Formalizing and simulating cross-layer elasticity strategies in Cloud systems

Khaled Khebbeb, Nabil Hameurlain, Faiza Belala

► **To cite this version:**

Khaled Khebbeb, Nabil Hameurlain, Faiza Belala. Formalizing and simulating cross-layer elasticity strategies in Cloud systems. Cluster Computing, 2020, 10.1007/s10586-020-03080-8 . hal-02525324

HAL Id: hal-02525324
https://univ-pau.hal.science/hal-02525324

Submitted on 30 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

This document shows the post-print version of an article in **Cluster Computing** journal (**Springer**)
The final article is available online since 27 March 2020.
DOI: https://doi.org/10.1007/s10586-020-03080-8

Formalizing and simulating cross-layer elasticity strategies in Cloud systems

Khaled Khebbeb · Nabil Hameurlain · Faiza Belala

Received: 30 April 2019 / **Revised**: 15 January 2020 / **Accepted**: 29 February 2020 / **Published**: 27 March 2020

Abstract Clouds are complex systems that provide computing resources in an elastic way. Elasticity allows their adaptation to input workloads by (de)provisioning resources as the demand rises and drops. Given the numerous overlapping factors that impact their elasticity and the unpredictable nature of the workload, providing accurate action plans to manage Cloud elasticity is a particularly challenging task. In this paper, we propose a formal approach based on Bigraphical Reactive Systems (BRS) to model Cloud structures and their elastic behavior. We design cross-layer elasticity strategies which operate at application and infrastructure Cloud layers to manage the elastic adaptations. We encode the elastic behaviors in Rewriting logic, through the Maude framework, to enable their generic executability. We provide a qualitative verification of the designed behaviors' correctness with a model-checking technique supported by the Linear Temporal Logic (LTL). Finally, we provide a tooled simulation-based methodology, through the Queuing theory, to conduct a quantitative analysis of the designed elasticity strategies.

Keywords Self-adaptation · Cloud computing · Cross-layer elasticity strategies · Formal methods · Bigraphical Reactive Systems · Rewriting logic · Maude · Linear Temporal Logic · Queuing theory

Khaled Khebbeb (✉)
LIUPPA Laboratory - University of Pau, France
E-mail: khaled.khebbeb@univ-pau.fr

Nabil Hameurlain
LIUPPA Laboratory - University of Pau, France
E-mail: nabil.hameurlain@univ-pau.fr

Faiza Belala
LIRE Laboratory - Constantine 2 University, Algeria
E-mail: faiza.belala@univ-constantine2.dz

1 Introduction

Cloud computing [32] is a recent paradigm that has known a great interest in both industrial and academic sectors. It consists of providing a pool of virtualized resources (servers, virtual machines, etc.) as on-demand services. These resources are offered by Cloud providers according to three fundamental service models: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). The most appealing feature that distinguishes the Cloud from other models is the elasticity property [22, 2, 18, 23]. Elasticity allows to efficiently control resources provisioning according to workload fluctuation in a way to maintain an adequate quality of service (QoS) while minimizing operating cost [15]. Such a behavior is implemented by an elasticity controller: an entity usually based on a closed control loop which decides of the elasticity actions to be triggered to adapt to the demand [25]. In fact, managing a Cloud system's elasticity can be particularly challenging. Elastic behaviors rely on many overlapping factors such as the available resources, current workload, the system's state of provisioning, etc. Managing these dependencies significantly increases the difficulty of modeling Cloud systems' elasticity controller. To address this challenge, formal methods characterized by their efficiency, reliability and precision, present an effective solution to deal with these numerous factors.

In this paper, we contribute to the design of elastic behaviors in the context of Cloud systems. We present a complete approach for formal modeling, qualitative verification and quantitative analysis of Cloud elasticity basing on formal models and mathematical theories. Figure 1 gives a global view of our solution. In terms of modeling, we adopt Bigraphical Reactive Systems (BRS) [34] as a meta-model for specifying struc-

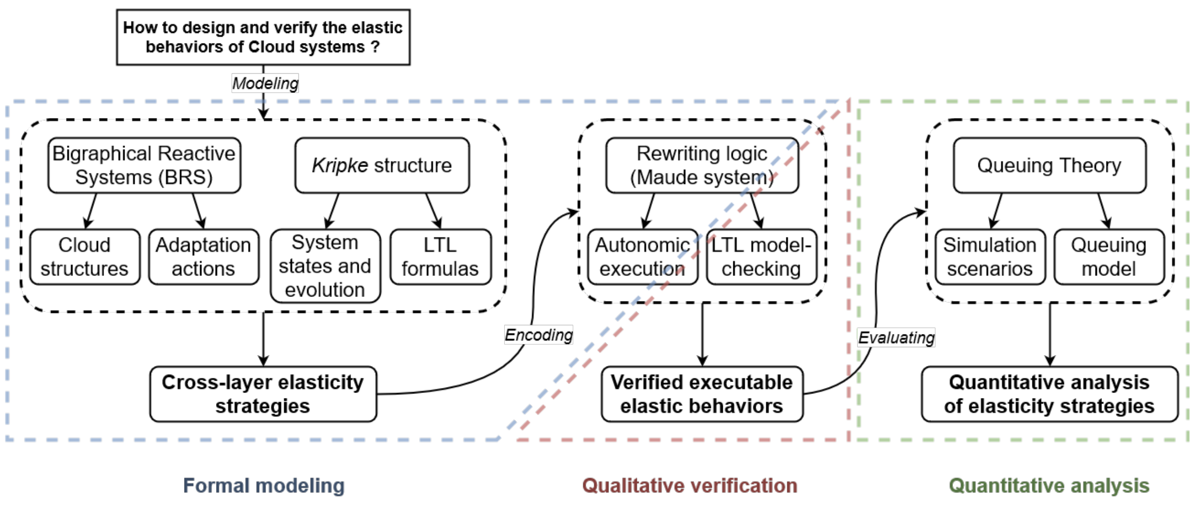

How to design and verify the elastic behaviors of Cloud systems ?
Modeling
Bigraphical Reactive Systems (BRS)
Kripke structure
Cloud structures
Adaptation actions
System states and evolution
LTL formulas
Cross-layer elasticity strategies
Encoding
Rewriting logic (Maude system)
Autonomic execution
LTL model-checking
Verified executable elastic behaviors
Evaluating
Queuing Theory
Simulation scenarios
Queuing model
Quantitative analysis of elasticity strategies
Formal modeling
Qualitative verification
Quantitative analysis

Fig. 1 A top view of our solution for formal modeling, qualitative verification and quantitative analysis of Cloud elasticity

tural and behavioral aspects of elastic Cloud systems. Bigraphs are used to model the structure of Cloud systems and the elasticity controller. Bigraphical reaction rules describe the elastic behavior of a Cloud system. As we aim at providing a generic solution, we focus on the infrastructure (IaaS) and application (SaaS) levels to define reactive elasticity strategies for provisioning and deprovisioning Cloud resources in a cross-layered way. A strategy provides a logic that governs resources provisioning. It enables the elasticity controller to manage the Cloud system's elastic behavior. It consists of a set of actions (bigraphical reaction rules) that are triggered according to the specified conditions. We design reactive strategies taht take the form: IF Condition(s) THEN Action(s). To describe the system's desirable states and evolution over time, we use a *Kripke* structure to define the desired behaviors as linear temporal logic (LTL) formulas.

Furthermore, we turn to Maude [11] as a semantic framework to encode the BRS modeling approach and to provide a generic executable solution of Cloud elastic behavior. Maude is a formal tool environment based on rewriting logic. It can be used as a declarative and executable formal specification language, and as a formal verification system. It provides good representation and verification capabilities for a wide range of systems including models for concurrency. Maude enables us to easily map the BRS specifications into Maude modules and to manage the non-determinism that characterizes a Cloud system's elastic behavior. In addition, Maude allows encoding the defined *Kripke* structure and LTL formulas. This enables conducting a LTL state-based model-checking technique to verify the introduced behaviors' qualitative correctness.

In order to illustrate the designed cross-layer strategies, it is important to analyze their induced behaviors in a quantitative point of view, i.e., in terms of cost, performance and efficiency [42]. This task is not trivial as it requires monitoring the system during its runtime to watch and control its adaptation [1]. Considering the fluctuating and unpredictable nature of its input workload, we propose a tooled queuing-based approach as an analytic support for simulating, monitoring and analyzing a Cloud system's elastic behavior. Precisely, we conduct an experimental case study on an existing Cloud-based service, through multiple execution scenarios, to provide a quantitative analysis of the introduced cross-layer elasticity strategies. By this simulation-based evaluation, our goal is to illustrate our main approach's capabilities in terms of elastic behaviors together with their hypothetical use in real environments.

As part of our contributions, in this paper we extend [26] by defining vertical scale elasticity strategies (i.e., adding/removing computer resources to resize virtual machines). This implies extending the bigraphical specifications to consider computational resource pools (CPU, memory) for Cloud servers and VMs. We also extend the bigraphical reaction rules to consider the vertical scale elasticity. Note that these extensions imply extending the Maude-based definitions as well as the LTL formulas for qualitative verification. Furthermore, we detail our tool for monitoring and simulating elasticity. With this tool, we conduct a deeper quantitative analysis of all the defined strategies. We provide a deeper comparative study of their induced behaviors and high-level policies resulting from the different cross-layer elasticity strategies at infrastructure (IaaS) and application (SaaS) layers. Note that we omit the Plat-

DOI:https://doi.org/10.1007/s10586-020-03080-8

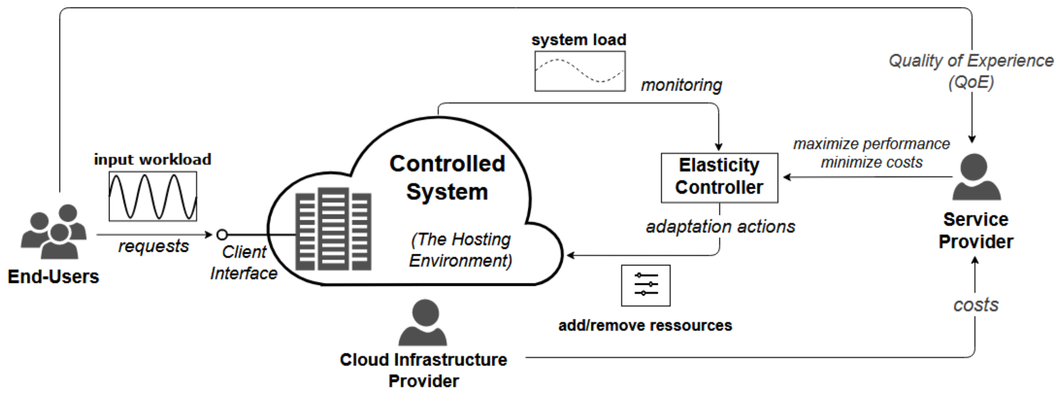


Fig. 2 High level view of Cloud systems' elastic behavior

form as a Service (PaaS) Cloud layer's elasticity [37] as it considers very specific and technology-dependent requirements including virtualization constraints (hypervisors types) for horizontal scaling, components upgrading and software aging. The solution we present in this paper tends to be more conceptual as it relies on formal methods and aims at reducing the design complexity and operating analysis.

The remainder of the paper is structured as follows. In Section 2, we present our vision of Cloud systems and explain how their elastic behavior is managed by the elasticity controller. In Section 3, we introduce and use BRS formalism to provide a modeling approach for Cloud systems. We model the elasticity controller, define elasticity strategies and describe the desirable behaviors with LTL. In Section 4, we encode the bigraphical specifications of elastic Cloud systems into Maude. In Section 5, we introduce our tooled queuing-based methodology and an experimental case study for the simulation and analysis of the defined elastic behaviors. In Section 6, we discuss some related works about formal specification of elasticity and about autonomic management and monitoring of elasticity. Finally, Section 7 summarizes and concludes the paper.

2 Cloud systems and elasticity

At a high level of abstraction, an elastic Cloud system can be divided in three parts: a front-end part, a back-end part and an elasticity controller. The front-end represents the client interface that is used to access the Cloud system and to interact with it. The back-end part refers to the Cloud system's hosting environment, i.e., the set of computing resources (servers, VMs: virtual machines, service instances, computing resources etc.) that are deployed in the system and that are provided to satisfy its incoming workload. Cloud systems offer their computing resources in an elastic way. Elasticity is a property that was defined as

"the degree in which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that, at each point in time, the available resources match the current demand as closely as possible." [22].

The main goal of elasticity is to avoid the system's *over-provisioning* and *under-provisioning* states, respectively categorized by too much or not enough provisioned resources to cope with the current demand. Elastic Cloud systems usually work according to a closed-loop architecture as shown in Figure 2, where the elastic Cloud system receives end-users' requests through its client interface. The amount of received requests (i.e., the input workload) can oscillate in an unpredictable manner. The growing workload, thus the system's load can cause users Quality of Experience (QoE) degradation (e.g. performance drop). A Cloud infrastructure provider hosts the controlled system (i.e., Cloud hosting environment). It provides costs to the Cloud service provider according to the provisioned resources (that are allocated to the service provider's running applications). When input workload drops, the eventually unnecessarily allocated resources are still billed. Elasticity controller monitors the controlled system and determines its adaptation (i.e., its elastic behavior). Adaptation actions (i.e., (de)provision Cloud resources) are triggered to satisfy high-level policies that are set by the service provider such as minimize costs, maximize performance, etc.

According to [18], we distinguish three methods of elasticity: horizontal scaling, vertical scaling and migration. Horizontal scaling consists of adding (*scale-out*) or removing (*scale-in*) Cloud resources (VMs, services) in order to adapt to the current demand, whereas vertical scaling consists of resizing VMs to allocate them with more (*scale-up*) or less (*scale-down*) computing resources (CPU, RAM). Figure 3 illustrates horizontal and vertical scaling. Horizontal scaling is closely linked to the notion of *load balancing* which consists of redirecting requests across services in order to balance the system's load and optimize the use of the deployed resources. Migration consists of relocating Cloud resources (VMs, services) to different hosts (servers, VMs) in order to optimize Cloud hosting environment's resources consumption.

The behavior of an elastic system can be intuitively described as follows. During its runtime, the system's load can increase, which might lead to overload the provisioned resources (the system is then *under-provisioned*). To avoid the saturation, an elastic system stretches, i.e., it scales by provisioning more computing resources. Conversely, when the system load decreases, some resources might become underused (leading the system to be *over-provisioned*). To reduce costs, the elastic system contracts, i.e., it scales by deprovisioning the unnecessarily allocated resources [8]. A system's elastic behavior is generally specified by elasticity strategies. Strategies gives the elasticity controller a logic that governs its decision making in order to satisfy the high-level policies, by triggering the suitable adaptation actions (according to the introduced elasticity methods), in an autonomic way.

Due to the complexity of Cloud systems and the multiplicity of the overlapping factors that impact their elasticity (i.e., input workload, available resources, logic that governs elasticity controller's behavior, etc.), specifying and implementing an elastic behavior is a particularly tedious task. In this paper, we address this challenge by relying on formal methods. We provide a BRS based modeling of Cloud systems' structure and the elasticity controller's behavior. We encode the proposed specification into Maude language to provide an executable solution of the elastic behaviors together with the verification of their correctness.

3 BRS-based specification of elastic Cloud systems

Bigraphical reactive systems (BRS) are a recent formalism introduced by Milner [34, 35], for modeling the temporal and spatial evolution of computation. It provides an algebraic model (and a graphical representation) which emphasizes both connectivity and locality via a link graph and a place graph respectively. A BRS consists of a set of bigraphs and a set of reaction rules, which define the dynamic evolution of a system by specifying how the set of bigraphs can be reconfigured.

