Towards a Generic Autonomic Model to Manage Cloud Services

Jonathan Lejeune[1], Frederico Alvares[2] and Thomas Ledoux[2]
[1]*Sorbonne Universités-Inria-CNRS, Paris, France*
[2]*IMT Atlantique, LS2N, Nantes, France*

Keywords: Cloud Computing, Cloud Modeling, Cloud Self-management, Constraint Programming, XaaS.

Abstract: Autonomic Computing has recently contributed to the development of self-manageable Cloud services. It provides means to free Cloud administrators of the burden of manually managing varying-demand services while enforcing Service Level Agreements (SLAs). However, designing Autonomic Managers (AMs) that take into account services' runtime properties so as to provide SLA guarantees without the proper tooling support may quickly become a non-trivial, fastidious and error-prone task as systems size grows. In fact, in order to achieve well-tuned AMs, administrators need to take into consideration the specificities of each managed service as well as its dependencies on underlying services (e.g., a Sofware-as-a-Service that depends on a Platform/Infrastructure-as-a-Service). We advocate that Cloud services, regardless of the layer, may share the same consumer/provider-based abstract model. From that model we can derive a unique and generic AM that can be used to manage any XaaS service defined with that model. This paper proposes such an abstract (although extensible) model along with a generic constraint-based AM that reasons on abstract concepts, service dependencies as well as SLA constraints in order to find the optimal configuration for the modeled XaaS. The genericity of our approach are showed and discussed through two motivating examples and a qualitative experiment has been carried out in order to show the approache's applicability as well as to point out and discuss its limitations.

1 INTRODUCTION

The Cloud computing service provisioning model allows for the allocation of resources in an on-demand basis, i.e., consumers are able to request/release compute/storage/network resources, in a quasi-instantaneous manner, in order to cope with varying demands (Hogan and al., 2011). 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 so as to enforce Service Level Agreements (SLAs) by keeping Quality of Service (QoS) at acceptable levels.

Autonomic Computing (Kephart and Chess, 2003) has been largely adopted to tackle that kind of dynamic environments. In fact, it proposes architecture references and guidelines intended to conceive and implement Autonomic Managers (AMs) that make Cloud systems self-manageable, while freeing Cloud administrators of the burden of manually managing them.

In order to achieve well-tuned AMs, administrators need to take into consideration specificities of each managed service as well as its dependencies on underlying systems and/or services. In other words, AMs must be implemented taking into account several managed services' runtime properties so as to meet SLA guarantees at runtime, which may require sometimes a certain level of expertise on fields that administrators are not always familiar to or supposed to master (*e.g.*, optimization, modeling, etc.). Furthermore, modeling autonomic behaviours without having a holistic view of the system, its dependency as well as the impacts incurred by reconfigurations could lead it to inconsistent states. Therefore, conceiving AMs from scratch or dealing with them at a low level, and without the proper tooling support, may quickly become a cumbersome and error-prone task, especially for large systems.

We advocate that Cloud services, regardless of the layer in the Cloud service stack, share many common characteristics and goals. Services can assume the role of both *consumer* and *provider* in the Cloud service stack, and the interactions among them are governed by SLAs. For example, an Infrastructure-as-a-Service (IaaS) may provide Virtual Machines (VMs) to its customers, which can be for instance Platform-

Lejeune, J., Alvares, F. and Ledoux, T.
Towards a Generic Autonomic Model to Manage Cloud Services.
DOI: 10.5220/0006302801750186
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 147-158
ISBN: 978-989-758-243-1

as-a-Service (PaaS) or Software-as-a-Service (SaaS) providers, or end-users, but it may also be a client of Energy-as-a-Service (EaaS) providers. Similarly, the SaaS provides software services to end-users, while purchasing VM services provided by one or several IaaS providers. In this sense, Anything-as-a-Service (XaaS)' objectives are very similar when generalizing it to a Service-Oriented Architecture (SOA) model: (i) 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 other words, any AM could be designed so as to find *XaaS configurations* according to these objectives.

In this paper, we propose an *abstract model* to describe autonomic Cloud systems at any XaaS level. The model basically consists of graphs and constraints formalizing the relationships between the Cloud service providers and their consumers in a SOA fashion and is encoded in a constraint programming model (Rossi et al., 2006). From the latter, we can automatically derive decision-making and planning modules that are later on integrated into an AM. The ultimate goal is to provide the means for administrators to easily define XaaS systems so they can focus on the core functionalities of each service while leaving the autonomic engineering, namely the decision-making and planning, to be performed by the generic AM.

The major advantage of our approach is that it is generic. In fact, Cloud administrators are able to define their own XaaS models by extending/specializing the *abstract model*. Even so the extended XaaS model can still benefit from the constraint programming model in a transparent way. That is to say, the generic AM and the underlying constraint solver reason on abstract concepts, service dependencies as well as SLA or internal constraints so as to find the appropriate XaaS configurations at a given time.

We evaluate our approach in terms of genericity and applicability. The genericity is showed and discussed throughout two motivating examples illustrating an IaaS and a SaaS self-managed systems as well as their respective customers and providers. Regarding the applicability, we provide a qualitative evaluation by showing the behaviour of the IaaS system over the time, *i.e.*, how its state autonomously evolves in response to a series of simulated events occurring not only at the customers (*e.g.*, requesting/releasing resources) and providers (e.g., changes in the price of offered services, new services available, etc.) sides but also inside itself (*e.g.*, a crash on a given resource).

