Formal modelling and verifying elasticity strategies in cloud systems

Khaled Khebbeb, Nabil Hameurlain, Faiza Belala, Hamza Sahli

Formal Modeling and Verifying Elasticity Strategies in Cloud Systems

Khaled Khebbeb[1,2*], Nabil Hameurlain[2], Faiza Belala[1], Hamza Sahli[1]

[1] LIRE Laboratory, Constantine 2 University – Abdelhamid Mehri, Constantine, Algeria
[2] LIUPPA Laboratory, University of Pau, Pau, France
[*]khaled.khebbeb@{univ-constantine2.dz, univ-pau.fr}

Abstract: Elasticity property allows cloud systems to adapt to their input workload by provisioning and deprovisioning resources as the demand grows and drops. However, due to the unpredictable nature of workload, providing accurate action plans to manage a cloud system's elasticity is a particularly challenging task. In this paper, we propose a BRS (short for *Bigraphical Reactive Systems*) based approach to provide a formal modeling of cloud systems' structure using *bigraphs,* and their elastic behaviors using *bigraphical reaction rules*. We introduce elasticity strategies to describe cloud systems' auto-adaptation behaviors. One step further, we encode the bigraphical specifications into Maude language to enable an autonomic executability of the elastic behaviors and verify their correctness. Finally, we propose a queuing-based approach to discuss and analyze elasticity strategies in cloud systems through different simulated scenarios.

1. Introduction

Cloud computing is a novel paradigm [1] that has gained a great interest in both industrial and academic sectors. It consists of providing a set of virtualized resources (servers, virtual machines, services, 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). A cloud system has many characteristics that make it very attractive such as high availability, flexibility and cost effectiveness. However, the most appealing feature for cloud users, and what distinguishes cloud computing from other models is elasticity property. Elasticity was defined as:

"The degree to 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" [2].

Elastic behaviors are implemented by an elasticity controller, an entity usually based on a closed control loop [3]. It *autonomically* decides of different *adaptation* actions to be triggered aiming at efficiently control cloud *resources* provisioning according to *workload* fluctuations. The adaptations (horizontal, vertical, etc.) [4] consist of provisioning and deprovisioning resources in order to maintain an adequate quality of service (QoS) while minimizing operating costs [5].

Controlling a system's elasticity rely on many overlapping factors such as the available resources, current workload, etc. Besides, defining desired elastic behaviors and checking their correctness, considering their hard-to-determine effects on the system's behavior, rise important concerns. Managing these dependencies significantly increases the difficulty of specifying and verifying cloud systems' elasticity. Thus, designing such behaviors can be a particularly challenging task. To address this challenge, formal methods characterized by their efficiency, reliability and precision present an effective solution to deal with all of these aspects.

In the last few years, some researches like [6–8] proposed formal modeling approaches of elasticity in cloud systems. They relied on different formalisms and theories such as *Petri nets*, *Markov Chains*, *Temporal logic* or *Queuing Theory*. These approaches globally proposed partial solutions for modeling elastic behaviors of cloud systems (i.e., specification, execution, verification and evaluation phases weren't fully covered). Precisely, most of these works focused on one single cloud layer (service or infrastructure) and didn't address cross-layer elasticity (at both levels). Most importantly, they lacked providing an autonomic executability of the introduced behaviors and globally relied on simulations for the verification and evaluation phases of the model. Finally, none of these works addressed modeling cloud systems' structures in the specification phase.

In this paper, we take a first step towards these directions. We contribute by providing a complete formal modeling approach that reduces the complexity of specifying, executing, verifying and evaluating cloud systems and their elastic behaviors. We adopt *Bigraphical Reactive Systems* (BRS) [9, 10] as a semantic framework for specifying structural and behavioral aspects of elastic cloud systems. We use *bigraphs* and *bigraphical reaction rules* to address both aspects. Bigraphs are used to model structures of cloud systems and the elasticity controller. We use bigraphical reaction rules to describe elastic behaviors of a cloud system. Precisely, we propose elasticity strategies for horizontal-scale (de)provisioning of cloud resources at service and infrastructure scopes (i.e., in a cross-layered manner).

One step further, we encode the BRS specifications into Maude language to provide an autonomic executability of the elastic behaviors and to formally verify their correctness. We proceed to a state-based model-checking [11] of elastic behaviors relying on *Linear Temporal Logic* (LTL).

Focusing on the definition of elasticity, some key concepts are important to consider. Witnessing *workload changes* (i.e., *the demand*) and estimating the *available resources* are significant in capturing the global system state regarding its elasticity. These tasks require *Monitoring* the system during its evolution in order to determine its performance, its elasticity and the correctness of the latter which is determined by the accuracy of the adaptations [12]. Actually, the analysis of performance in elastic cloud systems remains a notably challenging task due to the fluctuating and unpredictable nature of input workload [13]. Researchers have been using mathematical methods like *Queuing Theory* [14] and *Markov Chains* [15] to model input workload and its impact over the system's behavior.

Another contribution of this paper consists of providing an experimental analysis of cloud systems' elasticity strategies basing on a queuing approach. We study the system's adaptation capabilities according to different execution scenarios. We study and discuss the ability of a cloud system to adapt to its varying workload by (de)provisioning resources when needed, according to the adaptation strategies we introduced. To this purpose, we designed a tool to simulate and monitor these behaviors.

The remainder of the paper is structured as follows. In Section 2, we introduce the elasticity controller and explain its role in cloud elasticity management. In Section 3, we briefly give an overview of BRS formalism and detail our BRS-based approach to specify cloud systems and their elastic behaviors. In Section 4, we encode the bigraphical specifications into Maude language and provide a formal verification of the introduced behaviors' correctness. In Section 5, we introduce a queuing-based model to analyze elasticity and propose an experimental analysis and discussion of cloud systems elasticity. In Section 6, we review the state of art on formal specification and verification of elastic cloud systems. Finally, we summarize and conclude the paper in Section 7.

2. Elasticity Controller and the Elastic Behavior

In elastic cloud systems, resource provisioning can be adjusted by an elasticity controller. This entity decides of the adaptation rules to be triggered in order to scale the cloud system in such a way that resource provisioning matches the minimum requirements as closely as possible. This is done with taking into account many factors as the available resources, current workload, system state, etc. [2]. The elasticity controller is usually considered to operate according to a closed control loop derived from IBM's autonomic control loop known as MAPE for *Monitor*, *Analyze*, *Plan and Execute* [3]. In [16, 17], the controller is considered to be constituted by different entities that interact with each other to implement the main phases of the control loop. *Monitoring* and *Execution* phases are usually considered to be handled by entities that monitor the system (by means of sensors) and apply actions (using effectors) that *Planning* decides, in response to the flaws identified at *Analysis* phase.

At a high level of abstraction, the elastic behavior of a cloud system takes the form of a closed loop architecture as shown in Figure 1. A cloud system receives *end-users'* requests through its *front-end* interface. The intensity of received requests (i.e., *input workload*) might oscillate in an unpredictable manner. The growing workload, thus the *system's load* can result in a degradation of users *Quality of Experience (QoE)* (e.g. performance drop). Thus, more resources need to be provisioned to cope with the demand. The controlled system (i.e., cloud *hosting environment*) is hosted by the *cloud infrastructure provider* who provides costs to the cloud *service provider* proportionally to the provisioned resources (i.e., according to a *pay-per-use* policy). When input workload drops, the eventual unnecessarily allocated resources are still billed and need to be disposed.

To ensure these behaviors, the *elasticity controller* periodically 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. The challenging part here is how to implement a logic that enables the elasticity controller to ensure auto-adaptation behaviors over a managed cloud system. This is accomplished by triggering adaptation rules according to particular conditions that represent elasticity anomalies to resolve. To tackle this challenge, we adopt a bigraphical approach to model both structural and behavioral aspects of the cloud back-end part and the elasticity controller.

In [17], authors provide a cloud systems' design structured in three parts: the *front-end*, the *back-end* and the *elasticity controller*. In this paper, we focus on the elasticity controller and the managed back-end part. Besides, we extend their work by introducing elasticity strategies that describe an elastic behavior by means of bigraphical reaction rules. In addition, we provide an executability and correctness verification of the defined elastic behaviors. Thus, we endow the elasticity controller with autonomic management of the controlled cloud system's elasticity. Finally, we provide a quantitative analysis of elastic behavior using a queuing model.