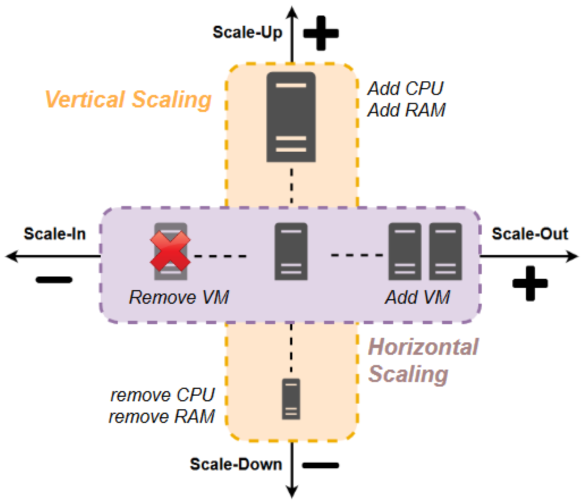


Fig. 3 Horizontal and vertical scaling at infrastructure level

3.1 Bigraphical modeling of Cloud systems

We model a Cloud system with a bigraph CS including all Cloud structural elements. We define a sorting logic to specify mapping rules and expresses all the constraints and construction rules, that CS needs to satisfy, to ensure proper and accurate encoding of the Cloud semantics into BRS concepts. Formal definitions are given in what follows.

Definition 1. Formally, a Cloud system is defined by a bigraph CS, where:

$$CS = (V_{CS}, E_{CS}, ctrl_{CS}, CS^P, CS^L) : I_{CS} \rightarrow J_{CS}$$

- V_{CS} and E_{CS} are sets of nodes and edges of the bigraph CS.
- $ctrl_{CS} : V_{CS} \rightarrow K_{CS}$ is a control map that assigns each node $v \in V_{CS}$ with a control $k \in K_{CS}$.
- $CS^P = (V_{CS}, ctrl_{CS}, prnt_{CS}) : m_{CS} \rightarrow n_{CS}$ is a parent map. m_{CS} and n_{CS} are the numbers of sites and regions of the bigraph CS.
- $CS^L = (V_{CS}, E_{CS}, ctrl_{CS}, link_{CS}) : X_{CS} \rightarrow Y_{CS}$ represents the link graph of CS, where $link_{CS} : X_{CS} \uplus P_{CS} \rightarrow E_{CS} \uplus Y_{CS}$ is a link map, X_{CS} and Y_{CS} are respectively inned and outer names, and P_{CS} is the set of ports of CS.
- $I_{CS} =< m_{CS}, X_{CS} >$ and $J_{CS} =< n_{CS}, Y_{CS} >$ are the inner and outer interfaces of the Cloud system bigraph CS.

Nodes V_{CS} represent the physical (servers, CPU and RAM units) or logical (VMs and service instances) elements of the Cloud system. Edges E_{CS} represent the links (e.g. communication canals) that connect the nodes

DOI:https://doi.org/10.1007/s10586-020-03080-8

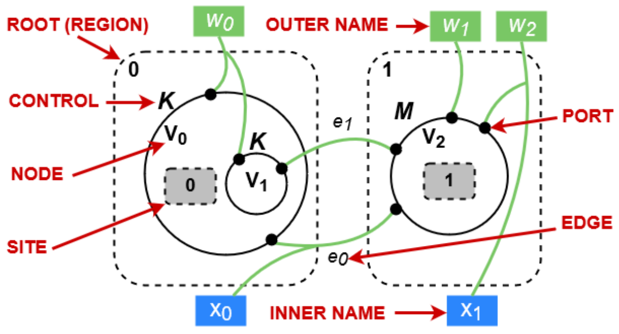


Fig. 4 The bigraph anatomy

via their ports P_{CS}. The control map $ctrl_{CS}$ associates semantics to the nodes. The place graph CS^P gives the hierarchical construction of the system basing on the parent map $prnt_{CS}$ for nodes and regions (e.g. a server node is a parent for a VM node, or hosts is). Regions represent the different parts of the system (e.g. the hosting environment). Sites are used to neglect parts of the system that are not included in the model. The link graph CS^L gives the link map $link_{CS}$ which shows all the connections between ports and names. Inner and outer interfaces I_{CS} and J_{CS} give the openness of the system to its external environment (other bigraphs). Inner and outer names X_{CS} and Y_{CS} give labels to different parts of the system for interfacing purposes. Figure 4 gives the bigraph anatomy and identifies graphically all the above-mentioned elements.

Definition 2. The sorting discipline associated to CS is a triple $\Sigma_{CS} = \{\Theta_{CS}, K_{CS}, \Phi_{CS}\}$. Where Θ_{CS} is a non-empty set of sorts. K_{CS} is its signature, and Φ_{CS} is a set of formation rules associated to the bigraph CS.

Table 1 gives for each Cloud concept the mapping rules for BRS equivalence. It consists of the control associated to the entity, its arity (number of ports) and its associated sort. Sorts are used to distinguish node types for structural constraints while controls identify states and parameters a node can have. For instance, a server noted SE has control SE^L when it is overloaded and SE^U when unused but all nodes representing servers are of sort e. Similarly, pools of computing resources have the same structure and are of sort r. However, we distinguish two types of resource pools: those indicating available resources at server level (through control RA) and those allocated to a VM (through control RV).

Table 2 gives the construction rules that define construction constraints over the bigraphical model. Rule Φ_0 specifies that servers are at the top of the hierarchical order of the deployed entities in the bigraph. Rules $\Phi1-4$ give the structural disposition of the Cloud hosting environment where a server hosts VMs, a VM runs service instances and a service instance handles requests. In addition, rules specify that resource pools are hosted by servers or VMs, and host CPU and RAM units. All connections are port-to-port links to illustrate possible links between the different Cloud entities. In $\Phi7-8$, we use the name w (for workload) to illustrate the connection the Cloud system has with its abstracted front-end part. A server is linked to its hosted VMs and available resource pool, and a VM is linked to the service instances it is running and its allocated resource pool [27]. Rule $\Phi4$ gives the active and passive elements, i.e., that may take part in reactions or not,

Table 1 The sorting discipline of the bigraph CS

Cloud element	Control	Arity	Sort
Physical machine layer			
Server	SE		
Overloaded server	SE^L	3	e
Unused server	SE^U		
Pool of available resources	RA	1	r
Virtual machine layer			
Virtual machine	VM		
Overloaded VM	VM^L	3	v
Overprovisioned VM	VM^P		
Unused VM	VM^U		
Pool of allocated resources	RV	1	r
Application layer			
Service instance	S		
Overloaded service	S^L	1	s
Unused service	S^U		
Request	q	0	r
Computing resources layer			
CPU unit	CU	0	c
RAM unit	M	0	m

Table 2 Construction rules Φ_{CS} of the bigraph CS

	Rule description
$\Phi0$	All children of a 0-region (hosting environement) have sort e
$\Phi1$	All children of a e-node have sort $\widehat{vr}$
$\Phi2$	All children of a v-node have sort $\widehat{sr}$
$\Phi3$	All children of a s-node have sort q
$\Phi4$	All children of a r-node have sort $\widehat{cm}$
$\Phi5$	All $\widehat{qcm}$-nodes are atomic
$\Phi6$	All $\widehat{evsqcm}$-nodes are active are r-nodes are passive
$\Phi7$	In a e-node, one port is always linked to a w-name, one port is always linked to the child r-node, and the other may be linked to children v-nodes
$\Phi8$	In a v-node, one port is always linked to a parent e-node, one port is always linked to a child r-node, and the other may be linked to children s-nodes

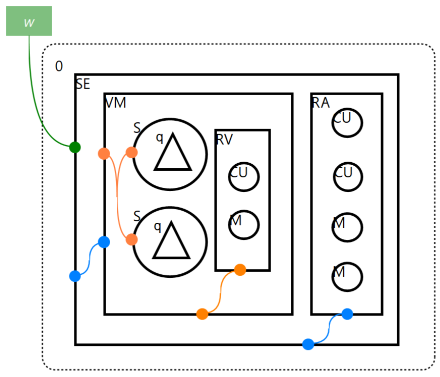


Fig. 5 Example of a bigraph CS modeling a Cloud system

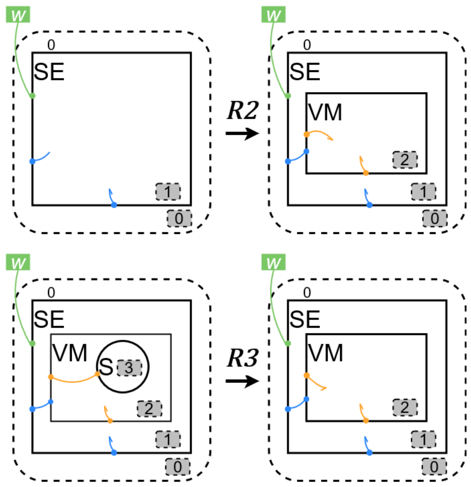


Fig. 6 Graphical notation of rules R2 and R3

respectively. Finally, rule $\Phi5$ gives the atomic entities, i.e., that do not host other nodes. We use a disjunctive notation to indicate that a node can be of different sorts. For example, a $\widehat{ab}$-node can be of sort a or sort b.

Example. Let's consider an arbitrary Cloud system where a physical server (SE node) is online and one VM (VM node) is deployed. Two service instances (S nodes) are running inside the VM and each service is handling one request (q nodes). The physical server has two CPU (CU nodes) and two RAM (M nodes) units in its available resource pool (RA node). The VM is allocated one unit of CPU and one unit of RAM into its allocated resource pool (RV node). Such a system can be modeled with a bigraph CS, according to the introduced bigraphical semantics, as shown in Figure 5. The shown bigraph CS has an interface $< 0, \emptyset > \rightarrow < 1, w >$ indicating that is contains 0 sites and 1 region, and that it has an outer name w and no inner names.

3.2 The elasticity controller as a behavioral entity

The elasticity controller determines the adaptations of the Cloud system's hosting environment. In our modeling approach, we consider this entity as (1) the set of adaptation actions that describe the system's behavior and (2) the logic that governs the rules' triggering. The adaptation actions are expressed as bigraphical reaction rules and the triggering logic is expressed as strategies that describe how different adaptations of a Cloud system are achieved in a *cross-layered* manner (i.e., at infrastructure and application Cloud levels).

3.2.1 Bigraphical reaction rules to model Cloud adaptation actions

In this section, we show how different Cloud elasticity adaptations can be expressed as bigraph structural rewriting. A bigraphical reaction rule Ri is a pair (B, B'), where *redex* B and *reactum* B' are bigraphs that respect the same sorting discipline and construction rules. The evolution of a given Cloud bigraph CS is derived by checking if B is a match (or occurs) in CS and by substituting it with B' to obtain a new system configuration CS'. This is made by triggering the corresponding reaction rule Ri. The evolution is noted $CS \overset{\text{Ri}}{\rightarrow} CS'$. In other words, when a rule Ri is triggered, a Cloud bigraph CS, on the left-hand side of the rule is rewritten to the right-hand side of the rule as a bigraph CS'. Note that a reaction rule produces a bigraph which is correct by definition, with respect to the specified structural constraints.

Table 3 gives the algebraic description of the different reaction rules that implement the adaptation actions of the elasticity controller. Sites (expressed as d) are used to neglect the elements that are not included in the reaction. The specified rules define horizontal and vertical scale elasticity actions together with load-balancing and migration actions at different Cloud levels. Reaction rules are applied for provisioning (adding) and deprovisioning (removing) resources by scaling-out/up ($R1 - R2/R5 - R6$) and scaling-in/down ($R3 - R4/R7-R8$) the hosting environment at infrastructure, application and resources levels. Rules ($R9-R10$) specify migration and load-balancing actions at application and infrastructure levels, and are used to balance the system's load.

Reaction rules graphical notation. To illustrate the defined bigraphical reaction rules' semantics. We give some example of the key features they provide using their graphical notations. Figure 6 illustrates the rule $R2$ (adding a new VM instance) and the rule $R3$ (removing a service instance) where a VM is deployed

Table 3 Reaction rules describing adaptation actions

Adaptation action	Reaction rule algebraic form
Scale-Out	
Deploy service instance	$R1 \overset{\text{def}}{=} SE.(VM.d2 \mid d1) \mid id \rightarrow SE.((VM.(S.d3) \mid d2) \mid d1) \mid id$
Deploy VM instance	$R2 \overset{\text{def}}{=} SE.d1 \mid id \rightarrow SE.((VM.d2) \mid d1) \mid id$
Scale-In	
Consolidate service instance	$R3 \overset{\text{def}}{=} SE.((VM.(S.d3) \mid d2) \mid d1) \mid id \rightarrow SE.(VM.d2 \mid d1) \mid id$
Consolidate VM instance	$R4 \overset{\text{def}}{=} SE.((VM.d1) \mid d1) \mid id \rightarrow SE.d1 \mid id$
Scale-Up	
Add CPU unit to a VM	$R5 \overset{\text{def}}{=} SE.((RA.CU \mid d4) \mid (VM.(RV.d3) \mid d2) \mid d1) \mid id \rightarrow SE.((RA.d4) \mid (VM.(RV.CU \mid d3) \mid d2) \mid d1) \mid id$
Add RAM unit to a VM	$R6 \overset{\text{def}}{=} SE.((RA.M \mid d4) \mid (VM.(RV.d3) \mid d2) \mid d1) \mid id \rightarrow SE.((RA.d4) \mid (VM.(RV.M \mid d3) \mid d2) \mid d1) \mid id$
Scale-Down	
Free CPU unit from a VM	$R7 \overset{\text{def}}{=} SE.((RA.d4) \mid (VM.(RV.CU \mid d3) \mid d2) \mid d1) \mid id \rightarrow SE.((RA.CU \mid d4) \mid (VM.(RV.d3) \mid d2) \mid d1) \mid id$
Free RAM unit from a VM	$R8 \overset{\text{def}}{=} SE.((RA.d4) \mid (VM.(RV.M \mid d3) \mid d2) \mid d1) \mid id \rightarrow SE.((RA.M \mid d4) \mid (VM.(RV.d3) \mid d2) \mid d1) \mid id$
Migration	
Migrate service instance	$R9 \overset{\text{def}}{=} SE.(((VM.(S.d4) \mid d3) \mid (VM.d2)) \mid d1) \mid id \rightarrow SE.((VM.d3) \mid (VM.(S.d4) \mid d2) \mid d1) \mid id$
Load Balancing	
Transfer request	$R10 \overset{\text{def}}{=} SE.((VM.(S.q \mid d4) \mid (S.d3) \mid d2) \mid d1) \mid id \rightarrow SE.((VM.(S.d4) \mid (S.q \mid d3) \mid d2) \mid d1) \mid id$

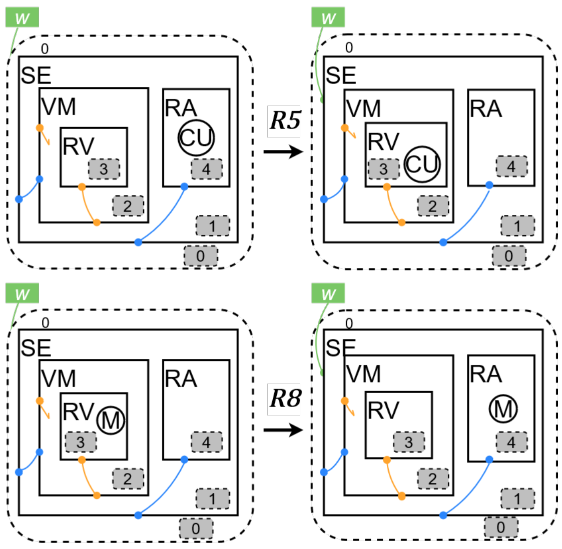


Fig. 7 Graphical notation of rules R5 and R8

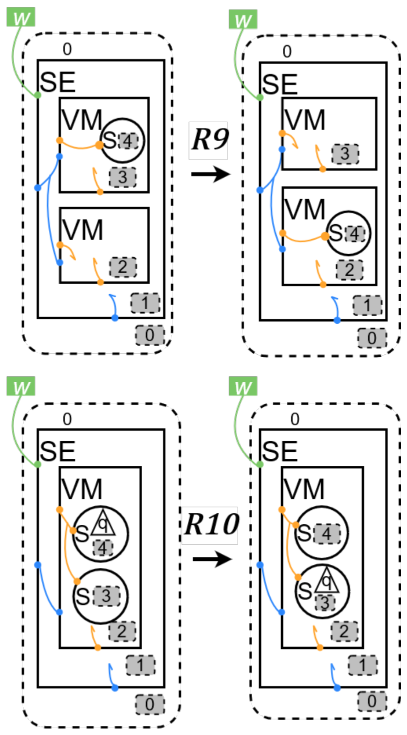


Fig. 8 Graphical notation of rules R9 and R10

on a physical server and a service instance is removed from a VM respectively. Figure 7 illustrates the rule $R5$ (adding a CPU unit to a VM) and the rule $R8$ (removing a RAM unit from a VM) where a CPU unit, respectively a RAM unit is allocated/freed to/from a VM instance. Figure 8 illustrates the rule $R9$ (migrating a service instance to another VM) and the rule $R10$ (trans-

DOI:https://doi.org/10.1007/s10586-020-03080-8

ferring a request to another service instance) where a service instance, respectively a request, is moved from a VM, respectively a service instance to another one.

