SLA-Driven Modeling and Verifying Cloud Systems: A Bigraphical Reactive Systems-based Approach

Oussama Kamel[a,b], Allaoua Chaoui[b], Gregorio Diaz[c], Mohamed Gharzouli[b]

[a] *Faculty of Medicine, University Constantine 3 Salah Boubnider, Constantine, Algeria*
[b]*MISC Laboratory, Department of Computer Science and its Applications, University Constantine 2 Abdelhamid Mehri, Constantine, Algeria*
[c]*School of Computer Science, University of Castilla-La Mancha, Campus Universitario s/n, 02071, Albacete, Spain*

Abstract

We propose a formal approach based on Bigraphical Reactive Systems (BRSs) and model checking techniques for modeling and verifying the interaction behaviours of SLA-based cloud computing systems. In the first phase of this approach, we address the modeling of the static structure and the dynamic behavior of cloud systems using BRSs. We show how bigraphs enable the description of the different cloud entities, including cloud actors, cloud services, service level agreements (SLAs), the diversity of their properties, and the complex interactions and dependencies among them. Furthermore, we propose a four-stages SLA lifecycle, and define a set of bigraphical reaction rules to abstract these stages and model the dynamic nature of the cloud. The second phase of this approach verifies that the behavior of services and cloud actors will cope with the agreed SLA terms during the lifecycle of the SLA. We map the proposed bigraphical models into SMV descriptions. Then, we express the interaction behaviors as a set of liveness and safety properties using Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) formulas, and we use the NuSMV model checker to verify them. Finally, we define a case study on which we illustrate the application of our proposed approach.

Keywords: Cloud computing, SLA, Bigraph, Bigraphical Reactive Systems, NuSMV, Formal verification

1. Introduction

During the last decades, several computing models and paradigms have been proposed to make the utility computing vision a reality. Cloud computing, one of the well-known emerging models, promises virtually infinite computing resources whenever and wherever they are required [1]. On-demand self-service, broad network access, resource pooling, measured service, and rapid elasticity, among other key characteristics of the cloud, have encouraged academia, industry, and government to move towards cloud computing solutions [2]. The cloud introduces a new model in which everything is offered and used as a service (the acronyms *aaS, XaaS or EaaS stand for Everything as a Service) [3]. According to the widely accepted definition, given by the *National Institute of Standards and Technology* (NIST) [4], cloud services are delivered under three main models: infrastructures (Infrastructure as a Service, i.e., IaaS), platforms (Platform as a Service, i.e., PaaS) and applications (Software as a Service, i.e., SaaS), and are deployed following four models (public cloud, private cloud, community cloud, and hybrid cloud).

Although the aforementioned gained benefits, this paradigm faces different issues such as the vendor *lock-in* problem [5]. This problem is due to the different models and descriptions used by providers to represent their cloud services. As a first objective of this paper, we will show *how to use formal methods for the modeling and description of cloud services and other cloud entities*, since formal methods offer standardized and rigorous descriptions. Another problem concerns the Service Level Agreement (SLA) and its management lifecycle. The SLA [6, 7] permits to regulate and control the provision and consumption of services and specifies the set of quality of service (QoS) that must be guaranteed. The realization of the SLA follows different stages [8, 9]. Through the whole lifecycle of the SLA, the different providers, end-users, and also the offered services may have different changing states and different kinds of interactions. Tackling the first problem using formal methods will pave the way towards formal verification, and therefore *ensuring the correctness of interaction behaviors among the different cloud entities during the SLA lifecycle*, which represents the second objective of this paper.

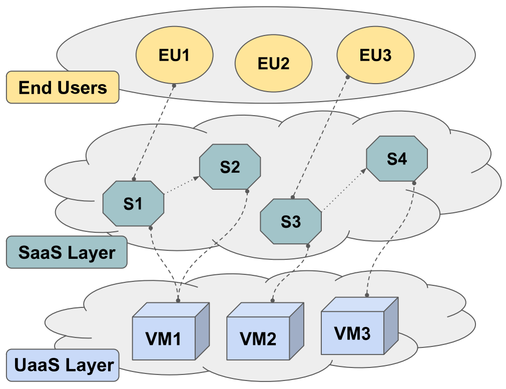


Figure 1: A layered cloud system

In the following, we present a motivating example through which we identify the different challenges that are faced when modeling and verifying SLA-based cloud systems, and also the specific features of the cloud that have been considered by such approach.

Let us consider the following motivating example, depicted in Figure 1, in which a cloud architecture comprises two layers. The lower one represents the UaaS (Utility as a Service) layer in which we find infrastructure services offered as virtual machines (VMs). The second layer represents SaaS providers that use services from the previous layer. On top of the cloud layers, Figure 1 shows end-users that consume services offered by the different providers. Note here that we have considered only the two layers following the classification proposed by [10, 11] in which it is considered that the cloud incorporates application services (SaaS) and utility services (UaaS). Utility services include both IaaS and PaaS services. Further explanation and examples, about cloud services classification, will be given in Section 4. Consider the example shown in Figure 1 where a SaaS provider wants to deploy two services. The first service depends on the second one. We start by presenting two scenarios for the deployment. In the first one, the provider wants to deploy the two services S1 and S2 in the same VM (VM1) offered by a public cloud provider. In the second scenario services S3 and S4, we suppose that service S4 will process confidential data, and therefore the provider wants to deploy it in his private cloud VM3 (in order to ensure data privacy and access control) and keep S3 deployed on the public cloud VM2.

Generally, providers of services give different details and properties of the offered services, for instance, the UaaS provider describes the offered VMs in terms of their CPU, RAM, Storage, Location, etc. On the other side, a consumer selects services depending on its requirements, for instance, the SaaS provider (playing the role of a consumer) may look for VMs having certain capacities, located in a specific region, and following particular security protocols. The consumer selects services that can meet its requirements and starts negotiating with their providers. The negotiation may be terminated by the definition and the establishment of an SLA. This includes a set of terms to be respected during the provision and consumption of services. Examples of SLA terms are: the availability of the offered service must be always greater than 99,99%, and the maximum number of violations is 3. As we can observe in the first example, these terms usually refers to QoS requirements. Deployed in a highly dynamic environment, such as the cloud, QoS of the delivered services are subjected to different changes. The monitoring process verifies that the execution of services is performed according to the signed SLA and reports any SLA violations. This process continues until the termination of SLA that can take place by one of two reasons: the expiration time has been reached or the occurrence of violations.

The focus of this paper is the modeling of cloud computing systems and the verification of interaction behaviors among their different cloud entities during the lifecycle of SLAs. Analyzing the motivating example, we have identified the following challenges:

1. *Diversity of cloud services models*: In the motivating example, we have considered two types of delivery models that are SaaS and UaaS. In the literature, however, we may find more and different classifications of the delivery models [4, 10, 11]. Therefore, the proposed approach should offer the possibility to consider any classification and define any type of cloud services.

2. *Service properties and QoS consideration*: Services can be described using different properties and QoS [12]. The example cites some resources features such as CPU, RAM, location, availability, etc. Note here that these parameters depend on the type of entity they describe. The proposed approach should offer means to define different properties and QoS.
3. *Cloud actors and SLA stakeholders*: In our example, three types of actors are identified: end-users, SaaS providers, and UaaS providers. These actors are involved in the establishment of the SLA as customers or providers. The proposed solution should permit the definition of different actors and SLA parties.
4. *Specific service compositions and diverse relationships*: Generally, a composition of services is required to offer a complete solution and satisfy the users' (functional and non-functional) requirements [13]. In cloud environments, service composition can be intra-layer composition (horizontal), which denotes composition of services that are on the same layer (between S1 and S2), or inter-layer composition (vertical), which involves services from different layers (between S1 from the SaaS layer and VM1 from the UaaS layer) [14]. This composition is always considered as a complex problem due, on one hand, to the wide variety and the growing number of services available in the cloud, and on the other hand, to the requirements of users that are more and more complicated [15]. Indeed, the establishment of such composition requires complex interactions between providers of these services and their consumers [16]. The proposed approach should allow the definition of such types of service compositions and relationships between cloud actors.
5. *SLA consideration*: In the motivating example, the SLA includes, among others, terms specifying the expected level of services (e.g., availability). The proposed solution should allow the specification of SLAs and their different terms.
6. *SLA lifecycle and dynamic change of cloud entities*: Cloud actors and services can go through different states during the lifecylce of SLA. The proposed solution should allow the modeling of the different SLA lifecycle stages. In addition, it should allow the description of the dynamic change of these entities states. For instance, an end-user can be satisfied or dissatisfied with the offered service.
7. *Correctness verification*: The proposed approach should be based on a formal foundation [17], and hence enables reasoning about the different entities such as services, end-users, providers, and SLAs. Specifically, it should permit to verify that the behavior of services and cloud actors will cope with the defined SLA terms during the lifecycle of the SLA. For instance, we may want to verify that: 1) *Always the monitoring process launches directly after the establishment stage*, 2) *Once the number of occurred violations reaches the maximum allowed number of violations, the SLA will be directly terminated with penalties*, or 3) *The number of occurred violations must not exceed the maximum allowed number of violations*. In this research we have considered two types of properties, introduced in [18], that are: liveness and safety. While the former describes that *something good will eventually happen* (e.g., properties 1 and 2), the latter denotes that *something bad will never happen* (e.g., property 3) during the lifecyle of the SLA. In other words, they express desirable or undesirable behaviors of the different cloud entities during the SLA lifecycle.

Based on Milner's Bigraphical Reactive Systems (BRSs) [19, 20] and model checking techniques [21, 22, 23], we aim to propose a novel formal approach for modeling and verifying SLA-based cloud systems to cover all the aforementioned challenges simultaneously. Taking into account these challenges, the contributions of this article are as follows:

- We cover the first five challenges by proposing a generic and complete formal description, based on the bigraph theory, for the different cloud entities, relationships thereof, and their properties. The cloud entities include cloud actors, cloud services, and cloud SLAs.

- We cover challenge 6 by proposing a four-stages cloud-SLA lifecycle and defining a set of bigraphical reaction rules to describe it. Moreover, the proposed reaction rules consider the dynamic change of these entities during the SLA lifecycle.

- We cover the last challenge by proposing a formal verification approach, based on the NuSMV model checker, to ensure the correctness of interaction behaviors among the different cloud entities, during the SLA lifecycle. This verification approach is based on a set of liveness and safety properties specified in Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). Moreover, the defined properties are ranging from general ones to specific properties which are depending on the signed SLA.

The rest of this paper is organized as follows. Section 2 presents an overview of the BRS theory. An overview of the proposed approach is detailed in Section 3. Our BRS-based modeling approach from static and dynamic perspectives are introduced in Sections 4 and 5, respectively. In Section 6, the formal verification approach is presented. In Section 7, we define a case study, with different scenarios, on which we apply the proposed approach. The related work and conclusion are presented in Sections 8 and 9, respectively.

2. Background

The theory of bigraphs has been introduced by Milner and colleagues for modeling ubiquitous systems [19, 20]. Through the combination of bigraphs and a bigraphical reaction rules-set, this theory can deal with both the static structure and the dynamic behavior of systems. This combination yields bigraphical reactive systems. Although in its infancy, BRSs have shown their successful adoption in many different domains such as cyber-physical systems [24, 25], context-aware systems [26, 27], wireless networks [28, 29], and cloud computing [30, 31].

2.1. Modeling static structure: bigraphs

Milner and co-workers have proposed a mathematical formalism called bigraphs [19, 20]. This model intends to unify earlier models (universal process algebra), such as the π-calculus [32] , CSS [20] and Petri nets [33] . The theory of bigraphs offers a graphical and a formal representation that can describe both locality and connectivity. This feature is achieved by specifying two extra types of graphs: the place graph and the link graph that compose bigraphs. These two graphs share the same node-set of their bigraph. Figures 2a and 2b show two bigraphs G and H modeling the first and the second scenarios, respectively, presented above. G^P and G^L are the corresponding place graph (Figure 2c) and link graph (Figure 2d) of G. Nodes in bigraphs can model physical (real) or logical (virtual) entities. Controls are used to define the different types of bigraph's nodes. For instance (see Figure 2a), E1 is an EUser-node (i.e., E1 has been assigned the control EUser) and S1 is a node of control SaaS. Each control has zero, one or many ports. The defined controls set K and the function $ar : K \to \mathbb{N}$, specifying the arity of each control (the number of its ports), are defined by the bigraph basic signature (K, ar). For instance, the signature of the bigraph G is defined as follows: $K = \{EUser : 1, SaaS : 4, UaaS : 4, SLA : 0, Req : 0, DepM(x) : 0\}$. Notice that the definition of the control $DepM(x)$ uses the parameter x that will be replaced by real values (for example, private or public) during its instantiation. This kind of controls is called parameterised controls [34] [29].

2.1.1. Place graph

This graph depicts the locality of the bigraph nodes as a containment relationship. For example, the node denoted by $E1$ contains the document SLA with the identifier *SL1* (Figure 2a). As shown in Figure 2c, the place graph is structured as a forest of trees, where, their roots represent the bigraph's regions (dashed boxes), and some of their leaves represent the bigraph's sites (shaded boxes). In figure 2a, we have three regions indexed from 0 to 2 and eight sites indexed from 0 to 7. Sites play an important role in the bigraph theory since they can be used to model details that are not the current interest. For instance, site 6 (Figure 2a) shows that in addition to the existing nodes contained in *VM1* (the SLA document and deployment model), this latter may contain other entities (unspecified nodes), such as response time, geographical location, and RAM capacity nodes.