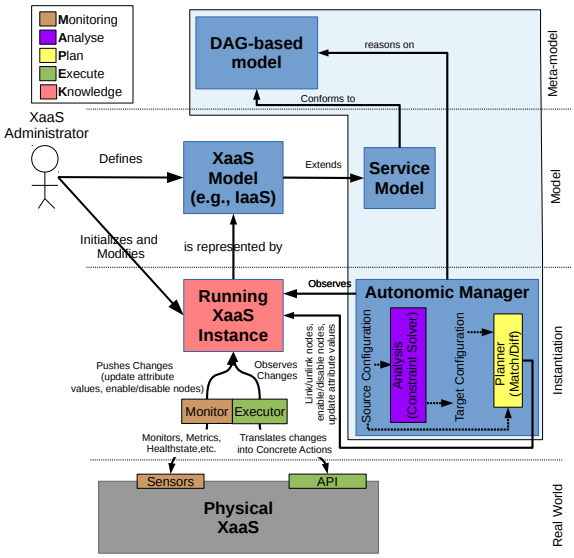


Figure 1: Approach Overview.

In the remainder of this paper, Section 2 gives an overview of the proposed approach. Section 3 presents a detailed and formal description of the abstract model. Examples of an IaaS and a SaaS model definitions are shown in Section 4. Section 5 shows the results on the qualitative evaluation performed on a IaaS model under the proposed generic autonomic manager. Related work is discussed in Section 6 and Section 7 concludes this paper.

2 ABSTRACT MODEL OVERVIEW

Our approach is based on a meta-model (Schmidt, 2006) allowing Cloud administrators to model any XaaS layer and on a MAPE-K loop providing autonomic features (Kephart and Chess, 2003). The rest of this section gives an overview of each modeling level of our approach as well as the generic AM, as depicted in Figure 1.

2.1 The Meta-model Level

We propose an abstract and generic model in which XaaS layers are architecturally composed of components and each component depends on other component in order to function. Thus, that can be modeled as a directed acyclic graph (DAG), where *nodes* represent atomic *components* of the system and *arrows* represent dependencies between the components. In other words, it exists an arc from a component A to a component B if and only if A depends on B. In the

following, the words *node* and *component* are interchangeable.

Each *node* may have several attributes defining its internal state and several constraints, which can be either *Link Constraints* or *Attribute Constraints*. The former specifies whether a component *A* may (or has to) use (or be used by) another component *B*, whereas the latter expresses a value depending on the value of other attributes located on the same node or on neighbor nodes.

2.2 The Model Level

The above mentioned meta-model provides a set of high-level DAG-based linguistic concepts allowing for the definition of components, attributes, dependencies among components and constraints on both attributes and dependencies. It is straightforward that the main advantage of relying on a DAG-based model is that it allows, if necessary, for checking properties such as connectivity or locality. At that level, however, the concepts remain quite far from the Cloud Computing domain, which makes it difficult to describe Cloud services equipped with autonomic capabilities.

We define a set of new linguistic concepts that allow the definition of a Cloud service in terms of relationships between *service providers* and *service consumers*, while taking into account the *SLAs* established in each relationship. The core of the service is modeled as a set of *internal components* that offer a set of services to *service clients* and may depend on a set of other services provided by *service providers*. In summary, we rely on the DAG-based meta-model to define a *Service Model* that introduces new SOA-related concepts while restraining the types of nodes, attributes and connections to be used. Thus, the *Service Model* is general enough to allow for the definition of *any* XaaS service and specific enough to simplify (by specialization) the task of the Administrator in defining *specific* XaaS models. For instance, an IaaS can be composed of a set of *internal components* (e.g., VMs with the attribute *ram_capacity*) that depend on a set of other *internal components* (e.g., PMs with the attribute *max_nb_vm*) or on a *service provider* (e.g., Energy Provider with the attribute *power_capacity*), that is, any service required by the service being modeled.

2.3 The Runtime Level

Once the Administrator has defined its XaaS model, he/she has to initialize the running instances, that is, the representation of the Physical XaaS entities (e.g., the real PMs) as well as their respective constraints in terms of dependencies, SLAs, attributes (e.g., CPU/RAM capacity). For instance, a running IaaS instance can be composed of a set of instances of the VM node with their initialization values (e.g., *ram_capacity=8GB*). This task is tremendously simplified by the adoption of a Model@run-time approach (Blair et al., 2009): the running XaaS instance represents the physical system and is linked in such a way that it constantly mirror the system and its current state and behavior; if the system changes, the representations of the system – the model – should also change, and vice versa.

A *XaaS configuration* is a snapshot of all running components, including the state of their current dependencies and their internal state. The *configuration* can then be modified by three actors: the XaaS Administrator, the Monitor and the AM. The XaaS Administrator modifies the configuration whenever he/she initializes the XaaS service by providing an initial configuration or for maintenance purposes.

The *Monitor* along with the *Executor* are responsible for keeping a causal link between the XaaS instance and the Physical XaaS. Hence, the *Monitor* modifies the configuration every time it detects that the state of the real Physical XaaS has changed by pushing the changes to the *XaaS Instance*. On the other way around, the *Executor* pushes the changes observed on the *XaaS instance* to the real system by translating them to concrete actions specific to the managed system.