Notice that these all the expressed entities (physical server, virtual machine, service instance, resource pools, CPU and RAM units) are expressed as nodes of the corresponding control, as defined by the sorting discipline in Table 1. In addition, the displayed (re)configurations obey to the construction rules (e.g., VM nodes are inside server nodes, service nodes are inside VM nodes, how all nodes are linked, etc.) as defined in defined in Table 2. In the presented examples, sites are used to abstract away elements that do not take part in the modeling.

Now that all the possible adaptation actions (i.e., reaction rules) are identified, we introduce elasticity strategies to define a logic expressing conditions about when, how and where adaptations are triggered.

3.2.2 Cross-layer elasticity strategies

As explained before, the specified strategies define a logic that governs the elastic behavior of the controlled Cloud system. We use reactive strategies to make decisions about the elastic adaptations of the deployed entities by reasoning on their states. A reactive strategy takes the form: *IF Condition(s) THEN Action(s).* Table 4 gathers and formally defines our proposed elasticity strategies for horizontal scaling, vertical scaling, migration and load balancing. For each strategy, the table gives its Cloud levels of action, its triggering condition and its triggered action. Conditions are expressed in first-order logic and actions are reaction rules. We informally explain the strategies in the following.

Horizontal scale strategies. In order to control the triggering of the different horizontal scale actions (reaction rules $R1 - R4$), we define the strategies *H_Out1*, *H_Out2* and *H_In* for (de)provisioning a Cloud's hosting environment resources at application and infrastructure levels.

H_Out1 (Scale-Out: High availability). This strategy ensures high Cloud resources availability. It triggers reaction rules to provision Cloud resources as follows.

- *Application level:* a new instance of service is provisioned by executing rule $R1$, when at least one available instance is *overloaded* (i.e., when it has control S^L) and no other service instance is *unused* (control S^U).
- *Infrastructure level:* a new VM instance is provisioned by executing rule $R2$, when at least one VM is *overloaded* (i.e., when it has control VM^O) and no other VM is *unused* (control VM^U).

H_Out2 (Scale-Out: Limited availability). Constraints Cloud resources to a limited availability as follows.

- *Application level:* a new instance of service is provisioned by executing rule $R1$, when all available service instances are overloaded.
- *Infrastructure level:* a new VM instance is provisioned by executing rule $R2$, when all available VMs are overloaded.

H_in (Scale-in). Describes how the Cloud hosting environment deprovisions resources as follows.

- *Application level:* the system deprovisions an empty service instance (which has control S^U) by executing rule $R3$, when one is detected, and no overloaded instance is available.
- *Infrastructure level:* an empty VM instance (of control VM^U) is deprovisioned by executing rule $R4$, if one is detected and no overloaded VM is available.

Note that the choice of deprovisioning an empty instance (VM/service) only when no overloaded one is detected is made to avoid loops in the elastic behavior (i.e., deleting an instance then provision another one right after).

Vertical scale strategies. Vertical scale elasticity consists of adapting a VM offering in terms of adding/removing processing (CPU) and memory (RAM) resources (reaction rules $R5 - R8$). Adapting a VM's offering depends on the amount of available resources at the hosting physical server. We introduce two functions *cpu(x)* and *ram(x)* which give the amount of available CPU and RAM units for a given Server or VM as a parameter x. We introduce two vertical scale strategies *V_Up* and *V_Down* for adding (*Scale-Up*) and reducing (*Scale-Down*) resources as follows.

V_Up (Scale-Up). This strategy describes how a VM that is at an *overloaded* state (i.e., which has a control VM^O) adapts its offering by provisioning more CPU/RAM as follows.

- *CPU:* an additional unit of CPU is allocated to the VM if its hosting server has more free CPU units than RAM.
- *RAM:* an additional unit of RAM is allocated to the VM if its hosting server has more free RAM units than CPU.

DOI:https://doi.org/10.1007/s10586-020-03080-8

Table 4 Elasticity strategies at application and infrastructure levels

Level	Condition	Action
	Horizontal scale strategies	
	H_Out1 (Scale-Out: High availability)	
Application	$\forall s \in V_{CS}\ \exists s' \in V_{CS}\ ctrl_{CS}(s') = S^O \wedge ctrl(s) \neq S^U$	$R1$
Infrastructure	$\forall v \in V_{CS}\ \exists v' \in V_{CS}\ ctrl_{CS}(v') = VM^O \wedge ctrl(e) \neq VM^U$	$R2$
	H_Out2 (Scale-Out: Limited availability)	
Application	$\forall s \in V_{CS}\ ctrl_{CS}(s) = S^O$	$R1$
Infrastructure	$\forall v \in V_{CS}\ ctrl_{CS}(v) = VM^O$	$R2$
	H_In (Scale-In)	
Application	$\forall s \in V_{CS} \exists s' \in V_{CS} ctrl_{CS}(s) \neq S^L \wedge ctrl_{CS}(s') = S^U$	$R3$
Infrastructure	$\forall v \in V_{CS}\ \exists v' \in V_{CS}\ ctrl_{CS}(v) \neq VM^L \wedge ctrl_{CS}(v') = VM^U$	$R4$
	Vertical scale strategies	
	V_Up (Scale-Up)	
CPU	$\exists v, e \in V_{CS}\ ctrl(v) = VM^O \wedge prnt(v) = e \wedge cpu(e) \geq ram(e) > 0$	$R5$
RAM	$\exists v, e \in V_{CS}\ ctrl(v) = VM^O \wedge prnt(v) = e \wedge ram(e) \geq cpu(e) > 0$	$R6$
	V_Down (Scale-Down)	
CPU	$\exists v \in V_{CS}\ ctrl(v) = VM^P \wedge cpu(v) \geq ram(v) \wedge cpu(v) > 1$	$R7$
RAM	$\exists v \in V_{CS}\ ctrl(v) = VM^P \wedge ram(v) \geq cpu(v) \wedge ram(v) > 1$	$R8$
	Migration and Load balancing strategies	
	Mig (Migration)	
Infrastructure	$\exists v, v' \in V_{CS}\ ctrl(v) = VM^O \wedge ctrl(v') \neq VM^O \wedge load(v) - load(v') > 1$	$R9$
	LB (Load balancing)	
Application	$\exists s, s' \in V_{CS}\ ctrl(s) = S^O \wedge ctrl(s') \neq S^O \wedge load(s) - load(s') > 1 \wedge prnt_{CS}(s) = prnt_{CS}(s')$	$R10$

V_Down (Scale-Down). This strategy describes how an *overprovisioned* VM (i.e., of control VM^P) adapts its offering by reducing its CPU/RAM offering as follows.

– *CPU:* a CPU unit is freed from the VM if the VM is using more CPU units than RAM.
– *RAM:* a RAM unit is freed from the VM if the VM is using more RAM units than CPU.

Migration and Load Balancing strategies. In order to control the triggering of reaction rules $R9 - R10$, we introduce the strategies *Mig* and *LB* to describe the migration and load balancing behaviors of a Cloud system at infrastructure and application levels as follows. We introduce the function *load(x)* which gives for a given VM or service instance, as a parameter x, the number of service instances or requests it is hosting. The *Mig* and *LB* strategies are defined as follows:

Mig. This strategy describes how a Cloud system balances its load at infrastructure level with the migration of a service instances from an *overloaded* VM to a less loaded one, by triggering rule $R9$.

LB. This strategy describes the load balancing behavior of a Cloud system at application level with the transfer of a request from an *overloaded* service instance to a less loaded one, by triggering rule $R10$.

Note that migration and load balancing strategies are designed to complement horizontal and vertical scale strategies. For instance, a newly provisioned hosting instance (i.e., VM/service) is initially *unused*. To reach a load equilibrium, *Mig* and *LB* strategies will balance the overall system load at infrastructure and application levels, as specified.

3.2.3 Modeling the desirable elastic behaviors with LTL

Linear Temporal Logic (LTL) [43] as an analytic support is particularly powerful. It is expressive enough to accurately describe (in a declarative fashion) a system's evolution over time. It is also generic enough to describe desired high-level goals. LTL semantics allow defining formulas that globally express the *liveness* fundamental property (i.e. the insurance that a given state is reachable). The satisfaction of such formulas can be interpreted as a qualitative indicator of behavioral correctness.

Modeling the introduced elastic behavior with Linear Temporal Logic allows the specification of formulas to verify the system's elastic adaptations. To this purpose, we define a model of temporal logic with a Kripke structure $\mathbf{A}_{CS}$, as follows.

Definition 3. Given a set AP_{CS} of *atomic propositions*, we consider a Kripke structure $\mathbf{A}_{CS} = (A, \rightarrow_A, L_{CS})$. Where A is a set of states, $\rightarrow_A$ is a *transition relation*, and $L_{CS} : A \rightarrow AP_{CS}$ is a labeling function associating to each state $a \in A$, the set $L_{CS}(a)$ of atomic propositions in AP_{CS} that hold in the state a. $LTL(AP_{CS})$ denotes the formulas of the *propositional linear temporal logic*. The semantics of $LTL(AP_{CS})$ is defined by a *satisfaction relation*: $\mathbf{A}_{CS}, a \models \varphi$, where $\varphi \in LTL(AP_{CS})$.

We consider the set of atomic propositions $AP_{CS} = \{Stable, Overloaded, Overprovisioned, Unbalanced\}$

that describe the hosting environment's states. For the sake of simplicity, these states are symbolic and relate to the elastic behavior of the system. The system is considered *Overloaded/Over-provisioned* when at least one entity (VM, Service) is overloaded/unused, and it is *Stable* otherwise. *Unbalanced* is a non-exclusive proposition that can hold together with *Stable*, *Overloaded* or *Overprovisioned* states (that are exclusive) when load balancing at VM or Service levels is applicable. In other terms, different structural states of the system in A (i.e., configurations) can be gathered (i.e., labeled) in the same class of equivalence with respect to the global symbolic state of of the system's elasticity in AP_{CS}.

To describe the desirable elastic behaviors that are triggered by the elasticity controller in LTL, we introduce the set

$LTL(AP_{CS}) = \{UpScale, DownScale, Balance, Elasticity\}$ of the propositional formulas, as follows.

- $UpScale \equiv \mathbf{G}(Overloaded \rightarrow \mathbf{F}Stable)$
- $DownScale \equiv \mathbf{G}(Overprovisioned \rightarrow \mathbf{F}Stable)$
- $Balance \equiv \mathbf{G}(Unbalanced \rightarrow \mathbf{F}Stable)$
- $Elasticity \equiv \mathbf{G}(\sim Stable \rightarrow \mathbf{F}Stable)$

Formulas *UpScale*, *DownScale* and *Balance* respectively state that a given system that is *Overloaded*, *Overprovisioned* and *Unbalanced* will eventually reach its *Stable* state. *Elasticity* formula states that a system that is not *Stable* will eventually reach its *Stable* state. We use the symbol $\sim$ for negation. The symbols **G** and **F** are LTL operators that respectively stand of "*henceforth*" and "*eventually*".

We represent system's transitions with Labeled Transition Systems (LTS) [47]. States are the introduced states of elasticity (S: *Stable*, O: *Overloaded*, P: *Overprovisioned* and B: *Unbalanced*). Transitions are the different adaptation actions for horizontal scaling ($R1 - R4$), vertical scaling ($R5 - R8$), migration and load balancing ($R9 - R10$). In addition, transitions *in* and *out* stand for receiving (input) and releasing (output) end-users' requests.

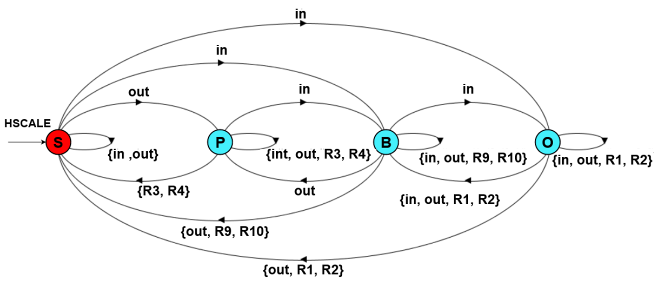


Fig. 9 LTS for horizontal scaling

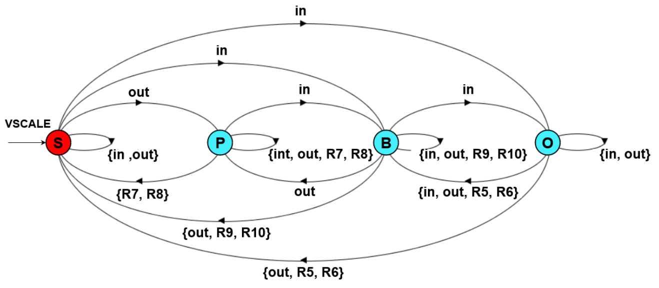


Fig. 10 LTS for vertical scaling

Figure 9 and Figure 10 show the system transitions when controlled with horizontal and vertical scale strategies respectively. Migration and load balancing actions are also shown as they complement the two scaling methods. Note that any state can be initial as it is determined at runtime, by monitoring. However, the shown transition systems depict the system's transitions with *Stable* as an initial state. This shows that there always exists a path that leads back to the *Stable* state even if the system transits by any other states. This behavior is related to the *non-plasticity* property, defined in [8].

4 Principles of Maude encoding and property verification

To verify the correctness of the introduced elasticity strategies and to watch the aimed cross-layered elasticity, it is important to provide an executable solution for the specified elastic behaviors. Theoretically, BRS provide good *meta-modeling* bases to specify Cloud systems' structure and their elastic behavior. As for their executable capabilities, the few existing tools built around BRS as BigraphER [48] and BPL Tool [21] are limited and only suitable for some specific application domains. Furthermore, the BRS model-checker BigMC [40] that was used in [45], allows formal verification of safety properties. However, the possible verifications rely on very limited predefined predicates. These tools lack of providing autonomic executability of the specified BRS models. In this paper, we turn to Maude language to tackle these limitations and to provide a generic executable solution of elasticity strategies together with the verification of their correctness.

4.1 Motivating the use of Maude

Maude [11] is a high-level formal specification language based on equational and rewriting logics. A Maude program is a *logical theory* and a Maude computation is *logical deduction* which uses the axioms specified in the program/theory. A Maude specification is structured in two parts. (1) A functional module that specifies a theory in membership equational logic. Such a theory is a pair $(\Sigma,\ E \cup A)$, where the signature Σ specifies the type structure (sorts, subsorts, operators etc.). E is the collection of the (possibly conditional) equations declared in the functional module, and A is the collection of equational attributes (associative, commutative, etc.) declared for the operators. (2) And a system module that specifies a rewrite theory as a triple $(\Sigma,\ E \cup A, R)$, where $(\Sigma,\ E \cup A)$ is the module's equational theory part, and R is a collection of the (possibly conditional) rewrite rules.

The Bigraphical specifications for Cloud systems' structure (i.e., sorting discipline and construction rules in Section 3.1) can be encoded in a functional module *Elastic_Cloud_System*, where the declared operations and equations define the constructors that build the system's elements and the predicates that determine their states. Similarly, BRS dynamics (i.e., reaction rules in Section 3.2) that describe the elasticity controller's behavior can be encoded in a system module *Elatic_Cloud_Behavior*, where the elasticity strategies are described as conditional rewrite rules. The set of rewrite rules R expresses the bigraphical reaction rules. Their triggering conditions, expressed as equations from the functional or system module, encode the strategies' predicates. To verify the correctness of the defined Cloud systems' elastic behavior as encoded in the system module, we define a Maude property specification based on Linear Temporal Logic. It consists of an additional system module *Elastic_Cloud_Properties* which encodes the specified *Kripke* structure and LTL formulas. It provides a support for the verification of their satisfaction through the Maude-integrated LTL-based model-checker.