2.1.2. Link graph

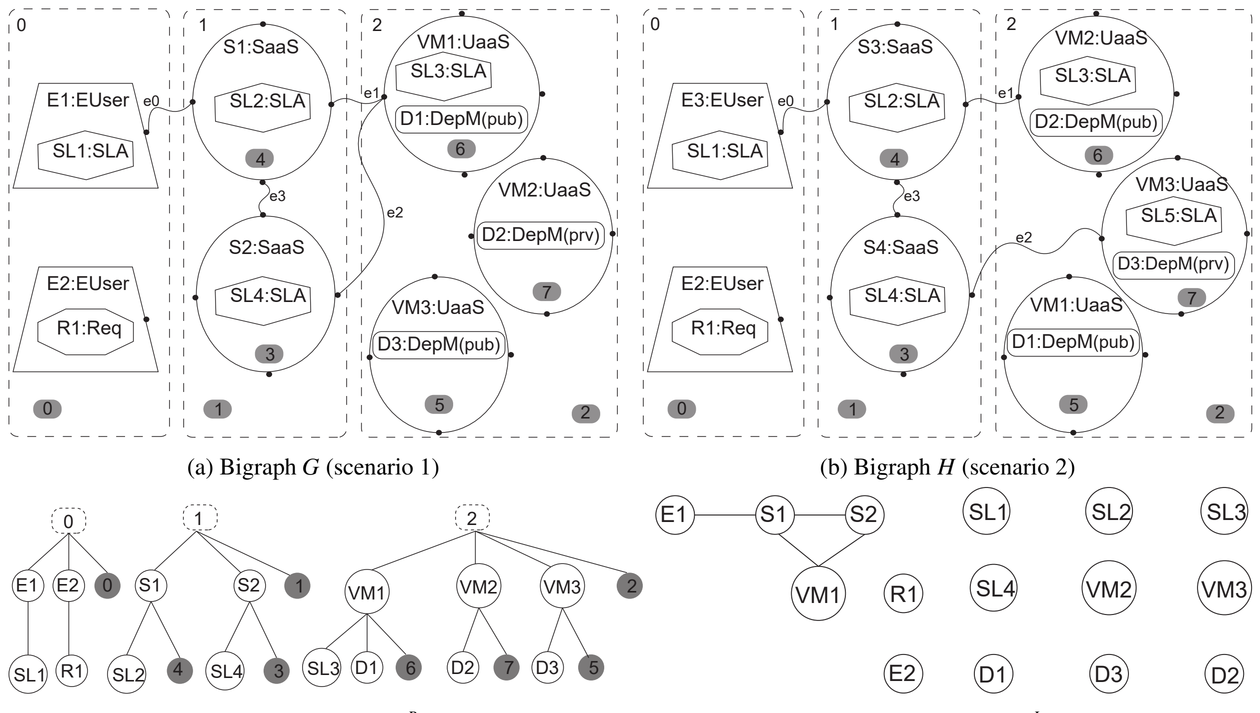
This graph expresses the interconnection between the bigraph nodes (see Figure 2d). Edges within this hypergraph may connect many ports (black dots) belonging to different nodes. As shown in Figure 2a, the edge $e2$ expresses that $S2$ is deployed on *VM1* and the edge $e0$ expresses that the end-user *E1* has started using *S1*.

2.2. Formal representations

According to Milner [19], $\uplus$ is used as the union operator between sets known or assumed to be disjoint, and a finite ordinal m is treated as the set $\{0, \ldots, m-1\}$. Milner [19] has defined bigraphs as follows:

Definition 1. *A bigraph G is a quintuplet* $(V, E, ctrl, prnt, link) : < m, X > \to < n, Y >$*, such that*

- *V and E denote finite sets of nodes and edges, respectively.*



(a) Bigraph G (scenario 1)

(b) Bigraph H (scenario 2)

(c) Place Graph G^P

(d) Link Graph G^L

Figure 2: Graphical representation of bigraphs, place graphs, and link graphs.

- *Nodes are assigned each a control using the control map $ctrl : V \to K$.*
- *While $prnt : m \uplus V \to V \uplus n$, the parent map, defines the containment relationship, the link map, $link : X \uplus P \to E \uplus Y$, defines how nodes are linked.*
- *The set $P = \{(v, i) \mid v \in V \text{ and } i \in ar(v)\}$ specifies ports.*
- *Both m and n are ordinals indexing sites and regions, respectively.*
- *The pair $< m, X >$ ($< n, Y >$, respectively) forms the inner (outer, respectively) interface of the bigraph, where X and Y are called inner and outer names.*

The bigraph G depicted in Figure 2a is defined as $(V, E, ctrl, prnt, link) : < 8, \phi > \to < 3, \phi >$, where:
$V = \{E1, E2, SL1, R1, S1, SL2, S2, SL4, VM1, SL3, D1, VM2, D2, VM3, D3\}$
$E = \{e0, e1, e2, e3\}$

$$ctrl(v) = \begin{cases} EUser : 1, & if \quad v \in \{E1, E2\} \\ SaaS : 4, & if \quad v \in \{S1, S2\} \\ UaaS : 4, & if \quad v \in \{VM1, VM2, VM3\} \\ SLA : 0, & if \quad v \in \{SL1, SL2, SL3, SL4\} \\ Req : 0, & if \quad v \in \{R1\} \\ DepM(x) : 0, & if \quad v \in \{D1, D2, D3\} \end{cases}$$

$$prnt(v) = \begin{cases} Region0, & if \quad v \in \{E1, E2, Site0\} \\ Region1, & if \quad v \in \{S1, S2, Site1\} \\ Region2, & if \quad v \in \{VM1, VM2, VM3, Site2\} \\ E1, & if \quad v \in \{SL1\} \\ E2, & if \quad v \in \{R1\} \\ S1, & if \quad v \in \{SL2, Site4\} \\ S2, & if \quad v \in \{SL4, Site3\} \\ VM1, & if \quad v \in \{SL3, D1, Site6\} \\ VM2, & if \quad v \in \{D2, Site7\} \\ VM1, & if \quad v \in \{D3, Site5\} \end{cases}$$

$$link(l) = \begin{cases} e0, & if \quad l \in \{(E1,0), (S1,0)\} \\ e1, & if \quad l \in \{(S1,2), (VM1,0)\} \\ e2, & if \quad l \in \{(S2,2), (VM1,0)\} \\ e3, & if \quad l \in \{(S1,1), (S2,3)\} \end{cases}$$

Note here that the set of ports of the node S1 is denoted as $P_{S1} = \{(S1,0), (S1,1), (S1,2), (S1,3)\}$ where, for instance, the notation $(S1,0)$ denotes the port 0 in the node $S1$.

2.3. Algebra of bigraphs

Milner [35] has defined an algebraic notation that can describe bigraphs and support their graphical representation as summarized in table 1. For instance, the bigraph G depicted in Figure 2a is expressed as :

$G = (E1_{e0}.SL1) \mid (E2.R1) \mid d_0 \parallel (S1_{e0,e1,e3}.(SL2 \mid d_4)) \mid (S2_{e2,e3}.(SL4 \mid d_3)) \mid d_1 \parallel (VM1_{e1,e2}.(SL3 \mid D1 \mid d_6)) \mid (VM3.(D3 \mid d_5)) \mid (VM2.(D2 \mid d_7)) \mid d_2$

The notation d_i denotes a site with the index i. For instance, d_0, d_1,...d_7 are the sites used in the bigraph G. The nesting operator $(U.V)$ is used to express the containment relationship. For example, $E1_{e0}.SL1$ expresses that the node $SL1$ is nested inside the node $E1$. The prime product operator $U \mid V$ denotes that U and V are contained into the same parent. For instance, $VM1_{e1,e2}.(SL3 \mid D1 \mid d_6)$ expresses that the nodes $SL3$, $D1$, and the site d_6 are juxtaposed under the node $VM1$. The parallel product operator defines new regions, for instance, in the bigraph G there are three regions that are juxtaposed using the operator || twice.

Table 1: Bigraphical algebraic representation

Bigraphical term	Meaning
d_i	Site indexed i
U.V	Nesting (U contains V)
U|V	Prime product
U||V	Parallel product

2.4. Bigraphical sorting discipline

The sorting mechanism for bigraphs was developed by Milner [19]. This mechanism categorizes controls and links in different sorts. In addition, it may impose more constraints on both the containment and the interconnection relationships (called place sorting and link sorting, respectively), thereby ensuring that only well-formed bigraphs will be obtained.

Definition 2. *A place sorting is a triple $\Sigma_P = (\Theta^P, K, \Phi)$, comprising a nonempty set Θ^P of place sorts. The signature K is place-sorted over Θ^P, i.e., the controls, from the basic signature, are assigned each a sort from Θ^P.*

Definition 3. *A link sorting is a triple $\Sigma_L = (\Theta^L, K, \Phi)$, comprising a non-empty set Θ^L of link sorts. The signature K is link-sorted over Θ^L, i.e., the arities of each control, from the basic signature, are assigned each a sort from Θ^L.*

In both place and link sorting, the component Φ (called formation rule) specifies properties that have to be fulfilled by the bigraph. For example, all children of c1-nodes are c2-nodes, or c1-nodes cannot be linked to c2-nodes.

2.5. Modeling dynamic structure: BRS

Bigraphs enable the description of only the static structure of a system through the modeling of the two concepts: locality and connectivity. To complete the description of that system, i.e., modeling its dynamic behavior, these bigraphs are furnished with a bigraphical reaction rules-set. This combination yields what is called *BRS*. The defined reaction rules allow the reconfiguration (rewriting) of bigraphs' placing or linking. A reaction rule is written $R \rightarrow R'$, where the redex (i.e., its left-hand side) R expresses the conditions that have to be satisfied to apply it, and the reactum (i.e., its right-hand side) R' represents the effect of the application of this rule [19]. In other words, a rule *ReacRul* is applicable on a bigraph G if the redex of *ReacRul* occurs in G, and its application means that the reactum of *ReacRul* will replace the matching part in G.

Definition 4. *A reaction rule is a triple* (R, R', η)*, where* $R : m \rightarrow J$*,* $R' : m' \rightarrow J$*, and* $\eta : m' \rightarrow m$ *represent the redex, the reactum, and a map of ordinals, respectively.*

Figure 3a depicts graphically a bigraphical reaction rule called $R_{migrate}$ modeling the migration of a service deployed on a VM offered by a public cloud provider to another VM in a private cloud. The algebraic representation of $R_{migrate}$ is :

$((SaaS_{e0}.d_2) \mid d_0) \parallel (UaaS_{e0}.(DepM(pub) \mid d_3)) \mid (UaaS.(DepM(prv) \mid d_4)) \mid d_1 \rightarrow ((SaaS_{e0}.d_2) \mid d_0) \parallel (UaaS.(DepM(pub) \mid d_3)) \mid (UaaS_{e0}.\ (DepM(prv) \mid d_4)) \mid d_1$

In the redex of $R_{migrate}$ (its left-hand side), a service from the SaaS layer is deployed on a VM from the UaaS layer. This VM contains a node of control DepM(pub) to denote that is provided by a public provider. In the reactum of $R_{migrate}$ (its right-hand side), the first link is removed and a new link between the SaaS service and another VM from a private provider (DepM(prv)) is created. The application of this rule on the bigraph G yields a new bigraph as depicted in Figure 3b. The algebraic representation of this application is :

$(EUser_{e0}.SLA) \mid (EUser.Req) \mid d_0 \parallel (SaaS_{e0,e1,e3}.(SLA \mid d_4)) \mid (SaaS_{e2,e3}.(SLA \mid d_3)) \mid d_1 \parallel (UaaS_{e1,e2}.(SLA \mid DepM(pub) \mid d_6)) \mid (UaaS.(DepM(pub) \mid d_5)) \mid (UaaS.(DepM(prv) \mid d_7)) \mid d_2 \rightarrow (EUser_{e0}.SLA) \mid (EUser.R1) \mid d_0 \parallel (SaaS_{e0,e1,e3}.(SLA \mid d_4)) \mid (SaaS_{e2,e3}.(SLA \mid d_3)) \mid d_1 \parallel (UaaS_{e1}.(SLA \mid DepM(pub) \mid d_6)) \mid (UaaS.(DepM(pub) \mid d_5)) \mid (UaaS_{e2}.(SLA \mid DepM(prv) \mid d_7)) \mid d_2$