The generic AM's role is to ensure that the current XaaS configuration: (i) respects the specified constraints; (ii) maximizes the balance between costs and revenues specified in SLA contracts. To that end, it observes regularly the running *XaaS instance* in both periodically or event-based basis (when severe events happen such as a SLA violation, a node that is no longer available, etc.) and triggers a constraint solver by taking as input the current configuration and produces as output a new configuration that is more suitable to the current *Physical XaaS* state. The *Planner* component produces a plan based on the difference between the current and new configurations in terms of components, attribute values and links, resulting in a set of reconfiguration actions (e.g., enable/disable, link/unlink and update attribute value) that have to be executed on the running *XaaS instance*. Lastly, *Executor* component pushes these actions to the *Physical XaaS*.

3 FORMAL DESCRIPTION

This section formally describes the DAG-based abstract model that is used to define the SOA-based model, from which a XaaS model can be extended.

3.1 Configurations and Transitions

Let T be the set of instants t representing the execution time of the system where t_0 is the instant of the first configuration. The XaaS configuration at instant t is denoted by c^t and includes all running nodes (e.g., PMs/VMs, Software Components, Databases, etc.), organized in a DAG. $CSTR_{c^t}$ denotes the set of constraints of configuration c^t.

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 consistent ($consistent(c^t)$), 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 configuration c at instant t, meaning that the higher this value, the better the configuration is (e.g., in terms of balance between costs and revenues).

We discretize the time T by the application of a transition function f on c^t such that $c^{t+1} = f(c^t)$. A configuration transition can be triggered in two ways by:

- an internal event (e.g., the XaaS administrator initializes a component, PM failure) or an external event (e.g., a new client arrival) altering the system configuration (cf. function *event* in Figure 2);
- the autonomic manager that performs the function *control*. This function ensures that $consistent(c^{t+1})$ is verified, while maximizing $\mathcal{H}(c^{t+1})$ [1] and minimizing the transition cost[2] to change the system state between c^t and c^{t+1}.

Figure 2 illustrates a transition graph among several configurations. It shows that an *event* function potentially moves away the current configuration from the optimal configuration and that a *control* function tries to get closer the optimal configuration while respecting all the system constraints.

3.2 Nodes and Attributes

Let n^t be a node at instant t. It is characterized by:

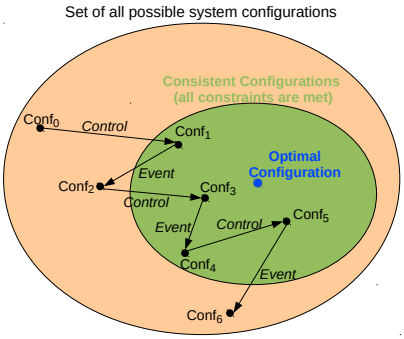


Figure 2: Examples of configuration transition in the set of configurations.

- 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 $\forall n^t_a, n^t_b \in c^t$, $id_{n^t_b} \neq id_{n^t_a}$
$$\exists id_{n^t_b} \in succs_{n^t_a} \Leftrightarrow \exists id_{n^t_a} \in preds_{n^t_b}$$
- a set of constraints $CSTR_{n^t}$ about links with neigborhood.
- 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: a 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); and a set of constraints $CSTR_{att^t}$ about its value (which may depends on local or remote attributes).

3.3 Service Model

XaaS services can assume the role of consumer or provider, and the interactions between them are governed by SLAs. According to these characteristics, we define our *Service Model* with the following node types where relationships between each one are illustrated and summarized in the Figure 3.

3.3.1 Root Types

We introduce two types of root nodes: the *RootProvider* and the *RootClient*. In *any* configuration, it exists exactly only one node instance of each root type respectively denoted n_{RP} and n_{RC}. These two nodes do not represent a real component of the system but they can be seen rather as theoretical nodes. The n_{RP} (resp. n_{RC}) node has no sucessor (resp. predecessor) and is considered as the only sink

[1] 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, we assume that the execution time of the *control* function is limited by a bound set by the administrator.

[2] Assuming that an approximate cost value for each reconfiguration action type is a priori known

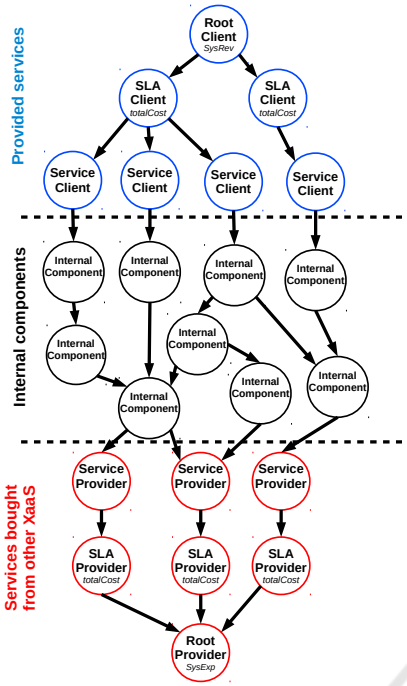


Figure 3: Example of a consistent configuration.

(resp. source) node in the DAG. The n_{RP} (resp. n_{RC}) node represents the set of all the providers (resp. the consumers) of the managed system. This allows to group 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 n_{RP}) and revenues thanks to services sold to all the consumers (represented by attribute *SysRev* in n_{RC}).