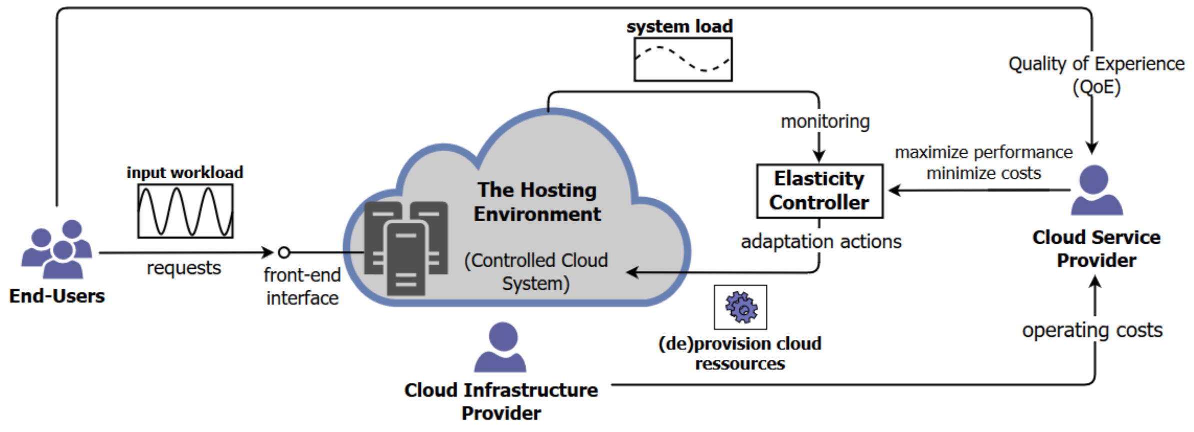


Fig. 1. *Top view of the elastic behavior loop*

3. A BRS Model for Elastic Cloud Systems

3.1. Bigraphical Reactive Systems Overview

Bigraphical reactive systems (BRS) are a recent formalism introduced by Milner [10] for modeling the temporal and spatial evolution of computation. It provides a graphical model that emphasizes both connectivity and locality. A BRS consists of a set of *bigraphs* and a set of *reaction rules* that define the dynamic evolution of the system by specifying how the set of bigraphs can be reconfigured.

Graphical notation and interface: Figure 2 depicts an example of a bigraph representation. Dashed rectangles denote *regions* describing separate parts of the system. *Nodes* are depicted by circles and represent the physical or logical components of the system. Each node has a type, called *control*, denoted by labels *A* and *B.* A *signature* is the set of controls of a bigraphs. A node can have zero, one or many *ports* which represent possible connections. Ports are depicted by bullets. In the example, connections are represented as *links*, depicted by curvy lines, which may connect ports and names (x, y and z). These links, also called *hyperedges*, indicate the bigraph's connectivity (e.g., they can be considered as (potential) links to other bigraphs). *Sites*, modeled with grey squares, encode parts of the model that have been abstracted away. A bigraph possibilities to interact with its external environment are visible through its *interface*. For example, B: 0 ⟶ <2, {x, y}> indicates that bigraph B has zero sites, two regions and its names are x and y.

Note that a bigraph also has algebraic notations that are equivalent to graphical ones. For instance, merge product F | G denotes the juxtaposition of bigraphs F and G which is then placed inside a single region. Nesting operation F.G allows to place bigraph G inside F and parallel product (||) term may be used to compose bigraphs by juxtaposing their roots and merging their common names. More details about bigraphs can be found in [9].

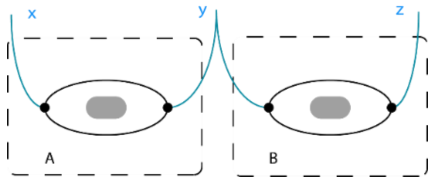


Fig. 2. *Example of a bigraph*

Bigraphs Sorting: Classification of controls and links for a bigraph is performed using sorts. A *sorting discipline* is a triple $\Sigma = \{\Theta, \mathrm{K}, \Phi\}$, where Θ is a non-empty set of sorts, K is a signature, and Φ is a set of *formation rules*. A formation rule is a set of properties a bigraph has to satisfy. Disjunctive sorts are written as $\widehat{ab}$, expressing that a node can either be of sort *a* or sort *b*.

Bigraphical Reactive Systems: A Bigraphical Reactive System (BRS) consists of a set of bigraphs representing the state of the system and a set of reaction rules defining how the system evolves (by going from one configuration to another). A reaction rule *Ri* is a pair (*R*, *R'*), where *redex R* and *reactum R'* are bigraphs that have the same interface. The evolution of a system *St* is derived by checking if *R* is a match [18] in *St* and by substituting it with *R'* to obtain a new system *St'*. The evolution is noted $St \xrightarrow{Ri} St'$.

Concrete and Abstract Bigraphs: A bigraph is defined by a place graph and a link graph on the same set of nodes. The difference between concrete and abstract bigraphs lies on a simple subtility [18]: concrete bigraphs are represented with named nodes and internal edges (i.e., that connect nodes only) thus providing an exhaustive *cliché* of a system configuration. Abstract bigraphs, equivalence class of the concrete ones, represent nodes with their controls only and omit internal edges' names. This allows the specification of more general system configurations. In this paper, we use abstract bigraphs in order to provide a generic modeling approach.

3.2. Modeling Cloud Structures

An elastic cloud system is represented by a bigraph *CS* involving all cloud architectural elements. Bigraph *CS* is composed of two regions, noted 0 and 1 that respectively represent the hosting environment and the elasticity controller parts of the elastic cloud system. This configuration is obtained by the parallel composition of hosting environment (back-end) and elasticity controller bigraphs as shown in [17]. The introduced sorting logic defines mapping rules and expresses all constraints and formation rules, that *CS* satisfies to ensure proper and precise encoding of cloud semantics into BRS concepts. Formal definitions are given in what follows.

Definition 1: Formally, a bigraph CS modeling an elastic cloud system is defined as follows.

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

- V_{CS} *and* E_{CS} are sets of nodes and edges of the bigraph *CS*.
- $ctrl_{CS} : V_{CS} \to K_{CS}$ a control map that assigns each node $v \in V_{cs}$ with a control $k \in K_{cs}$.
- $CS^P = (V_{CS}, ctrl_{CS}, prnt_{CS}): m_{CS} \to n_{CS}$ is the place graph of *CS* where $prnt_{CS}: m_{CS} \uplus V_{CS} \to V_{CS} \uplus n_{CS}$ is a parent map. m_{CS} and n_{CS} are the number of sites and regions of bigraph *CS*.
- $CS^L = (V_{CS}, E_{CS}, ctrl_{CS}, link_{CS}): X_{CS} \to Y_{CS}$ represents link graph of *CS*, where $link_{CS}: X_{CS} \uplus P_{CS} \to E_{CS} \uplus Y_{CS}$ is a link map, X_{CS} and Y_{CS} are respectively inner 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 *CS*.

Definition 2: The sorting discipline associated to *CS* is a triple $\Sigma_{CS} = \{\Theta_{CS}, \mathrm{K}_{CS}, \Phi_{CS}\}$.

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.

Table 1 gives for each cloud concept, mapping rules for BRS equivalence. This 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 purposes and 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. However, all nodes representing servers are of sort *e*.

Table 2 gives the formation rules Φ0-12 that bring construction constraints over the BRS specification. Formation rules give structural constraints over the BRS model.

Table 1 Controls and sorts for bigraph *CS*

Cloud element	Control	Arity	Sort
Hosting environment part (region 0)			
Server	SE	3	e
Overloaded Server	SE^{L}	3	e
Unused Server	SE^{U}	3	e
Virtual Machine	VM	2	v
Overloaded VM	VM^{L}	2	v
Unused VM	VM^{U}	2	v
Service instance	S	1	s
Overloaded service instance	S^{L}	1	s
Unused service instance	S^{U}	1	s
Request	q	0	q
Elasticity controller part (region 1)			
Evaluator	EV	1	o
Monitor	MO	2	m
Effector	E	2	f

Rule Φ0 specifies that servers are at the top of the hierarchical order of deployed entities in the back-end region. Rules Φ1-3 give the structural disposition of a hosting environment where a server hosts VMs, a VM runs service instances and a service instance handles requests. All connections are port-to-port or port-to-name links to illustrate possible communication capabilities between the different cloud entities. In Φ6-7, we use name *w*, for *workload*, to illustrate the connections the cloud system has with its abstracted front-end part. A server is linked to its hosted entities, that represent resources virtualization (VMs). A VM is linked to service instances it is running. The back-end is managed by the elasticity controller through c-name edge for *control* (Φ6 and Φ11). In Φ8, we structurally represent MAPE phases with nodes and consider that *Evaluator* node regroups *Analysis* and *Planning* phases. Φ12 states that monitor, effector and evaluator entities are always linked. In Rules Φ4 and Φ9, active elements may take part is reactions while passive ones won't. In Φ5 and Φ10, atomic nodes do not have children.

Table 2 Conditions of formation rules Φ_{CS} for bigraph *CS*

	Rule description
Φ0	All children of a 0-region (back-end part) have sort e
Φ1	All children of an e-node have sort v
Φ2	All children of a v-node have sort s
Φ3	All children of an s-node have sort q
Φ4	All $\widehat{evs}$-nodes are active
Φ5	All q-nodes are atomic
Φ6	In a e-node, one port is always linked to a w-name, another port is always linked to a c-name and the other may be linked to v-nodes
Φ7	In a v-node, one port may be linked to e-nodes and the other may be linked to s-nodes
Φ8	All children of a 1-region (elasticity controller part) have sort in {o, m, f}
Φ9	$\widehat{omf}$-nodes are passive
Φ10	$\widehat{omf}$-nodes are atomic
Φ11	$\widehat{mf}$-nodes are always linked to a c-name
Φ12	An o-node is linked to $\widehat{mf}$-nodes, a m-node is linked to $\widehat{of}$-nodes and a f-node is linked to $\widehat{om}$-nodes

Bigraphical Example of a Cloud System: Consider an online voting service S running on a cloud system VS. In its hosting environment part, as an initial configuration, the service is deployed on one single online server SE. The server hosts one virtual machine instance VM which is running one instance of the service S. Figure 3 shows a bigraphical representation of the cloud system VS. Its algebraic form focusing on its locality (i.e., place graph) is given with:
$VS \stackrel{\text{def}}{=} ((\mathrm{SE}.(\mathrm{VM}.(\mathrm{S}.\mathrm{q}|d3)|d2)|d1)|d0) \parallel (\mathrm{MO}|\mathrm{EV}|\mathrm{E}|d4)$.