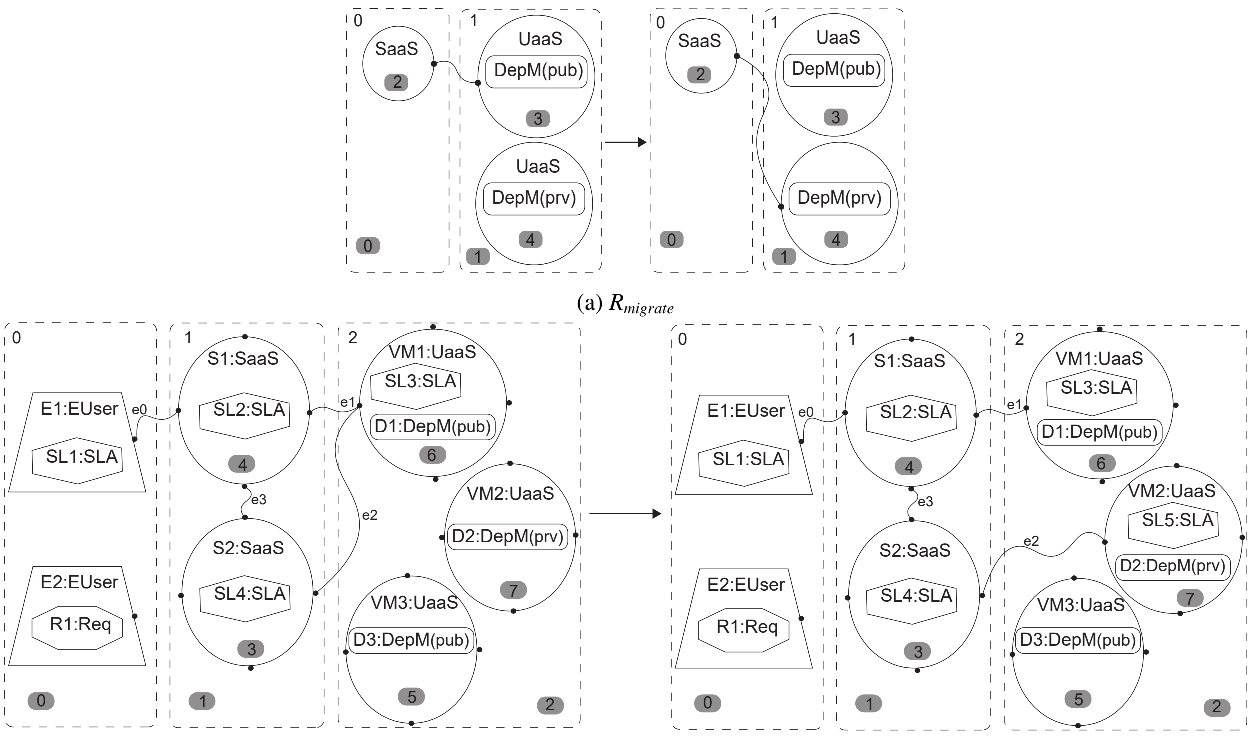
3. Approach overview

With respect to the selected formalism, we have found that BRSs are a well-suited formalism to address the first objective of this research as they can capture the different cloud entities, their properties, their relationships and interconnections, and their dynamic behaviors. Furthermore, BRSs are based on a rigorous mathematical foundation (category theory) and offer simple graphical notations.

Concerning the second objective, the formal verification, few tools are proposed to develop, simulate and formally verify BRS models such as BPLTool [36], BigMC [37], Big Red [38] and BigraphER [39]. Among all of these tools, BigMC [37] is the only one that supports formal verification. However, it does not support parametric controls, and it is not mature enough to express and verify complex properties. Different research works have tackled the formal verification problem in the cloud computing domain [40]. Model checking, as one of the verification techniques, represents an attractive approach that aims to ensure system correctness [21, 22]. The symbolic model checker NuSMV [23] allows to deal with real size models as those we can find out in the cloud computing context [40]. NuSMV offers model checking verification and simulation techniques. According to [40], the symbolic model checker NuSMV has been the most used in the verification of cloud case studies. In this work, we use NuSMV to address the second objective of this research.

Next, we present an overview of the proposed modeling and verification approach for SLA-based cloud systems. It consists of two phases: *modeling* and *verification* (see Figure 4).

- *The modeling phase*: In this phase, we propose a formal modeling approach based on BRS. It is composed of two steps:
  - *The structural modeling step* (label 1): During this step, we can address different aspects of the cloud. The first aspect to consider is the architecture modeling. Bigraphs enable us to model the layered architecture



(a) $R_{migrate}$

(b) Application on bigraph G

Figure 3: Example of a reaction rule and its application

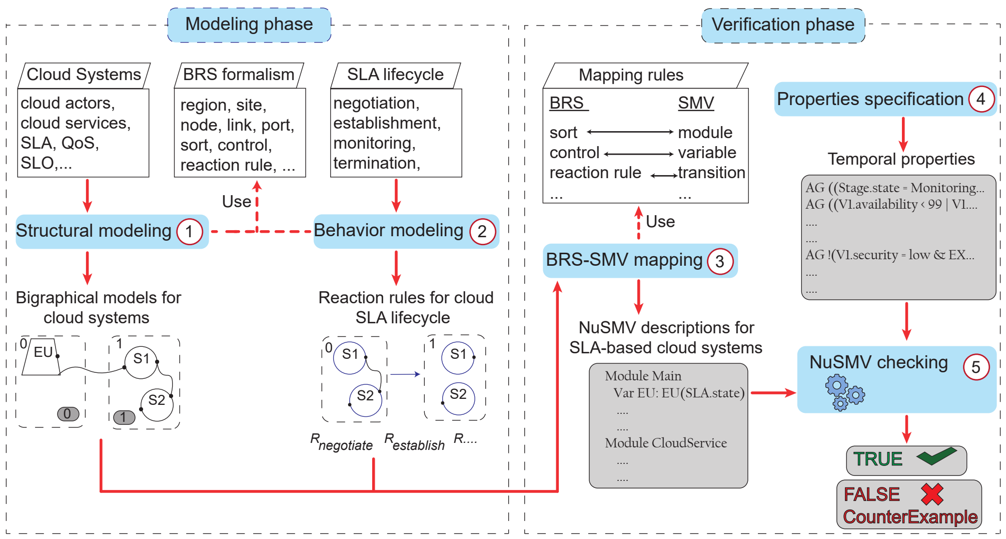

Modeling phase
Verification phase
Cloud Systems
cloud actors, cloud services, SLA, QoS, SLO,...
BRS formalism
region, site, node, link, port, sort, control, reaction rule, ...
SLA lifecycle
negotiation, establishment, monitoring, termination,
Use
Structural modeling 1
Behavior modeling 2
Biographical models for cloud systems
Reaction rules for cloud SLA lifecycle
EU
S1
S2
R_negotiate
R_establish
R....
Mapping rules
BRS
SMV
sort ⟷ module
control ⟷ variable
reaction rule ⟷ transition
...
...
Use
BRS-SMV mapping 3
NuSMV descriptions for SLA-based cloud systems
Module Main
Var EU: EU(SLA.state)
Module CloudService
Properties specification 4
Temporal properties
AG ((Stage.state = Monitoring...
AG ((V1.availability < 99 | V1....
AG !(V1.security = low & EX...
NuSMV checking 5
TRUE
FALSE
CounterExample

Figure 4: SLA-based cloud systems modeling and verification approach overview

of cloud computing. We use the concept of region in the bigraph theory to model each layer of the cloud. The second aspect is the service modeling. This approach proposes the use of bigraph nodes and sorting mechanism to define a generic model for cloud services. The use of place sorts (Σ_P) allows us to define different kinds of services and describe their properties. The third aspect is the service composition and interconnection modeling. We use bigraph links to model the relationships and dependencies between the different services. We define different ports (link sorts Σ_L), through which, services satisfy their requirements and expose their offerings. Another important aspect is the SLA. We use also nodes and sorting mechanism to represent this agreement and its components.

  - *The behavior modeling step* (label 2): First, based on the work [8], we propose an SLA lifecycle, which consists of four stages (see Figure 4). Then, we define different bigraphical reaction rules to describe these stages. First, we introduce two reaction rules to represent the negotiation and the establishment of an SLA. In the monitoring stage, we define a rule to model the SLA violation detection and other rules to describe how to fix the violation (either by a service adjustment or by an adjustment of the SLA). Finally, we present two rules expressing two possible cases of termination that are normal termination and termination due violations.

- *The verification phase*: After the modeling phase, the verification employs model checking techniques to check the correctness of interaction behaviors among the different cloud entities during the SLA lifecycle. It follows three steps:

  - *Mapping BRS to SMV descriptions* (label 3): We follow a set of proposed mapping rules to translate the obtained BRS models to SMV descriptions.
  - *Properties specification* (label 4): We express the interaction behaviors of the different cloud entities as a set of liveness and safety properties using LTL and CTL formulas [21, 22].
  - *NuSMV checking* (label 5): In this step, the NuSMV model checker takes as input the resulting SMV descriptions and the defined properties. As a result, it returns whether these properties are satisfied or not. In case a property is not satisfied a counterexample is provided with all the information necessary to reproduce the problem, which allows to trace errors.

The number and complexity of concepts addressed in both phases depend on the type and complexity of the properties we want to check. In addition, the use of sites allows us to focus on these concepts alleviating the model complexity. To implement this approach, there are different mechanisms to extract the information regarding these concepts and they are closely related to the used cloud provider. For instance, Amazon Web Services (AWS) provides a large set of SLAs [1] for the different featured services it offers such as the Amazon Elastic Compute Cloud (EC2) [2]. These SLAs are not enforced by Amazon, but by the users themselves via a manual claim management system using the request logs, which must be provided by the user. This practice is common among cloud providers, however it will be desirable their automatic enforcement. Initiatives like TOSCA (Topology and Orchestration Specification for Cloud Applications) [41] and OCCI (Open Cloud Computing Interface) [42] to standardize the specification of cloud applications may pave the path to accomplish this task.

4. Structural modeling

In this section, we define the static structure of services, actors and SLAs using bigraphs' nodes. Moreover, we use bigraphs' links to model the collaboration between these services, and the interaction between end-users and services.

4.1. Cloud service description

We consider services as the first-class entities. We use bigraphs' nodes to represent these services. We define $\Theta^P_{\{service\}}$ a set of place sorts for services as: $\Theta^P_{\{service\}} = \{XaaS_1, XaaS_2, \ldots, XaaS_n\}$ where each sort represents the type of service.

The content of $\Theta^P_{\{service\}}$ depends on the considered types of services. We will give more details about each sort later, i.e., we will present controls belonging to each sort and describing the states of services.

[1]`https://aws.amazon.com/legal/service-level-agreements/`

[2]`https://aws.amazon.com/compute/sla/`

Example 1. *In the literature, we can find different classifications of service models. Following the classification proposed by [2] and [4], the* $\Theta^{P}_{\{service\}}$ *is instantiated as:* $\Theta^{P}_{\{service\}} = \{SaaS, PaaS, IaaS\}$*. The IBM Cloud Computing Reference Architecture has extended this classification by adding the Business Process as a Service model (BPaaS) [43, 44]. Therefore the* $\Theta^{P}_{\{service\}}$ *will be:* $\Theta^{P}_{\{service\}} = \{SaaS, PaaS, IaaS, BPaaS\}$.

Example 2. *According to [10], cloud services fall into two main categories: application services (they refer to the* $SaaS$ *model) and utility services (denoted as* $UaaS$ *and they include the PaaS and IaaS models). Thus we instantiate* $\Theta^{P}_{\{service\}}$ *as:* $\Theta^{P}_{\{service\}} = \{\ SaaS,\ UaaS\ \}$.

Generally, cloud services are associated with different functional and non-functional properties [12]. To model these properties, a *service*-node may contain other nodes such as:

- $\Theta^{P}_{\{Offering\}} = \{offer\}$: this sort contains only one control $offer$. A node of this control describes what the service offers to end-users and/or other services.
- $\Theta^{P}_{\{Requirement\}} = \{req\}$: while the last control describes what the service provides to others, a *req*-node describes what the service requires from others, for example, services and resources on which it will be deployed. A node of this control marks that its parent (in this case the *service*-node in which it is contained) is not ready to provide its capabilities because it has some requirements to function.
- $\Theta^{P}_{\{property\}} = \{Prop_1(x), Prop_2(x), \ldots, Prop_m(x)\}$: We can add more nodes to describe other service's properties. We propose to use parameterized controls [34, 29] to describe them. For example, a node representing a resource (virtual machine) from the $UaaS$ layer can contain nodes of type $RAM(x)$ and $CPU(x)$ to describe its configuration in terms of RAM and CPU, respectively. The instantiation of them means that we give real values for the parameter x, for instance, $RAM(2)$ and $CPU(3)$ denote that its capacity of RAM is $2GB$ and its CPU has the capacity $3GHz$.

We note here that these nodes depend on their parents (in which they will be inserted). For instance, the *property*-nodes of services from the $SaaS$ layer will not be the same as the *property*-nodes of resources from the $UaaS$ layer. In addition, as previously mentioned, sites have an important role in bigraphs, since they help us to abstract away details that are not our current interest. Indeed, this feature allows us to control the level of abstraction depending on our necessities. For instance, we can abstract a set of *property*-nodes inside a *service*-node by a site and thus reducing the complexity of the model and show only relevant information currently under study.

