CoMe4ACloud: An End-to-end Framework for Autonomic Cloud Systems

Zakarea Al-Shara[a], Frederico Alvares[a], Hugo Bruneliere[a], Jonathan Lejeune[b], Charles Prud'Homme[a], Thomas Ledoux[a]

[a]*IMT Atlantique-Inria-LS2N, 4 rue Alfred Kastler, 44307, Nantes, France*
[b]*Sorbonne Université-Inria-CNRS 4 place Jussieu, 75005 Paris, France*

Abstract

Autonomic Computing has largely contributed to the development of self-manageable Cloud services. It notably allows freeing Cloud administrators of the burden of manually managing varying-demand services, while still enforcing Service-Level Agreements (SLAs). All Cloud artifacts, regardless of the layer carrying them, share many common characteristics. Thus, it should be possible to specify, (re)configure and monitor any XaaS (Anything-as-a-Service) layer in an homogeneous way. To this end, the CoMe4ACloud approach proposes a generic model-based architecture for autonomic management of Cloud systems. We derive a generic unique Autonomic Manager (AM) capable of managing any Cloud service, regardless of the layer. This AM is based on a constraint solver which aims at finding the optimal configuration for the modeled XaaS, i.e. the best balance between costs and revenues while meeting the constraints established by the SLA. We evaluate our approach in two different ways. Firstly, we analyze qualitatively the impact of the AM behaviour on the system configuration when a given series of events occurs. We show that the AM takes decisions in less than 10 seconds for several hundred nodes simulating virtual/physical machines. Secondly, we demonstrate the feasibility of the integration with real Cloud systems, such as Openstack, while still remaining generic. Finally, we discuss our approach according to the current state-of-the-art.

Keywords: Cloud Computing; Autonomic Computing; Model Driven Engineering; Constraint Programming

1. Introduction

Nowadays, Cloud Computing is becoming a fundamental paradigm which is widely considered by companies when designing and building their systems. The number of applications that are developed for and deployed in the Cloud is constantly increasing, even in areas where software was traditionally not seen as the core element (cf. the relatively recent trend on Industrial Internet of Things and Cloud

Email addresses: zakarea.al-shara@imt-atlantique.fr (Zakarea Al-Shara), frederico.alvares@imt-atlantique.fr (Frederico Alvares), hugo.bruneliere@imt-atlantique.fr (Hugo Bruneliere), jonathan.lejeune@lip6.fr (Jonathan Lejeune), charles.prudhomme@imt-atlantique.fr (Charles Prud'Homme), Thomas.Ledoux@imt-atlantique.fr (Thomas Ledoux)

Manufacturing [1]). One of the main reasons for this popularity is the Cloud's provisioning model, that allows for the allocation of resources in an on-demand basis. Thanks to this, consumers are able to request/release compute/storage/network resources, in a quasi-instantaneous manner, in order to cope with varying demands [2].

From the provider perspective, a negative consequence of this service-based model is that it may quickly lead the whole system to a level of dynamicity that makes it difficult to manage (*e.g.*, to enforce Service Level Agreements (SLAs) by keeping Quality of Service (QoS) at acceptable levels). From the consumer perspective, the large amount and the variety of services available in the Cloud market [3] may turn the design, (re)configuration and monitoring into very complex and cumbersome tasks. Despite of several recent initiatives intending to provide a more homogeneous Cloud management support, for instance as part of the OASIS TOSCA [4] initiative or in some European funded projects (*e.g.*, [5][6]), current solutions still face some significant challenges.

Heterogeneity. Firstly, the heterogeneity of the Cloud makes it difficult for these approaches to be applied systematically in different possible contexts. Indeed, Cloud systems may involve many resources potentially having various and varied natures (software and/or physical). In order to achieve well-tuned Cloud services, administrators need to take into consideration specificities (*e.g.*, runtime properties) of several managed systems (to meet SLA guarantees at runtime). Solutions that can support in a similar way resources coming from all the different Cloud layers (*e.g.*, IaaS, PaaS, SaaS) are thus required.

Automation. Cloud systems are scalable by definition, meaning that Cloud system may be composed of large sets of components and hence complex software structures to be handled manually in an efficient way. This concerns not only the base configuration and monitoring activities, but also the way Cloud systems should behave at runtime in order to guarantee certain QoS levels and expected SLA contracts. As a consequence, solutions should provide means for gathering and analyzing sensor data, making decision and re-configuring (to translate taken decisions into actual actions on the system) when relevant.

Evolution. Cloud systems are highly dynamic: clients can book and release "elastic" virtual resources at any moment at time, according to given SLA contracts. Thus, solutions need to be able to reflect and support transparently the elastic and evolutionary aspects of services. This may be non trivial, especially for systems involving many different services.

In this context, the CoMe4ACloud collaborative project [1] relies on three main pillars: Modeling/MDE [7], Constraint Programming [8], and Autonomic Computing [9]. Its primary goal is to provide a generic and extensible solution for the runtime management of Cloud services, independently from the Cloud layer(s) they belong to. We claim that Cloud systems, regardless of the layer in the Cloud service stack, share many common characteristics and goals, which can serve as a basis for a more homogeneous model. In fact, systems can assume the role of both *consumer/provider* in the Cloud service stack, and the interactions among them are governed by SLAs. In general, Anything-as-a-Service (XaaS) objectives are very similar when generalizing it to a Service-Oriented Architecture (SOA) model: (i)

[1] `https://come4acloud.github.io/`

finding an optimal balance between costs and revenues, *i.e.*, minimizing the costs due to other purchased services and penalties due to SLA violation, while maximizing revenues related to services provided to customers; (ii) meeting all SLA or internal constraints (*e.g.*, maximal capacity of resources) related to the concerned service.

In previous work, we relied on the MAPE-K Autonomic Computing reference architecture as a means to build generic an Autonomic Manager (AM) capable of managing Cloud systems [10] at any layer. The associated generic model basically consists of graphs and constraints formalizing the relationships between the Cloud service providers and their consumers in a SOA fashion. From this model, we automatically generate a constraint programming model [8], which is then used as a decision-making and planning tool within the AM.

This paper adds on our previous work in that we provide further details on the constraint programming models and translation schemes. Above all, in this work, we show how the generic model layer seamlessly connects to the runtime layer, *i.e.*, how monitoring data from the running system are reflected to the model and how changes in the model (performed by the AM or human administrators) are propagated to the running system. We provide some examples showing this connection, notably over an infrastructure based on the OpenStack [11]. We evaluate experimentally the feasibility of our approach by conducting a quantitative study over a simulated IaaS system. The objective is to analyze the AM behaviour in terms of adaptation decisions as well as to show how well it scales, considering the generic nature of the approach. Concretely, the results show the AM takes decisions in less than 10 seconds for several hundred nodes simulating virtual/physical machines, while remaining generic.

The remainder of the paper is structured as follows. In Section 2, we provide the background concepts for the good understanding of our work. Section 3 presents an overview of our approach in terms of architecture and underlying modeling support. In Section 4, we provide a formal description of the AM and explain how we designed it based on Constraint Programming. Section 5 gives more details on the actual implementation of our approach and its connection to real Cloud systems. In Section 6, we provide and discuss related performance evaluation data. We describe in details the available related work in Section 7 and conclude in Section 8 by opening on future work.

2. Background

The CoMe4Acloud project is mainly based on three complementary domains of Computer science.

2.1. Autonomic Computing

Autonomic Computing [9] emerged from the necessity to autonomously manage complex systems, in which the manual human-like maintenance becomes infeasible such as those in context of Cloud Computing. Autonomic Computing provides a set of principles and reference architecture to help the development of self-manageable software systems. Autonomic systems are defined as a collection of *autonomic elements* that communicate with each other. An *autonomic element* consists of a single *autonomic manager* (AM) that controls one or many *managed elements*. A *managed element* is a software or hardware resources similar to its counterpart found

in non-autonomic systems, except for the fact that it is adapted with *sensors* and *actuators* so as to be controllable by *autonomic managers*.

An autonomic manager is defined as a software component that, based on high-level goals, uses the *monitoring* data from *sensors* and the internal *knowledge* of the system to *plan* and *execute* actions on the *managed element* (via *actuators*) in order to achieve those goals. It is also known as a MAPE-K loop, as a reference to *Monitor*, *Analyze*, *Plan*, *Execute*, *Knowledge*.

As previously stated, the *monitoring* task is in charge of observing the data collected by software or hardware sensors deployed in the managed element. The *analysis* task is in charge of finding a desired state for the managed element by taking into consideration the monitored data, the current state of the managed element, and adaptation policies. The *planning* task takes into consideration the current state and the desired state resulting from the *analysis* task to produce a set of changes to be performed on the *managed elements*. Those changes are actually performed in the *execution* task within the desired time with the help of the actuators deployed on the *managed elements*. Last but not least, the *knowledge* in an autonomic system assemble information about the autonomic element (*e.g.*, system representation models, information on the managed system's states, adaptation policies, and so on) and can be accessed by the four tasks previously described.

2.2. Constraint Programming

In the context of autonomic computing systems to manage the dynamics of cloud systems, in order to take into consideration goals or utility functions, it is necessary to implement some methods. In this work, the goals and utility functions are defined in terms of constraint satisfaction and optimization problems. To this end we rely on Constraint Programming to model and solve these kind of problems.

Constraint Programming (CP) is a paradigm that aims to solve combinatorial problems defined by variables, each of them associated with a domain, and constraints over them [8]. Then a general purpose solver attempts to find a solution, that is an assignment of each variable to a value from its domain which meet the constraints it is involved in. Examples of CP solvers include free open-source libraries, such as Choco solver [12], Gecode [13] or OR-tools [14] and commercial softwares, such as IBM CPLEX CP Optimizer [15] or SICStus Prolog [16].