3.3.2 SLA Types

The relationship between the managed system and another system is modelled by a component representing a SLA. Consequently, we define in our model the *SLAClient* (resp. *SLAProvider*) type corresponding to a link between the modeled XaaS and one of its customer (resp. provider). A SLA defines the prices of each service level that can be provided and the amount of penalties for violations. Thus, a SLA component has different attributes representing the different prices, penalties and then the current cost or revenue (attribute *total_cost*) induced by current set of bought services (cf. service type below) associated with it. A *SLAClient* (resp. *SLAProvider*) has a unique predecessor (resp. successor) which is the *RootClient* (resp. *RootProvider*). Consequently, the attributes *SysRev* (resp. *SysExp*) is equal to the sum of all attribute *total_cost* of its successors (resp. predecessors).

3.3.3 Service Types

A SLA defines several Service Level Objectives (SLO) for each provided service (Kouki and Ledoux, 2012). Consequently, we have to model a service as a component. 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 modeled as attributes in the corresponding service component (e.g., configuration requirements, availability, response time, etc.). Since each *Service* is linked with a unique SLA component, we define for the service type an attribute designating the SLA node which the service is related to. For the *ServiceClient* (resp. *ServiceProvider*) type, this attribute is denoted by *sla_client* (resp. *sla_prov*) and its value is a node ID, which means that the node has a unique predecessor (resp. successor) corresponding to the SLA.

3.3.4 Internal Components Types

InternalComponent represents any kind of component of the XaaS layer that we want to manage with the AM. 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.

3.4 Autonomic Manager and Constraints Solver

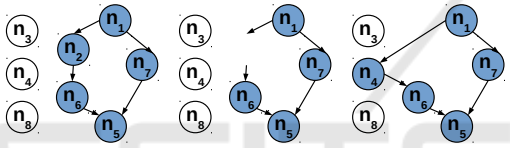
In the AM, the Analysis task is achieved by a constraint solver. A Constraint Programming Model (Rossi et al., 2006) needs three elements to find a solution: a static set of problem variables, a domain function, which associates to each variable its domain, and a set of constraints. In our model, the configuration graph 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 *activation attribute*. Thanks to this attribute the analyzer can decide whether a node has to be enabled (true value) or disabled (false value),

which corresponds respectively to a node adding/removing in the configuration.

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, meaning that a node will be removed in N^{t+1}.

Figure 4 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 $c^t = \{n_1, n_2, n_5, n_6, n_7\}$. The next configuration at $t+1$, the *Monitor* agent detects that component n_2 has failed, leading the managed system to an inconsistent configuration. At $t+2$, the control function detects the need to activate a disabled node in order to replace n_2 by n_4. This scenario matches the configuration transitions from $conf_1$ to $conf_3$ in Figure 2.



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

Figure 4: Examples of configuration transition.

3.5 Configuration Constraints

The graph representing the managed XaaS has to meet the following constraints:

1. any deactivated node n_a^t at $t \in T$ has no neighbor: n_a^t does not depend on other nodes and there is not a node n_b^t that depends on n_a^t. Formally,
$$\neg enable(n_a^t) \Rightarrow \left(succs_{n_a^t} = \emptyset \wedge preds_{n_a^t} = \emptyset\right)$$
2. except for root nodes, any activated node has at least one predecessor and one successor. Formally,
$$enable(n_a^t) \Rightarrow \left(\mid succs_{n_a^t} \mid > 0 \wedge \mid preds_{n_a^t} \mid > 0\right)$$
3. if a node $n_a^{t_i}$ 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,
$$enable(n^{t_i}) \Rightarrow \exists t_j \geq t_i, \forall cstr \in CSTR_{n_a^{t_i}}$$
$$\wedge cstr \in CSTR_{n_a^{t_j}} \wedge enable(n^{t_j}) \wedge satisfy(cstr, t_j)$$
4. the function $\mathcal{H}()$ is equal to the balance between the revenues and the expenses of the system.

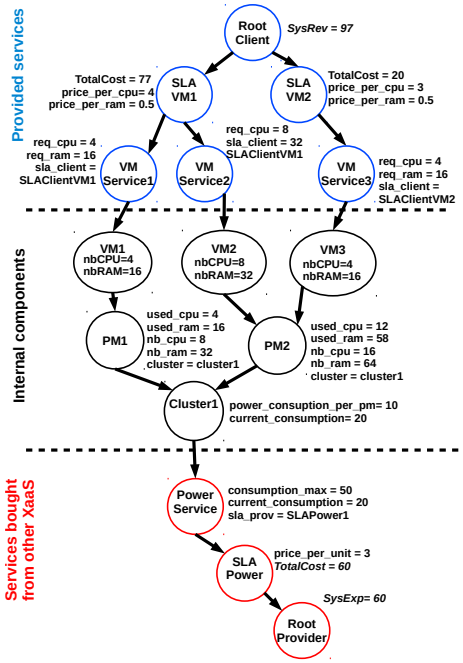


Figure 5: Example of a IaaS configuration.

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

4 IMPLEMENTATION EXAMPLES

The models presented in the previous sections rely on abstract provider/consumer relationships as well as on SLA constraints to describe any autonomic XaaS service. This section aims at showing the genericity of those models by applying them to two different XaaS: an IaaS and a SaaS. Figure 5 (resp. Figure 6) gives an example of a configuration at a given instant. Each enabled node is represented with its own attributes and their corresponding values.

4.1 Example of an IaaS Description

4.1.1 Provided Services

For sake of simplicity, we consider that such system 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 defined in the abstract model. This node type is responsible for bridging the IaaS and its customers. 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 abstract model. It should be noticed that prices may differ according to the customer.