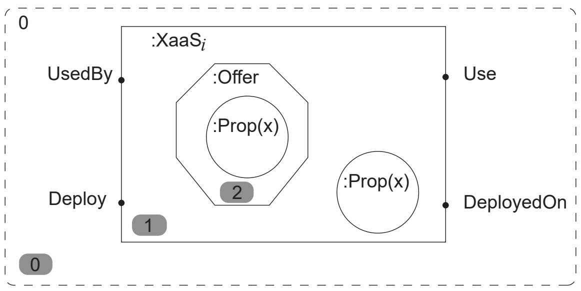
Example 3. *Figure 5a shows a generic model for cloud services. We use this model to describe an Amazon EC2 instance* [3]. *Figure 5b illustrates that this instance is provided by Amazon Web Service (*$Provider(AWS)$*), hosted in Europe-Ireland (*$Location(eu-west-1)$*) and running an Amazon Linux operating system (*$OS(AmazonLinux)$*). The type of this instance is m5.large (*$Type(m5.large)$*). The parametrized controls* $vCPU(2)$*,* $Memory(8)$*, and* $CPU(3,1)$ *represent other properties of this instance that are the number of processors, the capacity of its memory and its processors. In addition to the last controls that denote functional attributes, this instantiation may also include controls to describe QoS attributes such as* $Availability(99,99)$.

4.2. Cloud Actors

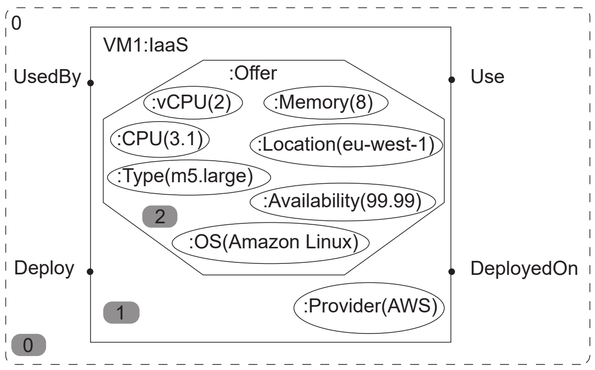
In this paper, we consider that a cloud actor can be a provider of cloud services or an end-user. In the following, we describe how to model each type:

- Provider of services: providers can be of different types depending on the services they are offering. In the bigraph model, we propose that each region corresponds to a layer in the cloud architecture in which we find providers of the same service model (services that belong to the same cloud layer). Hence, the set of nodes representing services are categorized in separate bigraph's regions depending on their types. For instance, Figure 6b depicts a bigraph with two regions that model $XaaS_i$ and $XaaS_j$ layers.

[3]https://aws.amazon.com/ec2/instance-types/

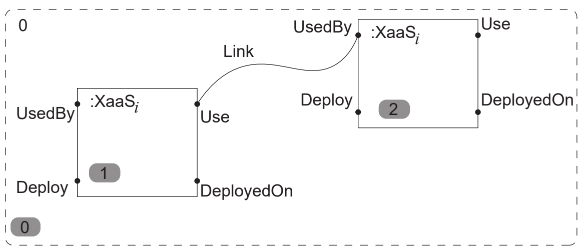


(a) Generic model

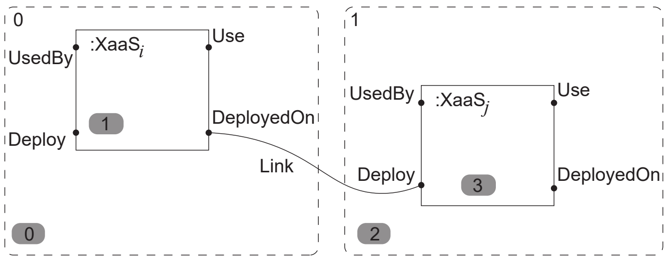


(b) Amzon EC2 instance description

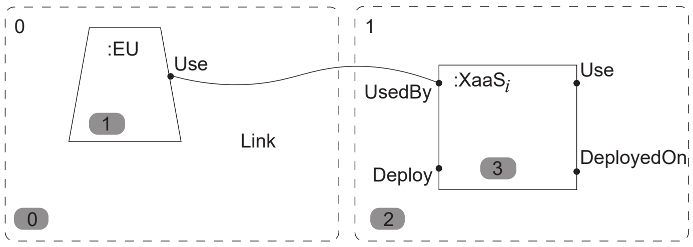
Figure 5: Generic model for cloud service and an example of instantiation



(a) Horizontal composition



(b) Vertical composition



(c) Interaction between End-user and services

Figure 6: Generic model for horizontal composition, vertical composition and End-users

- End-user: in addition to providers' regions, we define another region that represents end-users. We model these latter as a set of nodes from the sort: $\Theta^{P}_{\{Euser\}} = \{EU\}$. *Euser*-nodes may also contain other nodes, for example, to model requirements of an end-user we add a *req*-node in *Euser*-node. For instance, region 0 and 1 represent end-users and providers exposing $XaaS_i$ services, respectively (see Figure 6c).

Example 4. *Consider the hierarchical view of cloud computing presented in [45]. It involves four layers, namely SaaS, PaaS, IaaS, and Data Centers layers. The graphical representation of its bigraphical model is shown in Figure 7. In addition to the first region representing end users, there are four other regions denoting the SaaS, PaaS, IaaS, and Data Centers layers. This last layer provides physical machines (PM).*

4.3. Service Interaction

In this section, we continue the description of nodes we have just defined before. We start by the definition of different types of ports for *service*-nodes and *Euser*-nodes. Then, we use these ports to show how services are connected to each other (in terms of two types of compositions) and how end-users consume these services.

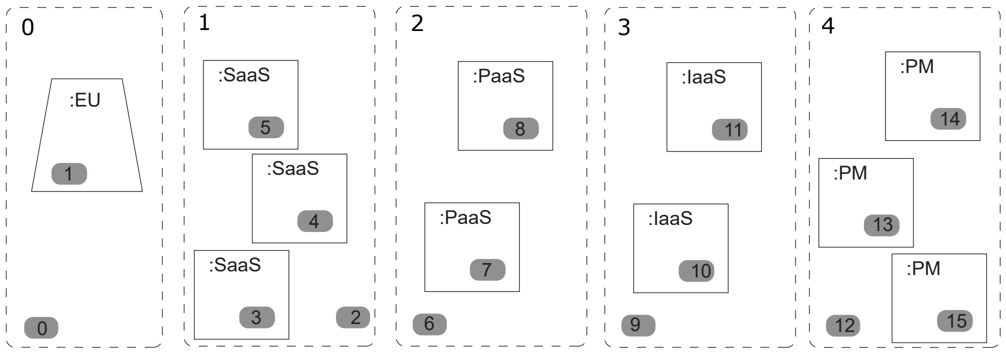


Figure 7: Example of Cloud computing actors

Cloud computing is composed of different layers that are related in a producer-consumer way [46]; the first layer consumes services from providers of the second layer, and these latter may also play the role of consumers as they consume services from the next layer, etc. Therefore, we associate for each *service*-node four ports: { *Deploy*, *DeployedOn*, *Use*, *UsedBy* } that allow them to play the role of consumer and/or provider, i.e., they use these ports to satisfy requirements and/or offer capabilities. On top of the cloud layers, end-users interact with and consume services offered by the different providers. Therefore, we associate only one port $\{Use\}$ to *Euser*-nodes because end-users just consume services and do not offer anything. We will give more details about these ports in what follows. We note here that the proposed model is extensible and we have the possibility to add more ports to represent other types of relationships if necessary.

Definition 5. *We define a service composition as a set of linked service-nodes. Bigraph's nodes having sort* $\Theta^{P}_{\{service\}}$ *specify services involved in the composition and links, connected through ports* { *Deploy, DeployedOn, Use, UsedBy* }*, define relationships between them.*

The notation $(v, portSort)$ expresses the port *portSort* on the node v. According to [14, 47, 48], the service composition in the cloud computing area is defined in two dimensions: horizontal composition (or intra-layer composition) and vertical composition (or inter-layer composition) :

- Horizontal composition: it expresses the combination of services that are on the same level (have the same service model). In order to model this type of composition, we assign two ports to *service*-nodes: $\Theta^{L}_{\{service\}} = \{Use, UsedBy\}$. A link connected to the first port expresses that the service consumes capabilities offered by another service, while the connection to the *UsedBy* port models that this service offers features to the others.

  Definition 6. *The horizontal composition of two services* $S1$ *and* $S2$ *is defined by the creation of a new link* e *between* $(S1, Use)$ *and* $(S2, UsedBy)$*.*

  In the example of Figure 6a, two services from the same layer are horizontally composed.

- Vertical composition: it was introduced by Mietzner [14]. It expresses the collaboration of services of different models to offer complete solutions and satisfy the functional and non-functional requirements of users [13]. To model this kind of composition, we extend the link sorts by two new ports and assign them to *service*-nodes: $\Theta^{L}_{\{service\}} = \Theta^{L}_{\{service\}} \cup \{Deploy, DeployedOn\}$. While the deployment feature is offered through the connection to the first port, a connection from the second port models the need to this feature.

  Definition 7. *The vertical composition of two services* $S1$ *and* $S2$ *(*$S1$ *is deployed on* $S2$*) is defined by the creation of a new link* e *between* $(S1, DeployedOn)$ *and* $(S2, Deploy)$*.*

  In the example of Figure 6b, an $XaaS_i$ service is vertically composed with an $XaaS_j$ service from another layer, meaning that the first one is deployed on the second one. For instance, a software application from the $SaaS$ layer needs other resources from $PaaS$ or $IaaS$ layers to be deployed.

In addition to the interaction between services, end-users can also interact with and consume services. Therefore, we define a new port Use and assign it to $Euser$-nodes: $\Theta^{L}_{\{Euser\}} = \{Use\}$.

Definition 8. *The interaction between an end-user $EU1$ and a service $S1$ is defined by the creation of a new link e between the port Use in the node $EU1$ and the port UsedBy in the service $S1$.*

Figure 6c models an end-user using an $XaaS_i$ service.

4.4. SLA-driven Modeling

The SLA permits to regulate and control the interaction of service providers with their consumers. The SLA consists of different components including among others [6, 8]:

- Parties: Different parties are involved in the establishment and management of the SLA such as signatory parties (customers and providers) and supporting parties (such as brokers and auditors). In this work, we focus on the first that represents providers and consumers of services. We have just defined sorts $\Theta_{\{Euser\}}$ and $\Theta_{\{service\}}$ representing end-users and services offered by providers, respectively.
- Service-level objectives (SLOs): The SLA indicates the functional and non-functional capabilities of services that have to be ensured. Service availability must be at least 99,99% and response time must be at most 2 sec. are some examples of SLOs. Now, we define a new sort $\Theta_{\{SLA\}} = \{SLA\}$ to represent the SLA between providers and their customers.
- Penalties: They will be applied if the expected SLOs are not achieved. We use a node of control *Pen* defined in the sort $\Theta_{\{Pen\}}$ to model these penalties: $\Theta_{\{Pen\}} = \{Pen\}$.

Moreover, we can define other sorts to model more components in the *SLA* or we can just use a site inside the *SLA*-node to abstract them away. According to [49], different cloud SLAs can be defined due to the layered architecture of the cloud and the producer-consumer relationship between these layers. Therefore, we add new *SLA*-nodes (in both nodes representing consumers and providers) each time an interaction between the parties is started. We note here that we will define different states of *SLA*-nodes that depend on the different stages of the *SLA* lifecycle. Indeed, once the interaction has started, the state of the inserted *SLA*-node is updated according to the stage of the *SLA* lifecycle.

Definition 9. *Let $S1$ and $S2$ be two services. $S2$ requires capabilities offered by $S1$. We define an SLA between the providers of $S1$ and $S2$ by the insertion (update of the states) of SLA-node inside the two nodes representing services $S1$ and $S2$.*

Definition 10. *Let $S1$ be a service and let $EU1$ be an end-user. The capabilities offered by $S1$ meet the requirements of $EU1$. We define the SLA between the provider of $S1$ and the end-user $EU1$ by the insertion (update of the states) of SLA-node inside the two nodes representing $S1$ and $EU1$.*

In the last two definitions we mean by the SLA not only the final established agreement but also this agreement in its different states during the *SLA* lifecycle.

Figure 8 shows two cases, from right to left-hand-side: in the first one, we have two providers offering $XaaS_j$ and $XaaS_k$ services, to model the interaction and the *SLA* between these providers we insert an *SLA*-node in both of them. In the second case, the two parties are an end-user (from region 0) and the provider of $XaaS_i$ service (from region 1). The *SLA* between them is modeled by the insertion of an *SLA*-node in both of them. In the two cases, we have also created a new link between nodes representing consumer and provider of services to show the interaction between the two parties. We note here that *SLA*-nodes contain sites to model the *SLA* components and properties such as the *SLOs*.

4.5. Modeling services, end-users and SLA states

We now give more details about the sorts that we have defined. We start by defining the states of services, end-users, and SLA as a set of controls inside $\Theta_{\{service\}}$, $\Theta_{\{Euser\}}$, and $\Theta_{\{SLA\}}$, respectively. We present a set of controls in each sort to represent real states of these entities through the different stages of the lifecycle of SLA.