2.2.1. Modeling a CSP

In Constraint Programming, a Constraint Satisfaction Problem (CSP) is defined as a tuple $\langle X, D, C\rangle$ and consists of a set of n variables $X = \{X_1, X_2, \ldots, X_n\}$, their associated domains D, and a collection of m constraints C. D refers to a function that maps each variable $X_i \in X$ to the respective domain $D(X_i)$. A variable X_i can be assigned to integer values (*i.e.*, $D(X_i) \subseteq \mathbb{Z}$), a set of discrete values (i.e. $D(X_i) \subseteq \mathcal{P}(\mathbb{Z})$) or real values (i.g. $D(X_i) \subset \mathbb{R}$). Finally, C corresponds to a set of constraints $\{C_1, C_2, \ldots, C_m\}$ that restrain the possible values variables can be assigned to. So, let $(v_1, v_2, \ldots, v_n^j)$ be a tuple of possible values for subset $X^j = \{X_1^j, X_2^j, \ldots, X_{n^j}^j\} \subseteq X$. A constraint C_j is defined as a relation on set X^j such that $(v_1, v_2, \ldots, v_n^j) \in C_j \bigcap (D(X_1^j)$ x $D(X_2^j)$ x $\ldots$ x $D(X_{n^j}^j))$.

2.2.2. Solving a CSP with CP

In CP, the user provides a CSP and a CP solver takes care of solving it. Solving a CSP $\langle X, D, C\rangle$ is about finding a tuple of possible values $(v_1, v_2, \ldots, v_n)$ for each

variable $X_i \in X$ such that $\forall i \in [\![1..n]\!], v_i \in D(X_i)$ and all the constraints $C_j \in C$ are met. In the case of a Constraint Optimization Problem (COP), that is, when a optimization criterion have to be maximized or minimized, a solution is the one that maximizes or minimizes a given objective function $f : D(X) \mapsto \mathbb{Z}$.

A constraint model can be achieved in a modular and composable way. Each constraint expresses a specific sub-problem, from arithmetical expressions to more complex relations such as `AllDifferent` [17] or `Regular` [18]. A constraint not only defines a semantic (`AllDifferent`: *variables should take distinct value in a solution*, `Regular` : *an assignment should respect a pattern given by an automaton*) but also embeds a *filtering* algorithm which detects values that cannot be extended to a solution. Modeling a CSP consists hence in combining constraints together, which offers both flexibility (the model can be easily adapted to needs) and expressiveness (the model is almost human readable). Solving a CSP consists in an alternation of a propagation algorithm (each constraint removes forbidden values, if any) and a Depth First Search algorithm with backtrack to explore the search space.

Overall, the advantages of adopting CP as decision-making modeling and solving tool is manifold: no particular knowledge is required to describe the problem, adding or removing variables/constraints is easy (and thus useful when code is generated), the general purpose solver can be tweaked easily.

2.3. Model-driven Engineering

Model Driven Engineering (MDE) [19, 20], more generally also referred to as Modeling, is a software engineering paradigm relying on the intensive creation, manipulation and (re)use of various and varied types of models. In a MDE/Modeling approach, these models are actually the first-class artifacts within related design, development, maintenance and/or evolution processes concerning software as well as their environments and data. The main underlying idea is to reason as much as possible at a higher level of abstraction than the one usually considered in more traditional approaches, e.g. which are often source code-based. Thus the focus is strongly put in modeling, or allowing the modeling of, the knowledge around the targeted domain or range of problems. One of the principal objectives is to capitalize on this knowledge/expertise in order to better automate and make more efficient the targeted processes.

Since several years already, there is a rich international ecosystem on approaches, practices, solutions and concrete use cases in/for Modeling [21]. Among the most frequent applications in the industry, we can mention the (semi-)automated development of software (notably via code generation techniques), the support for system and language interoperability (e.g. via metamodeling and model transformation techniques) or the reverse engineering of existing software solutions (via model discovery and understanding techniques). Complementarily, another usage that has increased considerably in the past years, both in the academic and industrial world, is the support for developing Domain-Specific Languages (DSLs) [22]. Finally, the growing deployment of so-called Cyber-Physical Systems (CPSs), that are becoming more and more complex in different industry sectors (thanks to the advent of Cloud and IoT for example), has been creating new requirements in terms of Modeling [23].

3. Approach Overview

This section provides an overview of the CoMe4ACloud approach. First, we describe the global architecture of our approach, before presenting the generic me-

tamodels that can be used by the users (Cloud Experts and Administrators) to model Cloud systems. Finally, to help Cloud users to deal with cross-layers and SLA, we provide a service-oriented modeling extension for XaaS layers.

3.1. Architecture

The proposed architecture is depicted in Figure 1 and is based on two main kinds of models, which conform to two different but complementary metamodels. On one hand, the topology metamodel is dedicated to the specification of the different topologies (*i.e.*, types) of Cloud systems. It is generic because this can be realized in a similar way for systems concerning any of the possible Cloud layers (*e.g.*, IaaS, PaaS or SaaS). On the other hand, the configuration metamodel is intended to the representation of actual configurations (*i.e.*, instances) of such systems at runtime. This is realized by referring to a corresponding (and previously specified) topology. Once again, this Configuration metamodel is generic because it is independent from any particular Cloud layer and topology/type of Cloud system. These two metamodels are the core elements of the XaaS modeling language we proposed in CoMe4ACloud (cf. next Section 3.2).

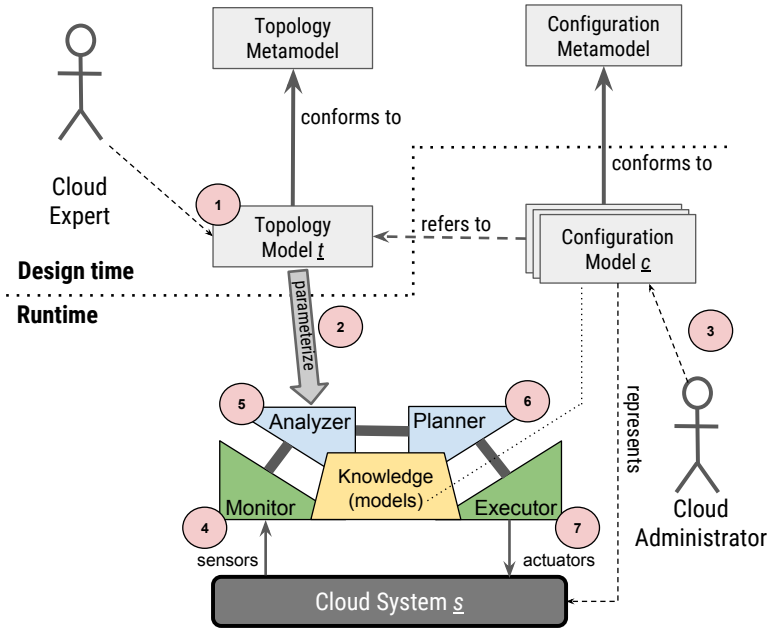


Figure 1: Overview of the model-based architecture in CoMe4ACloud.

In order to better grasp the fundamental concepts of our approach, it is important to reason about the architecture in terms of life-cycle. The life-cycle associated with this architecture involves both kinds of models. A topology model t has to be defined manually by a Cloud expert at design time (step 1). The objective is to specify a particular topology of system to be modeled and then handled at runtime, *e.g.*, a given type of IaaS (with Virtual/Physical Machines nodes) or SaaS (with webserver and database components). The topology model is used as the input of

a specific code generator that parameterizes a generic constraint program that is integrated into the *Analyzer* (step 2) of the generic AM.

The goal of the constraint program is to automatically compute and propose a new suitable system configuration model from an original one. Hence, in the beginning, the Cloud Administrator must provide an initial configuration model *c0* (step 3), which, along with the fore-coming configurations, is stored in the AM's *Knowledge* base. The state of the system at a given point in time is gathered by the *Monitor* (step 4) and represented as a (potentially new) configuration model $c0'$. It is important to notice that this new configuration model $c0'$ reflects the running Cloud system s current state (*e.g.*, a host that went down or a load variation), but it does not necessarily respect the constraints defined by the Cloud Expert/Administrator, *e.g.*, if a PM crashed, all the VMs hosted by it should be reassigned, otherwise the system will be in a inconsistent state. To that effect, the *Analyzer* is launched (step 5) whenever a new configuration model exists, whether it results from modifications that are manually performed by the Cloud Administrator or automatically performed by the *Monitor*. It takes into account the current (new) configuration model $c0'$ and related set of constraints encoded in the CP itself. As a result, a new configuration model $c1$ respecting those constraints is produced. The *Planner* produces a set of ordered actions (step 6) that have to be applied in order to go from the source ($c0'$) to the target ($c1$) configuration model. More details on the decision-making process, including the constraint program is given in Section 4.

Finally, the *Executor* (step 7) relies on actuators deployed on the real Cloud System to apply those actions. This whole process (from steps 4 to 7) can be re-executed as many times as required, according to the runtime conditions and the constraints imposed by the Cloud Expert/Administrator.

It is important to notice that configuration models are meant to be representations of actual Cloud systems at given points in time. This can be seen with configuration model c (stored within the *Knowledge* base) and Cloud system s in Figure 1, for instance. Thus, the content of these models has to always reflect the current state of the corresponding running Cloud systems. More details on how we ensure the required synchronization between the model(s) and the actual system are given in Section 5.2.

3.2. Generic Topology and Configuration Modeling

One of the key features of the CoMe4ACloud approach is the high-level language and tooling support, whose objective is to facilitate the description of autonomic Cloud systems with adaptation capabilities. For that purpose, we strongly rely on an MDE approach that is based on two generic metamodels.

As shown in Figure 2, 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 *RelationshipType*s that specify how to interconnect them. It can also have some global constraints attached to it.

Each *NodeType* has a name, a set of *AttributeType*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 (or "cost") of enabling/disabling nodes at runtime (*e.g.*, a given type of Physical Machine/PM node takes a certain time to be switched on/off).