Figure 11 shows an overview of our solution of modeling, executing and verifying Cloud system's elasticity. Straight arrows show encoding phases to build Maude modules from the specified BRS and Kripke formal definitions. Dashed arrows show the dependencies between the produced modules (a dashed arrow from A to B means that B requires A in order to be built).

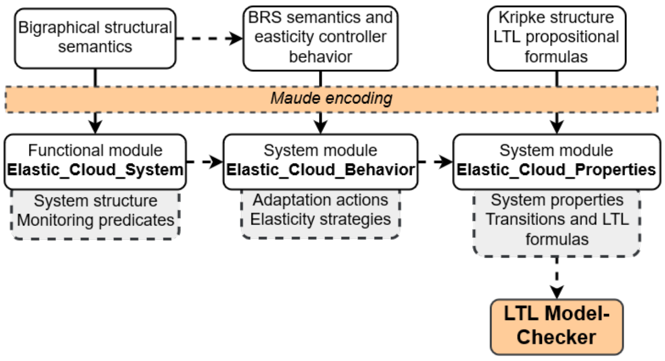

Bigraphical structural semantics
BRS semantics and easticity controller behavior
Kripke structure LTL propositional formulas
Maude encoding
Functional module Elastic_Cloud_System
System module Elastic_Cloud_Behavior
System module Elastic_Cloud_Properties
System structure Monitoring predicates
Adaptation actions Elasticity strategies
System properties Transitions and LTL formulas
LTL Model-Checker

Fig. 11 Top view of our solution for specifying and verifying Cloud elastic behaviors

4.2 Setting up the functional module

Table 5 gathers the main Maude definitions of the functional module *Elastic_Cloud_System*, where the specified bigraphical specifications in terms of system structure and system state predicates are encoded a follows.

Structure encoding. In the functional module, the bigraph sorts e, v, s and r (i.e., server, VM, service and resource pool) are encoded as Maude sorts `CS`, `VM`, `S` and `Res`. Note that we enriched Maude sorts with additional information as the maximum hosting thresholds and the entities states. A sort is built according to its associated constructor (`ctor`). For instance, a Cloud server is built by the operation `CS<x,y,z/VML:Res:state>`, where x, y and z are naturals that encode upper hosting thresholds at server, VM and service levels. *VML* is a list of VMs, as sort *VM* is defined as a subsort of sort *VML*. The term `state` gives a state out of the constructors (*overloaded*, *overprovisioned*, *stable*, *unbalanced* etc.). Sort `Res` stands for resource pool and gives values for CPU and RAM resources. It is given with the constructor `[cpu & ram]` where `cpu` and `ram` are expressed as units, using naturals. Similarly, sorts `VM` and `S` are defined with their constructors. For instance, a service instance is given with `S[z,load:state]` where `z` is its upper hosting threshold, `state` is its state of elasticity and `load` is a natural which gives the number of requests the service instance is handling.

System state predicates encoding. In order to express different states of the system, we define a set of operations and predicates in Maude, introduced with the keyword `op` or `ops` when the operations have the same signature. The defined operations are applied on the different Cloud entities such as `CS`, `VM`, `S`, etc. to

DOI:https://doi.org/10.1007/s10586-020-03080-8

Table 5 Encoding the BRS Cloud model into a Maude functional module

Functional module *Elastic_Cloud_System*	
Enriched bigraphical model	Maude specification

Structural semantic	`sorts CS VM S Res VML SL state . subsort VM < VML . subsort S < SL .`
	`op CS<_,_,_/_:_:_> : Nat Nat Nat VML Res state -> CS [ctor] .`
	`op VM{_,_:_:_} : Nat SL Res state -> VM [ctor] .`
	`op S[_,_:_] : Nat Nat state -> S [ctor] .`
	`op [_&_] : Nat Nat -> Res [ctor] .`
	`ops stable overloaded overprovisioned unbalanced : -> state [ctor] .`
	`...`

Operations and system state predicates	`op loadCS(_): CS -> Nat . op loadVM(_): VM -> Nat . op loadS(_): S -> Nat .`
	`op getResCS(_): CS -> Res . getResVM(_): VM -> Res .`
	`ops getCpu(_) getRam(_): CS -> Nat . ops getVCpu(_) getVRam(_): VM -> Nat .`
	`ops isStable(_) isOverloaded(_) isOverprovisioned(_) isUnbalanced(_) AoverV(_)`
	`     EoverV(_)  EunV(_) AoverS(_) EoverS(_) EunS(_) ... : CS -> Bool .`
	`ops stableV(_) overV(_) idelV(_) ... : VM -> Bool .`
	`ops stableS(_) overS(_) idleS(_) ... : S -> Bool .`
	`ops lessS(_) mostS(_): CS -> S . ops lessV(_) mostV(_): CS -> VM .`
	`...`

return information regarding their load, the amount of resources `Res` each entity (Server, VM) is allocated or a truth value on the applied state predicate. The defined operations and predicates are used to monitor the system during its runtime. For instance, *AoverV()* is a predicate for "all VMs are overloaded" and *EunS()* is a predicate for "there exists an unused service instance". We also encode system state predicates *isStable(), isOverloaded()*, *isOverprovisioned()* and *isUnbalanced()* that are true if the Cloud system is stable, overloaded, overprovisioned and unbalanced. In addition, we extend the model's expressivity by introducing operations like *lessS/V()* or *mostS/V()* in order to detect the most or less loaded service instance or VM. These operations are used to make choices regarding the aimed entities for migration and load balancing.

4.3 Setting up the system module

Table 6 gathers the main Maude definitions of the system module *Elatic_Cloud_Behavior*, where different scaling functions and the introduced elasticity strategies are encoded as follows.

Elasticity strategies encoding. Strategies are encoded as conditional rewrite rules in the system module. Their conditions are the states and monitoring predicates and their actions (bigraph reaction rules) are encoded as Maude functional computation. Like bigraphical reaction rules, Maude rewrite rules consist of rewriting the left-hand side of the rule to its right-side. For instance, migration strategy is specified as the following conditional rewrite rule (`crl`):

```
crl[migration]:cs => MigS(cs) if MigSpred(cs).
```

Where *cs* is a given Cloud system, *MigS(cs)* is an equation that reduces the term *cs* in such a way to apply migration at infrastructure level (by encoding reaction rule $R9$), and *MigSpred(cs)* is a predicate that is true if migration at infrastructure level in *cs* is possible (i.e., it implements the triggering condition of the previously introduced strategy *Mig*). *MigS()* and *MigSpred()* are defined as equations in the system module.

Note that rewrite rules in Maude are designed to be concurrent. Hence, we introduce two sorts `VSCALE` and `HSCALE` to specify structures that exclusively support vertical and horizontal scaling strategies respectively, without concurrency, as follows.

- *Vertical scaling:* sort `VSCALE` is introduced with a constructor "`VSCALE :: cs`" where `cs` is the controlled Cloud system. Such an expression is used to restrain the application of vertical scaling strategies over `cs`. For instance, strategy `V_Up` for scaling-up the system at CPU level is specified with :

```
crl[up-cpu]: VSCALE :: cs => VSCALE ::
addCpu(cs) if scaleUpPredCPU(cs).
```

- *Horizontal scaling:* sort `HSCALE` is given with a constructor "`HSCALE (V i, S j):: cs`" where parameters $i, j \in [1, 2]$ indicate which scale-out strategy (*H_Out1/2*) is applied at infrastructure and application levels of `cs`. For instance, strategy *H_Out1* at

Table 6 Encoding the BRS Cloud model into a Maude system module

System module *Elastic_Cloud_Behavior*	
Enriched bigraphical model	Maude specification
Sorts and elasticity predicates	`sorts VSCALE HSCALE .` `op VSCALE :: _ : CS -> VSCALE [ctor] .` `op HSCALE (V_ , S_) :: _ : Nat Nat CS -> HSCALE [ctor] .` `ops LBpred(_) MigSpred(_) scaleUpPredCPU(_) scaleUpPredRAM(_) scaleDownPredCPU(_) scaleDownPredRAM(_) ... : CS -> Bool .` `...`
Functions	`ops addCpu(_) addRam(_) subCpu(_) Hout1V(_) Hout1S(_) Hout2V(_) Hout2S(_) HinV(_) HinS(_) MigS(_) LB(_) ... : CS -> CS .` `...`
Reaction rules and elasticity strategies	Conditional rewrite rules of the form: `crl [rule-name] : term => term' if condition(s) .` `crl [migration] : cs => MigS(cs) if MigSpred(cs) .` `crl [load-balancing] : cs => LB(cs) if LBpred(cs) .` `crl [V-up-CPU] : VSCALE :: cs => VSCALE :: addCpu( cs ) if scaleUpPredCPU(cs) .` `crl [V-down-RAM] : VSCALE :: cs => VSCALE :: subRam( cs ) if scaleDownPredRAM(cs) .` `crl [H_Out1S] : HSCALE (V i, S j) :: cs => HSCALE (V i, S j) :: HoutS1(cs) if (j == 1 and EoverS(cs) and (not EunS(cs))) .` `crl [H_Out2V] : HSCALE (V i, S j) :: cs => HSCALE (V i, S j) :: HoutS1(cs) if (i == 2 and AoverV(cs)) .` `crl [H_inV] : HSCALE (V i, S j):: CS< ct,vt,st/v | vl :res: cst > => HSCALE (V i , S j):: HinV(CS< ct,vt,st/v | vl :res: cst > if ((not EoverV(vl)) and unV(v)).` `...`

application level is defined with:

```
crl[H_Out1-S]: HSCALE(V i, S j) :: cs =>
HSCALE(V i, S j) :: HoutS1(cs)
if (j==1 and EoverS(cs) and (not EunS(cs))
```

4.4 Formal verification of elasticity

Maude allows associating *Kripke* structures to the rewrite theory specified in the system module. The semantics introduced by the Kripke structure $\mathbf{A}_{CS}$ in Section 3.2 allows to conduct a generic LTL model-checking that can reason on any system configuration. For instance, determining that a Cloud configuration is stable in terms of elasticity is specified with: $cs \models Stable = true$ if $isStable(cs) == true$. Where cs is a given Cloud configuration. *Stable* is a proposition $\in AP_{CS}$ that represent the symbolic elastic state *Stable*. And *isStable(cs)* is a predicate for "the Cloud system cs is stable" which is defined in the functional module.

We execute Maude's LTL model-checker with, as parameters, (1) a Cloud configuration as an initial state and (2) a property formula in $LTL(AP_{CS})$ to verify. The model-checker can give counter examples showing the succession of the triggered rewrite rules that are applied on the initial state of the system, in such a way to verify the given property according to the specified elasticity strategies. For more detail about formal verification of Cloud elasticity in Maude, refer to our previous work [27]. Note that this paper also extends [27] in terms of (1) structure (to consider processing and memory computing resources), (2) behavior (to extend horizontal scale elasticity and to consider vertical scale elasticity and load balancing), (3) formal verification capabilities (to consider the correctness of all the provided elastic behaviors) and (4) quantitative study (by providing a deeper methodology and comparative analysis of the introduced elasticity strategies).

5 A Queuing approach for the simulation and quantitative analysis of Cloud elasticity

Elasticity strategies allow the elasticity controller to decide when, where and how to trigger the suitable adaptation given the system's state of elasticity. In order to validate the correctness of the designed strategies, their quantitative analysis is a mandatory task before their use in real Cloud environments. In this Section, we introduce a tooled queuing-based methodology to simulate and quantitatively analyze the introduced elasticity strategies over an original case-study.

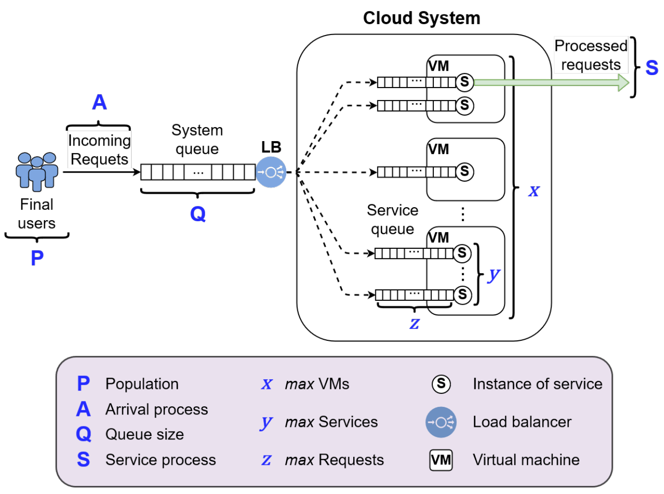


Fig. 12 A queuing-based view of Cloud systems

As its input workload rises, the congestion that may result in a system are in fact waiting queues that indicate the insufficiency of the provisioned resources. For this reason, we advocate that a queuing approach is a relevant support to illustrate the elastic behavior of a system. It allows to quantitatively analyze the hypothetical performance and costs induced by our elasticity strategies.

5.1 A queuing model for Cloud elasticity

We consider a queuing model, defined by a set of parameters as introduced by the *Kendall* notation: $A/S/C/Q/P/D$ [7], where C is the number of service instances. A is the arriving process describing how the requests arrive into the system. D is the serving discipline describing how the requests are processed (e.g., first come first served). The service process S gives the amount of time required to process the requests. Queue size Q gives the maximum number of requests that the system can hold and the population P is the number of requests expected to arrive into the system. In our study, we will consider that $Q = \infty$ and $N = \infty$. We consider that A is a Poisson process which gives an exponential distribution of the received requests (at each time unit) with the average value of λ. S also follows an exponential law with the average value of μ to give the number of requests that are processed by service instances. The essence of elasticity being the adaptations, we use a queuing model with on-demand number C of service instances, inspired from [31], to show how the system adapts to its varying input workload by (de)provisioning resources at service and infrastructure levels, using the defined elasticity strategies.

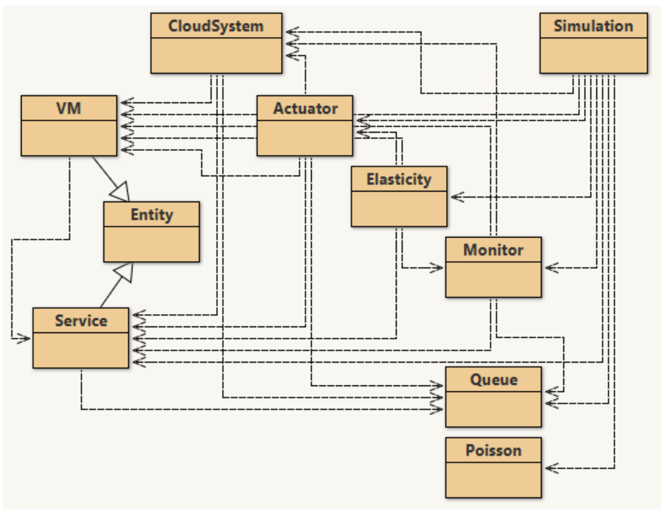


Fig. 13 Functional structure of the designed simulation and monitoring tool

Figure 12 illustrates the principle of our queuing-based vision of Cloud systems. This vision allows considering the *front-end* and *back-end* parts of a Cloud system (i.e., reception and processing of end users' requests) with a robust mathematical semantics (i.e., using the introduced queuing model). The load balancer (LB) component connects the front-end and the back-end parts of a Cloud system. It transfers incoming requests from the principal system queue to the available service instances deployed in the Cloud hosting environment. The values x, y and z correspond to upper hosting thresholds at server, VM and service layers, introduced in Section 4.2.

5.2 A simulation and monitoring tool for Cloud elasticity

In order to simulate and illustrate the defined elasticity strategies, we designed a program which works according to the MAPE (*Monitor*, *Analyze*, *Plan*, *Execute*) autonomic control loop [25]. The tool enables simulating and monitoring a managed elastic Cloud system during its runtime. It implements the defined elasticity controller and elastic behavior (elasticity strategies). In addition, the tool allows to reproduce a traffic of incoming requests (arrival process) and processed requests (service process) as specified by the introduced queuing model. The tool's functional structure is given in Figure 13. It implements the principal notions introduced in this paper such as the architectural entities (VMs, services) of a Cloud system and the introduced queuing approach (through the components *Queue* and *Poisson*). Furthermore, it integrates the main phases of the MAPE control loop through the components *Mon-*

itor, *Actuator* (for the *Execute* phase) and a component *Elasticity* which gathers both *Analyze* and *Plan* phases of the loop. Finally, the tool defines a *Simulation* component for setting-up, executing and recording the runtime of a simulated Cloud system. Note that *Entity* component is used to define a method to produce unique identifiers for VMs and services from CPU time stamps.