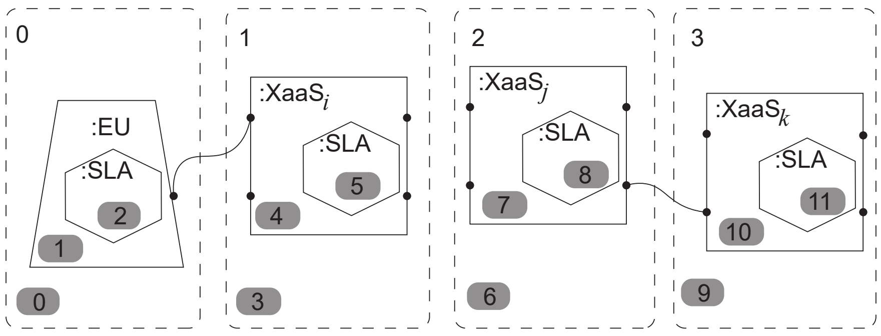


Figure 8: SLA at different levels

- Service states: In order to specify the states of services, we use three controls :

  $\forall XaaS \in \Theta_{\{service\}}, XaaS = \{XaaS^{Req}, XaaS^{PL}, XaaS^{BPL}\}$

  The control $XaaS^{Req}$ models a service that is not ready to be used because it has **requirements** that demand to be satisfied. For example, to be deployed, a $SaaS$ application requires other infrastructure resources. Once a service has satisfied all its requirements, it can be in one of the following two states. The first one is represented by the control $XaaS^{PL}$ denoting a service that is offering (can offer) the **promised level** of service. We use the control $XaaS^{BPL}$, representing the second case, when the level of service, that is in use, is decreased **below** the promised one.

- End-users states: The sort $EU = \{EU^{Req}, EU^{Sat}, EU^{DisSat}\}$ defines the different states that an end-user can experience during the SLA lifecycle. We use the first control EU^{Req} to represent an end-user looking for a service, i.e., he/she has some **requirements**. Once the end-user has found and started using the service, we use the two controls EU^{Sat} or EU^{DisSat} to model its states. While the former denotes that he/she is **satisfied** with the offered service, the latter denotes a **dissatisfied** end-user.

- SLA states: The different states of the SLA are regrouped in the sort $SLA = \{SLA^{negotiated}, SLA^{respected}, SLA^{violated}, SLA^{success}, SLA^{failure}\}$

  We use the first control to describe the **negotiation** between parties in order to define and establish the SLA.

  Once the SLA has been established, and in order to model its monitoring, we use two controls. A node of $SLA^{respected}$ control describes that the SLA is **respected**, and the control $SLA^{violated}$ denotes that an SLA **violation** is detected.

  According to [8] there are two cases in which the SLA may be terminated. We use the two controls $SLA^{success}$ and $SLA^{failure}$ to describe these cases. The former denotes **normal termination**, i.e., the SLA reaches its timeout, and the latter describes the case in which the termination is due an SLA violation (**termination due violations**).

Table 2 summarizes the correspondence between concepts in cloud computing and bigraphs.

Table 2: Correspondence between cloud computing concepts and bigraphs

Cloud computing	Bigraph
service, SLA, end-users, properties	nodes
cloud entities states	controls
interaction	links
cloud layers	region

5. Behavior modeling

We now complete the definition of the proposed bigraphical model by addressing its dynamic structure. The dynamic aspect describes the SLA lifecycle and how the states of end-users, services, and SLAs are evolving during the lifecycle of the SLA. The realization of the SLA follows different stages [8, 9]. In our work, we propose a lifecycle in four stages, based on the one defined in [8], which are "negotiation", "establishment", "monitoring", and "termination". We introduce a reaction rules-set that allows the transition between these stages and describes the dynamic changes of end-users states, cloud services states, and SLAs states. Figure 9 shows the proposed SLA lifecycle and the different reaction rules allowing the transition from one stage to another one. In the following figures, we show only the connected ports. In addition, the notations X/Y, used in figures, and $\widehat{XY}$, used in formulas, indicate that the control of the node can either be X or Y. For instance, the reaction rule (Figure 11) can be applied for end-users (by using EU^{Req} in the redex and EU^{Sat} in the reactum) or for $XaaS$ services (by using $XaaS^{Req}$ in the redex and $XaaS^{PL}$ in the reactum).

5.1. Negotiation

A consumer locates providers depending on its needs. Generally, a consumer selects cloud service providers based on its functional requirements and then negotiates nonfunctional properties of the desired services [50]. During the negation stage, the two parties negotiate in order to reach a mutual agreement and define the final SLA components such as SLOs, penalties, and budget. Reaction rules depicted in Figure 10 model this stage.

In the first rule (from left to right →), the left-hand side indicates a consumer, in region 0, and a service $XaaS^{PL}$, in region 1, that can satisfy this consumer's requirements. The consumer can be either an end-user with requirements (EU^{Req}) or a service requiring other services or resources ($XaaS^{Req}$). Starting the negotiation is modeled by the right-hand side, in which we create a new link between this consumer and the service $XaaS^{PL}$. Moreover, we add a node of control $SLA^{negotiated}$ in both nodes representing the consumer and the offered service. We can specify $R_{negotiate}$ in algebraic terms as follows:

$R_{negotiate} \overset{\text{def}}{=} ((\widehat{EU^{Req}XaaS^{Req}}.(d_1 \mid req)) \mid d_0) \parallel ((XaaS^{PL}.d_3) \mid d_2) \rightarrow ((\widehat{EU_e^{Req}XaaS_e^{Req}}.(d_1 \mid req \mid (SLA^{negotiated}.d_4)$ $)) \mid d_0) \parallel ((XaaS_e^{PL}.d_3 \mid (SLA^{negotiated}.d_5)) \mid d_2)$

The process of negotiation may terminate in two scenarios:

- In the first scenario, the two parties have not reached a mutual agreement and thus ceasing the negotiation process. This scenario is modeled by the opposite reaction rule depicted in Figure 10 (from right to left ←) , in which the bigraph goes back to the initial state (remove the link between the two parties and also the two nodes $SLA^{negotiated}$ from both of them).
- The second scenario represents a successful termination of the negotiation process. Thus we go to the next stage: establishment of SLA.

5.2. Establishment

After a successful negotiation, the SLA will be established, and the expected service will be delivered. The rule $R_{establish}$ represents this scenario (Figure 11). Its left-hand side represents the negotiation step, i.e., the consumer and provider are negotiating, and therefore they are linked together and both of them contain an $SLA^{negotiated}$-node. The application of this rule models that the two parties have agreed on the terms of the *SLA*, and therefore we replace the node of control $SLA^{negotiated}$ by the node of control $SLA^{respected}$ on its right-hand side. Moreover, the consumer's state is updated from EU^{Req} ($XaaS^{Req}$ respectively) to EU^{Sat} ($XaaS^{PL}$ respectively) and the *req*-node is removed from it. We note here that the link now describes that this consumer has begun consuming this $XaaS^{PL}$ that is expected to fulfill his/her requirements. In algebraic terms, $R_{establish}$ can be denoted as follows:

$R_{establish} \overset{\text{def}}{=} ((\widehat{EU_e^{Req}XaaS_e^{Req}}.(d_1 \mid req \mid (SLA^{negotiated}.d_4))) \mid d_0) \parallel ((XaaS_e^{PL}.d_3 \mid (SLA^{negotiated}.d_5)) \mid d_2) \rightarrow$ $((\widehat{EU_e^{Sat}XaaS_e^{PL}}.(d_1 \mid (SLA^{respected}.d_4))) \mid d_0) \parallel ((XaaS_e^{PL}.d_3 \mid (SLA^{respected}.d_5)) \mid d_2)$

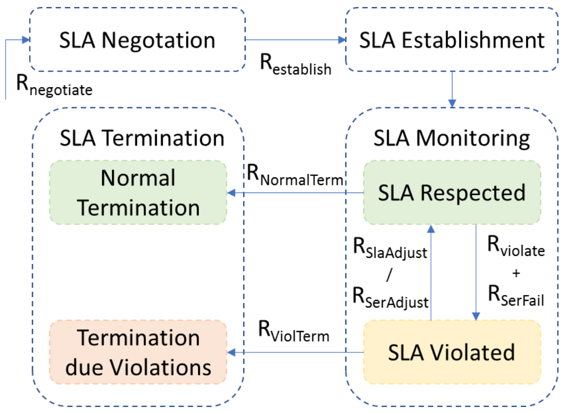


Figure 9: SLA lifecycle in the cloud

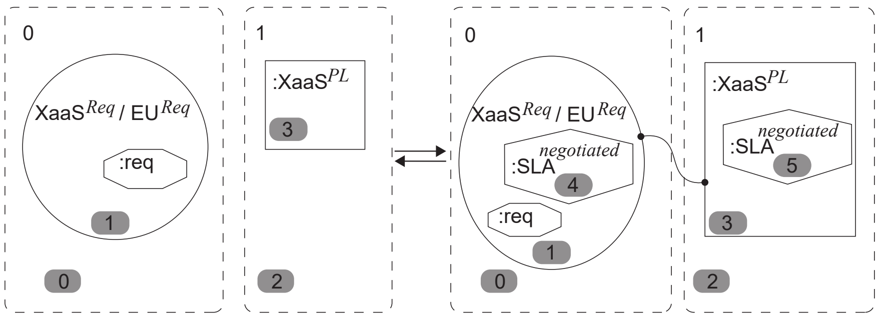


Figure 10: A consumer and provider start an SLA negotiation (→), negotiation failure (←)

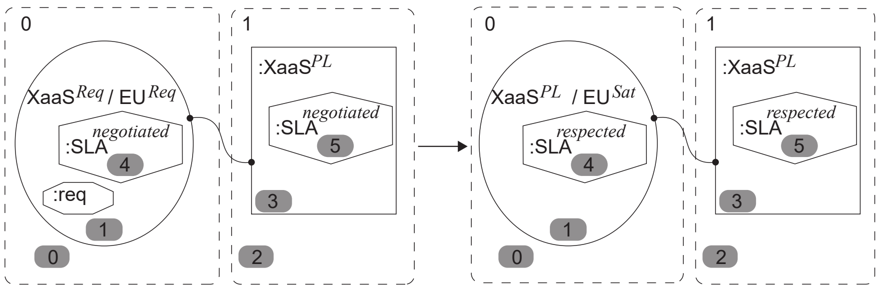


Figure 11: Reaction rule for an SLA establishment

5.3. Monitoring

The monitoring process is launched *immediately* once the SLA is established and the consumer commences the use of the provided service. Hence Figure 9 has not shown any reaction rules between the establishment and the monitoring stages. This process verifies the execution of services according to the agreed SLA. It constantly observes and compares the actual provisioned service against the pre-agreed level of service, and then, it reports any SLA violations. The process of monitoring will be continued until a termination state, defined in the SLA, is reached (see Sect. 5.4). In this stage, we first define a rule that tags a service as unable to offer the promised level (*Service level degradation*), then, we introduce a rule that models an *SLA violation*, and finally, we present rules describing different scenarios after an SLA violation (*SLA violation treatment*).

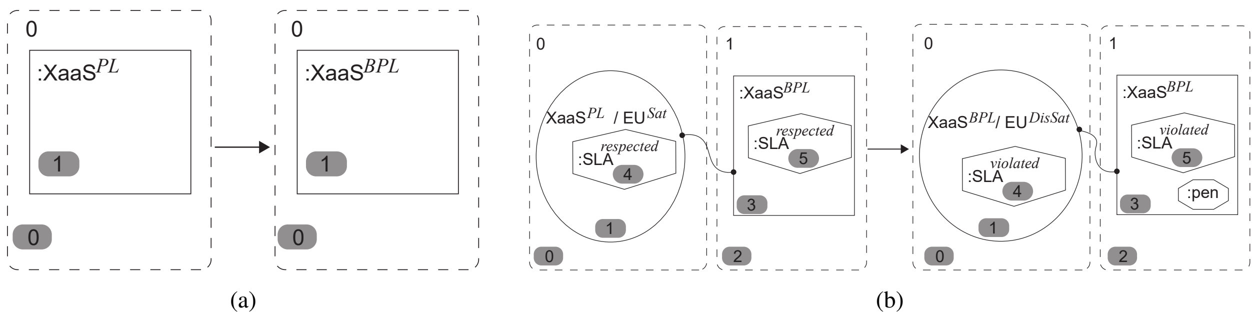


Figure 12: Service Failure And SLA Violation

5.3.1. *Service level degradation*

Deployed in a highly dynamic environment, such as the cloud, QoS of the delivered services are subjected to different changes. Figure 12a illustrates how to mark a service that *has failed* to satisfy the consumer's requirements (for example due to QoS degradation). In the reactum of this rule $R_{SerFail}$, the service's state is updated from $XaaS^{PL}$ to $XaaS^{BPL}$; meaning that its level has decreased below the agreed one. The algebraic form of $R_{SerFail}$ is:

$R_{SerFail} \overset{\text{def}}{=} (XaaS^{PL}.d_1) \mid d_0 \rightarrow (XaaS^{BPL}.d_1) \mid d_0$