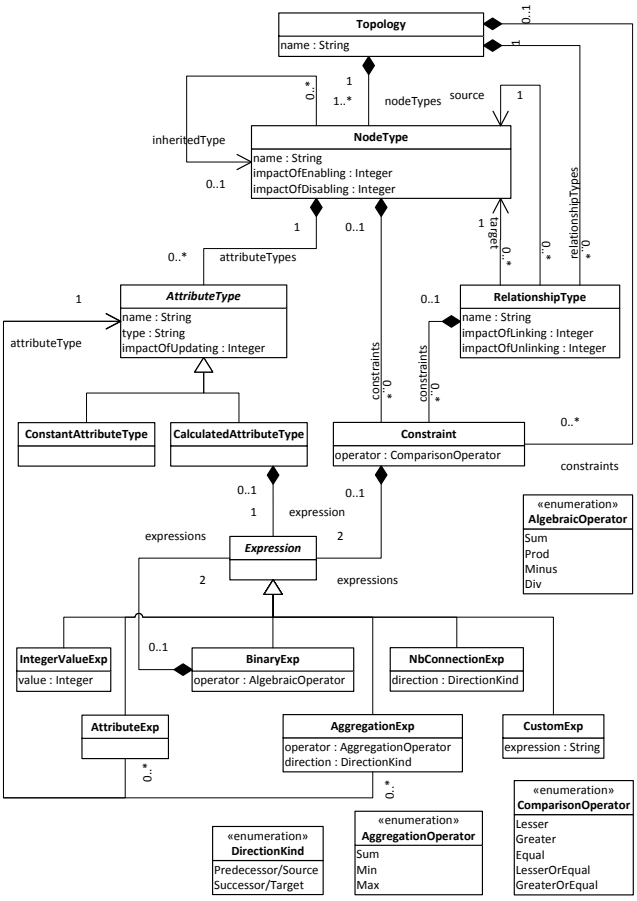


Figure 2: Overview of the Topology metamodel - Design time.

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*. 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 a *AggregationExpression* aggregating the values of a *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 3, the *Configuration* part of the language is lighter and directly refers to the *Topology* one. An actually running Cloud system *Configuration* is composed of a set of *Nodes* and *Relationships* between them.

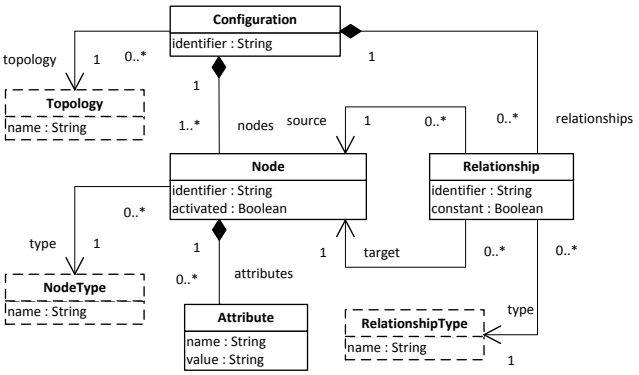


Figure 3: Overview of the Configuration metamodel - Runtime.

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 *Attributes* 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 *Nodes* together and indicates if the relationship can be possibly changed (*i.e.*, removed) over time, *i.e.*, if it is *constant* or not.

3.3. Service-oriented Topology Model for XaaS layers

The Topology and Configuration metamodels presented in the previous section provide a generic language to model XaaS systems. Thanks to that, we can model any kind of XaaS system that can be expressed by a Direct Acyclic Graph (DAG) with constraints having to hold at runtime. However, this level of abstraction can also be seen as an obstacle for some Cloud Experts and Administrators to model elements really specific to Cloud Computing. Thus, in addition to the generic modeling language presented before, we also provide in CoMe4ACloud an initial set of reusable node types which are related to the core Cloud concepts. They constitute a base Service-oriented topology model which basically represents XaaS systems in terms of their consumers (*i.e.*, the clients that consumes the offered services), their providers (*i.e.*, the required resources, also offered as services) and the Service Level Agreements (SLA) formalizing those relationships. Figure 4 shows an illustrative graphical representation of an example configuration model using the pre-defined node types.

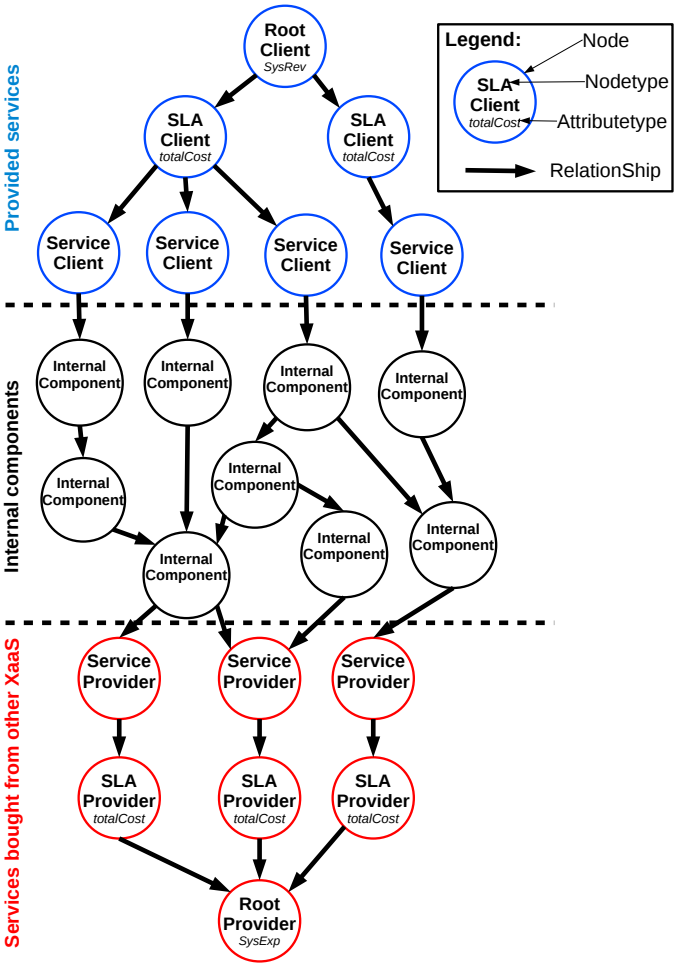


Figure 4: Example of configuration model using base Service-oriented node types (illustrative representation).

Root node types. We introduce two types of root nodes: *RootProvider* and *RootClient*. In any configuration model, it can only exist one node of each root node type. These two nodes do not represent a real component of the system but they can be rather seen as theoretical nodes. A *RootProvider* node (resp. *RootClient* node) has no target node (resp. source node) and is considered as the final target (resp. initial source). In other words, a *RootProvider* node (resp. *RootClient* node) node represents the set of all the providers (resp. the consumers) of the managed system. This allows grouping all features of both provider and consumer layers, especially the costs due to operational expenses of services bought from all the providers (represented by attribute *SysExp* in a *RootProvider* node) and revenues thanks to services sold to all the consumers (represented by attribute *SysRev* in a *RootClient* node).

SLA node types. We also introduce two types of SLA nodes: *SLAClient* and *SLAProvider*. In a configuration model, SLA nodes define the prices of each service level that can be provided and the amount of penalties for violations. Thus, both types of SLA nodes provide different attributes representing the different prices, penalties and then the current cost or revenue (*total_cost*) induced by current set of bought services (cf. the Service node types below) associated with it. A *SLAClient* node (resp. *SLAProvider* node) has a unique source (resp. target) which is the *RootClient* node (resp. *RootProvider* node) in the configuration. Consequently, an attribute *SysRev* (resp. *SysExp*) is equal to the sum of all attribute *total_cost* of its sources node (resp. target nodes).

Service node types. A SLA defines several Service Level Objectives (SLO) for each provided service [24]. Thus, we have to provide base Service node types: each service provided to a client (resp. received from a provider) is represented by a node of type *ServiceClient* (resp. *ServiceProvider*). The different SLOs are attributes of the corresponding Service nodes (e.g., configuration requirements, availability, response time, etc.). Since each Service node is linked with a unique SLA node in a configuration model, we define an attribute that designate the SLA node relating to a given service node. For a *ServiceClient* node (resp. *ServiceProvider* node), this attribute is named *sla_client* (resp. *sla_prov*) and its value is a node ID which means that the node has a unique source (resp. target) corresponding to the SLA.

Internal Component node type. *InternalComponent* represents any kind of node of the XaaS layer that we want to manage with the Generic AM (contrary to the previous node types which are theoretical nodes and provided as core Cloud concepts). Thus, it is kind of a common super-type of node to be extended by users of the CoMe4ACloud approach within their own topologies (e.g., cf. Listing 1 from Section 5.1). A node of this type may be used by another *InternalComponent* node or by a *ServiceClient* node. Conversely, it may require another *InternalComponent* node or a *ServiceProvider* node to work.

4. Decision Making Model

In this section, we describe how we formally modeled the decision making part (*i.e.*, the *Analyzer* and *Planner*) of our generic Autonomic Manager (AM) by relying on Constraint Programming (CP).

4.1. Knowledge (Configuration Models)

As previously mentioned, the *Knowledge* contains models of the current and past configurations of the Cloud system (*i.e.*, managed element). We define formal notations for a configuration at a given instant according to the XaaS model described in Figure 3.

4.1.1. The notion of time and configuration consistency

We first define T, the set of instants t representing the execution time of the system where t_0 is the instant of the first configuration (*e.g.*, the very first configuration model initialized by the Cloud Administrator, cf. Figure 1).

The XaaS configuration model at instant t is denoted by c^t , organized in a Directed Acyclic Graph (DAG), where vertices correspond to nodes and edges to relationships of the configuration metamodel (cf. Figure 3). $CSTR_{c^t}$ denotes the

set of constraints of configuration c^t. Notice that these constraints refer to those defined in the topology model (cf. Figure 2).