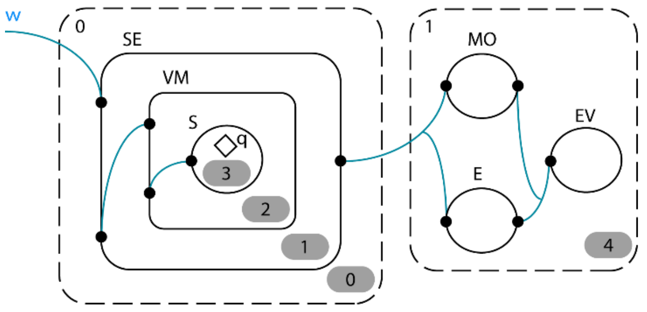


Fig. 3. *Example of a cloud-based system bigraph CS*

Elasticity controller bigraph is connected to hosting environment part by parallel composition and merging on name c (which is abstracted as we are using abstract bigraphs). Notice that the shown bigraph respects our defined construction rules.

3.3. Modeling Elastic Behaviors with BRS

The behavior of elasticity controller is given as bigraphical reactive rules that express dynamicity of an elastic cloud system. In this Section, we define a set of reaction rules that model horizontal actions over the cloud hosting environment (servers, VMs and service instances). In addition, we introduce two elasticity strategies that elasticity controller uses to manage a cloud's elasticity.

Table 3 gives the defined reaction rules *Ri* expressing a set of possible actions that can be applied over a cloud system's back-end part.

Table 3 Reaction rules modeling elasticity actions in cloud bigraph

Reaction rule	Algebraic form
Deploy a new service instance	$R1 \stackrel{\text{def}}{=} (\mathrm{SE}.(\mathrm{VM}.d1)|d0) \big| \mathrm{id} \rightarrow (\mathrm{SE}.(\mathrm{VM}.(\mathrm{S}.d2)|d1)|d0) \big| \mathrm{id}$
Deploy a new VM instance	$R2 \stackrel{\text{def}}{=} (\mathrm{SE}.d0) \big| \mathrm{id} \rightarrow (\mathrm{SE}.(\mathrm{VM}.d1)|d0) \big| \mathrm{id}$
Turn on a new server	$R3 \stackrel{\text{def}}{=} \mathrm{id} \rightarrow (\mathrm{SE}.d0) \big| \mathrm{id}$
Consolidate a service instance	$R4 \stackrel{\text{def}}{=} (\mathrm{SE}.(\mathrm{VM}.(\mathrm{S}^{U}.d2)|d1)|d0) \big| \mathrm{id} \rightarrow (\mathrm{SE}.(\mathrm{VM}.d1)|d0) \big| \mathrm{id}$
Consolidate a VM instance	$R5 \stackrel{\text{def}}{=} (\mathrm{SE}.(\mathrm{VM}^{U}.d1)|d0) \big| \mathrm{id} \rightarrow (\mathrm{SE}.d0) \big| \mathrm{id}$
Turn off a server	$R6 \stackrel{\text{def}}{=} (\mathrm{SE}^{U}.d0) \big| \mathrm{id} \rightarrow \mathrm{id}$

A reaction is applied by replacing the *redex* bigraph (left-hand side) with the *reactum* bigraph (right-hand side of the reaction). As both *redex* and *reactum* bigraphs respect the formation rules Φ_{CS}, the reaction rules always produce configurations that are structurally correct *by definition*.

Reactions won't execute if either bigraphs are malformed. The specified rules define horizontal scale elasticity actions for provisioning (R1-3) and de-provisioning (R4-6) resources by scaling-out and scaling-in the hosting environment at service, VM and server scopes.

Sites (expressed with d) nested in different entities (servers, VMs and services) are used to abstract elements that are not included in the reactions. Expression "id" stands for the *identity bigraph* (i.e., bigraph with one site inside one region) [19]. Note that using abstract bigraphs together with the notions of sites and id allow providing a generic description of reaction rules. It enables matching and rewriting a sub-configuration of the general context. However, the introduced rules describe instantaneous rewrites (i.e., rules are instantaneously triggered when *redex* matches in the context) [20]. This is not sufficient to express a logic which describes our desired elastic behavior (i.e., triggering reaction rules only when needed). In this paper, we provide this logic through *elasticity strategies* that describe a reasoning for the elasticity controller.

Elasticity Strategies: A strategy describes a behavior to be adopted to manage elastic adaptations in the system. It consists of a set of actions that are triggered in case the specified triggering conditions are fulfilled. We introduce two *reactive* elasticity strategies of the form [4]: $IF\ condition(s)\ THEN\ actions(s)$.

A strategy that reacts to a condition ($CS \vDash \varphi$) is expressed: $strat{:}\ if\ CS \vDash \varphi\ then\ Ri$. $CS \vDash \varphi$ is true iff $\exists$ a bigraph $B\varphi$, encoding the predicate φ, that is a match in the context of CS. The triggered actions Ri are modelled as bigraphical reaction rules and the triggering conditions are encoded into *predicates logic*.

Strategy 1 - hosting environment provisioning: When input workload increases by receiving growing number of client requests, the hosting environment needs to scale-out in a way to ensure *availability* along with *performance*. Strategy 1 can be expressed with three complementary actions that operate at service and infrastructure level as shown in Table 4. Predicates $\varphi1-3$ express universal quantifying on services, VMs and servers to determine the system's state. The predicates respectively stand for *"all services/VMs/servers are overloaded"* and need to scale-out at the equivalent level.

Table 4 Strategy 1 definition

Level	Condition	Action
Service	All service instances are overloaded $\varphi1 \equiv \forall s \in V_{CS}\ ctrl_{CS}(s) = \mathrm{S}^{\mathrm{L}}$	$R1$
Infrastructure	All VMs are overloaded $\varphi2 \equiv \forall v \in V_{CS}\ ctrl_{CS}(v) = \mathrm{VM}^{\mathrm{L}}$	$R2$
	All Servers are overloaded $\varphi3 \equiv \forall e \in V_{CS}\ ctrl_{CS}(e) = \mathrm{SE}^{\mathrm{L}}$	$R3$

In the context of the voting cloud-based system example VS, bigraph $\mathrm{A} \stackrel{\mathrm{def}}{=} (\mathrm{SE}.(\mathrm{VM}.(\mathrm{S}^{\mathrm{L}}.d3)|d2)|d1)|d0)$ expresses the back-end part of the system if the voting service instance is overloaded during its runtime (i.e., condition $VS \vDash \varphi1$ is satisfied). Hence, reaction rule $R1$ is triggered to create a new instance of service S. The produced bigraph is given with $\mathrm{A}' \stackrel{\mathrm{def}}{=} (\mathrm{SE}.(\mathrm{VM}.(\mathrm{S}.d4)|(\mathrm{S}^{\mathrm{L}}.d3)|d2)|d1)|d0)$. Figure 4 represents this adaptation graphically. Note that site $d3$ abstracts all the handled requests to avoid overloading the graphical representation.

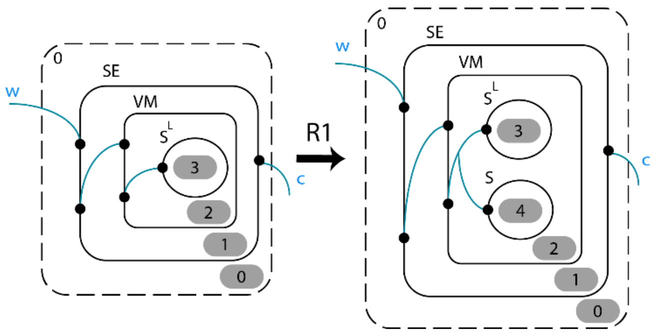


Fig. 4. *Triggering reaction rule R1*

Strategy 2 - hosting environment de-provisioning: When workload drops, the hosting environment is likely to be overprovisioned and has to scale-in. The elasticity controller enables this behavior at service and infrastructure levels by applying Strategy 2 as defined in Table 5. The predicates $\varphi4-6$ express existential quantifying over the entities (services instances, VMs, servers) to check their idleness. The predicates respectively express *"there exists a service/VM/server that is unused"*.

Table 5 Strategy 2 definition

Level	Condition	Action
Service	A Service instance is unused $\varphi4 \equiv \exists s \in V_{CS}\ ctrl_{CS}(s) = \mathrm{S}^{\mathrm{U}}$	$R4$
Infrastructure	A VM is unused $\varphi5 \equiv \exists v \in V_{CS}\ ctrl_{CS}(v) = \mathrm{VM}^{\mathrm{U}}$	$R5$
	A Server is unused $\varphi6 \equiv \exists e \in V_{CS}\ ctrl_{CS}(e) = \mathrm{SE}^{\mathrm{U}}$	$R6$

When workload drops in the context of the system VS, bigraph $\mathrm{B} \stackrel{\mathrm{def}}{=} (\mathrm{SE}.(\mathrm{VM}.(\mathrm{S}.d4)|(\mathrm{S}^{\mathrm{U}}.d3)|d2)|d1)|d0)$ is one expression of the back-end part when an instance of the voting service is unused (i.e., condition $VS \vDash \varphi4$ is satisfied). Reaction rule $R4$ is then applied to destroy the idle instance. After adapting, the produced bigraph is given with $\mathrm{B}' \stackrel{\mathrm{def}}{=} (\mathrm{SE}.(\mathrm{VM}.(\mathrm{S}.d3)|d2)|d1)|d0)$. Figure 5 represents this adaptation graphically. Notice that the system goes back to its initial configuration.