4.1.2 Internal Components

VMs are hosted on PMs which are themselves grouped into Clusters. We define thus three node types extending the *InternalComponent* type:

- the type *VM* represents a virtual machine and it has an attribute defining the current number of CPUs/RAM. Each enabled *VM* has exactly a successor node of type *PM* and exactly a unique predecessor of type *VMService*. 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 type *PM* represents a physical machine with several attributes such as the total number of CPUs/RAM, the number of allocated CPUs/RAM on *VM* and the node representing the cluster hosting the *PM*. The latter attribute allows to express a constraint that specifies the physical link between the *PM* and its *cluster*. The predecessors of a *PM* are the *VMs* currently hosted by it.
- the type *Cluster* represents a component hosting several PMs. It has an attribute representing the current power consumption of all hosted PMs. This attribute is computed according to the power consumption of each running *PM*, i.e., the number of predecessors.

4.1.3 Services Bought from Other Providers

The different clusters of the modeled IaaS system need electrical power in order to operate. That power is also offered in the form of service (Energy-as-a-Service, i.e., electricity), by an energy provider. We define the *PowerService* type by extending the *ServiceProvider* type of the abstract 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.

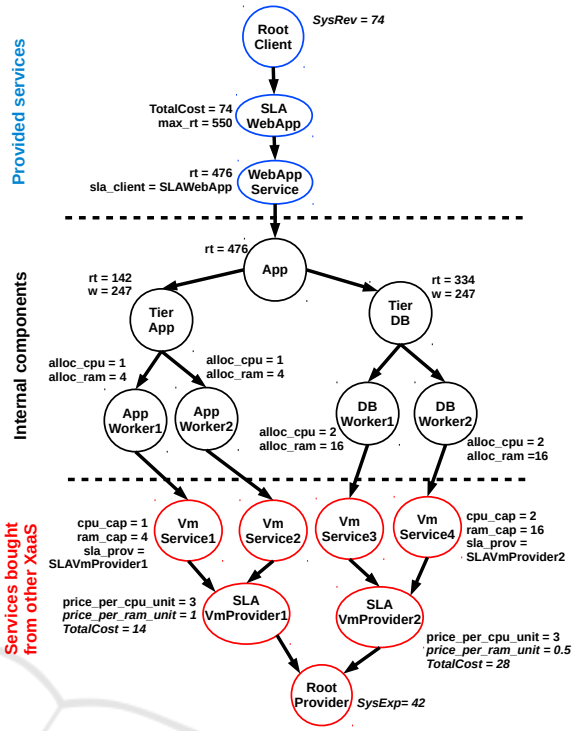


Figure 6: Example of a SaaS configuration.

4.2 Example of a SaaS Description

4.2.1 Provided Services

Also for the sake of simplicity and readability, we model a SaaS system so it provides a single Web Application service to its customers. It means, that there is a node type *WebAppService* extending *ServiceClient*. This node has two attributes corresponding to the current response time of the provided service and the node that defines SLA which the service provider and the client signed for. Customers can specify the maximum required response time in each corresponding *SLAWebApp* node, which extend the *SLAClient* type from the abstract model. The service price is defined as a utility function of the overall response time, that is to say that the price charged to customers is inversely proportional to the response time (in this case $max\_rt - rt$) and is also defined within node *SLAWebApp*. It should be noticed that prices may vary according to the client, i.e., according to the way *SLAWebApp* is defined for each client.

4.2.2 Internal Components

The web application is architecturally structured in tiers, and each tier is composed of workers that can be activated or deactivated to cope with workload varia-

tions while minimizing costs. That way, we define three *InternalComponent* nodes:

- the type *App* represents the web application itself and it has an attribute that defines current application overall response time. There is a constraint in *App* stating that the value of the response time is equal to the value of the response time of node *WebAppService*. Each *App* has two or more successor nodes of type *Tier* (in this case *TierApp* and *TierDB*). The *App* response time is calculated based on the sum of the response times of all its successors.
- the type *Tier* has also one or several successors of type *Worker* and two attributes: the income workload, which can be given as input to the model (i.e., monitored from the running system); and the tier response time, which is calculated based on the workload attribute and the amount of resources allocated to each worker associated to the concerned *Tier*. More precisely, we define the response time as a function of the amount of CPU and RAM currently allocated to the successor *Worker* nodes.
- the type *Worker* represents a replicated component of a given tier (e.g., application, database, etc.). It has three attributes corresponding to the currently allocated CPU and RAM; and specifying precisely which tier the worker belongs to so as to avoid the constraint solver to link a worker to a different tier (e.g., *AppWorker1* to *TierDB*).

4.2.3 Services Bought from Other providers

Each worker depends on compute/storage resources that are offered in terms of VMs by a VM provider. We define the node *VmService* by extending the *ServiceProvider* type of the abstract model. It corresponds to a VM offered by an IaaS provider. This node type consists of two attributes representing the CPU and RAM capacities and one attribute precising to which SLA Provider the service is associated to. Finally, the *SLAVmProvider* node extends the *SLAProvider* type from the abstract model and it corresponds to the signed SLA with the IaaS provider. This SLA specifies the price per unit of compute resources bought/rented (in terms of VM) by the SaaS.

5 PERFORMANCE EVALUATION

In this section, we present an experimental study of an implementation of our generic AM for an IaaS system modeled as the one depicted in the Figure 5. The main objective of our study is to analyse qualitatively the impact of the AM behaviour on the system configuration when a given series of events occur and the analysis time of the constraint solver to take a decision.