The property $satisfy(cstr, t)$ is verified at t if and only if the constraint $cstr \in CSTR_{c^t}$ is met at instant t. The system is satisfied ($satisfy(c^t)$) at instant t, if and only if $\forall cstr \in CSTR_{c^t}$, $satisfy(cstr, t)$. Finally, function $\mathcal{H}(c^t)$ gives the score of the configuration c at instant t : the higher the value, the better the configuration (*e.g.*, in terms of balance between costs and revenues).

4.1.2. Nodes and Attributes

Let n^t be a node at instant t. As defined in Section 3.2 it is characterized by:

- a node identifier ($id_n \in ID^t$), where ID^t is the set of existing node identifiers at t and id_n is unique $\forall t \in T$;
- a type ($type_n \in TYPES$)
- a set of predecessors ($preds_{n^t} \in \mathcal{P}(ID^t)$) and successors ($succs_{n^t} \in \mathcal{P}(ID^t)$) nodes in the DAG. Note that $\forall n_a^t, n_b^t \in c^t$, $id_{n_b^t} \neq id_{n_a^t}$

$$\exists id_{n_b^t} \in succs_{n_a^t} \Leftrightarrow \exists id_{n_a^t} \in preds_{n_b^t}$$

  . It is worth noting that the notion of predecessors and successors here is implicit in the notion of *Relationship* of the configuration metamodel.
- a set of constraints $CSTR_{n^t}$ specific to the type (cf. Figure 2).
- a set of attributes ($atts_{n^t}$) defining the node's internal state.

An attribute $att^t \in atts_{n^t}$ at instant t is defined by:

- name $name_{att}$, which is constant $\forall t \in T$,
- a value denoted $val_{att^t} \in \mathbb{R} \cup ID^t$ (*i.e.*, an attribute value is either a real value or a node identifier)

4.1.3. Configuration Evolution

The *Knowledge* within the AM evolves as configuration models are modified over the time. In order to model the transition between configuration models, the time T is discretized by the application of a transition function f on c^t such that $f(c^t) = c^{t+1}$. A configuration model transition can be triggered in two ways by:

- an internal event (*e.g.*, the Cloud Administrator initializes (add) a software component/node, a PM crashes) or an external event (*e.g.*, a new client arrival), which in both cases alters the system configuration and thus results in a new configuration model (cf. function *event* in Figure 5). This function models typically the *Monitor* component of the AM.
- the AM that performs the function *control*. This function ensures that $satisfy(c^{t+1})$ is verified, while maximizing $\mathcal{H}(c^{t+1})$ [2] and minimizing the transition cost to

[2] Since the research of optimal configuration (a configuration where the function $\mathcal{H}()$ has the maximum possible value) may be too costly in terms of execution time, it is possible to assume that the execution time of the *control* function is limited by a bound set by the administrator.

change the system state between c^t and c^{t+1}. This function characterizes the execution of the *Analyzer*, *Planner* and *Executor* components of the AM.

Figure 5 illustrates a transition graph among several configurations. It shows that an *event* function potentially moves away the current configuration from an optimal configuration and that a *control* function tries to get closer an new optimal configuration while respecting all the system constraints.

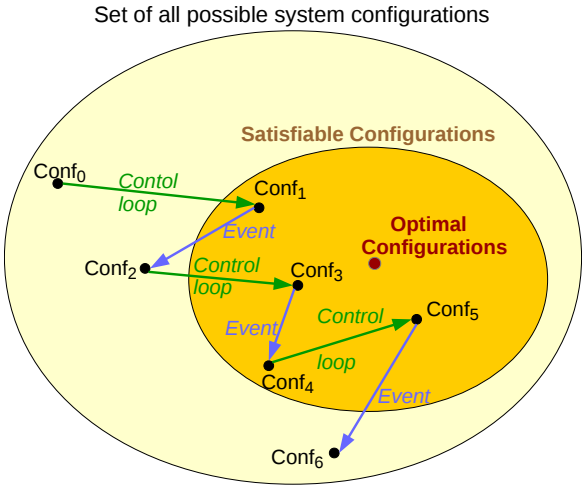


Figure 5: Examples of configuration transition in the set of configurations.

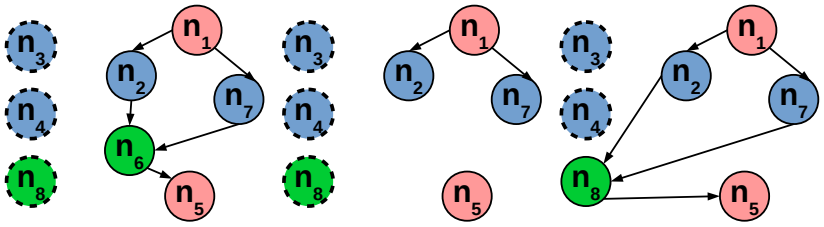
4.2. Analyzer (Constraint Model)

In the AM, the *Analyzer* component is achieved by a constraint solver. A Constraint Programming Model [8] needs three elements to find a solution: a static set of problem variables, a domain function, which associates to each variable its domain, and a set of constraints. In our model, the graph corresponding to the configuration model can be considered as a composite variable defined in a domain. For the constraint solver, the decision to add a new node in the configuration is impossible as it implies the adding of new variables to the constraint model during the evaluation. We have hence to define a set N^t corresponding to an upper bound of the node set c^t, *i.e.*, $c^t \subseteq N^t$. More precisely, N^t is the set of all existing nodes at instant t. Every node $n^t \notin c^t$ is considered as *deactivated* and does not take part in the running system at instant t.

Each existing node has consequently a boolean attribute called "activated" (cf. *Node* attribute *activated* in Figure 3). Thanks to this attribute the constraint solver can decide whether a node has to be enabled (true value) or disabled (false value).

The property $enable(n^t)$ verifies if and only if n is activated at t. This property has an incidence over the two neighbor sets $preds_{n^t}$ and $succs_{n^t}$. Indeed, when $enable(n^t)$ is false n^t has no neighbor because n does not depend on other node and no node may depend on n. The set N^t can only be changed by the *Administrator* or by the *Monitor* when it detects for instance a node failure or a new node in the running system (managed element), meaning that a node will be removed or added in N^{t+1}.

Figure 6 depicts an example of two configuration transitions. At instant t, there is a node set $N^t = \{n_1, n_2, \ldots, n_8\}$ and $c^t = \{n_1, n_2, n_5, n_6, n_7\}$. Each node color represents a given type defined in the topology (cf. Figure 3). The next configuration at $t+1$, the *Monitor* component detects that component n_6 of a green type has failed, leading the managed system to an unsatisfiable configuration. At $t+2$, the control function detects the need to activate a deactivated node of the same type in order to replace n_6 by n_8. This scenario may match the configuration transitions from $conf_1$ to $conf_3$ in Figure 5.



(a) Node set at t (b) Node set at $t+1$ (c) Node set at $t+2$

Figure 6: Examples of configuration transitions.

4.2.1. Configuration Constraints

The *Analyzer* should not only find a configuration that satisfies the constraints. It should also consider the objective function $\mathcal{H}()$ that is part of the configuration constraints. The graph representing the managed element (the running Cloud system) has to meet the following constraints:

1. any deactivated node n^t at $t \in T$ has no neighbor: n^t does not depend on other nodes and there is no node that depends on n^t. Formally,

$$\neg enable(n^t) \Rightarrow \left(succs_{n^t} = \emptyset \ \wedge \ preds_{n^t} = \emptyset\right)$$

2. except for root node types (cf. Section 3.3), any activated node has at least one predecessor and one successor. Formally,

$$enable(n^t) \Rightarrow \left(\mid succs_{n^t} \mid > 0 \wedge \ \mid preds_{n^t} \mid > 0\right)$$

3. if a node n^{t_i} is enabled at instant t_i, then all the constraints associated with n_a (link and attribute constraints) will be met in a finite time. Formally,

$$enable(n^{t_i}) \Rightarrow \exists t_j \geq t_i, \forall cstr \in CSTR_{n^{t_i}}$$

$$\wedge cstr \in CSTR_{n^{t_j}} \wedge enable(n^{t_j}) \wedge satisfy(cstr, t_j)$$

4. the function $\mathcal{H}()$ is equal to the balance between the revenues and the expenses of the system (cf. Figure 4). Formally,

$$\mathcal{H}(c^t) = att^t_{rev} - att^t_{exp}$$

where

$$att^t_{rev} \in atts_{n^t_{RC}} \wedge att^t_{rev} = SysRev$$

and

$$att^t_{exp} \in atts_{n^t_{RP}} \wedge att^t_{exp} = SysExp$$

4.2.2. Execution of the Analyzer

The *Analyzer* needs four inputs to process the next configuration:

a) The current configuration model which may be not satisfiable (*e.g.*, $c0'$ in Section 3.1);

b) The most recent satisfiable configuration model (*e.g.*, $c0$ in Section 3.1);

c) An expected lower bound of the next balance. This value depends on the reason why the AM has been triggered. For instance, we know that if the reason is a new client arrival or if a provider decreases its prices, the expected balance must be higher than the previous one. Conversely, if there is a client departure, we can estimate that the lower bound of the next balance will be smaller (in this case, the old balance minus the revenue brought by the client); This forces the solver to find a solution with a balance greater than or equal to this input value.

d) A boolean that indicates whether to use the *Neighborhood Search Strategy*, which is explained bellow.

```
1  Analyse (CurrentConf, SatisfiableConf, MinBalance, withNeighborhood)
   Result: a satisfiable Configuration
2  begin
3      solver ← buildConstraintModelFrom(CurrentConf);
4      initializer ← buildInitializer(SatisfiableConf, MinBalance, withNeighgorhood);
5      while not found solution and not error do
6          solver.reset();
7          initializer.nextVariableInitialization();
8          if variables correctly initialized then
9              solver.findsolution();
10             if a solution s found then
11                 return s;
12         else
13             error("impossible to initialize variables")
```

Algorithm 1: Global algorithm of the *Analyzer*

Algorithm 1 is the global algorithm of the *Analyzer* which mimics Large Neighborhood Search [25]. This strategy consists in two-step loop (lines 5 to 13) which is executed after the constraint model is instantiated in the solver (line 3) from the current configuration (*i.e.*, variables and constraints are declared).