Using our tool, a Cloud system is simulated with, as input: (1) an initial configuration of the Cloud system (in terms of deployed VMs and service instances), (2) initial values of maximum hosting thresholds at server, VM and service levels, (3) a request arrival rate λ and (4) a request service rate μ.

Proposed algorithm. We provide an algorithm to explain the introduced tool's approach for elasticity management and simulation. The Algorithm 1 shows the main MAPE control loop approach where the studied system is initialized then simulated. Note that the presented algorithms' purpose is to simplify the reader's understanding of the proposed approach. They give a simplified vision of the previously defined behaviors' semantics. Once the specified behaviors are formally verified, the purpose of our simulation tool is to (1) illustrate the capabilities of our approach in terms of modeling expressiveness and to (2) observe the proposed behaviors in terms of quantitative results. The overall idea is to analyze and discuss the hypothetical use of these defined behaviors (i.e. elasticity strategies) in real environments.

In the initialization phase, the initial system configuration is built. Sets of virtual machines (V_{set}) and service instances (S_{set}) running inside those VMs are specified. Requests arrival rate (λ), requests service rate (μ) and upper hosting thresholds (x, y, z) are initialized to set-up the system's queuing aspect. And the set of to be applied strategies ($Strat_{set}$) is specified. Note that the the number of VMs is bounded by the value of x.

During the whole simulation time, the tool applies the MAPE concept, at each step (*tick*). Input request flux (*workload*) is generated as a Poisson process around the value of λ. Incoming requests are queued in the system then distributed to the deployed service instances. At this point, Monitoring calculates the system global state (*overloaded, stable, etc.*) in function of the deployed hosting entities' sates as shown in The Algorithm 2. For each VM (V_i) and service instance (S_i), the state of elasticity (*overloaded, unused, stable*) is obtained regarding their respective upper hosting thresholds (respectively y and z).

Once the system's state obtained, the elasticity controller (component Elasticity or *Elast* in the algorithm) applies the desired elasticity strategies as shown in the Algorithm 3. The controller analyzes monitoring data to evaluate the triggering conditions of each strategy $strat_i$ in in $Strat_{set}$ and the Actuator applies the associated adaptation actions A_i in $Actions_{set}$, as specified in Section 3.2.2. After the suitable adaptation actions are applied, requests are processed (i.e., freed from services queues) according to the service process rate μ. Finally, the Monitor prints information logs to keep monitoring

Algorithm 1 Pseudo code for elasticity control loop
```
1: Initialization: instantiate initial VMs ($V_{set}$) and services ($S_{set}$); init $\lambda$, $\mu$ and hosting thresholds; specify applied strategies ids ($Strat_{set}$)
2: Begin
3: while (tick < simulation time) do
4: generate incoming requests ($workload = Poisson.gen(\lambda)$)
5: put requests in system queue ($Actuator.queue(workload)$)
6: distribute requests to services ($Actuator.dist(S_{set})$)
7: calculate system global state ($Monitor.monitor(V_{set}, S_{set})$)
8: adapt according to strategies ($Elast.adapt(Strat_{set})$)
9: process requests ($Actuator.output(\mu, S_{set})$)
10: output monitoring log ($Monitor.log()$)
11: end while
12: End
```

Algorithm 2 Pseudo code for states monitoring
```
1: Begin
2: /* calculate deployed VMs states */
3: for (every virtual machine $V_i$ in $V_{set}$) do
4: if ($V_i.load() == 0$) then $V_i.state = unused$
5: elseif ($V_i.load() < y\text{-}threshold$) then $V_i.state = stable$
6: elseif ($V_i.load() >= y\text{-}threshold$) then $V_i.state = overloaded$
7: end if
8: end for
9: /* calculate deployed services states */
10: for (every service $S_i$ in $S_{set}$) do
11: if ($S_i.load() == 0$) then $S_i.state = unused$
12: elseif ($S_i.load() < z\text{-}threshold$) then $S_i.state = stable$
13: elseif ($S_i.load() >= z\text{-}threshold$) then $S_i.state = overloaded$
14: end if
15: end for
16: determine system global state /* from $V_{set}$ and $S_{set}$ states */
17: End
```

Algorithm 3 Pseudo code for elastic adaptation
```
1: Begin
2: for (every strategy $strat_i$ in $Strat_{set}$) do
3: /*$Strat_{set} \subseteq \{H\_Out1, H\_Out2, H\_In, V\_Up, V\_Down, Mig, LB\}$*/
4: evaluate triggering predicates /*using monitoring data*/
5: apply the suitable adaptation actions $A_i$ in $Actions_{set}$
6: /*$Actions_{set} \subseteq \{scale\text{-}up, scale\text{-}down, scale\text{-}out, scale\text{-}in, migration, load\text{-}balancing\}$*/
7: end for
8: End
```

traces, and the adaptation loop (starting with incoming requests) is reiterated.

5.3 Simulation and analysis methodology

According to the shown tool's elasticity control algorithms, a Cloud system's runtime is simulated and monitored from a given initial configuration. Information is gathered from the conducted simulations order to quantitatively analyze the obtained behaviors.

Recorded parameters. At each time step, monitoring records the following information:

- Number of input requests
- Number of requests waiting in the system queue
- Number of deployed VMs and service instances
- Number of requests waiting in the service queues

Calculated metrics. At the end of a simulation, the following metrics are calculated from the recorded values listed above:

- Average number of deployed VMs and service instances
- Average waiting rate of requests before processing (delay)
- Average usage rate of VMs and service instances

The obtained metrics indicate the performance and cost induced by the elastic behavior of the simulated Cloud system. Performance is given by the average waiting rate of requests before processing (i.e., response time). Operating costs and scaling accuracy are given with the average number of deployed resources (i.e., VMs and service instances).

Analysis and validation. The system's elastic behavior efficiency is analyzed in terms of performance, costs and average usage rate of resources [44]. In order to validate the obtained results we propose the following solutions for horizontal and vertical scaling:

- *Horizontal scaling:* as horizontal scale elasticity consists of adding/removing Cloud resources in a cross-layered manner (i.e., VMs, services), we compare the obtained simulation results with those given by the *Erlang-C* formula [16]. This formula calculates, from a given arrival and service rates λ and μ, the minimal needed number of servers (i.e., service instances) to ensure a given level of service (i.e., average waiting rate).

- *Vertical scaling:* unlike horizontal scaling, vertical scale elasticity consists of resizing the already existing VM instances in the initial system configuration without adding/removing them. As the number of deployed instances remains unchanged, the *Erlang-C* formula is no longer of use. Vertical scaling is more about the efficiency of computing resources allocation (i.e., minimum resources for maximum performance). However, it is not a trivial task to define a generic correlation between input workload and an ideal and minimal resources (CPU, RAM) allocation [4]. Therefore, we propose an arbitrary solution to deal with resources consumption: we consider that a VM is initially allocated one unit of CPU and one unit of RAM which give it a initial capacity z of upper threshold in terms of requests in service queues and an initial service rate μ. Adding or removing a CPU/RAM unit simply adds or removes a constant amount to its z/μ (as adding/removing resources modifies the system's capacity in terms of handled and processed requests by time unit). Finally, we study the obtained results by analyzing the system states of *over-provisioning* and *under-provisioning* regarding the deployed computing resources.

5.4 The *Steam* digital library: a case study

Steam is a platform for online contents distribution, of rights management and communication developed by Valve in 2003 [50]. Mainly focusing on the market of video-games, Steam platform enables users to buy games, software and automatically update their products. Since 2013, Steam benefits from a complete Cloud-based support for distributed hosting of its offered services. In this case study, we focus on the Steam digital library online store.

In January 2018, Steam recorded about 125 Million registered users and a catalog of about 28000 items for sale. Steam bases most of its business model on the high availability of its services and ubiquitous accessibility worldwide. According to [51], nearly 1 Million products have been sold on Steam in 2018, with an average of $83,000$ sales monthly. Actually, the online store is the object of permanent solicitation but activity peaks are recorded during special events such as seasonal sales and periodic discounts. For instance, about $200,000$ sales were recorded during the last week of December 2017.

From these reports, we can see that Steam store knows an important activity (excluding browsing, researches, display of multimedia content, uncompleted transactions, etc.) and is of a highly variable nature in terms of solicitation fluctuations. This makes of Steam store a suitable case study to illustrate our solution for managing the elastic behaviors of a Cloud system. We propose an experimental study of its elasticity, according to our introduced strategies, in order to validate their correctness under a quantitative point of view.

5.4.1 Setting up the experimental study

In order to illustrate the defined elasticity strategies, we simulate the execution of the Steam store. First, we apply our bigraphical modeling over the Steam store to identify an initial state of the service as a Cloud configuration. Then, we set values for the different simulation parameters (i.e., thresholds, arrival and service rates). Finally, we explain the study protocol and identify some simulation scenarios.

System initial state. We consider the Steam store as an initial instance of service (S node) deployed inside a virtual machine (VM node), which is running on top of a physical server (SE node) as shows in Figure 14. In terms of computing resources, CPU and RAM units (CU and M nodes) are deployed inside the available and allocated resource pools (RA and RV node) for the server and the VM respectively. The VM is allocated one CPU unit and one RAM unit. The server has 3 available units of both CPU and RAM. Remember that sites (numbered from 0 to 5 here) are used to abstract some parts of the system. For instance, site 5 nested inside the service instance is used to abstract all the requests the service is handling to avoid overloading the graphical representation.

Such a configuration is encoded in Maude and enriched with the upper hosting thresholds x, y and z for services, VMs and servers respectively, according to the introduced constructors in Section 4.2, as follows:

```
CS< x,y,z/ VM{y,S[z,q: stateS]:[1 & 1]:
 stateVM}:[3 & 3]: stateSE > .
```

The terms `stateS`, `stateV` and `stateSE` are states (out of *overloaded*, *unused*, *stable*, etc.) of the service instance, the VM and the physical server respectively. The term `q` gives the number of handled requests by the service instance. Finally, the structures `[a & b]` encode the resource pools where `a` and `b` give the number of CPU and RAM units respectively.

Simulation inputs. We define input values for the simulations as follows:

- *Arrival rate* λ: We consider that $\lambda = 500$. This value gives the maximal request arrival rate that will be reached in our simulation.

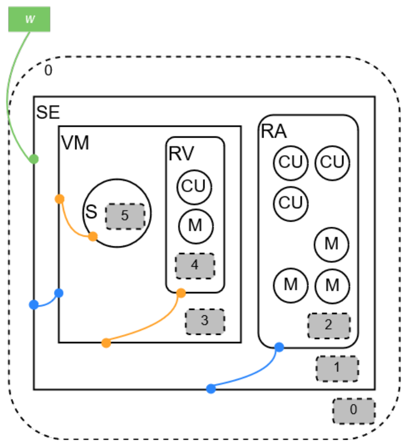


Fig. 14 Bigraphical modeling of the Cloud based Steam store service initial state

- *Service rate* μ: We consider that $\mu = 50$. This values indicates the initial capacity of the system in terms of requests processing.
- *Upper server hosting threshold* x: We consider that $x = 4$ which means that the system can provision up to 4 VM instances.
- *Upper VMs hosting threshold* y: We consider that $y = 4$ which indicates that each VM can host up to 4 instances of the Steam store service.
- *Upper services hosting threshold* z: We consider that $z = 50$ which means that a service instance can handle up to 50 requests at a time (i.e., service queue size).

Experiment and scenarios. We face the Cloud based Steam store service to an unpredictable model of incoming workload as shown in Figure 15(a). This model describes a highly fluctuating activity over 200 units of time (t). It is obtained with an algorithm shown in Figure 15(b) and describes three main simulation phases. The obtained activity represent the Steam store solicitations following the announcement of short duration discount offers as follows.

- Phase 1 ($t = [0, 70]$): Slow and progressive rise of the workload. The value of λ goes from 5% to 70%.
- Phase 2 ($t = [71, 100]$): Workload peak rise. The value of λ goes from 70% to 100%.
- Phase 3 ($t = [101, 200]$): Fast drop of workload. The value of λ goes from 100% to 1%.

Once set up, we simulate the system's runtime to study its elastic behavior according to the following scenarios, focusing on the horizontal then the vertical scale strategies:

Horizontal scale scenarios: we propose four scenarios where we compose the defined *Scale-Out* strategies at infrastructure and application levels. The *Scale-Down* strategy (*H_in*) is also applied for all scenarios:

– H1: we apply strategy *H_Out1* at both infrastructure and application levels.
– H2: we apply strategy *H_Out2* at both infrastructure and application levels.
– H3: we apply strategy *H_Out1* at infrastructure level and strategy *H_Out2* at application level.
– H4: we apply strategy *H_Out2* at infrastructure level and strategy *H_Out1* at application level.

Vertical scale scenario: we propose one scenario where we apply *Scale-Up* and *Scale-Down* strategies. As explained before, in the scenario we focus on the system's capacity in terms of queue size and request processing rate. The capacity is given with: $c = s \times \mu$ where s in the number of deployed service instances and μ is the service rate.

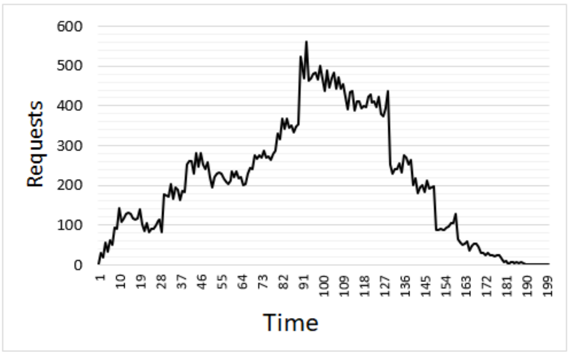
– V: we apply strategies *V_Up* and *V_Down* to add/remove computing resources (CPU,RAM) to cope with the varying demand.

5.4.2 Experiment results and analysis

In this Section, we present the obtained simulation experiment results. In Table 7, we give, for the horizontal scale scenarios, the obtained metrics values in terms of average waiting rate, average deployed VMs, average deployed service instances and the average value of λ for the three simulation phases. For the vertical scale scenario, we give the obtained results in terms average waiting rate, the average amount of deployed CPU and RAM, and the average value of λ for the three simulation phases.

Horizontal scale scenarios. Intuitively, applying high resources availability *scale-out* strategy *H_Out1* at infrastructure and application levels (scenario H1), the systems behaves in a way to achieve *High performance* high-level policy. In summary, the system runtime ends with a request processing waiting rate of 0.03%. However this implies important deploying costs with 78% of total VM capacity (out of $x = 4$) and 70% total service instances capacity (out $x \times y = 16$). In addition, the system presents a relatively low resources use rate efficiency (i.e., system load) of 37% on average.

In terms of validation, the *Erlang-C* formula indicates that at a minimum of 14 service instances are required to cope with the workload peak (phase 2) given the value of λ during this phase. However, H1 shows that 16 instances were deployed which indicates the



(a)

```
if (tick == 2) lambda = (lambdaMax * 0.05);
else if (tick == 4) lambda = (lambdaMax * 0.1);
else if (tick == 8) lambda = (lambdaMax * 0.2);
else if (tick == 10) lambda = (lambdaMax * 0.25);
else if (tick == 20) lambda = (lambdaMax * 0.2);
else if (tick == 30) lambda = (lambdaMax * 0.35);
else if (tick == 40) lambda = lambdaMax * 0.5;
else if (tick == 50) lambda = lambdaMax * 0.45;
else if (tick == 70) lambda = (lambdaMax * 0.55);
else if (tick == 80) lambda = lambdaMax * 0.7;
else if (tick == 90) lambda = lambdaMax;
else if (tick == 100) lambda = lambdaMax * 0.95;
else if (tick == 110) lambda = lambdaMax * 0.8;
else if (tick == 130) lambda = lambdaMax * 0.5;
else if (tick == 140) lambda = lambdaMax * 0.4;
else if (tick == 150) lambda = (lambdaMax * 0.2);
else if (tick == 160) lambda = (lambdaMax * 0.1);
else if (tick == 170) lambda = (lambdaMax * 0.05);
else if (tick == 180) lambda = (lambdaMax * 0.01);
else if (tick == 190) lambda = (lambdaMax * 0.001);
workload = Poisson.getPoisson(lambda);
```

(b)

Fig. 15 A model of unpredictable input workload

system's trend to an *over-provisioning* state in terms of deployed resources. Note that the *Erlang-C* formula gives the same values for all scenarios as all the input data are the same. The recorded system's monitoring traces are given in Figure 16.