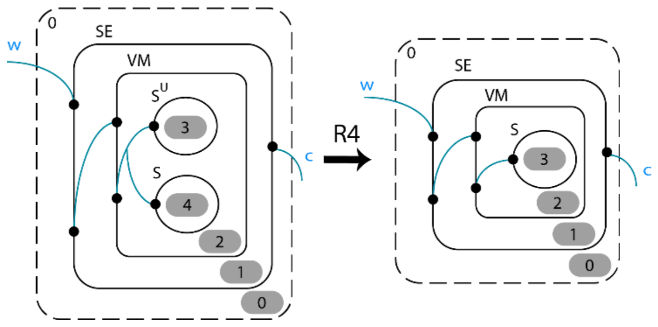


Fig. 5. *Triggering reaction rule R4*

In the introduced elasticity strategies, we express triggering conditions in predicates logic. These conditions reason on sets of cloud resources (i.e., servers, VMs and service instances) that are well-defined and which state (overloaded, unused) is known at every moment of the system's evolution. By quantifying elements in the sets to check their states, we enable capturing (and monitoring) the system's global state almost instantaneously (thus tackling monitoring concerns). Moreover, the introduced elastic auto-adaptations (scaling-out/in in this paper) are triggered when the specified conditions are fulfilled. As these conditions reason on the system's global state, the managed cloud system is considered *self-aware* [21] in terms of its elasticity.

4. Executability and Formal Verification of Elastic Behaviors

To verify and validate the correctness of the proposed elasticity strategies and to watch the desired elastic behaviors, we provide an executable solution for the proposed BRS specification. In theory, Bigraphical Reactive Systems provide good *meta-modeling* bases to specify cloud systems' structure and their elastic behaviors. As for their executable capabilities, providing a generic bigraphical solution requires designing generic and parametric reaction rules that can consider more aspects than those allowed by *Bilog* predicates [22–24]. To the best of our knowledge, there exists no tool built around BRS that enables to: (1) quantitatively reason about the system's global state (e.g. whether a threshold has been reached) and to (2) express predicate-based universal and existential conditions in order to trigger the reaction rules. Besides, to the best of our knowledge, classical bigraphs do not allow to define additional quantitative data over nodes (e.g. setting variable attributes to nodes such as thresholds) [25]. Furthermore, the few existing tools built around BRS such as BigraphER [19] and BPL Tool [26] are limited and only suitable for some specific application domains. BRS *model-checker* BigMC [27] that was for example used in [28], allows formal verification of safety properties. However, possible verifications rely on very limited predefined predicates. Globally, these tools lack of providing an autonomic executability of the specified BRS models. In this paper, we turn to Maude language to tackle these limitations in terms of encoding (strategies' triggering conditions) and executing in order to provide a generic executable solution of elasticity strategies together with verifying their correctness.

4.1. Motivating the relevance of Maude

Maude is a high-level formal specification language based on equational and rewriting logics. A Maude program defines a logical theory and a Maude computation implements a logical deduction which uses axioms specified in the program/theory. A Maude specification is structured in two parts [29]:

1. A functional module which specifies a *theory* in membership equational logic. Such a theory is a pair $(\Sigma, E \cup A)$, where *signature* Σ specifies the type structure (sorts, subsorts, operators etc.). E is the collection of possibly conditional equations declared in the functional module, and A is the collection of equational attributes declared for the operators (associative, commutative, etc.).
2. 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 possibly conditional rewrite rules.

The defined bigraphical specifications for cloud systems' structure can be encoded in a functional module, where the declared operations and equations define constructors that build the system's elements. Similarly, the specified BRS dynamics describing the elasticity controller's behavior can be encoded in a system module. Where elasticity strategies are described as conditional rewrite rules. The set of rewrite rules R express bigraphical reaction rules. The strategies' triggering conditions (predicates) can be expressed as equations from the functional module.

4.2. Principles of BRS encoding into Maude

To enable a generic executability of the introduced elastic behaviors, we encode the BRS-based specifications into Maude language as shown in Table 6.

Table 6 Mapping the bigraphical cloud model into Maude

Bigraphical model	Maude specification
	Functional module
Sorting discipline	**sorts** CS SE VM S SEL VML SL gstate state . **subsort** SE < SEL . **subsort** VM < VML . **subsort** S < SL . **op** CS<_/_:_> : Nat SEL gstate -> CS [**ctor**] . **op** SE<_,_,_/_:_> : Nat Nat Nat VML state -> SE [**ctor**] . **op** VM{_,_:_} : Nat SL state -> VM [**ctor**] . **op** S[_,_:_] : Nat Nat state -> S [**ctor**] . **ops** gstable underprovisioned overprovisioned : -> gstate [**ctor**] . **ops** stable overloaded unused ... : -> state [**ctor**] . **op** nilse : -> SEL [**ctor**] . **op** _*_ : SEL SEL -> SEL [**ctor assoc comm** id: nilse] . **op** nilv : -> VML [**ctor**] . **op** _|_ : VML VML -> VML [**ctor assoc comm** id: nilv] . **op** nils : -> SL [**ctor**] . **op** _+_ : SL SL -> SL [**ctor assoc comm** id: nils] . ...
System state predicates	**ops** isStable(_) isUnderprovisioned(_) isOverprovisioned(_) AoverSE(_) EunSE(_) AoverV(_) EunV(_) AoverS(_) EunS(_): CS -> Bool . ...
	System module
Elasticity strategies	Conditional rewrite rules of the form: **crl** [rewrite-rule-name] : term => term' **if** condition(s) .

Encoding Cloud Structures: In the functional module, bigraph sorts e, v and s (i.e., server, VM and service) are built according to their associated Maude constructors (`ctor`). We map bigraphical sorts as `SE`, `VM` and `S` and we introduce sort `CS` to define a cloud system. Notice that we enrich sorts with additional information in Maude to consider maximum hosting thresholds and entities states. For instance, a cloud system is defined by constructor `CS<`m`/SEL:gstate>`. And `SE<`x`,`y`,`z`/VML:state>` defines a cloud server. Parameters x, y and z are naturals that encode upper hosting thresholds at server, VM and service levels. m gives a maximum number of possible online servers. `SEL` is a list of servers and `VML` is a list of VMs (hosted by a server). These relationships are expressed by declaring sorts SE and VM as subsorts of SEL and VML respectively. `state` gives a symbolic elastic state for each element out of constructors `overloaded`, `unused` and `stable`. Term `gstate` gives a global state to the cloud system out of constructors `underprovisioned`, `overprovisioned` and `gstable` for "*globally stable"*.

Encoding System State Predicates: In the functional module, we define a set of predicates that describe a global elastic state of the system. For instance, *AoverSE()* is a predicate that stands for *"all servers are overloaded"*. Predicate *EunVM()* stands for *"there exists an unused VM instance"*. Typically, predicates of the form *Aover(SE/VM/S)* and *Eun(SE/VM/S)* encode our strategies' triggering predicates $\varphi 1 - \varphi 3$ and $\varphi 4 - \varphi 6$ respectively. We also encode system global state predicates *isStable(), isUnderprovisioned()* and *isOverprovisioned()* that are true when the system is *Stable, Underprovisioned* and *Underused*.

Encoding Elasticity Strategies: In the system module, we encode elasticity strategies as conditional rewrite rules. Their triggering conditions are state predicates encoded above and their triggered actions (mapped from the introduce bigraphical reaction rules) are encoded as Maude functional computation. For instance, Strategy 1 at service level is specified as follows: `crl[S1-service]:cs => addService(cs) if AoverS(cs) .` Where `cs` is a cloud system, `AoverS(cs)` is a predicate that is true if all service instances in the system are overloaded (φ1). Function `addService(cs)` is an equation that rewrites the term `cs` in such a way to deploy a new service instance (rule R1). This function is defined as an equation in the functional module.

4.3. Formal Verification of Elastic Behaviors

To verify their correctness, we model our defined elastic behaviors with *Linear Temporal Logic* (LTL). To proceed, we first define a model of temporal logic with a *Kripke* structure $\mathbf{A}_{CS}$ [30] as follows.

Definition 3: Given a set AP_{CS} of *atomic propositions*, we consider the Kripke structure $\mathbf{A}_{CS} = (A, \rightarrow_{\mathbf{A}}, L_{CS})$. Where A is the *set of states*, $\rightarrow_{\mathbf{A}}$ is the *transition relation*, and $L_{CS}: A \rightarrow AP_{CS}$ is the *labeling function* associating to each state $a \in A$, the set $L_{CS}(a)$ of the 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*: $\mathbf{A}_{CS}, a \vDash \phi$, where $\phi \in LTL(AP_{CS})$.