First, in line 7, some variables of the solver model are selected to be fixed to their value in the previous satisfiable configuration (in our case, the b) parameter). This reduces the number of values of some variables X_s which can be assigned to, $D^0(X_j) \subset D(X_j), \forall X_j \in X_s$.

Variables not selected to be fixed represent a *variable area* (VA) in the DAG. It corresponds to the set of nodes in the graph the solver is able to change their successors and predecessors links. Such a restriction makes the search space to explore by the solver smaller which tends to reduce solving time. The way variables are selected is managed by the initializer (line 4). Note that when the initializer is built, the initial variable area, VA_i where $i = 0$, contains all the deactivated nodes and any nodes whose state has changed since the last optimal configuration (ex : attribute value modification, disappearance/appearance of a neighbour).

Then, the solver tries to find a solution for the partially restricted configuration (line 9). If a solution is found, the loop breaks and the new configuration is returned (line 11). Otherwise, the variable area is extended (line 7). A call for a new initialization at iteration i means that the solver has proved that there is no solution in iteration $i-1$. Consequently, a new initialization leads to relax the previous VA_{i-1}, $D^{i-1}(X_j) \subseteq D^i(X_j) \subseteq D(X_j), \forall X_j \in X_s$. At iteration i, VA_i is equal to VA_{i-1} plus the sets of successors and predecessors of all nodes in VA_{i-1}. Finally, if no solution is found and the initializer is not able to relax domains anymore, $D^i(X_j) = D(X_j), \forall X_j \in X_s$, the *Analyzer* throws an error.

This mechanism brings three advantages: (1) it reduces the solving time because domains cardinality is restrained, (2) it limits the set of actions in the plan, achieving thus one of our objective and (3) it tends to produce configurations that are close to the previous one in term of activated nodes and links.

Note that without the *Neighborhood Search Strategy*, the initial variable area VA_0 is equal to the whole graph leading thus to a single iteration.

4.2.3. Planner (Differencing and Match)

The *Planner* relies on differencing and match algorithms for object-oriented models [26] to compute the differences between the current configuration and the new configuration produced by the *Analyze*. From a generic point of view it exists five types of action : enable and disable node; link and unlink two nodes; and update attribute value.

5. Implementation Details

In this section, we provide some implementation details regarding the modeling languages and tooling support used by the users to specify Cloud systems as well as the mechanisms of synchronization between the models within the Autonomic Manager and the actual running Cloud systems.

5.1. A YAML-like Concrete Syntax

We propose a notation to allow Cloud experts quickly specifying their topologies and initializing 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 [27] featuring most of the structural constructs we needed (for topologies and configurations). We decided to start from this syntax and complement it with the elements specific to our language, notably concerning expressions and constraints as not supported in YAML (cf. Section 3.2). We also ignored some constructs from TOSCA YAML that are not required in our language (*e.g.*, related to interfaces, requirements or capabilities). Moreover, we can still rely on other existing notations. 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 [28].

As shown on Listing 1, for each node type the user gives its name and the node type it inherits from (if any) (cf. Section 3.3). Then she describes its different attribute types via the *properties* field, following the TOSCA YAML terminology. Similarly, for each relationship type the expert gives its name and then indicates its source and target node types.

As explained before (and not supported in TOSCA YAML), 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 user 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.

5.2. Synchronization with the running system

We follow the principles of Models@Runtime [29], by defining a bidirectional causal link between the running system and the model. The idea is to decouple the specificities of the causal link, w.r.t. the specific running subsystems, while keeping the Autonomic Manager generic, as sketched in Figure 7. It is important to recall that the configuration model is a representation of the running system and it can be modified in three different situations: (i) when the Cloud administrator manually changes the model; (ii) when it is the time to update the current configuration with data coming from the running system, which is done by the *Monitor* component; and (iii) when the *Analyzer* decides for a better configuration (*e.g.*, with higher balance function), in which case the *Executor* performs the necessary actions on the running Cloud systems. Therefore, the causal link with the running system is defined by two different APIs, which allows to reflect both the changes performed by the generic AM to the actual Cloud systems, and the changes that occur on the system at runtime to the generic AM. To that effect, we propose the implementation of an adaptor for each target running system (managed element).

From the *Executor* component perspective, the objective is to translate generic actions, *i.e.*, enable/disable, link/unlink nodes, update attribute values, into concrete operations (*e.g.*, deploy VM at a given PM) to be invoked over actuators of the different running subsystems (*e.g.*, Openstack, AWS, Moodle, etc.). From the *Monitor* point of view, the adaptors' role is to gather information from sensors deployed at the running subsystems (*e.g.*, a PM failure, a workload variation) and translate it into the generic operations to be performed on the configuration model by the *Monitor*, *i.e.*, add/remove/enable/disable node, link/unlike nodes and update attribute value.

It should be noticed that the difference between the two APIs is the possibility to add and remove nodes to the configuration model. In fact, the resulting configuration from the *Analyzer* does not imply the addition or removal of any node, since

Listing 1: Topology excerpt.

```
Topology: IaaS

node_types:

InternalComponent:
...

PM:
    derived_from: InternalComponent
    properties:
      impactOfEnabling: 40
      impactOfDisabling: 30
      ...

VM:
    derived_from: InternalComponent
    properties:
      ...

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:
  valid_source_types: PM
  valid_target_types: Cluster

Cluster_To_Power:
  valid_source_types: Cluster
  valid_target_types: Power

...
```

Listing 2: Configuration excerpt.

```
Configuration:
  identifier: IaaSSystem_0
  topology: IaaS

Node Power0:
    type: IaaS.Power
    activated: 'true'
    properties:
      PowerCapacity: 1500
      PowerCurConsumption: 0

...

Node Cluster0:
    type: IaaS.Cluster
    activated: 'true'
    properties:
      ClusterCurConsumption:
          0
      ClusterNbCPUActive: 0
      ClusterConsOneCPU: 1
      ClusterConsOneRAM: 0
      ClusterConsMinOnePM: 5

...

Node PM0:
    type: IaaS.PM
    activated: 'true'
    properties:
      ...

...

Relationship PM_To_Cluster0:
  type: IaaS.PM_To_Cluster
  constant: true
  source: PM0
  target: Cluster0

...
```

the constraint solver may not add/remove variables during the decision-making process, as already explained in Section 4.2. The Cloud Administrator and the *Monitor*, on the contrary may modify the configuration model (that is given as input to the constraint solver) by removing and adding nodes as a reflect of both the running Cloud system (*e.g.*, a PM that crashed) or new business requirements or agreements (*e.g.*, a client that arrives or leaves). Notice also that both adaptors and the *Monitor* component are the entry-points of the running subsystems and the generic AM, respectively. Thus, the adaptors and the *Monitor* are the entities that actually have to implement the APIs.

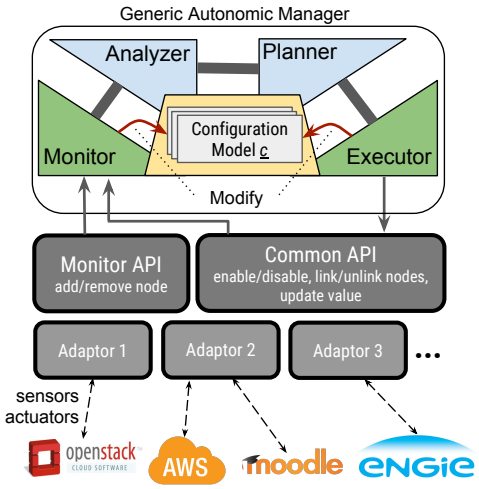


Figure 7: Synchronization with real running system.

We rely on a number of libraries (*e.g.*, AWS Java SDK [3], Openstack4j [4]) that ease the implementation of adaptors. For example, Listing 3 shows an excerpt of the implementation of the *enable* action for a *VM* node in Openstack4j. For the full implementation and for more examples, please see `https://gitlab.inria.fr/come4acloud/xaas`.

Listing 3: Excerpt of an IaaS adaptor with OpenStack.

```
1  OSClientV3 os = OSFactory.builderV3()
2                      .endpoint(...)
3                      .credentials(...)
4                      .authenticate();
5  ...
6
7  ServerCreate vm = Builders.server()
8                      .name(nodeID)
9                      .flavor(flavorID)
10                     .image(imageID)
11                     .availabilityZone(targetCluster + ":" + targetPM)
12                     .build();
13
14 Server vm = os.compute().servers().boot(vm);
```

[3] `https://aws.amazon.com/fr/sdk-for-java/`
[4] `http://www.openstack4j.com`

6. Performance Evaluation

In this section, we present an experimental study of our generic AM implementation that has been applied to an IaaS system. The main objective is to analyze qualitatively the impact of the AM behaviour on the system configuration when a given series of events occurs, and notably the time required by the constraint solver to take decisions. Note that the presented simulation focuses on the performance of the controller. Additionnally, we also experimented with the same scenario on a smaller system but in a real OpenStack IaaS infrastructure[5]. In a complementary manner, a more detailed study of the proposed model-based architecture (and notably its core generic XaaS modeling language) can be found in [30] where we show the implementation of another use case, this time for a SaaS application[6].

