (2014). Towards a Generic Language for Scalability Rules. In *ESOCC*, pages 206–220.

Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C., and Hu, B. (2015). Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. In *IEEE CLOUD 2015*, pages 621–628.

Eclipse Foundation (2017). Winery project. https://projects.eclipse.org/projects/soa.winery.

Ferry, N., Rossini, A., Chauvel, F., Morin, B., and Solberg, A. (2013). Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In *IEEE CLOUD 2013*, pages 887–894.

Ferry, N., Song, H., Rossini, A., Chauvel, F., and Solberg, A. (2014). CloudMF: Applying MDE to Tame the Complexity of Managing Multi-cloud Applications. In *IEEE/ACM UCC 2014*, pages 269–277.

García-Galán, J., Pasquale, L., Trinidad, P., and Ruiz-Cortés, A. (2014). User-centric Adaptation of Multi-tenant Services: Preference-based Analysis for Service Reconfiguration. In *ACM SEAMS 2014*, pages 65–74.

Gouvas, P., Fotopoulou, E., Zafeiropoulos, A., and Vassilakis, C. (2016). A Context Model and Policies Management Framework for Reconfigurable-by-design Distributed Applications. *Procedia Computer Science*, 97(Supplement C):122 – 125.

Hamdaqa, M. and Tahvildari, L. (2015). Stratus ML: A Layered Cloud Modeling Framework. In *IEEE IC2E 2015*, pages 96–105.

Jouault, F. and Kurtev, I. (2005). Transforming Models with ATL. In *ACM/IEEE MODELS 2005*, pages 128–138.

Jussien, N., Rochart, G., and Lorca, X. (2008). Choco: an Open Source Java Constraint Programming Library. In *OSSICP@CPAIOR'08*.

Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50.

Kounev, S., Huber, N., Brosig, F., and Zhu, X. (2016). A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures. *IEEE Computer*, 49(7):53–61.

Krupitzer, C., Roth, F. M., VanSyckel, S., Schiele, G., and Becker, C. (2015). A Survey on Engineering Approaches for Self-adaptive Systems. *Pervasive and Mobile Computing*, 17:184–206.

Lejeune, J., Alvares, F., and Ledoux, T. (2017). Towards a Generic Autonomic Model to Manage Cloud Services. In *CLOSER 2017*.

Menychtas, A., Konstanteli, K., Alonso, J., Orue-Echevarria, L., Gorronogoitia, J., Kousiouris, G., Santzaridou, C., Bruneliere, H., Pellens, B., Stuer, P., Strauss, O., Senkova, T., and Varvarigou, T. (2014). Software Modernization and Cloudification Using the ARTIST Migration Methodology and Framework. *Scalable Computing : Practice and Experience*, 15(2):131–152.

Narasimhan, B. and Nichols, R. (2011). State of Cloud Applications and Platforms: The Cloud Adopters' View. *IEEE Computer*, 44(3):24–28.

Nguyen, D. K., Lelli, F., Taher, Y., Parkin, M., Papazoglou, M. P., and van den Heuvel, W.-J. (2011). Blueprint Template Support for Engineering Cloud-based Services. In *ServiceWave 2011*, pages 26–37.

OASIS (2017). Topology and Orchestration Specification for Cloud Applications (TOSCA) and YAML (TOSCA Simple Profile). https://www.oasis-open.org/standards.

OpenStack Foundation (2017). OpenStack Open Source Cloud Computing Software. https://www.openstack.org.

Pop, D., Iuhasz, G., Craciun, C., and Panica, S. (2016). Support Services for Applications Execution in Multi-clouds Environments. In *ICAC 2016*, pages 343–348.

Quinton, C., Romero, D., and Duchien, L. (2016). SALOON: a Platform for Selecting and Configuring Cloud Environments. *Software: Practice and Experience*, 46:55–78.

Rossini, A. (2015). Cloud Application Modelling and Execution Language (CAMEL) and the PaaSage Workflow. In *ESOCC 2015*, pages 437–439.

Sandru, C., Petcu, D., and Munteanu, V. I. (2012). Building an Open-Source Platform-as-a-Service with Intelligent Management of Multiple Cloud Resources. In *IEEE UCC 2012*, pages 333–338.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

Xu, X. (2012). From Cloud Computing to Cloud Manufacturing. *Robotics and Computer-Integrated Manufacturing*, 28(1):75–86.