5.3.2. *SLA violation*

Once a service has failed to reach the expected level, an *SLA violation* is occurred. To model this situation we use the rule depicted in Figure 12b. The redex of this rule denotes a service that cannot fulfill the consumer's requirements, thus we assign the control $XaaS^{BPL}$ to the offered service (we use the last rule $R_{SerFail}$ to mark that this service is unable to offer the promised level of service). On the right-hand side, we update the consumer's state from EU^{Sat} ($XaaS^{PL}$ respectively) to EU^{DisSat} ($XaaS^{BPL}$ respectively) to model that he/she is not satisfied with the offered service. The reactum updates also the state of the SLA from respected ($SLA^{respected}$) to violated ($SLA^{violated}$). Updating the SLA state in the two nodes models that both parties (provider and consumer) are informed by this violation. In addition, we insert a new *Pen*-node in the offered $XaaS^{BPL}$ to express that the corresponding penalties defined in the SLA (in the case of its violation) are applied to the provider of this service (in this case we have considered that the provider is the party who violate the SLA). This rule is defined algebraically as follows:

$R_{violate} \overset{\text{def}}{=} ((\widehat{EU_e^{Sat} XaaS_e^{PL}}.(d_1 \mid (SLA^{respected}.\ d_4))) \mid d_0) \parallel ((XaaS_e^{BPL}.\ d_3 \mid (SLA^{respected}.d_5)) \mid d_2) \rightarrow ((\widehat{EU_e^{DisSat} XaaS_e^{BPL}}.(d_1 \mid (SLA^{violated}.\ d_4))) \mid d_0) \parallel ((XaaS_e^{BPL}.d_3 \mid (SLA^{violated}.d_5) \mid Pen) \mid d_2)$

5.3.3. *SLA violation treatment*

We consider the following three scenarios when an SLA violation is occurred:

- Service adjustment: the provider, in this first scenario, possesses the ability to resolve the detected SLA violation and re-provide the pre-agreed level of service (it has remediation strategies for the detected problem). We define the rule $R_{ServiceAdjust}$, depicted in Figure 13a, to model this scenario. After an SLA violation (denoted by the redex of this rule), the provider applies the appropriate strategy and executes the corrective actions (add new instance, service substitution , resource reconfiguration, etc.). We note that the corrective strategies are not the scope of this work and readers can refer to [51, 52] in which the authors have modeled elasticity actions as reaction rules. The reactum of $R_{ServiceAdjust}$ models that the provider has successfully resolved the problem; note that the bigraph goes back to its state before the SLA violation where the two parties are satisfied (EU^{Sat} and $XaaS^{PL}$) and the agreement is respected ($SLA^{respected}$). Moreover, the reactum of $R_{ServiceAdjust}$ has substituted the service $XaaS$ with a new one $XaaS'$ (we have used the prime symbol to differ them) to denote the application of the service adjustment and the corrective strategies. The algebraic notation of this rule is:

  $R_{SerAdjust} \overset{\text{def}}{=} ((\widehat{EU_e^{DisSat} XaaS_e^{BPL}}.(d_1 \mid (SLA^{violated}.\ d_4))) \mid d_0) \parallel ((\ XaaS_e^{BPL}.d_3 \mid (SLA^{violated}.d_5) \mid Pen) \mid d_2) \rightarrow ((\widehat{EU_e^{Sat} XaaS_e^{PL}}.(d_1 \mid (SLA^{respected}.\ d_4))) \mid d_0) \parallel ((XaaS'^{PL}_e.\ d_3 \mid (SLA^{respected}.d_5)) \mid d_2)$

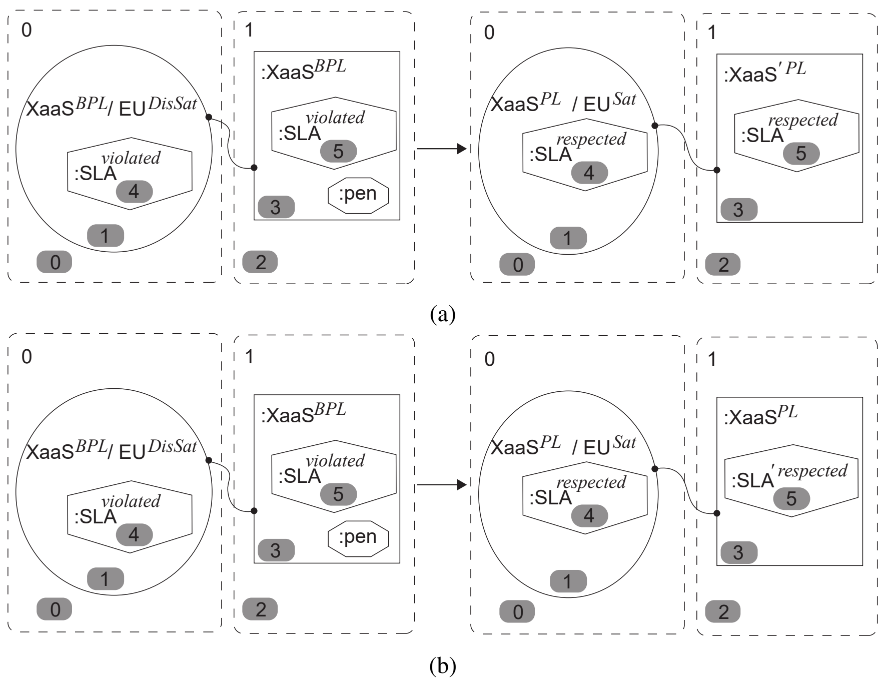


Figure 13: Service and SLA Adjustment

- SLA adjustment: in this scenario, the two parties renegotiate the current SLA and agreed on the definition of a new one by updating some of the $SLOs$. In contrast to static and predefined SLAs that cannot be changed after their establishment, recent studies have addressed the problem of dynamic SLAs and their adjustment during the service provisioning [53, 54]. The rule $R_{SlaAdjust}$, depicted in Figure 13b, illustrates this scenario. Analogy to the first scenario, the application of this rule brings the bigraph back to its state before the SLA violation. In addition, in the reactum of this rule, we have used a new node SLA' in the place of SLA to model that the agreement is renegotiated and a new one is defined. The algebraic terms of this rule is:

$$R_{SlaAdjust} \overset{\text{def}}{=} ((\widehat{EU_e^{DisSat}XaaS}_e^{BPL}.(d_1 \mid (SLA^{violated}.\, d_4))) \mid d_0) \parallel ((\, XaaS_e^{BPL}.d_3 \mid (SLA^{violated}.d_5) \mid Pen) \mid d_2) \rightarrow ((\widehat{EU_e^{Sat}XaaS}_e^{PL}.(d_1 \mid (SLA^{respected}.\, d_4))) \mid d_0) \parallel ((XaaS_e^{PL}.\, d_3 \mid (SLA'^{\,respected}.d_5)) \mid d_2)$$

- Provider limitations: if the encountered problems are beyond remedy (the provider cannot remedy the problem or the parties have not agreed on new terms in the renegotiation process), the *termination due violations* will be the next stage of the SLA lifecycle.

5.4. Termination

In this stage we model how the SLA can be terminated. We consider two scenarios [8] in which an SLA termination may be occurred:

- Normal termination: In this first case, the consumer has continued using the service until the predefined expiry of the agreement. In the left-hand side of the rule (Figure 14a), we have a satisfied consumer (EU^{Sat} or $XaaS^{PL}$) using an offered service $XaaS^{PL}$ and a respected SLA between the two parties ($SLA^{respected}$). The right-hand side models the successful fulfillment of the SLA, thus the node of control $SLA^{respected}$ is replaced by the node of control $SLA^{success}$. This rule can be defined as:

$$R_{NormalTerm} \overset{\text{def}}{=} ((\widehat{EU_e^{Sat}XaaS}_e^{PL}.(d_1 \mid (SLA^{respected}.d_4))) \mid d_0) \parallel ((\, XaaS_e^{PL}.(d_3 \mid (SLA^{respected}.d_5))) \mid d_2) \rightarrow ((\widehat{EU_e^{Sat}XaaS}^{PL}.((SLA^{success}\, .d_4) \mid d_1)) \mid d_0) \parallel ((XaaS^{PL}.(d_3 \mid (SLA^{success}.d_5))) \mid d_2)$$

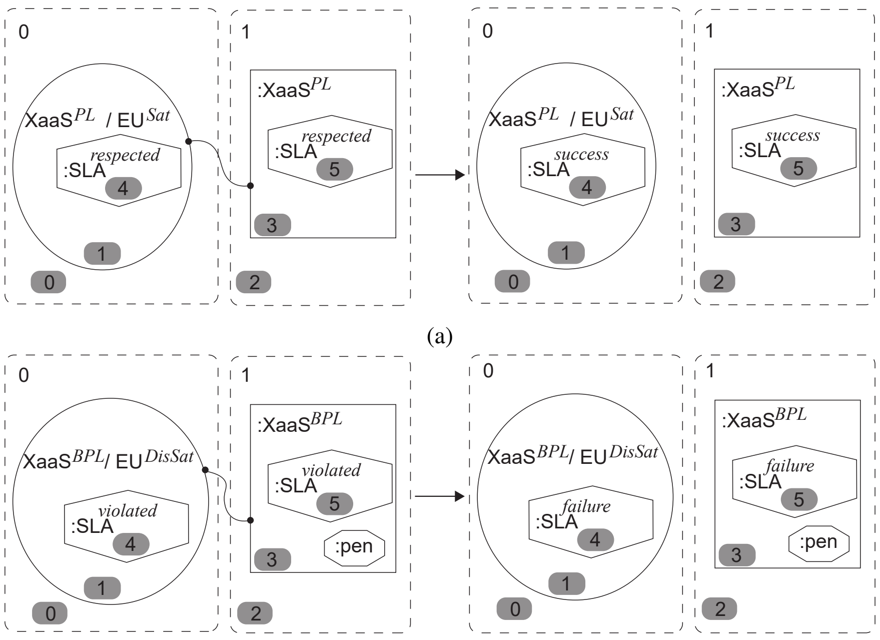


Figure 14: Normal termination And Termination due violations

- Termination due violations: In this scenario, the consumer ceases the utilization of the offered service before the contract expired. This termination may occur in some cases, e.g., one of the two parties cancels the contract, the number of violations reaches the maximum allowed and defined in the SLA [55]. To specify this scenario, we use the rule depicted in Figure 14b. It is applied after SLA violations, thus the state of the SLA in its redex is violated ($SLA^{violated}$). The redex shows also that the consumer is dissatisfied (EU^{DisSat} or $XaaS^{BPL}$) with the consumed service ($XaaS^{BPL}$). The reactum models that the SLA has not been fulfilled ($SLA^{failure}$) and has been terminated due violations. This rule can be defined as:

  $R_{ViolTerm} \overset{\text{def}}{=} ((EU_e^{\widehat{DisSat}}\widehat{XaaS}{}_e^{BPL}.(d_1 \mid (SLA^{violated}.d_4))) \mid d_0) \parallel ((XaaS_e^{BPL}.(d_3 \mid Pen \mid (SLA^{violated}.d_5))) \mid d_2) \rightarrow$
  $((EU_e^{\widehat{DisSat}}\widehat{XaaS}{}^{BPL}.((SLA^{failure}.d_4) \mid d_1)) \mid d_0) \parallel ((XaaS^{BPL}.(d_3 \mid Pen \mid (SLA^{failure}.d_5))) \mid d_2)$

Note that once this stage is achieved (in both termination scenarios), the allocated resources and services are released and thus no link between the two parties remains in the reactum of $R_{NormalTerm}$ and $R_{ViolTerm}$,

6. Verification

In this work, we use the NuSMV [23] model checker to verify the correctness of interaction behaviors of cloud entities during the whole lifecycle of SLA. The verification phase starts by mapping the proposed bigraphical models to SMV descriptions. In the second step, we specify the set of liveness and safety properties we want to check. Finally, the NuSMV model checker verifies the resulting SMV descriptions against the defined properties.

6.1. Mapping BRS to SMV descriptions

In this subsection, we describe how to translate the proposed bigraphical models to SMV descriptions. Table 3 summarizes the mapping rules between BRS concepts and SMV language. We associate each sort in the bigraphical models with a module in the SMV code. For instance, we create two SMV modules called EndUser and CloudService to represent the two sorts EU and XaaS, respectively. Furthermore, the BRS using these latter sorts is defined as the SMV main module and in which the other modules are instantiated. We create a variable of type enumeration called state in each SMV module to encode controls of each sort. For instance, the variable state in the EndUser module has three enumerative values : EUreq, EUsat and EUdis to denote the different controls of the EU sort. Modules may also include other variables to describe more properties. Reaction rules are expressed as transitions in SMV code. For instance, we define in the module CloudService a transition that changes the state of this service from Sbpl to Spl. Moreover, a set of transitions in the SLAstages module are implemented to model the transitions between the SLA lifecycle stages. We note here that we have used modules with parameters in order to model the relationships between the different cloud entities. For instance, the two modules EndUser and CloudService use the variable state defined in the SLAagreement module (modeling the SLA) as a parameter to denote their relationships through the SLA.