6.1. The IaaS system

We relied on the same IaaS system whose models are presented in Listings 1 and 2 to evaluate our approach. In the following, we provide more details. For sake of simplicity, we consider that the IaaS provides a unique service to their customers: compute resource in the form of VMs. Hence, there exists a node type $VMService$ extending the *ServiceClient* type (cf. Section 3.3). A customer can specify the required number of CPUs and RAM as attributes of $VMService$ node. The prices for a unit of CPU/RAM are defined inside the SLA component, that is, inside the $SLAVM$ node type, which extends the $SLAClient$ type of the service-oriented topology model. Internally, the system has several $InternalComponent$s: VMs (represented by the node type VM) are hosted on PMs (represented by the node type PM), which are themselves grouped into Clusters (represented by the node type $Cluster$). Each enabled VM has exactly a successor node of type PM and exactly a unique predecessor of type $VMService$. This is represented by a relationship type stating that the predecessors of a PM are the VMs currently hosted by it. The main constraint of a VM node is to have the number of CPUs/RAM equal to attribute specified in its predecessor $VMService$ node. The main constraint for a PM is to keep the sum of allocated resources with VM less or equal than its capacity. A PM has a mandatory link to its $Cluster$, which is also represented by a relationship in the configuration model. A $Cluster$ needs electrical power in order to operate and has an attribute representing the current power consumption of all hosted PMs. The $PowerService$ type extends the $ServiceProvider$ type of the service-oriented topology model, and it corresponds to an electricity meter. A $PowerService$ node has an attribute that represents the maximum capacity in terms of kilowatt-hour, which bounds the sum of the current consumption of all $Cluster$ nodes linked to this node ($PowerService$). Finally, the $SLAPower$ type extends the $SLAProvider$ type and represents a signed SLA with an energy provider by defining the price of a kilowatt-hour.

6.2. Experimental Testbed

We implemented the *Analyzer* component of the AM by using the Java-based constraint solver Choco [12]. For scalability purposes the experimentation simulates the interaction with the real world, *i.e.*, the role of the components *Monitor* and

[5] CoMe4ACloud Openstack Demo:`http://hyperurl.co/come4acloud_runtime`
[6] CoMe4ACloud Moodle Demo: `http://hyperurl.co/come4acloud`

Executor depicted in Figure 1, although we have experimented the same scenario with a smaller system (fewer PMs and VMs) in a real OpenStack infrastructure[7]). The simulation has been conducted on a single processor machine with an Intel Core i5-6200U CPU (2.30GHz) and 6GB of RAM Memory running Linux 4.4.

The system is modeled following the topology defined in Listing 1, *i.e.*, compute services are offered to clients by means of Virtual Machines (VM) instances. VMs are hosted by PM, which in turn are grouped by Clusters of machines. As Clusters require electricity in order to operate, they can be linked to different power providers, if necessary (cf. section 6.1). The snapshot of the running IaaS configuration model (the initial as well as the ones associated to each instant $t \in T$) is described and stored with our configuration DSL (cf. Listing 2). At each simulated event, the file is modified to apply the consequences of the event over the configuration. After each modification due to an event, we activated the AM to propagate the modification on the whole system and to ensure that the configuration meets all the imposed constraints.

The simulated IaaS system is composed of 3 clusters homogeneous PMs. Each PM has 32 processors and 64 GB of RAM memory. The system has two power providers: a classical power provider, that is, brown energy provider and a green energy provider.

The current consumption of a turned on PM is the sum of its idle power consumption (10 power units) when no guest VM is hosted with an additional consumption due to allocated resources (1 power unit per CPU and per RAM allocated). In order to avoid to degrade the analysis performance by considering too much physical resources compared to the number of consumed virtual resources, we limit the number of unused PM nodes in the configuration model while ensuring a sufficient amount of available physical resources to host a potential new VM.

In the experiments, we considered five types of event:

- *AddVMService* (a): a new customer arrival which requests for x $VMService$ (x ranges from 1 to 5). The required configuration of this request (*i.e.*, the number of CPUs and RAM units and the number of $VMService$) is chosen independently, with a random uniform law. The number of required CPU ranges from 1 to 8, and the number of required RAM units ranges from 1 to 16 GB. The direct consequences of such an event is the addition of one $SLAVM$, x $VMService$ nodes and x VM nodes in the configuration model file. The aim of the AM after this event is to enable the x new VM and to find the best PM(s) to host them.

- *leavingClient* (l): a customer decides to cancel definitively the SLA. Consequently, the corresponding $SLAVM$, $VMService$ and VM nodes are removed from the configuration. After such an event the aim of the AM is potentially to shut down the concerned PM or to migrate other VMs to this PM in order to minimize the revenue loss.

- *GreenAvailable* (ga): the Green Power Provider decreases significantly the price of the power unit to a value below the price of the Brown Energy Provider. The consequence of that event is the modification of the price attribute

[7]CoMe4ACloud Openstack Demo: `http://hyperurl.co/come4acloud_runtime`

of the green $SLAPower$ node. The expected behaviour of the AM is to enable the green $SLAPower$ node in order to consume a cheaper service.

- *GreenUnAvailable* (gu): contrary to the *GreenAvailable* event, the Green Power Provider resets its price to the initial value. Consequently, the Brown Energy Provider becomes again the most interesting provider. The expected behaviour of the AM is to disable the green $SLAPower$ node to the benefit of the classical power provider.

- $CrashOnePM$ (c): a PM crashes. The consequence on the configuration is the suppression of the corresponding PM node in the configuration model. The goal of the AM is to potentially turn on a new PM and to migrate the VMs which were hosted by the crashed PM.

In our experiments, we consider the following scenario over the both analysis strategies *without_neigborhood* and *with_neigborhood* depicted in Section 4.2.2. Initially, the configuration at t_0, no VM is requested and the system is turned off. At the beginning, the unit price of the green power provider is twice higher than the price of the other provider (8 against 4). The unit selling price is 50 for a CPU and 10 for a RAM unit. Our scenario consists to repeat the following sequence of events: 5 $AddVMService$, 1 *leavingClient*, 1 *GreenAvailable*, 1 $CrashOnePM$, 5 $AddVMService$, 1 *leavingClient*, 1 *GreenUnAvailable* and 1 $CrashOnePM$. This allows to show the behaviour of the AM for each event with different system's sizes. We show the impact of this scenario over the following metrics:

- the amount of power consumption for each provider (Figures 8a and 8c);

- the amount of $VMService$ and size of the system in terms of number of nodes (Figure 8f);

- the configuration balance (function $\mathcal{H}()$) (Figure 8e).

- the latency of the Choco Solver to take a decision (Figure 8g)

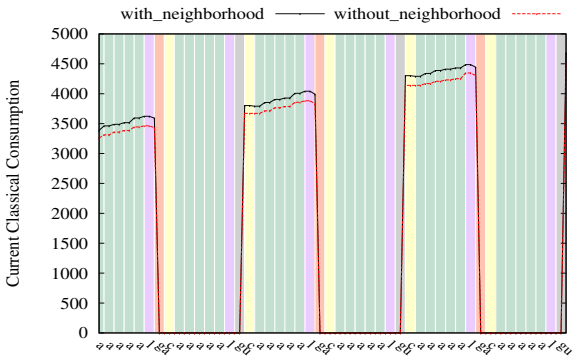
- the number of PMs being turned on (Figure 8d)

- the size of generated plan, *i.e.*, the number of required actions to produce the next satisfiable configuration (Figure 8b)

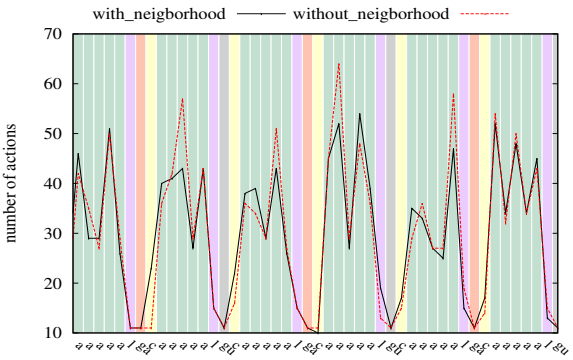
The x-axis in Figure 8 represents the logical time of the experiment in terms of configuration transition. Each colored area in this figure includes two configuration transitions: the event immediately followed by the control action. The color differs according to the type of the fired event. For the sake of readability, the x-axis does not begin at the initiation instant but when the number of node reaches 573 and events are tagged with the initials of the event's name.

6.3. Analysis and Discussion

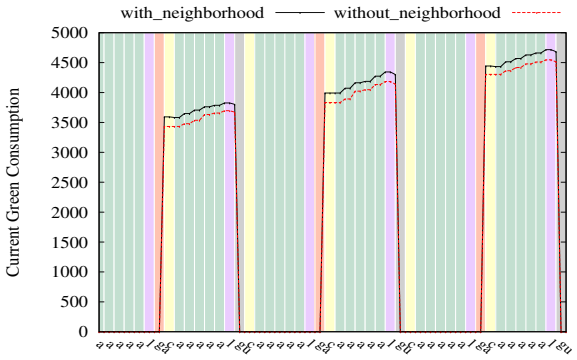
First of all, we can see that both strategies have globally the same behaviour whatever the received event. Indeed, in both cases a power provider is deactivated when its unit price becomes higher than the second one (Figures 8a and 8c). This shows that the AM is capable of adapting the choice of provided service according to their current price and thus benefit from sales promotions offered by its providers.



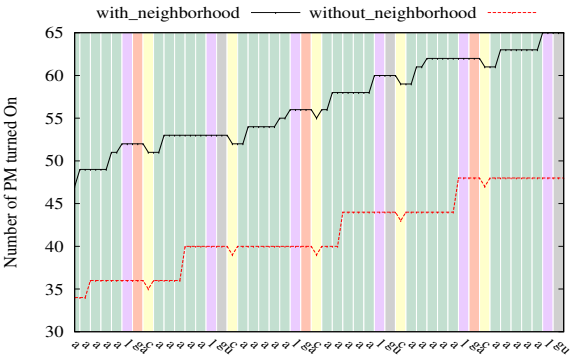
(a) Classical Power consumption.