Setting up the Kripke Structure: We consider the set of atomic propositions $AP_{CS} = \{\varphi 1, \varphi 2, \varphi 3, \varphi 4, \varphi 5, \varphi 6\}$. These propositions are indicative of our elasticity strategies' triggering conditions (i.e., proposition φi holds when predicate φi is satisfied, with $i \in [1..6]$). Knowing that a cloud system evolves in a highly dynamic environment, multiple adaptations can be triggered during its runtime according to our specified elasticity strategies. Thus, the set of possible structural states of the system (i.e., configurations defined by a cloud system `cs`) is theoretically infinite. For this reason, we consider three symbolic states: *Stable*, *Underprovisioned* and *Overprovisioned*, respectively denoted in the set of states $A = \{\mathrm{S}, \mathrm{U}, \mathrm{O}\}$. The considered symbolic states express classes of equivalence with respect to the global elastic state of the cloud system (i.e., different structural configurations can have the same elastic symbolic state). Precisely, a cloud system has *Stable* state when no proposition in $\varphi 1 - \varphi 6$ holds (i.e., $L_{CS}(\mathrm{S}) = \emptyset$). The system is *Underprovisioned* when one or more propositions in $\varphi 1 - \varphi 3$ hold (i.e., $L_{CS}(\mathrm{U}) \subseteq \{\varphi 1, \varphi 2, \varphi 3\}$). It is *Overprovisioned* when one or more propositions in $\varphi 4 - \varphi 6$ hold (i.e. $L_{CS}(\mathrm{O}) \subseteq \{\varphi 4, \varphi 5, \varphi 6\}$). In other terms, the system is *Underprovisioned* or *Overprovisioned* when scaling-out or scaling-in actions are required, and it is *Stable* when no adaptation is needed.

Representing Transitions: We use *Labeled Transition Systems* [31] to represent transition relations between the considered states. For the sake of clarity, we give for each state the set of propositions that hold in it. Besides, we label the transition relations using adaption actions R1 − R6 and with two actions *in* and *out* that stand for input/output (i.e., receiving and satisfying an end-user's request). We split the graph into two parts to improve the readability of the transitions. The first part, shown in Figure 6, focuses on a view of system's evolution when it is *Underprovisioned* (i.e., mainly managed with Strategy 1). The second part, shown in Figure 7, gives a view of system's evolution when is it *Overprovisioned* (i.e., managed with Strategy 2). To facilitate the comprehension and the linking of the two parts, we represent edge states (i.e., that represent the connection between the two views) in red. Notice that most of these states are denoted U/O. They describe states where parts of the system are overloaded while others are unused. This leads to an "instable" state where the system is *Overprovisioned* and *Underprovisioned* at the same time. For example, it is possible to have an empty VM while all available service instances (deployed in other VMs) are overloaded (i.e., propositions φ1 and φ5 hold together). This perfectly depicts the impact of workload fluctuations on the system resources management's efficiency. Moreover, it shows that the two specified strategies are complementary. In addition, other "instable" states are possible to occur during the system's evolution as a direct result of workload traffic. However, besides the obvious readability concern, we don't show all possible states as they globally represent intermediate states. Some other states are impossible to occur. For example, it is impossible that the propositions pair $(\varphi 1, \varphi 4)$ hold at the same moment. The propositions respectively stand for "*all service instances are overloaded"* and *"a service instance is unused"* and are therefore contradictory. Idem for pairs $(\varphi 2, \varphi 5)$ and $(\varphi 3, \varphi 6)$ for the same reason.

Finally, the transition system shows the cross-layered behavior of the system. When workload grows, scaling-out at service level might result in overloading the system at VM level. Scaling-out at VM level is then necessary and could overload the system at server level. Inversely, when workload drops, scaling-in at service level could result in unused VMs then to unused servers when scaling-in at VM level. This behavior shows the importance of designing strategies than can be applied at service, VM and server levels of a cloud system in order to provide a complete management of horizontal elasticity.

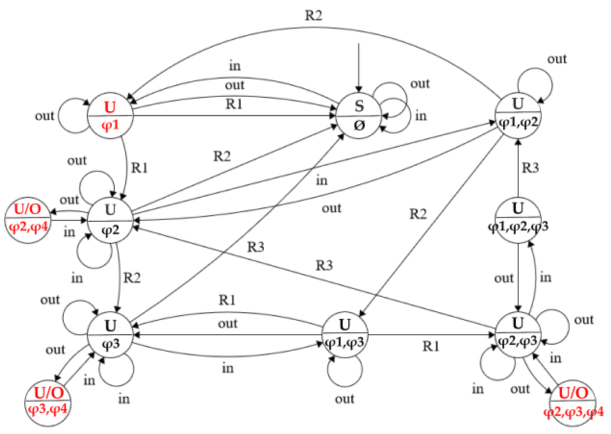


Fig. 6. A view of system transitions for Strategy 1

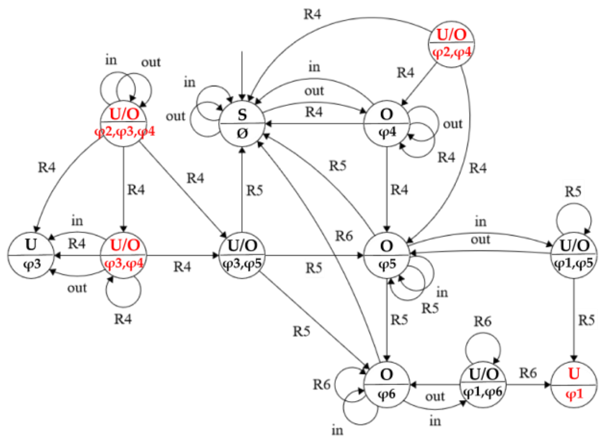


Fig. 7. A view of system transitions for Strategy 2

Defining LTL propositional Formulas: We introduce $LTL(AP) = \{Scale\text{-}Out, Scale\text{-}In, Stabilize\}$ the set of the propositional formulas in *Linear Temporal Logic* as follows.

– $Scale\text{-}Out \equiv \boldsymbol{G}\ (Underprovisioned \rightarrow \boldsymbol{F}\ Stable)$
– $Scale\text{-}In \equiv \boldsymbol{G}\ (Overprovisioned \rightarrow \boldsymbol{F}\ Stable)$
– $Stabilize \equiv \boldsymbol{G}$ (Underprovisioned $\wedge$ $Overprovisioned \rightarrow \boldsymbol{F}\ Stable)$

Where formulas *Scale-Out* / *Scale-In* specify that the managed cloud system that is *Underprovisioned* / *Overprovisioned* will eventually end up by reaching its *Stable* state. Formula *Stabilize* is applied when the system is in an "instable" state, as explained before, making it reach its *Stable* state eventually. Symbols ***G*** and ***F*** are LTL operators that stand for *"always"* and *"eventually"*. Note that every state can be initial since it is determined by monitoring. Nevertheless, the graphs show that every possible state is accessible, that it has a successor and that the *stable* state S is always accessible from any other state during system's evolution. We represented the system's transitions with the *Stable* state as initial state to show that regardless its evolution, there is always a path that leads back to its (initial) *Stable* state. This shows the *non-plasticity* [6] and *safety* properties of the system's auto-adaptation behaviors. The specified formulas are used to ensure *reachability* property of the managed elastic cloud system's *Stable* state, thus the *correctness* of our introduced elasticity strategies.

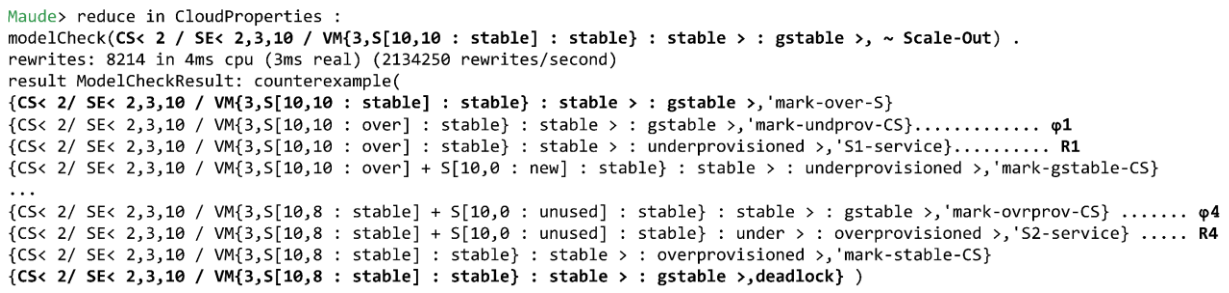
Encoding Kripke Structure into Maude: Maude allows associating *Kripke* structures to the specified rewrite theory (in system module) to define a module for *property specification*. Precisely, the introduced *Kripke* structure (Definition 3) enables conducting generic LTL state-based model checking that can reason on any system configuration. For instance, determining that a cloud configuration is *Stable* in terms of elasticity is specified with a conditional equation: `ceq cs ⊨ Stable = true if isStable(cs) == true.` Where `cs` is a given cloud system, `Stable` is a proposition in AP_{CS} representing the symbolic elastic state *Stable*. `isStable(cs)` is a predicate for *"the cloud system cs is stable"* defined in the functional module.