Table 3: Correspondence between BRS concepts and SMV language

BRS	SMV concept	SMV code

BRS	main module	`MODULE main`
		 `...`

sort	module	`MODULE SortName(Parameters)`
		 `...`

control	variable	`VAR`
		 `VariableName: VariableValues;`
		 `...`

reaction rule	transition	`ASSIGN`
		 `init(VariableName):= InitialValues;`
		 `next(VariableName):= NextValues;`
		 `...`

6.2. *Properties specification*

According to Lamport [18], there are two types of properties to be verified which are: liveness and safety. In our work, these types state that *something good will eventually happen* or *something bad will never happen* during the lifecyle of the SLA, respectively. In other words, they describe desirable or undesirable behaviors of the different cloud entities during the SLA lifecycle. Moreover, properties can be also classified as general or specific properties. The former type does not depend directly on the agreed SLA terms as the latter type does. In addition, general properties must hold at any scenario but not specific properties. For instance, *always the monitoring process launches directly after the establishment stage*, and, *the SLA cannot be declared violated when the offered service is provisioning the expected level*, both of them represent general properties. An example of specific properties is: *once the number of occurred violations reaches the maximum allowed number of violations, the SLA will be directly terminated with penalties*. Notice that the maximum number of violations and the amount of penalties (defined in this property) depend on the signed SLA and may vary since each scenario consists of different stakeholders with varying features and specific requirements.

Regarding the specification of these properties, we have used temporal logic. The NuSMV model checker supports both LTL and CTL [21, 22] for properties expression. LTL specifications use different operators: X (next state), G (globally), F (finally) and U (until). In addition, LTL logic supports past-tense operators such as Y (previous state) and H (historically). In the CTL logic, the two path quantifiers : E (some path) and A (all paths) precede temporal operators. Some examples of CTL operators are: exists next state (EX), forall next state (AX), exists finally (EF), forall finally (AF), exists globally (EG), and forall globally (AG). A set of LTL and CTL properties that will be verified are given below. We note here that we use the dot notation as *Mod.Var* to encode the value of the variable *Var* in the module *Mod*. In addition, we use the symbol -> to denote the implication operator.

6.3. *NuSMV checking*

We employ the NuSMV model checker to verify the obtained SMV descriptions against the defined CTL and LTL properties. The NuSMV model checker returns TRUE if the SMV descriptions satisfy the defined properties. Otherwise, it returns FALSE and generates a counterexample (execution traces that violate the property). We create a file with the extension (.smv) that includes the SMV descriptions and the set of properties. Then, we will input this file to the NuSMV model checker that will process it using different commands. We use *check_ctlspec* and *check_ltlspec* to check the different properties expressed in CTL and LTL, respectively. Moreover, we use the *check_fsm* and *compute_reachable* commands to check the deadlock problem and compute reachability states, respectively.

7. Use case

As a proof of concept, we present a use case on which we will apply the proposed modeling and verification approach. First, we follow the BRS-based approach, as defined in sections 4 and 5, to model this use case. Then, we define the corresponding SMV descriptions of the obtained BRS models, as described in section 6, to execute and verify them using the NuSMV tool.

7.1. BRS modeling

Let assume that a company (called TrustUS) needs an IaaS provider to deploy its new SaaS application (called S1). This initial state is represented by bigraph BGstate0 in Figure 15. S1 will process confidential data, therefore, TrustUS looks for an IaaS provider that can guarantee a very high level of security regarding data confidentiality. Among several providers, offering different security policies and mechanisms [56], TrustUS negotiates and establishes an SLA with a provider called IaaS4All. The negotiation step is represented by bigraph BGstate1 (Figure 15). Reaction rule $R_{establish}$ is applied and the new bigraph BGstate2 represents the establishment step. The signed agreement is represented by the SLA-node in which it is specified the SLA objectives. Some of these objectives are depicted in table 4. In this case study, we consider two parameters that are availability and security. According to this table, the availability must be always greater than or equal 99%, and the level of security has to be guaranteed is high (we suppose that IaaS4All proposes 3 levels: low, medium, and high). In addition, the maximum number of violations is 3 for the availability and for the level of security is limited to 1, i.e., if any degradation is detected in security, the SLA is directly terminated with penalties. Moreover, table 4 shows penalties applied to each violation. The amount of penalty is calculated using the number of occurred violations (Numbrviol) and the penalty unit (PenUnity) defined during the negotiation (in our example PenUnity equals 20 euro). The provider is penalized with *PenUnity∗Numbrviol* euro if the availability is violated and it is greater than or equal to 98%, and with *2∗PenUnity∗Numbrviol* euro if the availability is less than 98%. Regarding the security parameter, the amount of penalties equals to *100∗PenUnity* euro. We note here that, in Figure 15 and 16, we have kept only relevant nodes and we have used sites to abstract others.

Table 4: Agreed SLA between TrustUS and IaaS4All

SLA parameters	Values	Penalties
Availability (Av)	$Av \geq 99$ %	0 *(SLA respected)*
	$98\% \leq Av < 99\%$	*PenUnity∗Numbrviol*
	$Av<98\%$	*2∗PenUnity∗Numbrviol*
	Numbrviol(Av)= 3	*Termination (SLA Failed)*
Security (Sec)	*Sec=high*	0 *(SLA respected)*
	Sec=medium or Sec=low	*Termination (SLA Failed)*
		and 100∗PenUnity

After the establishment of the SLA and during service provisioning, the IaaS4All provider has faced three successive problems:

- Problem 1: We suppose that the availability of resources offered by IaaS4All is dropped to 98%. This new situation (BGstate3 , i.e., IaaS4All has failed to offer the expected availability) is obtained after the application of reaction rule $R_{SerFail}$ on bigraph BGstate2 depicted in Figure 15. This problem leads to an SLA violation (application of $R_{violate}$) represented by bigraph BGstate4 (Figure 15). This latter bigraph shows also that the corresponding penalties are applied to IaaS4All (*20∗1*). After the detection of this failure, TrustUS fixes it (application of $R_{serAdjust}$) and the the availability comes back to 99% (BGstate5 shown in Figure 15).

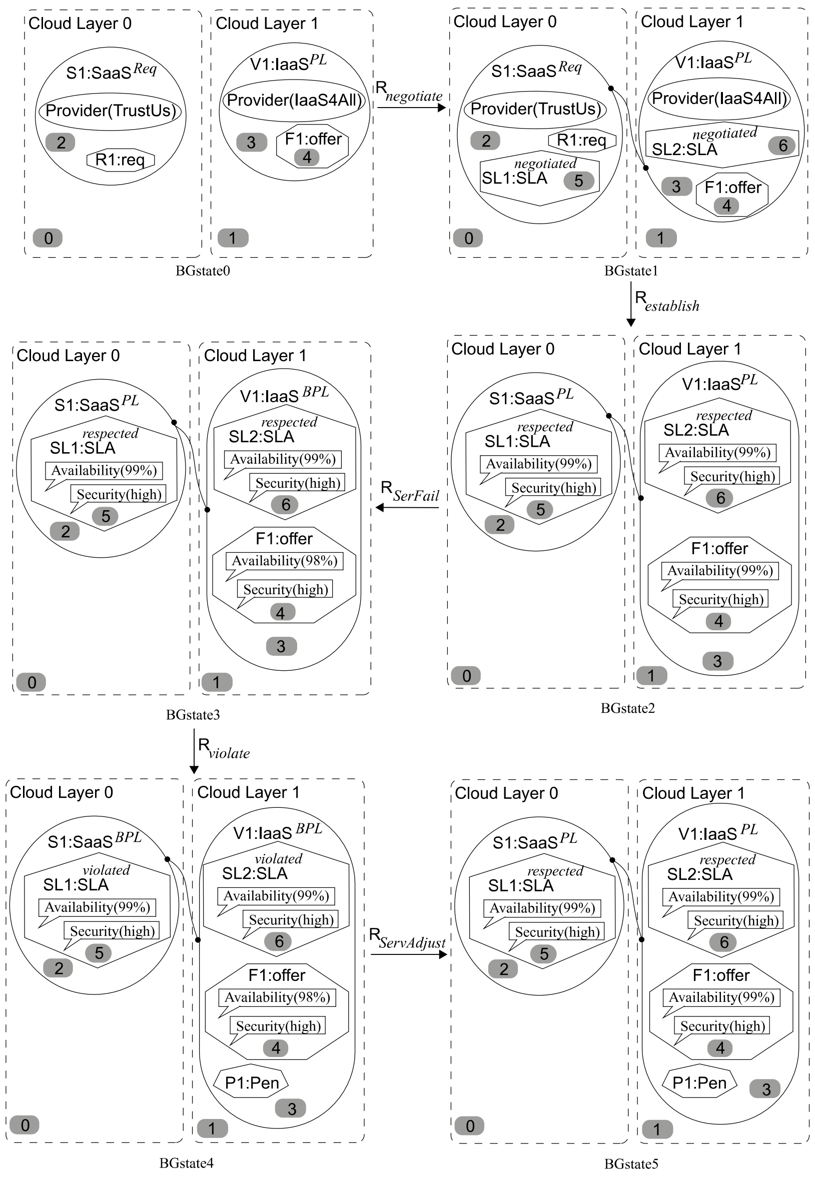


Figure 15: BRS modeling for the use case: problem 1

- Problem 2: Similar to the first problem, the availability is decreased again to 97% and an SLA violation is detected. Thereafter, TrustUS has successfully resolved this problem and the availability comes back to 99%. The only difference between the two problems is in the amount of penalties applied to IaaS4All (in this problem, the penalty amount will be *2∗20∗2*).This problem is illustrated through the following reaction rules sequence (see Figure 16):

$$BGstate5 \overset{\mathrm{R_{SerFail}}}{\longrightarrow} BGstate6 \overset{\mathrm{R_{violate}}}{\longrightarrow} BGstate7 \overset{\mathrm{R_{serAdjust}}}{\longrightarrow} BGstate8$$

- Problem 3: We suppose that TrustUS encounters a security problem and cannot deal with it. The SLA violation occurs, penalty amount is calculated (100∗20) and the agreement is directly terminated according to table 4 (if any security violation is detected, the SLA terminates directly with penalties). A sequence of three reaction rules is used to describe this problem as follows (see Figure 16):

641 $$BGstate8 \xrightarrow{\text{R}_{\text{SerFail}}} BGstate9 \xrightarrow{\text{R}_{\text{violate}}} BGstate10 \xrightarrow{\text{R}_{\text{ViolTerm}}} BGstate11$$

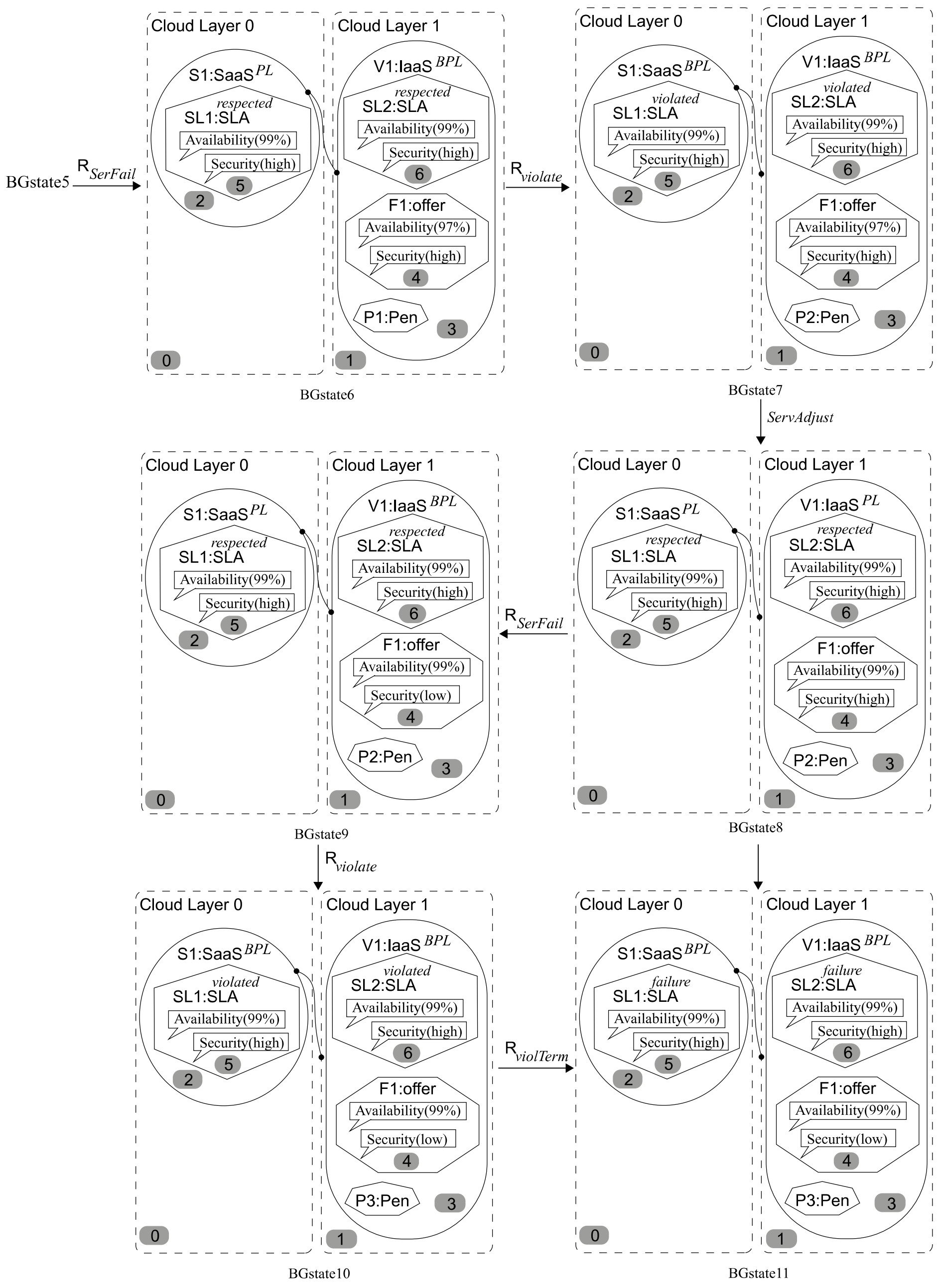


Figure 16: BRS modeling for the use case: problem 2 and problem 3

7.2. NuSMV-based verification and execution scenarios

We now follow the mapping rules defined in Section 6.1 to translate the obtained BRS models to SMV descriptions. We create a file called CaseStudy.smv that contains five modules: main, SaaS, IaaS, SLAagreement, and SLAstages. In the main module, we instantiate the other four modules. In these latter modules, we define a variable state that describes the state of the SaaS service, IaaS service, SLA, and the stage of the SLA lifecycle, respectively.