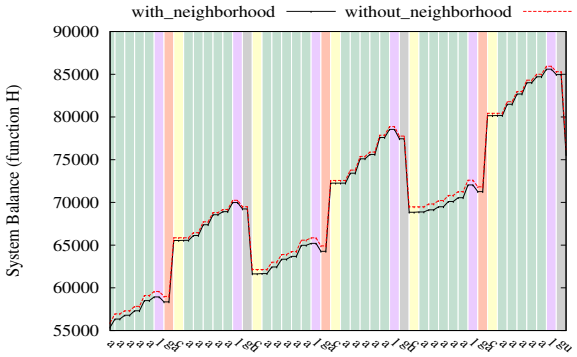
(b) Number of generated actions in planning phase



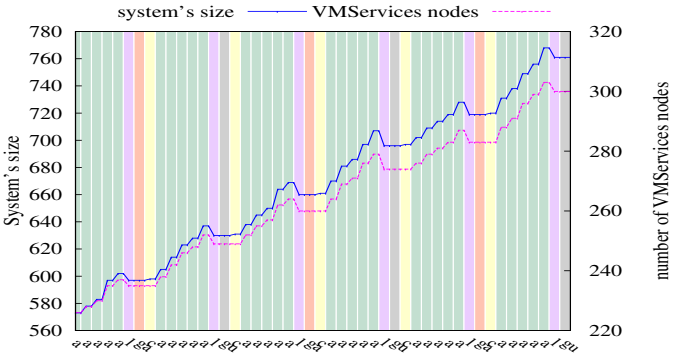
(c) Green Power consumption.



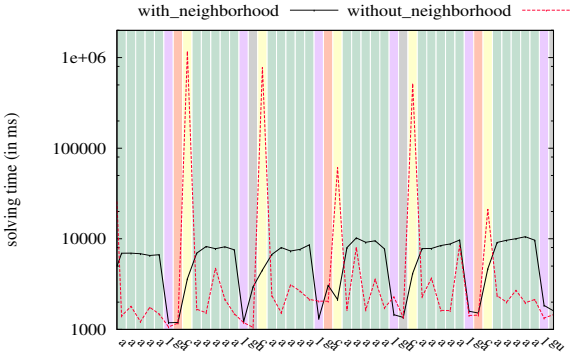
(d) Turned on PMs.



(e) System balance.



(f) Total number of nodes (left y axis) and required VMs (right y axis)



(g) Latency of the solver to take a decision

Figure 8: Experimental results of the simulation.

When the amount of requests for $VMService$ increases (Figure 8f) in a regular basis, the system power consumption increases (Figures 8a and 8c) sufficiently slowly so that the system balance also increases (Figure 8e). This can be explained by the ability of the AM to decide to turn on a new PM in a just-in-time way, that is, the AM tries to allocate the new coming VMs on existing enabled PM. On the other way around, when a client leaves the system, as expected, the number of $VMService$ nodes decreases but we can see that the number of PMs remains constant during this event, leading to a more important decrease of the system balance. Consequently, we can deduce that the AM has decided in this case to privilege the reconfiguration cost criteria at the expense of the system balance criteria. Indeed, we can notice in Figure 8b that the number of planning actions remains limited for the event l.

However, we can observe some differences on the values between both strategies. The main difference is in the decision to turn on a PM in the events $AddVMService$ and $CrashOnePM$. In the $AddVMService$ event, the neighborhood strategy favors the start-up of new PM, contrary to the other strategy which favors the use of PM already turned on. Consequently, the neighborhood strategy increases the power consumption leading to a less interesting balance. This can be explained by the fact that the neighborhood strategy avoids to modify existing nodes which limits its capacity for actions. Indeed, this is confirmed in Figure 8b where the curve of the neighborhood strategy is mostly lower than the other one. However, the solving time is worse (Figure 8g) because the minimal required variable area (VA) to find a solution needs several iterations.

Conversely, in the $CrashOnePM$ event, we note that the number of PM is mostly the same with the neighborhood strategy while the other one starts up systematically a new PM. This illustrates the fact that, in case of node disappearance, the neighborhood strategy tries to use as much as possible the existing nodes by modifying it as less as possible. Without neighborhood, the controller is able to modify directly all variables of the model. As a result, it is more difficult to find a satisfiable configuration, which comes at the expense of a long solving time (Figure 8g)

Finally, in order to keep an acceptable solving time while limiting the number of planning actions and maximizing the balance, it is interesting to choose the strategy according to the event. Indeed, the neighborhood strategy is efficient to repair nodes disappearance but the system balance may be lower in case of new client arrival. Although our AM is generic, we could observe that with the appropriate strategy, it can take decisions in less than 10 seconds for several hundred nodes. In terms of a CSP problem, the considered system's size corresponds to an order of magnitude of 1 million variables and 300000 constraints. Moreover, the taken decisions increase systemically the balance in case of favorable events (new service request from a client, price drop from a provider, etc.) and limits its degradation in case of adverse events (component crash, etc.) .

7. Related Work

In order to discuss the proposed solution, we identified common characteristics we believe important for autonomic Cloud (modeling) solutions. Table 1 compares our approach with other existing work regarding different criteria: 1) *Genericity* - The solution can support all Cloud system layers (*e.g.*, XaaS), or is specific to some particular and well-identified layers; 2) *UI/Language* - It can provide a proper user interface and/or a modeling language intended to the different Cloud actors; 3) *Interoperability* - It can interoperate with other existing/external solutions, and/or

is compatible with a Cloud standard (*e.g.*, TOSCA); 4) *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 muti-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 [4] 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 [9] 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.

Recently, OCCI (Open Cloud Computing Interface) has become one of the first standards in Cloud. The kernel of OCCI is a generic resource-oriented metamodel [31], which lacks a rigorous and formal specification as well as the concept of (re)configuration. To tackle these issues, the authors of [32] specify the OCCI Core Model with the Eclipse Modeling Framework (EMF)[14], whereas its static semantics is rigorously defined with the Object Constraint Language (OCL)[15]. An EMF-based OCCI model can ease the description of a XaaS, which is enriched with OCL constraints and thus verified by a many MDE tools. The approach, however, does not cope with autonomic decisions at runtime that have to be done in order to meet those OCL invariants.

The European project 4CaaSt proposed the *Blueprint Templates* abstract language [33] to describe Cloud services over multiple PaaS/IaaS providers. In the same direction, the Cloud Application Modeling Language [35] studied in the ARTIST EU project [52] 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 [36]. Finally, StratusML [37] provides another language for Cloud applications dealing with different layers to address the various Cloud stakeholders concerns. All these approaches focus on how to enable the deployment of applications (SaaS or PaaS) in different

[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://getcloudify.org`
[13] `https://brooklyn.apache.org`
[14] https://eclipse.org/modeling/emf
[15] http://www.omg.org/spec/OCL

	Generi-city	UI / Language	Interop-erability	Runtime support
APIs/DevOps		✓	✓	
Cloudify		✓	✓	✓
Brooklyn		✓	✓	∼
[32]	✓	✓	✓	
[33]		✓		
[34]		✓		∼
[35]		✓	✓	
[36]		✓		∼
[37]	✓	✓		
[38, 39]		✓		∼
[40]		✓		∼
[41]	∼	✓		✓
[42]	✓	✓		
[43][44][45]	✓	✓	-	-
[46]		✓		✓
[47]		✓		✓
[48]		✓	✓	✓
[49]	✓	✓		✓
[50]		✓		✓
[51]				✓
CoMe4ACloud	✓	✓	∼	∼

Table 1: Comparison of Cloud (modeling) solutions - ✓ for full support, ∼ for partial support

IaaS providers. Thus they are quite layer-specific and do not provide support for autonomic adaptation.

The MODAClouds EU project [5] introduced some support for runtime management of multiple Clouds, notably by proposing CloudML as part of the Cloud Modeling Framework (CloudMF) [38, 39]. 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 [29]. In the PaaSage EU project [6], CAMEL [40] extended CloudML and integrated other languages such as the Scalability Rule Language (SRL) [41]. However, contrary to our generic approach, in these cases the adaptation decisions are delegated to 3rd-parties tools and tailored to specific problems/constraints [53]. The framework Saloon [42] was also developed in this same project, relying on feature models to provide support for automatic Cloud configuration and selection. Similarly, [44] proposes the use of ontologies were used to express variability in Cloud systems. Finally, Mastelic *et al.*, [45] propose a unified model intended to facilitate the deployment and monitoring of XaaS systems. These approaches fill the gap between application requirements and cloud providers configurations but, unlike our approach, they focus on the initial configuration (at deploy-time), not on the run-time (re)configuration.

Recently, the ARCADIA EU project proposed a framework to cope with highly

adaptable distributed applications designed as micro-services [43]. 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 [46] which is based on high-level metamodels to describe resources, applications, adaptation policies, etc. On top of Descartes, a generic control loop is proposed to fulfill some requirements for quality-of-service and resource management. Quite similarly, Pop*et al.*, [47] propose an approach to support 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 [48], the authors extend OCCI in order to support autonomic management for Cloud resources, describing the needed elements to make a given Cloud resource autonomic regardless of the service level. This extension allows autonomic provisioning of Cloud resources, driven by elasticity strategies based on imperative Event–Condition–Action rules. The adaptation policies are, however, focused on the business applications, while our declarative approach, thanks to a constraint solver, is capable of controlling any target XaaS system so as to keep it close to the a consistent and/or optimal configuration.

In [49], 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.

Several approaches on SLA-based resource provisioning – and based on constraint solvers – have been proposed. Like in our approach, the authors of [50] rely on rely on MDE techniques and constraint programming to find consistent configurations of VM placement in order to optimize energy consumption. But no modeling or high-level language support is provided. Nonetheless, the focus remains on the IaaS infrastructure, so there is no cross-layer support. In [51], the authors propose a new approach to autoscaling that utilizes a stochastic model predictive control technique to facilitate resource allocation and releases meeting the SLO of the application provider while minimizing their cost. They use also a convex optimization solver for cost functions but no detail is provided about its implementation. Besides, the approach addresses only the relationship between SaaS and IaaS layers, while in our approach any XaaS service can be defined.