5.1 Experimental Testbed

We implemented the *Analysis* component of the AM by using the Java-based constraint solver Choco (Prud'homme et al., 2014). The experimentation simulates the interaction with the real world, i.e., the role of the components *Monitor* and *Executor* depicted in Figure 1. This 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.

We rely on the XML language to specify the initial configuration of our IaaS system. The snapshot of the running IaaS system configuration (the initial as well as the ones associated to each instant $t \in T$) is stored in a file. At each simulated event, this file was modified to apply 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. By trying to maximize the system balance between costs and revenues and to minimize the reconfiguration time, the AM produces a reconfiguration plan and generates then a new configuration file.

The simulated IaaS system is composed of 3 clusters physical homogeneous machines (PM). Each physical machine has 32 processors and 64 GB of RAM memory. The system has two power providers: a classical power provider, that is, brown energy power provider and a green energy power provider. The current consumption of a turned on PM is the sum of its idle power consumption (40 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 analysis performance by considering too much physical resources compared to the number of consummed virtual resources, we limit the number of unused PM nodes in the graph while ensuring a sufficient amount of available physical resources to host a potential new VM.

In the experiments, we considered four types of event:

- *AddVMService*: a customer requests for a new *VMService*. The required configuration of this request (i.e., the number of CPUs and RAM units) 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 a *VMService* node and a VM node in the configuration file. The aim of the AM after this event is to enable the new VM and to find the best PM to host it.

- *leavingClient*: a customer decides to cancel definitively the SLA. Consequently, the corresponding *SLAVM*, *VMService* and VM nodes are removed from the configuration. After a 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*: 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 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.
- *CrashOnePM*: a PM crashes. The consequence on the configuration is the suppression of the corresponding PM node. The goal of the AM is to potentially turn on a new PM and to migrate VM which was hosted by the broken PM.

In our experiments, we consider the following scenario. 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 considerably higher than the price of the other provider (70 against 5). The system has four clients which requests VM services. The number of requested services per client is not necessary equal. The unit selling price is 50 for a CPU and 10 for a RAM unit. We first consider a sequence of several *AddVMService* events until having around 40 *VMService* nodes. Then, we trigger a *leavingClient* event, a *GreenAvailable* event and finally a *CrashOnePM* event.

We shows the impact of this scenario over the following metrics: the amount of power consumption for each Power Provider in the Figure 7(a); the amount of enabled PMs and *VMService* in the Figure 7(b); and the configuration balance (function $\mathcal{H}()$) in the Figure 7(c). The x-axis in Figures 7(a), 7(b) and 7(c), 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 sake of readability, the x-axis does not begin at the

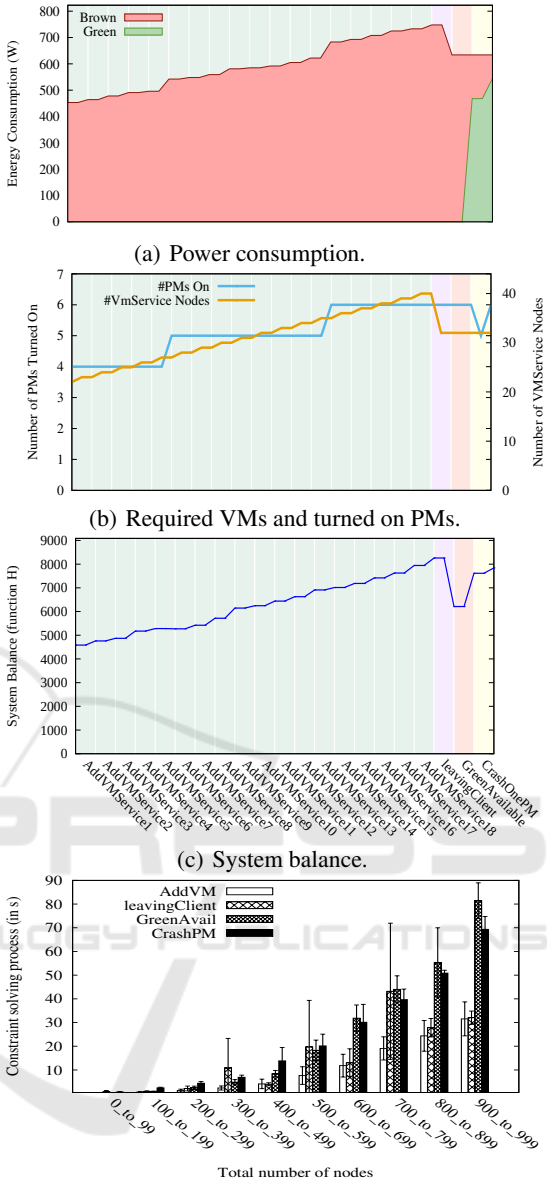


Figure 7: Experimental results of the simulation.

initiation instant but when the number of *VMService* reaches 20. In figure 7(d), we show the time of the Choco Solver to take a decision according to the number of nodes in the graph. Actually, while the experiment of figures 7(a), 7(b) and 7(c) considers a size of 0 to 99 nodes, we replay the same scenario of events described above until reaching around 1000 nodes.

5.2 Analysis and Discussion

As expected, when the amount of requests of *VMService* increases in a regular basis (Figure 7(b)), the system power consumption increases (Figure 7(a))

sufficiently slowly so that the system balance also increases (Figure 7(c)). 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. Indeed, we can see at the fifth *AddVMService* event that starting a new PM can be costly (especially when the new VM is small in terms of resources), since the balance does not increase after this event, which would be the expected outcome after selling new services (VMs in this case).