Conversely, applying limited availability *scale-out* strategy *H_Out2* in a cross-layered manner, i.e., at infrastructure and application levels (scenario H2), the systems achieves a *High economy* high-level policy. In summary, it achieves an average of 52% and 35% of total VMs and service instances capacity deployment. However, this implies relatively low performance with an average waiting rate of 15%. Finally, the system achieves a high system load of 73%. The system's monitoring records are given in Figure 17. In terms of validation, the system deploys 12 service instances to cope with the growing demand in phase 2, which is better than the *Erlang-C* results.

Table 7 Simulation scenarios quantitative results

Horizontal scale scenarios				
H1				
Metrics	Phase 1	Phase 2	Phase 3	Total
Avg. wait (%)	0,08	0	0	0,03
Avg. VMs	3,7	4	2,1	3,1
Avg. services	13,5	16	6,7	11,3
Avg. λ	201	500	129	210
H2				
Metrics	Phase 1	Phase 2	Phase 3	Total
Avg. wait (%)	12	18	0	15
Avg. VMs	1,6	4	2	2
Avg. services	4,8	12	5	5,6
Avg. λ	194	490	154	207
H3				
Metrics	Phase 1	Phase 2	Phase 3	Total
Avg. wait (%)	6	12	0	8
Avg. VMs	3,5	4	3	3
Avg. services	5,5	14	5,3	7
Avg. λ	197	486	155	208
H4				
Metrics	Phase 1	Phase 2	Phase 3	Total
Avg. wait (%)	9	0	0	2
Avg. VMs	3,1	4	2,4	2,6
Avg. services	12,5	16	7,7	10,2
Avg. λ	204	484	156	209

Vertical scale scenario				
V				
Metrics	Phase 1	Phase 2	Phase 3	Total
Avg. wait (%)	4,7	8,6	0,6	6,4
Avg. CPU	4,9	11,6	4,5	5,42
Avg. RAM	3,5	11,47	4,6	4,8
Avg. λ	201	473	155	208

By combining *H_Out1* and *H_Out2 scale-out* strategies at infrastructure and application levels respectively (scenario H3), the systems achieves a compromise between H1 and H2 in terms of deployment costs and overall performance. This combination leads the system to achieve a *High infrastructure availability* high-level policy, where it rapidly provisions VMs when the workload suddenly rises. During its runtime (as shown in Figure 18), the system ends up with a waiting rate of 8% (against 0.03% in scenario H1 and 15% in H2). It achieves a similar yet inferior resources usage rate, comparing to H2 with a system load of 68%. However, H3 presents higher infrastructure deployment costs than H2, with an average rate of 74% deployed VMs against 52% in H2. In terms of validation, this scenario is coherent with the given results with the *Erlang-C* formula as it gives the same total amount of deployed service instances (i.e., 14).

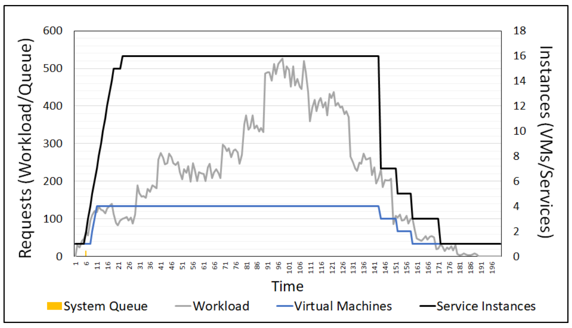


Fig. 16 H1 scenario monitoring traces

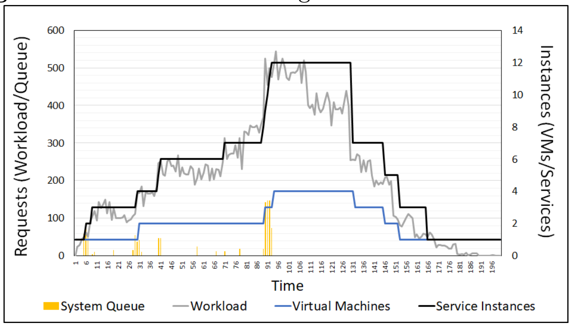


Fig. 17 H2 scenario monitoring traces

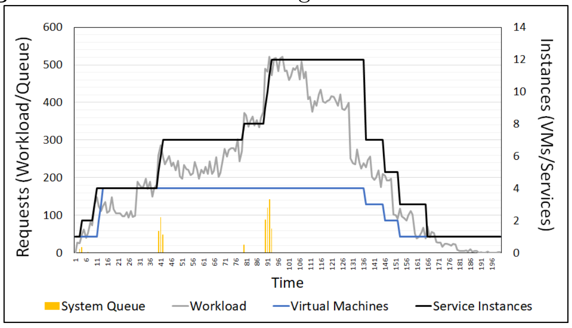


Fig. 18 H3 scenario monitoring traces

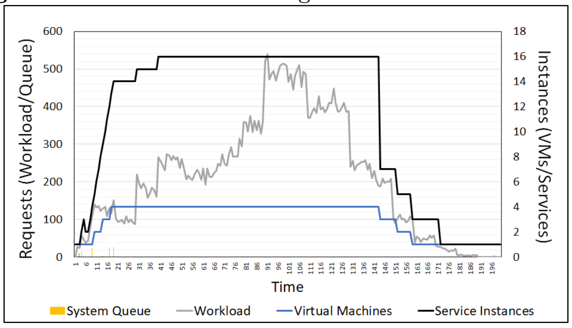


Fig. 19 H4 scenario monitoring traces

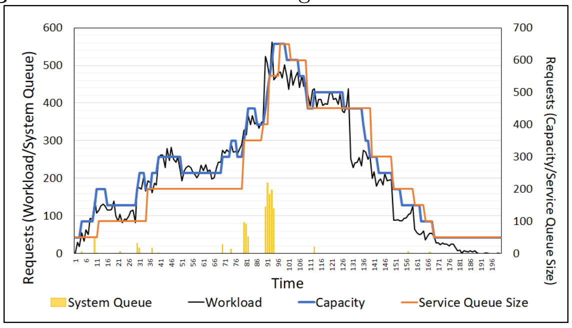


Fig. 20 V scenario monitoring traces

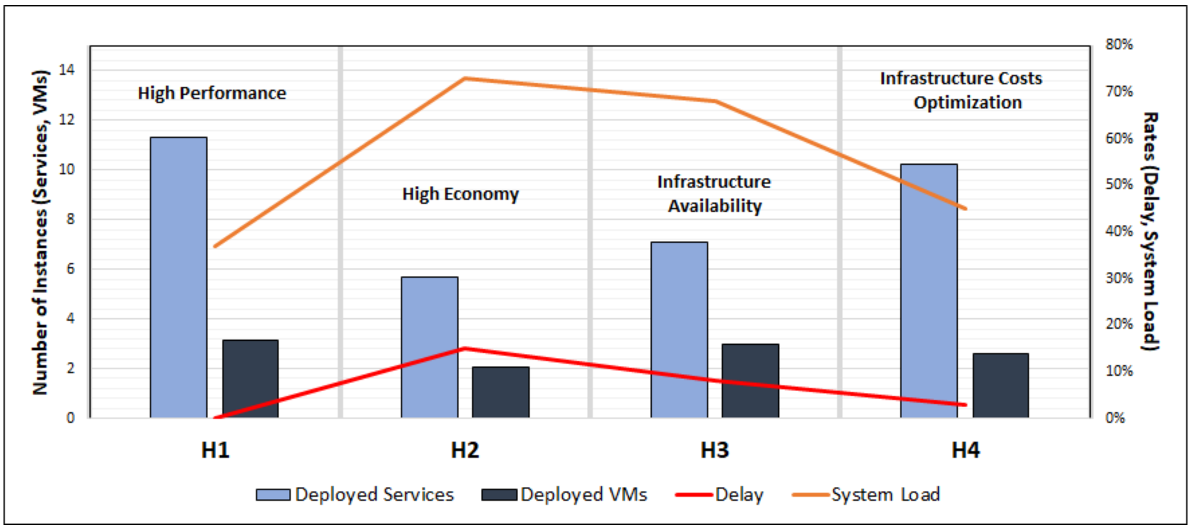
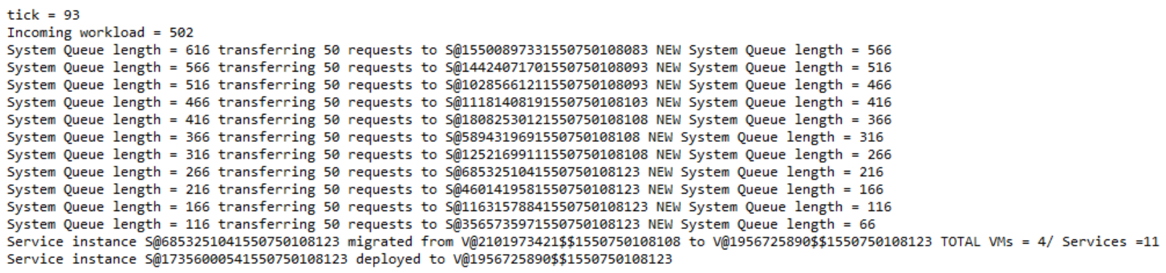


Fig. 21 Horizontal scale strategies cross-layer high-level policies



```
tick = 93
Incoming workload = 502
System Queue length = 616 transferring 50 requests to S@15500897331550750108083 NEW System Queue length = 566
System Queue length = 566 transferring 50 requests to S@14424071701550750108093 NEW System Queue length = 516
System Queue length = 516 transferring 50 requests to S@10285661211550750108093 NEW System Queue length = 466
System Queue length = 466 transferring 50 requests to S@11181408191550750108103 NEW System Queue length = 416
System Queue length = 416 transferring 50 requests to S@18082530121550750108108 NEW System Queue length = 366
System Queue length = 366 transferring 50 requests to S@5894319691550750108108 NEW System Queue length = 316
System Queue length = 316 transferring 50 requests to S@12521699111550750108108 NEW System Queue length = 266
System Queue length = 266 transferring 50 requests to S@6853251041550750108123 NEW System Queue length = 216
System Queue length = 216 transferring 50 requests to S@4601419581550750108123 NEW System Queue length = 166
System Queue length = 166 transferring 50 requests to S@11631578841550750108123 NEW System Queue length = 116
System Queue length = 116 transferring 50 requests to S@3565735971550750108123 NEW System Queue length = 66
Service instance S@6853251041550750108123 migrated from V@2101973421$$1550750108108 to V@1956725890$$1550750108123 TOTAL VMs = 4/ Services =11
Service instance S@1735600054155075010812 3 deployed to V@1956725890$$1550750108123
```

Fig. 22 Monitoring trace for $t = 93$ in H2 scenario

Finally, by combining *H_Out2* and *H_Out1 scale-out* strategies at infrastructure and application levels respectively (scenario H4), the system depicts a subtle behavior (as shown in Figure 19). It achieves *Infrastructure costs optimization* high-level policy with an equivalent, yet inferior, VMs and service instances total capacity provisioning of respectively 65% and 63% against 78% and 70% in H1. In addition, the Cloud system achieves a better resources usage efficiency than H1 with 45% against 37%. Finally, it shows a slightly worse performance with a waiting rate of 3% against 0.03% in H1. In terms of validation, H4 gives the same results as H1 in terms of deployed service instances (i.e., 16), which is higher than the minimal value given with the *Erlang-C* formula.

In conclusion, we observe that the different obtained behaviors, resulting from the different strategies combinations widely influence the studied Cloud system in terms of performance, costs and efficiency. On the one hand, limiting resources deployment in terms of VMs and service instances leads to better efficiency. However, it implies worse overall performance for lower costs. On the other hand, the preference of a higher resources availability globally implies a higher infrastructure deployment cost (due to the system's frequent *over-provis-ioning* tendency), and less efficient resources usage rate. However, it implies a higher overall performance. A comparison of the obtained horizontal scale strategies' results is shown in Figure 21 in terms of average amount of deployed resources in a cross-layered manner, average system load (i.e., services usage rate) and average requests processing waiting rate.

Vertical scale scenario. In our experiment, we have set request service rate to $\mu = 50$ and upper requests hosting threshold (i.e., service queue size) to $z = 50$. As explained before, we consider that these values arbitrarily correspond to one unit of CPU and one unit of RAM respectively. To illustrate the system's capacity growth and decrease, adding/removing an unit of CPU/RAM results in adding/subtracting 50 to the values of z/μ. Consequently, it becomes possible to deduce the amount of deployed CPU and RAM units during the system's runtime from the values of the system's capacity c and the sum of services queue size q, according to the applied *scale-up* and *scale-down* strategies. Thus, the amount of CPU/RAM units is given with $CPU = c/50$ and $RAM = q/50$.

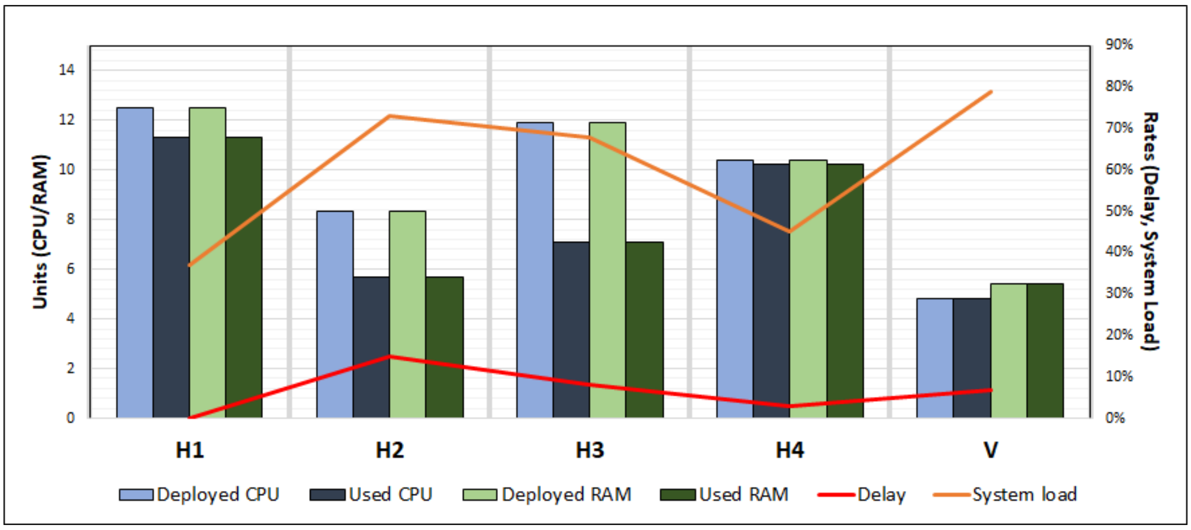


Fig. 23 Horizontal scaling vs. vertical scaling

By applying vertical scale strategies, the studied Cloud system achieves a *High resource efficiency* high-level policy. The scenario V (Figure 20) shows that the systems adapts particularly efficiently to the workload fluctuations by provisioning resources quite closely to the actual demand, in a way to diminish the occurrence of *over-provisioning* and *under-provisioning* system states. In summary, the system ends up with an average waiting rate of 6.4%. In terms of deployed resources, average values of 5.42 CPU units and 4.8 RAM units are recorded.

Migration and load balancing. Notice that the shown graphs does not provide information about the strategies *Mig* and *LB* behaviors. However, those behaviors are visible at the level of the log outputs produced by our monitoring tool, where traces are kept of all the operations triggered regarding the system's state. Figure 22 gives the log trace printed by the tool for $t = 93$ (beginning of phase 2) of scenario H2 activity. The log shows how requests are transferred from the system queue to the different service queues. In addition, it shows when a service instance is migrated from a VM to another one.