To the best of our knowledge, there is currently no work in the literature 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 an initial step in this direction.

8. Conclusion

The CoMe4ACloud architecture is a generic solution for the autonomous runtime management of heterogeneous Cloud systems. It unifies the main characteristics and objectives of Cloud services. This model enabled us to derive a unique and generic Autonomic Manager (AM) capable of managing any Cloud service, regardless of the layer. The generic AM is based on a constraint solver which reasons on very abstract concepts (e.g., nodes, relations, constraints) and tries to find the best balance between costs and revenues while meeting constraints regarding the established Service

Level Agreements and the service itself. From the Cloud administrators and experts point of view, this is an interesting contribution because it frees them from the difficult task of conceiving and implementing purpose-specific AMs. Indeed, this task can be now simplified by expressing the specific features of the XaaS Cloud system with a domain specific language based on the TOSCA standard. Our approach was evaluated experimentally, with a qualitative study. Results have shown that yet generic, our AM is able to find satisfiable configurations within reasonable solving times by taking the established SLA and by limiting the reconfiguration overhead. We also showed how we managed the integration with real Cloud systems like such as Openstack, while remaining generic.

For future work, we intend to apply CoMe4ACloud to other contexts somehow related to Cloud Computing. For instance, we plan experiment our approach in the domain of Internet of Things or Cloud-based Internet of Things, which may incur challenges regarding the scalability in terms of model size. We also plan to investigate how our approach could be used to address self-protection, that is, to be able to deal with security aspects in an autonomic manner. Last but not least, we believe that the constraint solver may be insufficient to make decisions in a durable way, *i.e.*, by considering the past history or even possible future states of the managed element. A possible alternative to overcome this limitation is to combine our constraint programming based decision making tool with control theoretical approaches for computing systems.

References

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

[2] Peter Mell and Tim Grance, The NIST Definition of Cloud Computing (2011).

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

[4] OASIS, Topology and Orchestration Specification for Cloud Applications (TOSCA) (Apr. 17).
URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html

[5] D. Ardagna, E. D. Nitto, G. Casale, D. Petcu, P. Mohagheghi, S. Mosser, P. Matthews, A. Gericke, C. Ballagny, F. D'Andria, et al., MODAClouds: A Model-driven Approach for the Design and Execution of Applications on Multiple Clouds, in: Proceedings of the 4th international workshop on modeling in software engineering (MISE 2012), IEEE Press, Zurich, Switzerland, 2012, pp. 50–56.

[6] A. Rossini, Cloud Application Modelling and Execution Language (CAMEL) and the PaaSage Workflow, in: Advances in Service-Oriented and Cloud ComputingÃĆÂŮWorkshops of ESOCC 2015, Vol. 567, Taormina, Italy, 2015, pp. 437–439.

[7] D. C. Schmidt, Guest editor's introduction: Model-driven engineering, Computer 39 (2) (2006) 0025–31.

[8] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier, 2006.

[9] J. O. Kephart, D. M. Chess, The Vision of Autonomic Computing, Computer 36 (1) (2003) 41–50.

[10] J. Lejeune, F. Alvares, T. Ledoux, Towards a generic autonomic model to manage cloud services, in: CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science, Porto, Portugal, April 24-26, 2017., 2017, pp. 147–158.

[11] OpenStack Foundation, OpenStack Open Source Cloud Computing Software (Apr. 2017).
URL https://www.openstack.org

[12] C. Prud'homme, J.-G. Fages, X. Lorca, Choco Documentation, TASC, LS2N CNRS UMR 6241, COSLING S.A.S. (2017).
URL http://www.choco-solver.org

[13] The Gecode Team, Gecode: A generic constraint development environment (2006).
URL http://www.gecode.org

[14] The OR-Tools Team, Google optimization tools (2017).
URL https://developers.google.com/optimization/

[15] Cplex cp optimizer, http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/.

[16] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. SjÃűland, J. Widen, Sicstus prolog user's manual, release 4, Tech. rep., SICS - Swedish Institute of Computer Science (2007).

[17] J. Régin, Generalized arc consistency for global cardinality constraint, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1., 1996.

[18] G. Pesant, A regular language membership constraint for finite sequences of variables (2004).

[19] D. C. Schmidt, Model-Driven Engineering, Computer 39 (2) (2006) 25.

[20] J. Bézivin, Model Driven Engineering: An Emerging Technical Space, Lecture Notes in Computer Science 4143 (2006) 36.

[21] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2012.

[22] M. Fowler, Domain-Specific Languages, Pearson Education, 2010.

[23] P. Derler, E. A. Lee, A. S. Vincentelli, Modeling Cyber–Physical Systems, Proceedings of the IEEE 100 (1) (2012) 13–28.

[24] Y. Kouki, T. Ledoux, Csla: a language for improving cloud sla management, in: Int. Conf. on Cloud Computing and Services Science, CLOSER 2012, 2012, pp. 586–591.

[25] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in: Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings, 1998, pp. 417–431.

[26] Z. Xing, E. Stroulia, Umldiff: An algorithm for object-oriented design differencing, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, ACM, New York, NY, USA, 2005, pp. 54–65.

[27] OASIS, YAML (TOSCA Simple Profile) (Apr. 2017).
URL http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.html

[28] Eclipse Foundation, Winery project (Apr. 2017).
URL https://projects.eclipse.org/projects/soa.winery

[29] G. Blair, R. B. France, N. Bencomo, Models@ run.time, Computer 42 (2009) 22–27.

[30] H. Bruneliere, Z. Al-Shara, F. Alvares, J. Lejeune, T. Ledoux, A Model-based Architecture for Autonomic and Heterogeneous Cloud Systems, in: Proc. 8th International Conference on Cloud Computing and Services Science (CLOSER 2018), Funchal, Madeira, Portugal, 2018.

[31] R. Nyren, A. Edmonds, A. Papaspyrou, T. Metsch, Open cloud computing interface - core, specification document, Tech. rep., Open Grid Forum, OCCI-WG (June 2011).

[32] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, S. Tata, A precise metamodel for open cloud computing interface, in: CLOUD 2015, 2015, pp. 852–859.

[33] D. K. Nguyen, F. Lelli, Y. Taher, M. Parkin, M. P. Papazoglou, W.-J. van den Heuvel, Blueprint Template Support for Engineering Cloud-based Services, in: Proceedings of the 4th European Conference on Towards a Service-based Internet, ServiceWave'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 26–37.

[34] F. Galán, A. Sampaio, L. Rodero-Merino, I. Loy, V. Gil, L. M. Vaquero, Service Specification in Cloud Environments Based on Extensions to Open Standards, in: Proceedings of the 4th International ICST Conference on COMmunication System softWAre and middlewaRE, COMSWARE '09, ACM, New York, NY, USA, 2009, pp. 19:1–19:12.

[35] A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, G. Kappel, UML-based Cloud Application Modeling with Libraries, Profiles, and Templates, in: Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud, CloudMDE@MODELS 2014, Valencia, Spain, 2014., 2014, pp. 56–65.

[36] C. Sandru, D. Petcu, V. I. Munteanu, Building an Open-Source Platform-as-a-Service with Intelligent Management of Multiple Cloud Resources, in: Proceedings of the 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing, UCC '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 333–338.

[37] M. Hamdaqa, L. Tahvildari, Stratus ML: A Layered Cloud Modeling Framework, in: 2015 IEEE International Conference on Cloud Engineering, 2015, pp. 96–105.

[38] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg, Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems, in: 2013 IEEE Sixth International Conference on Cloud Computing, 2013, pp. 887–894.

[39] N. Ferry, H. Song, A. Rossini, F. Chauvel, A. Solberg, CloudMF: Applying MDE to Tame the Complexity of Managing Multi-cloud Applications, in: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, 2014, pp. 269–277.

[40] A. P. Achilleos, G. M. Kapitsaki, E. Constantinou, G. Horn, G. A. Papadopoulos, Business-Oriented Evaluation of the PaaSage Platform, in: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), 2015, pp. 322–326.

[41] J. Domaschka, K. Kritikos, A. Rossini, Towards a Generic Language for Scalability Rules, Springer International Publishing, Manchester, UK, 2015, pp. 206–220.

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

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

[44] A. Dastjerdi, S. Tabatabaei, R. Buyya, An effective architecture for automated appliance management system applying ontology-based cloud discovery, in: CCGrid 2010, 2010, pp. 104–112.

[45] T. Mastelic, I. Brandic, A. Garcia Garcia, Towards uniform management of cloud services by applying model-driven development, in: COMPSAC 2014, 2014, pp. 129–138.

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

[47] D. Pop, G. Iuhasz, C. Craciun, S. Panica, Support Services for Applications Execution in Multiclouds Environments, in: 2016 IEEE International Conference on Autonomic Computing (ICAC), 2016, pp. 343–348.

[48] M. Mohamed, M. Amziani, D. Belaid, S. Tata, T. Melliti, An autonomic approach to manage elasticity of business processes in the cloud, FGCS 50 (2015) 49 – 61.

[49] J. García-Galán, L. Pasquale, P. Trinidad, A. Ruiz-Cortés, User-centric Adaptation of Multi-tenant Services: Preference-based Analysis for Service Reconfiguration, in: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, ACM, New York, NY, USA, 2014, pp. 65–74.

[50] B. Dougherty, J. White, D. C. Schmidt, Model-driven auto-scaling of green cloud computing infrastructure, FGCS 28 (2) (2012) 371–378.

[51] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, G. Iszlai, Optimal autoscaling in a iaas cloud, in: ICAC 2012, ACM, 2012, pp. 173–178.

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

[53] M. A. A. d. Silva, D. Ardagna, N. Ferry, J. F. Perez, Model-Driven Design of Cloud Applications with Quality-of-Service Guarantees: The MODAClouds Approach, in: 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2014, pp. 3–10.