On the other way around, when a client leaves the system, as expected, the number of *VMService* nodes decreases (from 40 to 32). In spite of that, the power consumption also decreases from 748 to 634 (around 15%) due to the amount of resources which are not used anymore, the decrease from 8260 to 6210 of the system balance is not proportional (around 24 %). In fact, we can see that the number of PMs is constant during this event and consequently, the power consumption is higher than at the previous instant, where the number of *VMService* nodes is the same (at the tenth *AddVMService* event). 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: the cost in terms of planning actions (in our case VM migrations) leading to the configuration at the tenth *AddWMService* event is too costly compared to the cost due to system balance loss.

When the *GreenAvailable* event occurs, we can observe the activation of the Green Energy Provider (cf. Figure 7(a)) and, as expected, an increase of the system balance. This shows that the AM is capable of adapting the choice of provided service according to their current price. Thus, the modeled XaaS can benefit from sales promotions offered by its providers.

Finally, when a PM (*CrashOnePM* event), we can see that the AM starts a new PM to replace the old one. Moreover, in order to optimize the system balance (Figure 7(c)), the new PM is started on a cluster that uses the green energy, i.e., the current cheapest energy.

In figure 7(d), we can see that the decision time globally increases with the system size while keeping the same order of magnitude. However, it is not regular according to the event type showing that the impact of each event is very variable. Indeed, the *AddWMService* event concerns the adding of a unique VM on a PM which explains the fact that it is the fastest processed event, contrary to the *CrashOnePM* event which concerns a cluster, several PMs and VMs to migrate leading to a decision on a larger scale. Moreover we can see a huge variance especially for the *leavingClient* event. This shows that its impact over the system reconfiguration is unpredictable. Indeed, it depends on several factors like the number of concerned VM and their locality on the PMs, leading thus to make sometimes costly consolidation operations. In spite of that, as shown in Figure 7(d), our constraint model is capable of managing systems with reasonable sizes (e.g., 1000 nodes), with acceptable solving time.

6 RELATED WORK

Model-driven Approach and Cloud Management. Recent work have proposed the use a Model-driven Engineering for engineering the Cloud services. Some for reusing existing deployment procedures (Mastelic et al., 2014), other for optimizing VM configuration (Dougherty et al., 2012) or managing multi-cloud applications (e.g., migrate some VMs from a IaaS to another that offers better performance) (Ardagna and al., 2012). These approaches typically focus on supporting either IaaS or PaaS configuration, but do not address SaaS layer nor cross-layer modelisation. StratusML provides a modeling framework and domain specific modeling language for cloud applications dealing with different layers to address the various cloud stakeholders concerns (Hamdaqa and Tahvildari, 2015). The OASIS TOSCA specification aims at enhancing the portability of cloud applications by defining a modeling language to describe the topology across heterogeneous clouds along with the processes for their orchestration (Brogi and Soldani, 2016). However, those approaches do not deal with autonomic management.

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 (Nyrén et al., 2011), which lacks a rigorous and formal specification as well as the concept of (re)configuration. To tackle these issues, the authors of (Merle et al., 2015) specify the OCCI Core Model with the Eclipse Modeling Framework (EMF)[3], whereas its static semantics is rigorously defined with the Object Constraint Language (OCL)[4]. A 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 that have to be done in order to meet those OCL invariants.

Another body of work propose ontologies (Dastjerdi et al., 2010) or a model-driven approach based

[3]https://eclipse.org/modeling/emf

[4]http://www.omg.org/spec/OCL

on Feature Models (FMs) (Quinton et al., 2013) to handle cloud variability and then manage and create Cloud configurations. 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. In (García-Galán et al., 2014), the authors rely on FMs to define the space of configurations along with user preferences and game theory as decision-making tool. While the work focuses on features that are selected in a multi-tenant context, our approach provides support for ensuring the selection of SLA-compliant configurations in a cross-layer manner, i.e., by considering the relationships between providers and consumers in a single model.

Generic Autonomic Manager. In (Mohamed et al., 2015), 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. These rules require expertise at each service level and is error-prone as the number of rules grows. In contrast, our generic autonomic manager is based on a declarative approach of consumer/provider relationships and – thanks to a constraint solver – it is capable of controlling the target XaaS system so as to keep it close to the optimal configuration.

In (Ferry et al., 2014), the authors propose a support for management of multi-cloud applications for enacting the provisioning, deployment and adaptation of these applications. Their solution is based on a models@run-time (Blair et al., 2009) engine which is very close to our autonomic manager (with a reasoning in a cloud provider-agnostic way and a diff between the current and the target configuration). However, the authors focus on the IaaS or PaaS levels, but do not address SaaS, nor the relationships between layers.

Relationships between Cloud layers are addressed in (Marquezan et al., 2014) where the authors propose a conceptual model to represent the entities and relationships inside the cloud environment that are related to adaptation. They identify relationships among the cloud entities and dependencies among adaptation actions. However, their proposal is only an early work without a formal representation neither implementation.

In (Kounev et al., 2016), the authors propose a generic control loop to fit the requirements of their model-based adaptation approach based on an architecture-level modeling language (named Descartes) for quality-of-service and resource management. Their solution is very generic and do not focus specifically on cross-layers SLA contracts.