Formulas in $LTL(AP)$ are also encoded into the property specification module. For instance, *Scale-Out* formula is specified with an equation: `eq Scale-Out = [] (Underprovisioned -> <> Stable).` Where `Underprovisioned` is a proposition in AP_{CS}. `[]` and `<>` encode LTL operators ***G*** and ***F***.

Running Model-Checking: Maude LTL model-checker is executed with, as parameters: (1) a cloud configuration `cs` as initial state and (2) a property formula in $LTL(AP_{CS})$ to verify.

Figure 8 gives an execution trace of a model-checked cloud configuration example for a violation of the formula *Scale-Out* (given with its negation through symbol ~). The model-checker gives a counter example showing a succession of different executed rewrite rules that are applied on the given cloud system.

The trace shows that strategies 1 and 2 are triggered at service level (rules R1 and R4) when propositions φ1 and φ4 are satisfied during the system's evolution which is impacted by workload fluctuations. Notice that newly deployed service instance has *new* as state. As it is unused when deployed, *new* state enables not considering this fact during the elasticity evaluation of its state. This can be interpreted as a *cool-down period* and ensures *resources thrashing* i.e., avoiding opposite adaptations (deploying an instance than disposing it right after). These notions were defined in [6]. We also apply this reasoning at VM and server scopes. Besides, we define marking rules for understanding and clarity purposes. These rules update each entity state as well as the global state of the system if it changes during its execution. The reached *deadlock* state doesn't stand for some critical error, it genuinely means that no further adaptation is needed.



```
Maude> reduce in CloudProperties :
modelCheck(CS< 2 / SE< 2,3,10 / VM{3,S[10,10 : stable] : stable} : stable > : gstable >, ~ Scale-Out) .
rewrites: 8214 in 4ms cpu (3ms real) (2134250 rewrites/second)
result ModelCheckResult: counterexample(
{CS< 2/ SE< 2,3,10 / VM{3,S[10,10 : stable] : stable} : stable > : gstable >,'mark-over-S}
{CS< 2/ SE< 2,3,10 / VM{3,S[10,10 : over] : stable} : stable > : gstable >,'mark-undprov-CS}............ φ1
{CS< 2/ SE< 2,3,10 / VM{3,S[10,10 : over] : stable} : stable > : underprovisioned >,'S1-service}......... R1
{CS< 2/ SE< 2,3,10 / VM{3,S[10,10 : over] + S[10,0 : new] : stable} : stable > : underprovisioned >,'mark-gstable-CS}
...
{CS< 2/ SE< 2,3,10 / VM{3,S[10,8 : stable] + S[10,0 : unused] : stable} : stable > : gstable >,'mark-ovrprov-CS} ....... φ4
{CS< 2/ SE< 2,3,10 / VM{3,S[10,8 : stable] + S[10,0 : unused] : stable} : under > : overprovisioned >,'S2-service} ..... R4
{CS< 2/ SE< 2,3,10 / VM{3,S[10,8 : stable] : stable} : stable > : overprovisioned >,'mark-stable-CS}
{CS< 2/ SE< 2,3,10 / VM{3,S[10,8 : stable] : stable} : stable > : gstable >,deadlock} )
```

Fig. 8. *Maude LTL model-checker counter-example*

Finally, the trace shows that the adaptations in response of workload changes make the system reach its stable state. This allows verifying the effectiveness of our introduced elastic behaviors, thus confirming the assumptions we made at design time regarding their correctness and their ensured non-functional properties.

5. Quantitative Analysis of Cloud Elasticity

In this Section, we proceed to an analysis of the introduced strategies using *Queuing Theory* [14], a mathematical method of analyzing congestions and delays in waiting in line. Generally, a queuing process consists of customers arriving at a system to receive some service. If the servers (which offer the service) are busy, the customers wait in line in a queue until they are served, then leave the system. This kind of systems (that work according to a queuing process) can be described by a queuing model.

5.1. A Queuing Model for Cloud Elasticity

We advocate that a queuing model is a suitable support for analyzing an elastic system. In such systems that adapt to their incoming workload by provisioning computational resources, these available resources, at every moment, are not always sufficient to cope with the growing throughput. This makes congestions to appear in the system resulting in waiting queues that impact its quality of service (QoS). The main goal of our analysis is to watch the ability of a cloud system to adapt to its varying input workload. This implies provisioning and deprovisioning resources when workload rises and drops, using the two adaptation strategies that we introduced.

A queuing model introduces a set of parameters using the Kendall notation: A/S/C/Q/N/D [32]. These parameters are defined as follows. A system provides one or more servers, or more generally, a number C of resources (offering a service). The customers arriving at the system according to an arriving process A, join the queue to get a service from a server following some serving discipline D, e.g., they get served one by one, generally with FCFS (First Come First Served) law or in batches. The amount of time required to serve the customers is given by a service process S. The system capacity Q gives the maximum number of customers that the system can hold. It includes the customers that are waiting in the queue to be served and those being served. Finally, the size of the arriving population N gives the number of customers expected to arrive in the system.

Setting-up a Queuing Model: Applying a queuing reasoning over an elastic cloud system, we can consider that the queuing relationship customers/servers corresponds to the cloud concept of requests/service instances. In other words, in a queuing point of view, the servers of a cloud system (that offer a service) are in fact service instances that are deployed to serve requests (customers) arriving into the system. It is interesting here to notice that service instances' availability depends on cloud infrastructure deployment (provisioned virtual machines, etc.). This allows to easily understand how input workload impacts cloud hosting environment as it adapts by (de)provisioning resources, at service and infrastructure levels (i.e., in a cross-layered manner)

To conduct quantitative evaluation, we will assume that the system hosting capacity $Q = \infty$ and the size of arriving population $N = \infty$. Requests' arrivals A is given by a Poisson process which gives an exponential distribution of received requests at each time unit, with an average value λ. The service process S also follows an exponential law with an average value μ to give the number of requests that are processed by every service instance. The essence of our work being adaptation behaviors, we inspire from a queuing model with on-demand number of servers C, as presented in [33], to adjust the number of provisioned resources at different levels of the cloud hosting environment by applying the strategies we introduced. The serving discipline D follows a batching principle as proposed in [34].

5.2. Experimentation

To study the introduced elastic behaviors, we run simulations where inputs are: (1) a requests arrival rate λ, (2) a service rate μ, (3) an initial configuration of the cloud system and (4) constant values of hosting thresholds *x*, *y*, *z* and *m* (introduced in Section 4). To evaluate *performance* and *cost efficiency* of elasticity, we consider the following metrics:

- Average number of deployed service instances and VMs.
- Avg. usage rate of hosting environment.
- Avg. rate of successfully processed requests.
- Avg. requests processing delay (processing/waiting ratio).

The system's *performance* is indicated by the rate of treated requests and processing delay. The system's *load* is given by the average amount of handled requests at each unit of time (not to be confused with workload or received requests). *Processing delay* indicates the proportion in which requests are waiting to be processed. The number of deployed service and VM instances gives the accuracy of adaptations in response to workload variations. The *system's cost* *effectiveness* is given by analyzing the relationship between treated requests rate and the average hosting environment usage rate (in function of its maximum provisioning capacity).

Note that introducing thresholds over the hosting environment makes the system bounded in terms of resources that can be deployed (e.g., when $x = 2$ and $y = 2$, the maximum amount of service instances that can be deployed is $x \times y = 4$), this makes the model more realistic regarding its physical resources limitations. To estimate the correctness of resources' provisioning, we compare the average number of service instances in our simulations with results of the *Erlang-C* formula [35] that calculates the minimum and sufficient number of servers for a given arrival and service rates λ and μ. Erlang model states that the customers are impatient and can leave the system if their tolerance threshold in terms of waiting time is reached. Since our model does not consider requests timeout, we will only consider the minimum number of servers as a pertinent result of *Erlang-C* formula.

The Analyzed Cloud System: We use the introduced example of an online cloud-based voting system in Section 3. We keep the same initial configuration (i.e., one online server, one deployed VM and one running instance of the service). Hosting thresholds' values are defined as follows: maximum number of VMs running in the server is given by $x = 2$, maximum amount of service instances running in each VM is $y = 2$, and maximum number of requests a service instance can hold is $z = 30$. We run simulations on a single physical server (i.e., $m = 1$). For the same initial deployment and same values of thresholds, we analyze how the system behaves and adapts to different arrival and process rates patterns. We study the simulations execution traces within 50 units of time according to the following two scenarios.

Scenario 1 ($\lambda = 50$, $\mu = 25$): In this scenario, we assume that input workload is generated with an average value $\lambda = 50$. The service process is given with the average value of $\mu = 25$ (every service instance process around 25 requests per time unit). Within the simulation time, monitoring shows that 100% of the received requests are successfully processed with an average delay of 6,7%. As for the hosting environment's provisioning, the system achieves almost 100% of its capacity in terms of deployed VM instances ($x = 2$), and 65,5% of its maximum capacity of service instances with an average number of 2,6 deployed service instances out of 4 (i.e., $x \times y = 4$). This result is coherent with *Erlang-C* formula which states, for $\lambda = 50$ and $\mu = 25$, that at least 3 servers (service instances in a cloud point of view) are required to achieve full level of service (i.e., processing all received requests).