Horizontal scaling vs. vertical scaling. In order to compare the horizontal and vertical scaling induced by the defined strategies, we need to calculate the amount of computing resources (CPU and RAM) deployed for horizontal scale scenarios (H1 to H4). We consider that a service instance car run only if one unit of CPU and one unit of RAM are available in its hosting VM. Since we set upper hosting threshold of hosted service instances to $y = 4$, a VM needs to provide at least 4 units of each CPU and RAM to be able to run this number of service instances. Thus, the average amount of deployed resource units is given with $CPU_D = RAM_D = 4 \times v$ where v is the average amount of recorded deployed VMs during each simulation. As the full potential of deployed of deployed service instance is not always reached, we can also calculate the average amount of really used resources with $CPU_U = RAM_U = s$, where s is the average amount of deployed service instances.

Table 8 Horizontal scaling vs. vertical scaling

Results	H1	H2	H3	H4	V
Depl. CPU	12,52	8,32	11,92	10,4	4,8
Used CPU	11,32	5,67	7,1	10,2	4,8
Depl. RAM	12,52	8,32	11,92	10,4	5,42
Used RAM	11,32	5,67	7,1	10,2	5,42
Waiting (%)	0.03	15	8	3	7
Load (%)	37	73	68	45	79

In the case of vertical scaling, we consider that the deployed resources are de-facto used, since only one instance of service is deployed. Thus, the amount of deployed and used resources is given with $CPU_D = CPU_U$ and $RAM_D = RAM_U$.

Figure 23 shows a comparison between horizontal scaling and vertical scaling scenarios in terms of deployed and used resources. Table 8 shows the displayed values which are calculated, as described, from the previously shown values. The data is analyzed in the following:

– *Vertical scaling advantages:* vertical scale scenario showed overall better results than horizontal scale ones, with a high system usage rate of 79% (which is similar to H2) for a moderate waiting rate of 7% (which is similar to H3). However, vertical scenario truly distinguishes with a more optimized resources deployment whereas horizontal scenarios showed an overall system *over-provisioning* tendency.

– *Horizontal scaling advantages:* if vertical scale elasticity puts an accent on efficiency, horizontal scale elasticity distinguishes from the vertical with a high reliability and system availability. In fact, having multiple deployed Cloud resources (i.e., VMs and service instances) provides a distributed deployment for the running Cloud service (i.e., the Steam store service in our study). Such a deployment results on a high reliability of the system in front of software and hardware breakdown (e.g., network failure, software and VM crash, etc.). In addition, the service availability is maintained in case of system node sudden crash. On the other hand, vertical scaling consists of a SPOF (*single point of failure*) kind of deployment, where a single node stop or failure (e.g., crash, update, ect.) results on the potentially extended unavailability of the whole service [52].

5.5 Discussion

The presented experimental study shows the elastic behavior of a simulated Cloud system. Globally, the defined horizontal scale strategies gave results that were in adequacy with those obtained with the *Erlang-C* formula. Through the conducted simulation-based quantitative study, we showed that horizontal scaling focuses on availability and reliability whereas vertical scaling focuses on resources usage efficiency.

Given the different nature of the obtained results, one can ask: "which scenario shows the better results ?" or "which strategy is the best to be adopted ?". To answer these questions, let's discuss the concept of a "good" strategy. We advocate that this notion remains of very subjective nature and is not trivially formalizable, even by considering the objective nature of the obtained performance, costs and efficiency estimations. Thus, it is legitimate to consider that a "good" strategy is one which leads to maximized performance and efficiency for minimized costs. However, results depend heavily on the case study, the workload nature and the system constraints in terms of hosting thresholds.

Our modeling and simulation approach gives a wide notion of freedom and choice in setting-up desirable elastic behaviors. We provide a way to design behaviors (i.e., elasticity controller's strategies) which can be adapted to a Cloud system's requirements and constraints, in terms of budget and nature of activity (i.e., workload intensity and fluctuations). Thus, our proposed solution of elasticity management enables a Cloud service provider to set-up its product's elastic behavior considering its constraints and requirements. Precisely, the introduced hosting thresholds (x, y, z) can be set in a way to model a real-life system. The upper VMs hosting threshold x models a providers financial constraints (budget) on affording VMs provisioning. The bigger is the value of x, the more VMs can be provisioned to provide a distributed deployment of a Cloud service. The upper service instances threshold y indicates the nature of the provisioned VM profiles in terms of resources (CPU,RAM) offering. The bigger is the value of y, the more a provisioned VM carries resources (CPU, RAM) to host more service instances.

Finally, the values of the system queue size (Q), service rate μ and the upper requests hosting threshold y can be adapted in function of the running service's nature. For example, a lightweight service (e.g., simple data access service) could handle more requests at a time than a complex time consuming one (e.g., file conversion service).

6 Related work

This paper's contributions are structured in two main parts. In the first part, we proposed a formal approach for the modeling and specification of Cloud systems structures and elastic behavior using the BRS formalism in Section 3, and an execution and verification solution of the defined behaviors, using the rewriting-logic-based Maude system in Section 4. In the second part (Section 5), we introduced a tooled queuing-based approach for the simulation, monitoring and quantitative analysis of the introduced elastic behavior. Hence, this Section discusses related works in two parts. First, we discuss some related papers about formal modeling of Cloud system's elastic behaviors. The second part of this Section is dedicated to some papers about solutions for monitoring and managing elasticity in Cloud systems.

6.1 Formal approaches for Cloud elasticity specification and management

There have been several works in the literature proposing solutions and frameworks for Cloud systems' elasticity management, such as [29, 3, 53]. Overall, these solutions are mainly from the academic world and provide MAPE-based approaches to control elasticity. Recently, formal methods have been increasingly used to deal with Cloud elasticity [28]. Woks such as [17, 41], were for example proposed to study elasticity using formal models. We will discuss in this sub-Section some similar work in terms of Cloud systems and their elastic behaviors modeling.

In [8], the temporal logic named CLTLt(D)(Timed Constraint LTL) was used to model some properties related to Cloud systems such as *elasticity*, *resource management* and *quality of service*. In their work, authors provided horizontal scale (scale-out/in) elastic behavior and vertical scale. Other elasticity methods are not addressed. In terms of modeling, Cloud system's architecture has been abstrated to only consider Cloud resources as virtual machines (number of VMs). The application layer is also not addressed. The proposed approach is verified with an offline SAT/SMT based tool to analyze execution traces of online simulations that gave the system's evolution in terms of number of deployed VMs.

Authors in [5] proposed a Petri Nets based formalization to describe Cloud-based business processes' elastic behaviors. They introduced elasticity strategies for routing, duplicating and consolidating Cloud components at application level. Authors focused on the application layer of a Cloud configuration but did not address the Cloud infrastructure in their model. Authors simulate their approach at design time to observe its hypothetical performance in terms of deployed service instances. The approach was also verified at design time via mathematical proofs using SNAKES, a Petti nets based tool for reachability graphs analysis.

In [54], authors proposed an analytical model based on a queuing approach with variable number of servers. They represented service-based business processes horizontal scale adaptation (scale-out/in) focusing on the application layer only. In their approach, authors modeled the input workload as a Poisson process and the whole system as a *Markov Chain*, where Cloud-based services are seen as queuing systems, which global state is given as the size of the waiting queue. Metrics such as the number of deployed instances and average response time were calculated using probabilistic formulas. Authors did not provide a formal verification support.

In [38], authors proposed a formal model for the quantitative analysis of Cloud elasticity at infrastructure level. They used a Markov decision process to model proactive strategies for horizontal scaling elasticity. The elasticity actions are modeled as a non-deterministic behavior and their impact on the system state are modeled as a probabilistic function. In their approach, authors model the global system state as the number of VMs. To verify their approach, authors used a continuous online verification technique. This verification is performed before an adaptation is applied to check and verify its impact on the system.

Globally, these approaches based on the CLTLt(D) [8], Petri nets [5], Markov decision process [38] or Markov chains [54] formal supports, allow considering a Cloud system at a very high level of abstraction. In these works, authors respectively considered number of VMs, number of requests and system queue size as main variables impacting the elasticity controller's decision-making in adapting the controlled system. Given the simple modeling, the approaches provide elasticity solutions at a single Cloud level (i.e., infrastructure or application). A similiar approach for modeling elastic Cloud systems using BRS was proposed in [46]. Authors modeled Cloud structures with bigraphs in three parts: the front-end part, the back-end part and the elasticity controller. They relied on bigraphical reaction rules to express the front/back-end interactions along with the adaptation actions of Cloud configurations at service and infrastructure levels and according to horizontal scale, vertical scale and migration elasticity methods. However, they lacked providing elasticity strategies which operate in an autonomic manner, they did not provide an executable support for their solution and did not quantitatively evaluate the introduced behaviors.

In this paper, we provide an overall deeper modeling expressiveness comparing to [8, 5, 54, 38, 46]. We propose a more complete solution for the design of Cloud elasticity using formal methods which covers the essential phases of modeling, qualitative verification and quantitative analysis. We use Bigraphical reative systems (BRS) formalism to model Cloud systems' hosting environment structure, and to model their elasticity controller as a behavioral entity. The controller is modeled using bigraphical reaction rules alongside with the logic which triggers the reactions. This logic is represented by elasticity strategies which specify the elastic behavior of the Cloud system in a cross-layered manner (i.e., at service and infrastructure levels). This modeling approach enables seeing the elasticity controller as an intrinsic entity of the Cloud system. Therefore, monitoring tasks over the controlled Cloud system enables considering it as "*self-aware*"; and the adaptation actions which are triggered in function of its state enables considering it as "*self-adaptive*" [10]. In addition, we answer to the limitations of [46] by defining a set of elasticity strategies for horizontal scale, vertical scale and migration methods, and for load balancing. We encode the proposed specifications into the Maude formal framework to enable their executability and the formal verification of their correctness basing on an LTL state-based model-checking technique. Furthermore, we

Table 9 Comparative study of formal approaches for Cloud elasticity specification and management

Approach		[8]	[5]	[54]	[38]	[24]	[46]	**Our approach**
Modeling Cloud structure		-	-	-	-	-	√	√
Elasticity methods	Horizontal	√	√	√	√	-	√	√
	Vertical	-	-	-	-	-	√	√
	Migration	-	-	-	-	√	√	√
	Load Balancing	-	√	-	-	√	-	√
Cloud layer	Infrastructure	√	-	-	√	-	√	√
	Platform	-	-	-	-	-	-	-
	Application	-	√	√	-	√	√	√
Elasticity strategies	Reactive	√	√	√	-	-	-	√
	Proactive	√	-	-	√	√	-	-
Formalism / formal model		CLTLt(d)	Petri Nets	Markov chains	Markov decision process	Game theory	BRS	BRS/ Maude
Formal verification technique		SAT/ SMT solvers	Proof verification	-	Continuous verification	Stochastic games model checking	Model-checking	LTL state-based model-checking
Quantitative analysis	Technique	Simulation	Simulation	Probabilistic calculation / Queuing theory	Probabilistic calculation	Simulation	-	Queuing theory / Simulation
	Tooled support	-	-	-	-	√	-	√

provided a queuing based modeling as a quantitative solution for analyzing the designed behaviors in terms of hypothetical performance, costs and efficiency.

Also as related work using mathematical analysis on self-adapting Cloud systems, authors of [24] use game theory to study the decision-making behaviors to perform adaptations in a Cloud environment. Elements of SLA contracts are encoded into a set of quality-oriented objectives as desired output for the adaptations. resources variation and uncertainty concerns in Cloud environments are considered to model the adaptation behaviors as stochastic games and the quality objectives as a variant of temporal logic. Authors consider a Cloud environment as a set of Cloud collaborators (servers). When a Cloud realizes that it cannot execute an application due to its limited resources, it requires to find the optimal collaborator that can do the job. Game theory is used to provide proactive optimal decision-making strategies to find the optimal collaborator. In our work, we propose a solution to study how a single Cloud system could self-reconfigure in terms of resources capability, i.e., by (de)provisioning resources at infrastructure (VMs, offering) and application (service instances) layers. Particularly, we focus more on the modeling on all elasticity methods (horizontal/vertical scaling, migration and load balancing) in a complemetary way. However, the migration and load balancing strategies that we proposed are similar to the behaviors proposed in [24]. The difference is that we describe how service instances and requests are migrated and redirected across VMs and services; especially given the adapting, thus varying, amount of such VMs and services (according to our strategies given the fluctuating workload). Our behaviors are reactive and go towards ensuring (or at least converging towards) the desired "stable" state of the entire managed Cloud system's hosting environment. Authors of [24] focus on the proactive optimization problem of finding a Cloud collaborator in order to meet the modeled quality objectives. Cloud elasticity mechanisms are not studied in their approach. The approach is verified using a stochastic games model-checking technique. It is simulated for quantitative study purposes using the PRISM-games tool.

Table 9 gives a comparative study of our approach with the presented papers in the literature about formal solutions for Cloud elasticity specification and management. Globally, it shows that the presented approach in this paper incarnates a complete formal solution that distinguishes from the others with a considerable modeling expressiveness, particularly in terms of Cloud structure and elastic behaviors. Furthermore, it shows that it is formally verified and quantitatively studied by simulation.

Besides, control theory was used for resource management in distributed [30] and Cloud [33] systems. Control theory is commonly used to design controllers that manage the behavior of dynamical systems. Such systems have a desired output, called "*the reference*", which is often defined as mathematical functions of various complexity. Control theoretical-based systems are often designed as control loops which principle is the following [49]: when one or more output variables of the system need to follow a certain reference over time, a controller manipulates the inputs to a system to obtain the desired effect on the output of the system. One of the main limitations of this approach is the non-linearity of most inter-relationships in computing systems [55]. This requires designing non-linear and adaptive controllers that are particularly difficult to implement and to verify qualitatively. In this paper, we inspire from closed-loop based approaches to design our elasticity controller. It aims at having the controlled Cloud system reach a "*stable*" global state (which is defined in first-order logic) by relying on elasticity strategies we specified using BRS. The Maude-based encoding of these behaviors ensures autonomic execution of the elastic adaptations. And the Maude's LTL model-checker enables verifying the correctness of the adaptations regarding the reachability of the "*stable*" state.

Furthermore, authors of [9] discuss the verification of non-functional properties (NFPs) in Cloud systems. In their work, the authors describe how a formal verification technique, called "runtime quantitative verification", can be used to (1) verify evolving Cloud systems continually and (2) to guide this evolution towards meeting configurations that are guaranteed to satisfy the system non-functional requirements (NFRs). According to the authors, continual verification can be used to manage the reliability of service deployed on Cloud infrastructure. Such verification enables quantifying the impact of both planned and unexpected variations on the reliability of services. It can provide answers to some questions such as : "*how many additional VMs should be used to run a service over time considering their resources variations ?*". The authors expose different formal verification techniques such as "*compositional NFR verification*" and "*incremental NFR verificaiton*". These techniques can be used to continually verify NFPs of service-based systems *on the fly*, using the PRISM probabilistic model checker. In our work, we used a state-based model-checking technique supported by the temporal lineal logic (LTL) to verify the NFRs linked to elastic Cloud systems. Precisely, we showed that the reachability of the desired "stable" state is ensured using the defined behaviors. In other words, we ensure the *liveness* NFP. Most importantly, unlike the strict quantitative nature of the NFRs to be met by the verification technique in [9], our conducted verification is mainly qualitative. It is categorized with flexible verification goals that depend on the controlled system's configuration itself (i.e., the cross-layer defined thresholds and the deployed resources). Precisely, we define symbolic states as first order logic predicates to identify the desirable and undesirable states. We defined a *Kripke* structure to identify the high-level goals (i.e., NFRs) as LTL formulas describing the desirable system evolution over time (i.e., the correctness of the designed behaviors). In future work, we think that our approach could significantly benefit from integrating a continual verification technique to extend its verification capabilities, especially in terms of quantitative verification on runtime.

6.2 Solutions for Cloud elasticity monitoring and management

In [20], authors present an autonomic resource provisioning approach based on the MAPE control loop concept. The proposed approach is implemented as a resource provisioning framework which supports the MAPE control loop. The proposed approach is simulated and evaluated in the CloudSim Toolkit to analyze how the managed system could dynamically adapt to uncertainties, sudden changes and workload spikes, dealing with the undesirable states of over-provisioning and under-provisioning. The solution is evaluated in terms of performance under both smooth and bursty workloads. The obtained results are compared with other approaches, and showed that the proposed solution increases the resources utilization and decreases the total cost, while avoiding SLA violations.