SLA-based Resource Provisioning and Constraint Solver. Several approaches on SLA-based resource provisioning – and based on constraint solvers – have been proposed. Like in our approach, the authors of (Hermenier et al., 2009) rely on the Choco solver, but their focus remains on the IaaS infrastructure, and more precisely on VM migration. In (Ghanbari et al., 2012), 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.

7 CONCLUSION AND FUTURE WORK

This paper presented a generic and abstract service-based model that unifies the main characteristics and objectives of Cloud services: finding an optimal balance between costs and revenues while meeting constraints regarding the established Service Level Agreements and the service itself. This model enabled us to derive a unique and generic Autonomic Manager (AM) capable of managing any Cloud service, regardless of the layer. From the Cloud Administrators point of view, this is an interesting contribution, not only because frees them from the difficult task of conceiving and implementing purpose-specific AMs, but also because the proposed model, although generic and abstract, is extensible. The generic AM relies on a constraint solver that reasons on very abstract concepts (e.g., nodes, relations, constraints) to perform the analysis phase in a MAPE-K loop. We showed the genericity of the abstract model by illustrating two possible implementations: a IaaS and a SaaS systems. The IaaS implementation was evaluated experimentally, with a qualitative study and the results show that the AM is able to adapt the configuration accordingly by taking into account the established SLAs and the reconfiguration costs. Further, results show that although generic, the AM can

capture the specificities and runtime properties of the modeled Cloud service.

As an on-going work, we are currently improving the constraint resolution model so we can have better performance in terms of decision-making. Also, we are implementing a real IaaS AM on top of OpenStack [5] and evaluating it [6]. For future work, we plan to tackle issues related to the coordination of many inter-related AMs, which may cause problems of conflicting actions and other synchronization issues that come with (Alvares de Oliveira et al., 2012). Finally, we plan also to provide full Domain Specific-Language (DSLs) (van Deursen et al., 2000) and tooling support allowing Administrators for a clearer, easier and more expressive description of XaaS models.

REFERENCES

Alvares de Oliveira, F., Sharrock, R., and Ledoux, T. (2012). Synchronization of multiple autonomic control loops: Application to cloud computing. In *Proceedings of the 14th Int. Conf. on Coordination Models and Languages (Coordination)*, pages 29–43. Springer-Verlag.

Ardagna, D. and al. (2012). Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *4th Int. Workshop on Modeling in Software Engineering*, pages 50–56.

Blair, G., Bencomo, N., and France, R. B. (2009). Models@ run.time. *Computer*, 42(10):22–27.

Brogi, A. and Soldani, J. (2016). Finding available services in tosca-compliant clouds. *Science of Computer Programming*, 115–116:177 – 198.

Dastjerdi, A., Tabatabaei, S., and Buyya, R. (2010). An effective architecture for automated appliance management system applying ontology-based cloud discovery. In *CCGrid 2010*, pages 104–112.

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

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 *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 *SEAMS 2014*, SEAMS 2014, pages 65–74, New York, NY, USA. ACM.

Ghanbari, H., Simmons, B., Litoiu, M., Barna, C., and Iszlai, G. (2012). Optimal autoscaling in a iaas cloud. In *ICAC 2012*, pages 173–178. ACM.

Hamdaqa, M. and Tahvildari, L. (2015). Stratus ml: A layered cloud modeling framework. In *2015 IEEE International Conference on Cloud Engineering*, pages 96–105.

Hermenier, F., Lorca, X., Menaud, J.-M., Muller, G., and Lawall, J. (2009). Entropy: A consolidation manager for clusters. In *VEE 2009*, pages 41–50.

Hogan, M. and al. (2011). Nist cloud computing standards roadmap, version 1.0.

Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Kouki, Y. and Ledoux, T. (2012). Csla: a language for improving cloud sla management. In *Int. Conf. on Cloud Computing and Services Science, CLOSER 2012*, pages 586–591.

Kounev, S., Huber, N., Brosig, F., and Zhu, X. (2016). A model-based approach to designing self-aware it systems and infrastructures. *Computer*, 49(7):53–61.

Marquezan, C. C., Wessling, F., Metzger, A., Pohl, K., Woods, C., and Wallbom, K. (2014). Towards exploiting the full adaptation potential of cloud applications. In *PESOS 2014*, pages 48–57.

Mastelic, T., Brandic, I., and Garcia Garcia, A. (2014). Towards uniform management of cloud services by applying model-driven development. In *COMPSAC 2014*, pages 129–138.

Merle, P., Barais, O., Parpaillon, J., Plouzeau, N., and Tata, S. (2015). A precise metamodel for open cloud computing interface. In *CLOUD 2015*, pages 852–859.

Mohamed, M., Amziani, M., Belaïd, D., Tata, S., and Melliti, T. (2015). An autonomic approach to manage elasticity of business processes in the cloud. *FGCS*, 50:49 – 61.

Nyrén, R., Edmonds, A., Papaspyrou, A., and Metsch, T. (2011). Open cloud computing interface - core, specification document. Technical report, Open Grid Forum, OCCI-WG.

Prud'homme, C., Fages, J.-G., and Lorca, X. (2014). *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.

Quinton, C., Haderer, N., Rouvoy, R., and Duchien, L. (2013). Towards multi-cloud configurations using feature models and ontologies. In *Int. Workshop on Multi-cloud Applications and Federated Clouds*, pages 21–26.

Rossi, F., van Beek, P., and Walsh, T., editors (2006). *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA.

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

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36.

[5] https://www.openstack.org/

[6] https://www.grid5000.fr/