Scenario 2 ($\lambda = 35$, $\mu = 25$): In this scenario, we assume that input workload is around 35 arriving requests per time unit and that service processing is around 25 treated requests per unit of time. As the arrival rate λ is dropped from 50 to 35 comparing to Scenario 1, monitoring shows that less resources are provisioned overall. During this simulation, 100% of the requests are successfully processed with an average delay of 7,2%. The system only provisions 50% of its service instances capacity with an average number of 2 deployed service instances out of 4. However, around 93% of VM instances capacity is achieved with an average number of 1,86 deployed VMs out of 2. These results nonetheless correspond to *Erlang-C* formula which states that at least 2 servers are required for the same values of λ=35 and μ=25.

Figure 9 summarizes the obtained results for both scenarios 1 and 2. Overall, the system shows better performance in Scenario 1 than in Scenario 2 with 6,71% processing delay versus 7,18%. Besides, more important resource provisioning is recorded in Scenario 1 as input workload intensity is bigger. However, the obtained service instances' provisioning in both scenarios was coherent with the results given by *Erlang-C* formula.

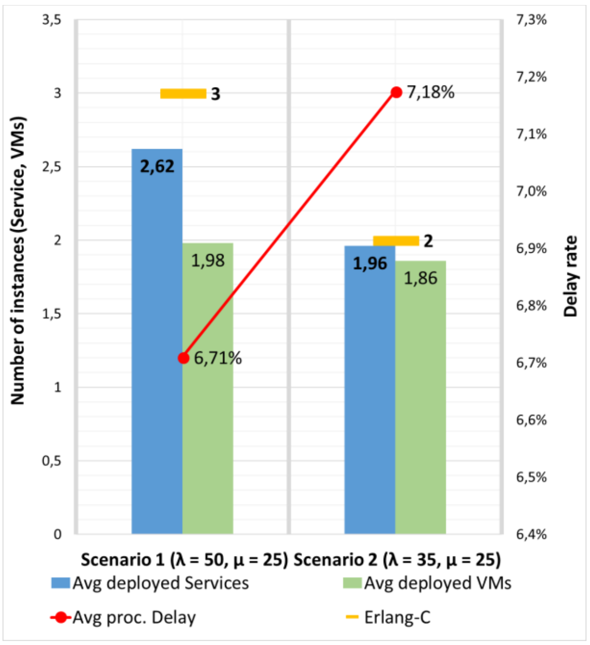


Fig. 9. *Experimentation quantitative results summary*

5.3. Discussion

The conducted experiment shows that the elastic behaviors, described by the strategies we introduced, are pertinent and in accordance with our expectations during design phase. The results given by simulation are globally convincing as the minimum required number of service instances always corresponds to results given by *Erlang-C* formula. Nevertheless, we think that scaling capabilities can be improved by introducing another strategy for load balancing (i.e., uniformizing hosting environment entities' load to maximize deployment efficiency). In fact, the simulations showed that the average service instances load was about 63% in Scenario 1 and around 60% in Scenario 2. Maximizing efficiency could lead to deploy less cloud resources (VMs) and consequently, to provide less expensive cloud deployments. It is important to keep in mind that this task is tedious regarding the fluctuating nature of input workload and the unpredictable congestions it could cause, even around a known average value λ.

In addition, one could consider a "good" strategy as one that brings good compromise between performance and cost (i.e., minimum infrastructure deployment for maximum processing rate and minimum delay). However, finding a right compromise between performances and cost efficiency might imply to scarify some performance. In Scenario 2, monitoring recorded 50% of the system's usage of service instances while it reached almost full VM capacity. This means that VM usage is not optimized as the deployed VMs were averagely half used. Threshold *x* (maximum deployed VMs) can be dropped to its half to avoid overprovisioning the system, thus avoiding unnecessary operating costs. However, this often implies to have bigger processing delays at high workload peaks. For instance, authors in [36] show that it is very challenging to give a definition to what should a "good" strategy be. Resources' consumption depended on the fluctuating nature of input workload that was the main parameter for describing a good elastic behavior.

Furthermore, we consider that the concept of strategy can be enlarged. In fact, our experiment showed that not only adaptations policies can affect the system's behavior. In terms of modeling, thresholds (x, y, z, m) are very important and can give implicit yet pertinent details about the system. Indeed, the number *x* of VMs that can be deployed, could be significative of a cloud service provider's financial ability to afford deploying VMs. The number *y* of maximum running service instances, can indicate the VM hardware profile [37] (allocated physical resources). The maximum processing requests threshold *z* can describe the service's nature (e.g., a light task could handle more requests at a time than a complex time consuming one). To illustrate, Scenario 1 showed 65,5% of service instances usage rate for full capacity ($x = 2$) of deployed VMs. This indicates that one of the two deployed VMs is always half used at the best. One could set *x* to 1 and *y* to 3 to maximize system's efficiency whilst reducing operating costs. For example, these values could mean that provisioning one Amazon EC2 Medium VM instance is more efficient than provisioning two Small VM instances. Moreover, the system's initial deployment could be considered as a strategy. Having a suitable initial configuration could lead to minimize processing delays and the need to adapt. It enables the system to efficiently absorb its input workload.

In conclusion, the experiment shows that simulating a cloud system's elastic behavior with multiple parameters enables a cloud administrator to plan for optimal effort (cloud resources) that should be allocated to a cloud-based service to provide an optimal compromise between cost and performance.

6. Related Work

There have been multiple research studies in the literature using formal methods to specify elastic behaviors in cloud systems. In [6], authors proposed a formalization based on CLTLtD) temporal logic of several concepts and properties related to elastic behaviors of cloud systems. Qualitative properties of elastic cloud systems have been formally introduced and detailed, such as elasticity and resources management. Authors validate their approach using an offline SAT and SMT-solvers based verification tool. The tool checks the elasticity mechanisms' (scale-in/out) correctness by reasoning on execution traces obtained by online simulation. Different input workload patterns are generated in the process to trigger elastic behaviors. In terms of modeling, precise cloud cross-layered composition has been abstracted to only address resources at infrastructure level. Precisely represented by a number of virtual machines.

Authors in [7] adopted a Petri nets formalization to describe cloud-based business processes' elastic behaviors. Elasticity strategies for *routing*, *duplicating* and *consolidating* cloud components at service level were defined. Strategies are compared in terms of reliability and performance (resources consumption). In their work, authors focus on the application layer of a cloud configuration and infrastructure details are not addressed in the model. Besides, the formal approach is verified using a verification-based evaluation. Authors use SNAKES, a Petri nets-based reachability graph which verifies the correctness of the introduced strategies that are simulated at design time.

In [17], authors introduced a formal approach based on *Bigraphical Reactive Systems* for modeling both structural and behavioral aspects of elastic cloud systems. Cloud elastic behaviors are represented in terms of client/application interaction. Elasticity methods at service, platform and infrastructure levels are modeled with *bigraphical reaction rules* to define a range of adaptation actions that describe horizontal, vertical and migration scales elasticity. However, no elasticity strategies are presented to describe a logic that governs the autonomic management of the adaptations. Besides, no verification of elasticity mechanisms is provided.

In [8], authors proposed an analytical model based on a queuing approach with variable number of servers. They represented service-based business processes adaptation to workload variations and evaluate elasticity strategies (scale-

Table 7 Comparison of elastic cloud systems formal modeling approaches

Approach	Formalism / formal model	Modeling elastic cloud systems			Verification and evaluation	
		System structures	Elastic behaviors	Elasticity strategies	Qualitative verification technique	Quantitative evaluation
[6]	CLTLt(D)	-	Infrastructure	Horizontal scale	SAT and SMT solvers	Simulation
[7]	Petri nets	-	Service		Reachability graph	
[8]	Markov Chains, Queuing Theory	-			-	Queuing model
[17]	BRS	Bigraphs	Service and Infrastructure	-	-	-
Our approach	BRS, Queuing Theory			Horizontal scale	LTL state-based model-checking	Simulation, Queuing model

out/in) that operate at service level. In this work, authors modeled input workload as a Poisson process and the queuing system as a *Markov Chain*. The Markov Chain describes the system's state with the size of the waiting queue. Metrics as number of servers and average response time are then calculated using probabilistic formulas. Authors provided a quantitative evaluation based on conceptual scenarios to validate their approach. However, no formal qualitative verification is provided.

In this paper, we extend [17] by defining two elasticity strategies for scaling-out/in cloud systems at service and infrastructure levels. These strategies are reactive to conditions (designed in predicates logic) that reason on the global state of the system. When triggered, the strategies apply adaptation actions that we modeled using bigraphical reaction rules. In our approach, BRS modeling enabled us to consider a complex global state of an elastic cloud system which is determined by the jointure of all states of hosting environment elements. These elements (services, VMs, servers) are expressed as sets of nodes that are linked by a hosting relationship representing dependencies between the three considered scopes. Reasoning over the elements' states at different levels, thus capturing the system global state ensures providing accurate adaptation capabilities to cope with the varying demand. To the best of our knowledge, there have not been published works that addressed the question of cloud systems' state in the way we propose it. The other approaches, based on Markov Chains, Petri nets or CLTLt(D), allow considering the system state at a very high level of abstraction. Using these formalisms, the cited works respectively considered the size of the queue, number of requests and number of VMs as main variable impacting adaptation decisions. Therefore, both [7, 8] propose horizontal-scale elasticity strategies that operate at service level and [6] address it at infrastructure level. In our model, we address horizontal scaling in a cross-layered manner (i.e., at both service and infrastructure levels). We show how managing elastic adaptations at service scope impacts the system's state at VM scope (idem for VM and server scopes).