Authors of [13] proposed a framework to perform Cloud systems' elastic behaviors and to monitor these behaviors at runtime. In their approach, authors interpreted service-based Cloud application platforms as sensor networks in order to apply sensor web techniques for PaaS-level autonomic management. The proposed solution promotes extensibility, scalability, platform independence, genericity and complementarity to other approaches of auto-scaling management.

In [36] authors introduced concepts and techniques for monitoring and analyzing the elastic behavior of Cloud services according to their elasticity controller introduced in [12]. Cross-layer metrics were introduced to link service level with its underlying infrastructure level monitoring information and to derive higher level information from it. Authors introduced a framework which provides features and functions for real-time cross-layer analysis of elastic Cloud services.

Authors of [14] presented a framework for maximizing Cloud service availability. In their approach, Cloud service availability is preserved through virtual machines migration. Authors proposed a solution to specify which services should be migrated, and when and where these services should be migrated in response to anomalous events, such as workload peaks, which impact performance and availability. Algorithms for monitoring and controlling a Cloud service were proposed and evaluated through an experimental study.

In this paper, we propose a complete solution of robust mathematical foundations to specify Cloud systems and their elastic behaviors. We model Cloud systems using the BRS formalism in order to (1) integrate the complexity of their cross-layer architectures (i.e., application, infrastructure and computing resources) and to (2) specify an elasticity controller which governs their elastic behaviors. Precisely, we propose elasticity strategies which provide a logic that governs the elasticity controller's decision-making for horizontal scale, vertical scale and migration elasticity methods together with load-balancing in a cross-layered manner. In the first part of our contributions, we propose a rewriting-logic-based solution for executing the BRS specifications, which is supported by the formal specification language and system called Maude. In addition, Maude, as a semantic framework, enables conducting a qualitative verification of the introduced approach through a LTL state-based model-checking technique [6]. Typically, we reason over a controlled Cloud system's global state which is obtained in function of the conjunction of all the entities (service instances, virtual machines, etc.) that compose the system. The qualitative verification consists of checking the reachability of the system's global *stable* state, which categorizes the absence of *over-provisioning* and *under-provisioning* states.

In the second part of our contributions, we propose a tooled queuing-based solution to simulate and to illustrate, in a quantitative point of view, the hypothetical performance, costs and efficiency induced by the designed elasticity controller's behaviors. Similarly to [20], we propose an autonomic MAPE-based control loop to simulate the runtime of a Cloud system according to the designed elasticity strategies. Regarding the monitoring methodoloy, our solution reaches the philosophy of [13] in their search for identifying virtual sensors to gather information about the system state. Precisely, our proposed modeling approach enables representing a Cloud system cross-layer configuration as a forest of overlapping nodes which enables isolating sets of services according to their host VMs, for example. The obtained sets are then analyzed to gather information (i.e., monitored) using first-order logic predicates, regarding the introduced upper hosting thresholds. This solution provides a similar solution than [36] for specifying complex cross-layer monitoring metrics.

Unlike [14] and the other referenced works, we do not validate our approach's performance by comparison to other approaches. We use the *Erlang-C* formula which gives a mathematical ideal to be reached in terms of provisioned resources and response time. In this paper, our goal is not to provide the best approach for elasticity management. Instead, we intend to provide an original solution to design and illustrate different composable elasticity strategies in order to obtain different high-level policies. Precisely, we showed through our simulation-based experimental study that different strategies compositions allow to achieve *high performance*, *high economy*, *high infrastructure availability*, *infrastructure cost optimization* or *high resource efficiency* high-level policies.

Unlike [20] and [13] who evaluate elastic behaviors to diagnose SLAs (*Service Level Agreements*) [39] violations, our approach can be applied in real-life services in order to establish SLAs between a Cloud service provider and a Cloud infrastructure provider rather than diagnose them. We showed that the introduced strategies induce correct behaviors regarding a system's elasticity, in front of unpredictable workload activity. By applying the proposed tooled support, a service provider could simulate their product in an infinity of situations while taking into account their constraints and requirements. They could ultimately estimate the required amount of resources to achieve a desired level of service. By varying the possibilities of scaling adaptation (i.e., elasticity strategies), a service provider could ultimately elaborate the suitable agreements with a Cloud provider (i.e., SLA) which satisfy the most their needs and requirements while ensuring, at the best, their desired high-level policies and constraints.

7 Conclusion

In this paper, we propose a formal-based approach to design, qualitatively verify and quantitatively analyze Cloud elasticity, basing on an original and complementary combination of formal models and logics. Namely: Bigraphical Reative Sysmtes (BRS), rewriting logic, Linear Temporal Logic (LTL) and Queuing theory.

We have provided a modeling approach for Cloud systems' structure and elastic behaviors based on BRS. We have used bigraphs and bigraphical reaction rules to express both aspects respectively. These behaviors implement the elasticity controller and are described by elasticity strategies. We have proposed different strategies for horizontal scale, vertical scale, migration and load-balancing adaptations to (de)provision and optimize Cloud system resources consumption in a cross-layered manner (i.e., at service and infrastructure levels). Strategies describe the logic that enables the elasticity controller to reason over the entire Cloud system's state and manage its elastic adaptations. In order to identify the desirable behaviors of the designed elasticity controller (i.e., their correctness), we have defined a *Kripke* structure which identifies the desirable and undesirable states, and which enables describing the desirable temporal evolution of the system as formulas expressed in LTL.

One step further, we have encoded the BRS-based modeling approach and the LTL specifications into the Maude formal language, which is an implementation of Rewriting logic, to provide a generic executable solution for elasticity in Cloud systems. We have also provided a formal verification of the designed elastic behaviors' correctness, basing on the LTL model-checker integrated in Maude. Besides, we have presented an original way to compose different elasticity strategies at both service and infrastructure levels to provide multiple high-level elastic behaviors.

Finally, we have proposed a queuing-based approach to conduct experimental analysis by simulation of the different elasticity strategies combinations in order to provide a quantitative study of the adaptations. We showed that the introduced elasticity strategies could be composed in a cross-layered manner in order to provide several high-level policies for different patterns of performance, cost and efficiency. We have demonstrated that our introduced elasticity strategies ensure different high-level policies: *high performance, high economy, infrastructure availability, infrastructure costs optimization* and *high resource efficiency*.

As future work, we aim at enlarging the specifications of Cloud system's elastic behavior. Our goal is to provide a more complete solution which considers horizontal and vertical scale elasticity coordination to achieve hybrid Cloud resources' dynamic management.

References

1. Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.
2. Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2017.
3. Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*, pages 204–212. IEEE, 2012.
4. Maryam Amiri and Leyli Mohammad-Khanli. Survey on prediction models of applications for resources provisioning in cloud. *Journal of Network and Computer Applications*, 82:93–113, 2017.
5. Mourad Amziani. *Modeling, evaluation and provisioning of elastic service-based business processes in the cloud.* PhD Thesis, Institut National des Télécommunications, 2015.
6. Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.
7. Bruno Baynat. Théorie des files d'attente. *Hermès, Paris*, 2000.
8. Marcello M Bersani, Domenico Bianculli, Schahram Dustdar, Alessio Gambi, Carlo Ghezzi, and Sr\d jan Krstić. Towards the formalization of properties of cloud-based elastic systems. In *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, pages 38–47. ACM, 2014.
9. Radu Calinescu, Kenneth Johnson, Yasmin Rafiq, Simos Gerasimou, Gabriel Costa Silva, and Stanimir N Pehlivanov. Continual verification of non-functional properties in cloud-based systems. Citeseer, 2013.
10. Tao Chen, Rami Bahsoon, and Xin Yao. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Surveys (CSUR)*, 51(3):61, 2018.
11. Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude Manual (Version 2.7. 1). 2016.
12. Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Multi-level elasticity control of cloud services. In *International Conference on Service-Oriented Computing*, pages 429–436. Springer, 2013.
13. Rustem Dautov, Iraklis Paraskakis, and Mike Stannett. Towards a framework for monitoring cloud application platforms as sensor networks. *Cluster Computing*, 17(4):1203–1213, Dec 2014.
14. Mamadou H. Diallo, Michael August, Roger Hallman, Megan Kline, Scott M. Slayback, and Christopher T. Graves. Automigrate: a framework for developing intelligent, self-managing cloud services with maximum availability. *Cluster Computing*, 20(3):1995–2012, 2017.
15. Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of elastic processes. *IEEE Internet Computing*, 15(5):66–71, 2011.
16. Mohamed Firdhous, Osman Ghazali, and Suhaidi Hassan. Modeling of cloud system using Erlang formulas. In *Communications (APCC), 2011 17th Asia-Pacific Conference on*, pages 411–416. IEEE, 2011.
17. Leo Freitas and Paul Watson. Formalizing workflows partitioning over federated clouds: multi-level security and costs. *International Journal of Computer Mathematics*, 91(5):881–906, 2014.
18. Guilherme Galante and Luis Carlos E de Bona. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 263–270. IEEE Computer Society, 2012.
19. Mostafa Ghobaei-Arani, Sam Jabbehdari, and Mohammad Ali Pourmina. An autonomic approach for resource provisioning of cloud services. *Cluster Computing*, 19(3):1017–1036, Sep 2016.

20. Arne John Glenstrup, Troels Christoffer Damgaard, Lars Birkedal, and Espen Højsgaard. An implementation of bigraph matching. *IT University of Copenhagen*, page 22, 2007.
21. Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (${$ICAC$}$ 13)*, pages 23–27, 2013.
22. Abdul R Hummaida, Norman W Paton, and Rizos Sakellariou. Adaptation in cloud resource configuration: a survey. *Journal of Cloud Computing*, 5(1):7, 2016.
23. Azlan Ismail and Marta Kwiatkowska. Synthesizing pareto optimal decision for autonomic clouds using stochastic games model checking. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 436–445. IEEE, 2017.
24. Bart Jacob, Richard Lanyon-Hogg, Devaprasad K Nadgir, and Amr F Yassin. A practical guide to the ibm autonomic computing toolkit. *IBM Redbooks*, 4(10), 2004.
25. Khaled Khebbeb, Nabil Hameurlain, and Faiza Belala. Modeling and evaluating cross-layer elasticity strategies in cloud systems. In El Hassan Abdelwahed, Ladjel Bellatreche, Mattéo Golfarelli, Dominique Méry, and Carlos Ordonez, editors, *Model and Data Engineering*, pages 168–183, Cham, 2018. Springer International Publishing.
26. Khaled Khebbeb, Nabil Hameurlain, Faiza Belala, and Hamza Sahli. Formal modelling and verifying elasticity strategies in cloud systems. *IET Software*, 13(1):25–35, 2018.
27. Shinji Kikuchi and Kunihiko Hiraishi. Improving reliability in management of cloud computing infrastructure by formal methods. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–7. IEEE, 2014.
28. Loic Letondeur. *Planification pour la gestion autonomique de l'élasticité d'applications dans le cloud.* PhD Thesis, Université de Grenoble, 2014.
29. Xue Liu, Xiaoyun Zhu, Sharad Singhal, and Martin Arlitt. Adaptive entitlement control of resource containers on shared servers. In *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005.*, pages 163–176. IEEE, 2005.
30. Vladimir V Mazalov and Andrei Gurtov. Queuing system with on-demand number of servers. *Mathematica Applicanda*, 40(2):1–12, 2012.
31. Peter Mell, Tim Grance, and others. The NIST definition of cloud computing. 2011.
32. Milton Mendieta, César A Martín, and Cristina L Abad. A control theory approach for managing cloud computing resources: a proof-of-concept on memory partitioning. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6. IEEE, 2017.
33. Robin Milner. Bigraphs and their algebra. *Electronic Notes in Theoretical Computer Science*, 209:5–19, 2008.
34. Robin Milner. *The space and motion of communicating agents.* Cambridge University Press, 2009.
35. Daniel Moldovan, Georgiana Copil, Hong Linh Truong, and Schahram Dustdar. MELA: elasticity analytics for cloud services. *IJBDI*, 2(1):45–62, 2015.
36. Francesc D Muñoz-Escoí and José M Bernabéu-Aubán. A survey on elasticity management in paas systems. *Computing*, 99(7):617–656, 2017.
37. Athanasios Naskos, Emmanouela Stachtiari, Anastasios Gounaris, Panagiotis Katsaros, Dimitrios Tsoumakos, Ioannis Konstantinou, and Spyros Sioutas. Cloud elasticity using probabilistic model checking. *arXiv preprint arXiv:1405.4699*, 2014.
38. Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009.
39. Gian Perrone, Søren Debois, and Thomas T Hildebrandt. A model checker for bigraphs. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1320–1325. ACM, 2012.
40. Mariam Rady. Formal definition of service availability in cloud computing using OWL. In *International Conference on Computer Aided Systems Theory*, pages 189–194. Springer, 2013.
41. Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE, 2011.
42. Kristin Y Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.
43. Manuj Sabharwal, Abhishek Agrawal, and Grace Metri. Enabling green it through energy-aware software. *IT Professional*, 15(1):19–27, 2013.
44. Hamza Sahli, Faiza Belala, and Chafia Bouanaka. Model-Checking Cloud Systems Using BigMC. In *VECoS*, pages 25–33, 2014.
45. Hamza Sahli, Nabil Hameurlain, and Faiza Belala. A bigraphical model for specifying cloud-based elastic systems and their behaviour. *International Journal of Parallel, Emergent and Distributed Systems*, 32(6):593–616, 2017.
46. Rob Schoren. Correspondence between kripke structures and labeled transition systems for model minimization. In *Seminar project, Technische Universiteit Eindhoven, Department of Computer Science*, 2011.
47. Michele Sevegnani and Muffy Calder. BigraphER: rewriting and analysis engine for bigraphs. In *International Conference on Computer Aided Verification*, pages 494–501. Springer, 2016.
48. Stefan Simrock. Control theory. 2008.
49. Steam. Steam, the ultimate online game platform. https://store.steampowered.com/about/, 2019. [Browsed on 2019-02-23].
50. SteamSpy. Steamspy - all the data about steam games. https://steamspy.com/year/, 2019. [Browsed on 2019-02-23].
51. Basem Suleiman, Sherif Sakr, Ross Jeffery, and Anna Liu. On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure. *Journal of Internet Services and Applications*, 3(2):173–193, 2012.
52. Demetris Trihinas, Chrystalla Sofokleous, Nicholas Loulloudes, Athanasios Foudoulis, George Pallis, and Marios D Dikaiakos. Managing and monitoring elastic cloud applications. In *International Conference on Web Engineering*, pages 523–527. Springer, 2014.
53. Lydia Yataghene, Mourad Amziani, Malika Ioualalen, and Samir Tata. A queuing model for business processes elasticity evaluation. In *Advanced Information Systems for Enterprises (IWAISE), 2014 International Workshop on*, pages 22–28. IEEE, 2014.
54. Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, Pradeep Padala, and Kang Shin. What does control theory bring to systems research? *ACM SIGOPS Operating Systems Review*, 43(1):62–69, 2009.

DOI:https://doi.org/10.1007/s10586-020-03080-8

Khaled Khebbeb received a Ph.D. degree in computer science from the University of Pau, France and the University of Constantine 2, Algeria in 2019. Since September 2018, he is a research and teaching assistant at the University of Pau and is affiliated to the MOVIES team of the LIUPPA laboratory. His research interests include software engineering and formal modeling of self-adaptive software systems applied on cloud and service-oriented computing.

Nabil Hameurlain received a Ph.D. degree in computer science from the University of Toulouse, France in 1998 and a HdR (French Habilitation to become Research Activity Supervisor) in computer science from the University of Pau in 2011. Since October 1999, he is associate professor at the University of Pau and the head of MOVIES team at LIUPPA Laboratory. His main research interests include software engineering for distributed and self-adaptive software systems, with a particular focus on cloud and service oriented computing.

Faiza Belala received a Ph.D. degree in computer science from Mentouri University of Constantine in 2001. She is currently a professor at the same university and head of the GLSD team (LIRE Laboratory). Her current research focuses on architecture description languages, formal refinement (Rewriting Logic, Bigraphs, Petri nets, ect.), mobility and concurrency aspects in software architectures, formal analysis of distributed systems. She has organized and chaired the international conferences on Advanced Aspects of Software Engineering ICAASE 2014/2016/2018, she is the author of many refereed journal articles and peer reviewed international and regional conference papers. She has supervised over sixty Master and Ph.D. theses.

DOI:https://doi.org/10.1007/s10586-020-03080-8