To validate our contributions, we formally verify our approach using LTL state-based model-checking technique [11]. We encode the BRS specifications into Maude language to enable their autonomic executability and verify the correctness of the elastic behaviors. The cited papers do not provide an executable autonomic support for their modeling approaches. One step further, we designed a queuing-based simulation tool to provide a quantitative evaluation and analysis of elastic adaptations.

Table 7 summarizes this Section by comparing our approach with the referenced papers. As comparison criteria, we consider (1) the used formalism or formal model, (2) the provided modeling features in terms of cloud structures, elastic behavior and elasticity strategies and (3) the provided qualitative verification and quantitative evaluation of the modeling approach.

7. Conclusion

In this paper, we provided a view of cloud systems' hosting environment including all cloud components that are involved in elastic behaviors. Structural and behavioral aspects of elastic cloud systems have been modeled using the Bigraphical Reactive Systems formalism. Precisely, we use bigraphs and bigraphical reactive rules to express both aspects respectively. These behaviors implement an elasticity controller and are described by elasticity strategies. We propose two horizontal scale strategies for (de)provisioning cloud system resources at service and infrastructure levels. They describe a logic that enables the elasticity controller to reason over the entire cloud system state. Precisely, a strategy specifies conditions expressed as predicates. When satisfied, the conditions trigger adaptation actions that we expressed as bigraphical reaction rules.

One step further, we encoded our BRS specifications into Maude language in order to enable their autonomic executability. We also verify the correctness of the proposed elastic behaviors according to a state-based model-checking, relying on Linear Temporal Logic (LTL). Besides, we adopted a queuing approach as a support for an analysis of adaptation capabilities of elastic cloud systems. Clearly, we designed a tool that enables simulating and monitoring a cloud system's execution. Moreover, we provided an experimental analysis over two different execution scenarios. Finally, we discussed and analyzed elasticity strategies aiming at giving a deeper comprehension of cloud systems' elastic adaptation.

In this present work, we attempt to take a first step towards the formalization of cloud systems elastic behaviors. In the next step, we plan to enlarge our specifications to provide more adaptation capabilities by introducing strategies for load balancing and vertical scaling. Finally, our objective is to provide a complete executable and verifiable formalization of cloud systems' elastic behaviors.

8. References

1 Mell, P., Grance, T.: 'The NIST Definition of Cloud Computing'2011, p. 7.

2 Herbst, N.R., Kounev, S., Reussner, R.: 'Elasticity in Cloud Computing: What It Is, and What It Is Not', in 'Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)' (USENIX, 2013), pp. 23–27

3 Kephart, J.O., Chess, D.M.: 'The vision of autonomic computing'*Computer*, 2003, **36**, (1), pp. 41–50.

4 Galante, G., Bona, L.C.E. de: 'A Survey on Cloud Computing Elasticity', in 'Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing' (IEEE Computer Society, 2012), pp. 263–270

5 Dustdar, S., Guo, Y., Satzger, B., Truong, H.-L.: 'Principles of Elastic Processes'*IEEE Internet Computing*, 2011, **15**, (5), pp. 66–71.

6 Bersani, M.M., Bianculli, D., Dustdar, S., Gambi, A., Ghezzi, C., Krstić, S.: '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' (ACM, 2014), pp. 38–47

7 Amziani, M.: 'Modeling, evaluation and provisioning of elastic service-based business processes in the cloud'. phdthesis, Institut National des Télécommunications, 2015

8 Yataghene, L., Ioualalen, M., Amziani, M., Tata, S.: 'Using Formal Model for Evaluation of Business Processes Elasticity

in the Cloud', in 'Service-Oriented Computing – ICSOC 2016 Workshops' International Conference on Service-Oriented Computing, (Springer, Cham, 2016), pp. 33–44

9 Milner, R.: 'Bigraphs and Their Algebra'*Electronic Notes in Theoretical Computer Science*, 2008, **209**, pp. 5–19.

10 Milner, R.: 'The Space and Motion of Communicating Agents' (Cambridge University Press, 2009, 1st edn.)

11 Souri, A., Navimipour, N.J., Rahmani, A.M.: 'Formal verification approaches and standards in the cloud computing: A comprehensive and systematic review'*Computer Standards & Interfaces*, 2018, **58**, pp. 1–22.

12 Aceto, G., Botta, A., de Donato, W., Pescapè, A.: 'Cloud monitoring: A survey'*Computer Networks*, 2013, **57**, (9), pp. 2093–2115.

13 Roy, N., Dubey, A., Gokhale, A.: 'Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting', in 2011 IEEE 4th International Conference on Cloud Computing, (2011), pp. 500–507

14 Stern, G.J.A., Kleinrock, L.: 'Queueing Systems, Volume 2: Computer Applications.'*Applied Statistics*, 1978, **27**, (2), p. 186.

15 Meyn, S.P., Tweedie, R.L.: 'Markov Chains and Stochastic Stability' (Springer-Verlag, 1993)

16 Letondeur, L.: 'Planification pour la gestion autonomique de l'élasticité d'applications dans le cloud' (Grenoble, 2014)

17 Sahli, H., Hameurlain, N., Belala, F.: 'A bigraphical model for specifying cloud-based elastic systems and their behaviour'*International Journal of Parallel, Emergent and Distributed Systems*, 2017, **32**, (6), pp. 593–616.

18 Birkedal, L., Damgaard, T.C., Glenstrup, A.J., Milner, R.: 'Matching of Bigraphs'*Electronic Notes in Theoretical Computer Science*, 2007, **175**, (4), pp. 3–19.

19 Sevegnani, M., Calder, M.: 'BigraphER: Rewriting and Analysis Engine for Bigraphs', in Chaudhuri, S., Farzan, A. (Eds.): 'Computer Aided Verification' (Springer International Publishing, 2016), pp. 494–501

20 Calder, M., Sevegnani, M.: 'Modelling IEEE 802.11 CSMA/CA RTS/CTS with stochastic bigraphs with sharing'*Form Asp Comp*, 2014, **26**, (3), pp. 537–561.

21 Chen, T., Bahsoon, R., Yao, X.: 'A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems'*ACM Comput. Surv.*, 2018, **51**, (3), pp. 61:1–61:40.

22 Mansutti, A., Miculan, M., Peressotti, M.: 'Multi-agent Systems Design and Prototyping with Bigraphical Reactive Systems', in 'Distributed Applications and Interoperable Systems' IFIP International Conference on Distributed Applications and Interoperable Systems, (Springer, Berlin, Heidelberg, 2014), pp. 201–208

23 Conforti, G., Macedonio, D., Sassone, V.: 'Spatial Logics for Bigraphs', in 'Automata, Languages and Programming' International Colloquium on Automata, Languages, and Programming, (Springer, Berlin, Heidelberg, 2005), pp. 766–778

24 Faithfull, A.J., Perrone, G., Hildebrandt, T.T.: 'Big Red: A Development Environment for Bigraphs'*Electronic Communications of the EASST*, 2013, **61**, (0).

25 Krivine, J., Milner, R., Troina, A.: 'Stochastic Bigraphs'*Electronic Notes in Theoretical Computer Science*, 2008, **218**, pp. 73–96.

26 Glenstrup, A.J., Damgaard, T.C., Birkedal, L., Højsgaard, E.: 'An Implementation of Bigraph Matching'no date, p. 22.

27 Perrone, G., Debois, S., Hildebrandt, T.T.: 'A Model Checker for Bigraphs', in 'Proceedings of the 27th Annual ACM Symposium on Applied Computing' (ACM, 2012), pp. 1320–1325

28 Sahli, H., Belala, F., Bouanaka, C.: 'Model-Checking Cloud Systems Using BigMC', in Proceedings of the 8th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS 2014), (2014), pp. 25–33

29 Clavel, M., Duran, F., Eker, S., *et al.*: 'Maude Manual (Version 2.7.1)'2017, p. 521.

30 Baier, C., Katoen, J.-P.: 'Principles of model checking' (The MIT Press, 2008)

31 Schoren, R.: 'Correspondence between Kripke Structures and Labeled Transition Systems for Model Minimization' (2011)

32 Baynat, B.: 'Théorie des files d'attente: des chaînes de Markov aux réseaux à forme produit' (Hermes Science Publications, 2000)

33 Mazalov, V., Gurtov, A.: 'Queueing System with On-Demand Number of Servers'*Mathematica Applicanda*, 2012, **40**, (2), pp. 1–12.

34 Dragović, B., Park, N.-K., Zrnić, N.Đ., Meštrović, R.: 'Mathematical Models of Multiserver Queuing System for Dynamic Performance Evaluation in Port'*Mathematical Problems in Engineering*, 2012.

35 Firdhous, M., Ghazali, O., Hassan, S.: 'Modeling of cloud system using Erlang formulas', in The 17th Asia Pacific Conference on Communications, (2011), pp. 411–416

36 Netto, M.A.S., Cardonha, C., Cunha, R.L.F., Assuncao, M.D.: 'Evaluating Auto-scaling Strategies for Cloud Computing Environments', in 'Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems' (IEEE Computer Society, 2014), pp. 187–196

37 'Amazon EC2 Instance Types – Amazon Web Services (AWS)', https://aws.amazon.com/ec2/instance-types/, accessed September 2018