Moreover, we define other variables to describe properties and parameters of services and SLAs. For instance, both CloudService and SLAagreement modules have a variable called availability that denotes the current offered availability and the expected availability, respectively. Furthermore, two constants maxviolav and maxviols defined in module SLAagreement denote the maximum number of violations for availability and security, respectively. Modules include also a set of transitions describing how states, properties or parameters change during the SLA lifecycle. Figure 17 shows an excerpt of the obtained SMV program. After the creation of this file, we use the interactive simulation mode to generate the trace corresponding to our use case. Figure 18 shows the resulting simulation trace that contains 12 states. From this figure, we notice that IaaS4All has violated the SLA three times (V1.numbviolav=2 and V1.numbviols=1) and it has fixed it twice (Stage.state=ViolFixing has occurred twice in the trace). The NuSMV tool reports that the SLA has terminated with penalties and the IaaS4All is penalized with a total of 2100 euro.

```
MODULE main
   VAR
      S1:SaaS(SLA...);
      V1: IaaS(SLA...);
      SLA:SLAagreement(Stage...);
      ...
CTLSPEC NAME LP2 :=  AG ((V1.availability < 99 | V1.sec...
...
-----------------------------------------------
MODULE SLAagreement(Stage)
   VAR
      state: {neg,resp,viol,fail,succ};
      availability: {99};  -- expected Availability
      security:{high};     -- expected security
      ...
   ASSIGN
      init(state) := neg;
      next(state) := case
      (Stage = Negotiation) &  next(Stage)=Negotiation : neg;
      ...
-----------------------------------------------
MODULE SaaS(Agreement)
   VAR
      state:{SaaSreq,SaaSpl,SaaSbpl};
      ...
-----------------------------------------------
MODULE IaaS(Agreement...)
   VAR
      security:{low,medium,high};
      ...
 -------------------------------------------------
MODULE SLAstages(...)
   VAR
      state:{Negotiation,Establishment,...Ptermination};
      ...
```

Figure 17: NuSMV descriptions

7.3. *Verification results*

We now verify the following set of properties using the NuSMV model checker. We specify these properties informally and in NuSMV as follows:

- *Liveness properties*:

  During the monitoring stage, any *degradation* in the *service level* will be *detected*.

  LP1: *AG ((Stage.state = Monitoring & V1.state = IaaSbpl) -> AX Stage.state = ViolDetection).*

```
NuSMV > show_traces -v
    <!-- ################### Trace number: 1 ################### -->
Trace Description: Simulation Trace
Trace Type: Simulation
 - > State: 1.1 < -             - > State: 1.4 < -             - > State: 1.7  < -            - > State: 1.10 < -
    S1.state = SaaSreq             S1.state = SaaSpl              S1.state =  SaaSpl             S1.state = SaaSpl
    V1.state= IaaSreq              V1.state= IaaSbpl              V1.state =  IaaSbpl            V1.state= IaaSbpl
    V1.availability   =  99        V1.availability  =  98         V1.availability = 97           V1.availability  =  99
    V1 .numbviolav = 0             V1 .numbviolav = 0             V1.numbviolav = 1              V1.numbviolav = 2
    V1.numbviols  =  0             V1.numbviols  =  0             V1.numbviols  =  0             V1.numbviols  =  0
    V1.security =  high            V1.security =  high            V1.security  =  high           V1.security = low
    V1.PenAmount = 0               V1.PenAmount = 0               V1.PenAmount=20                V1.PenAmount = 100
    SLA.state= neg                 SLA.state =  resp              SLA.state =  resp              SLA .state =  resp
    Stage.state  = Negotiation     Stage.state = Monitoring       Stage.state = Monitoring       Stage.state = Monitoring
    S1.provider = TrustUs          S1.provider  =  TrustUs        S1.provider  =  TrustUs        S1.provider  =  TrustUs
    V1.provider= IaaS4All          V1.provider  =  IaaS4All       V1.provider  =  IaaS4All       V1.provider  =  IaaS4All
    SLA.availability = 99          SLA.availability = 99          SLA.availability = 99          SLA.availability = 99
    SLA.security = high            SLA.security = high            SLA.security = high            SLA.security = high
    SLA.maxviolav=3                SLA.maxviolav = 3              SLA.maxviolav = 3              SLA.maxviolav = 3
    SLA.maxviols = 1               SLA .maxviols =  1             SLA.maxviols = 1               SLA.maxviols = 1
    SLA.PenUnit =  20              SLA.PenUnit = 20               SLA.PenUnit =  20              SLA.PenUnit =  20
 - > State: 1.2 < -             - > State: 1.5 < -             - > State: 1.8 < -             - >State: 1.11 < -
    S1.state = SaaSreq             S1.state  = SaaSbpl            S1.state= SaaSbpl              S1.state  = SaaSbpl
    V1.state= IaaSpl               V1.state= IaaSbpl              V1.state= IaaSbpl              V1.state= IaaSbpl
    V1.availability   =  99        V1. availability =  98         V1.availability  =  97         V1. availability =  99
    V1 .numbviolav = 0             V1.numbviolav = 1              V1.numbviolav  = 2             V1.numbviolav =2
    V1.numbviols  =  0             V1.numbviols  = 0              V1.numbviols   =  0            V1.numbviols  =  1
    V1.security =  high            V1.security = high             V1.security=high               V1.security= low
    V1.PenAmount = 0               V1.PenAmount=20                V1.PenAmount=100               V1.PenAmount =2100
    SLA.state= neg                 SLA.state= viol                SLA.state = viol               SLA.state = viol
    Stage.state  = Negotiation     Stage.state =  ViolDetection   Stage.state =  ViolDetection   Stage.state =  ViolDetection
    S1.provider = TrustUs          S1.provider  =  TrustUs        S1.provider  =  TrustUs        S1.provider  =  TrustUs
    V1.provider= IaaS4All          V1.provider  =  IaaS4All       V1.provider  =  IaaS4All       V1.provider  =  IaaS4All
    SLA.availability = 99          SLA.availability = 99          SLA.availability = 99          SLA.availability = 99
    SLA.security = high            SLA.security = high            SLA.security = high            SLA.security = high
    SLA.maxviolav=3                SLA.maxviolav=3                SLA.maxviolav=3                SLA.maxviolav=3
    SLA.maxviols = 1               SLA.maxviols = 1               SLA.maxviols = 1               SLA.maxviols = 1
    SLA.PenUnit =  20              SLA.PenUnit = 20               SLA.PenUnit =  20              SLA.PenUnit =  20
 - > State: 1.3 < -             - > State: 1.6 < -             - > State: 1.9 < -             - > State: 1.12< -
    S1.state = SaaSpl              S1.state = SaaSpl              S1.state = SaaSpl              S1.state= SaaSbpl
    V1.state= IaaSpl               V1.state= IaaSpl               V1.state= IaaSpl               V1.state= IaaSbpl
    V1.availability   =  99        V1. availability =  99         V1.availability  =  99         V1. availability =  99
    V1 .numbviolav = 0             V1.numbviolav = 1              V1.numbviolav  = 2             V1.numbviolav =2
    V1.numbviols  =  0             V1.numbviols  = 0              V1.numbviols   =  0            V1.numbviols  =  1
    V1.security =  high            V1.security =  high            V1.security=high               V1.security= low
    V1.PenAmount = 0               V1.PenAmount = 20              V1.PenAmount=100               V1.PenAmount  = 2100
    SLA .state =  resp             SLA .state =  resp             SLA .state =  resp             SLA.state= fail
    Stage .state =  Establishment  Stage.state= ViolFixing        Stage.state=ViolFixing         Stage.state =   Ptermination
    S1.provider  =  TrustUs        S1.provider  =  TrustUs        S1.provider=TrustUs            S1.provider = TrustUs
    V1.provider  =  IaaS4All       V1.provider  =  IaaS4All       V1.provider=IaaS4All           V1.provider = IaaS4All
    SLA.availability = 99          SLA.availability = 99          SLA.availability = 99          SLA.availability = 99
    SLA.security = high            SLA.security = high            SLA.security=high              SLA.security = high
    SLA.maxviolav=3                SLA.maxviolav = 3              SLA.maxviolav = 3              SLA.maxviolav=3
    SLA.maxviols = 1               SLA.maxviols = 1               SLA.maxviols = 1               SLA.maxviols = 1
    SLA.PenUnit =  20              SLA.PenUnit = 20               SLA.PenUnit =  20              SLA.PenUnit =  20
```

Figure 18: The corresponding NuSMV trace for the use case

A service is *not able* to offer the *expected level* if *at least one* of the *SLA objectives* is *not met*.

LP2: *AG ((V1.availability < 99 | V1.security != high) -> V1.state = IaaSbpl)*

Always the *monitoring* process *launches directly after* the *establishment* stage.

LP3: *AG (Stage.state = Establishment -> AX Stage.state = Monitoring)*

During the *provisioning* of a service, *its level* can go through *different states* (in our work we have proposed *two levels* either *able* to offer the promised level or *not*).

LP4: *AG (Stage.state = Monitoring -> (V1.state = IaaSbpl | V1.state = IaaSpl))*

Once an *SLA violation* is *detected* an *amount of penalties* will *be imposed* on the provider of this service.

LP5: *AG (SLA.state = viol -> V1.PenAmount > 0)*

Once the number of *occurred violations* reaches the *maximum allowed number* of violations, the *SLA* will *be directly terminated* with *penalties*.

LP6: *AG ((V1.numbviolav = SLA.maxviolav | V1.numbviols = SLA. maxviols) -> AX Stage.state = Ptermination)*

The SLA is declared *successful* if the service level *has been always the expected one*, or if a *degradation* in the service level has been *detected*, the provider of this service *has fixed it*.

LP7: *G (SLA.state = succ -> ((H V1.state = IaaSbpl & X Stage.state = ViolFixing) | G V1.state = IaaSpl))*

- *Safety properties*:

  SLA establishment cannot be done if the required service *has some requirements* or it *is not able* to offer the required quality.

  SP1: *AG (Stage.state = Establishment -> !(V1.state = IaaSbpl | V1.state = IaaSreq))*

  The number of *occurred violations* must *not exceed* the *maximum allowed number* of violations.

  SP2: *AG !(V1.numbviolav > SLA.maxviolav | V1.numbviols > SLA. maxviols)*

  The *SLA* cannot be declared *violated* when the offered service is *provisioning the expected level*.

  SP3: *AG !(SLA.state = viol & V1.state = IaaSpl)*

  Once the *security* is *violated* there is *no way to fix it*.

  SP4: *AG !(V1.security = low & EX Stage.state = ViolFixing)*

  When the level of *security is respected*, the *SLA* will be *never violated*.

  SP5: *AG !(V1.security = high & Stage.state = ViolDetection)*

It is worthy to note that the specification of these properties may vary from this scenario to a new one since the agreed SLA may be different. For instance, in this scenario the two parties agree on the level of availability to be greater than or equal 99%. This QoS parameter may be different depending on the requirements and offerings of the SLA parties (customers and providers). Therefore, a customization of these properties is necessary to be addressed for different scenarios.

The results of the verification are shown in Figure 19. We notice that our model verifies all the properties except property SP5. For this property, NuSMV returns false and gives back a counterexample in which the SLA was violated due to the violation of the availability parameter (The SLA will never be violated if all its objectives are respected, i.e., both security and availability). We have introduced this property to show a NuSMV counterexample generation. Moreover, Figure 20 shows that there is no deadlock problem and all the states are reachable.

8. Related work

Service description, cloud systems modeling and verification, and SLA management have been investigated in different research works. This section discusses related works in three categories: (i) general approaches not using bigraphs ([57, 58, 59, 60, 61, 62, 63]), (ii) SLA management in the cloud ([64, 65, 66, 49, 9, 67, 68, 69, 70]), and (iii) BRS-based approaches ([30, 71, 51, 52, 72, 73, 31]). Note here that although the intensive work that has been done in this domain, to the best of our knowledge, these research works do not consider all the challenges identified in section 1 simultaneously. In addition, they do not consider the different states of cloud entities and their dynamic change during the SLA lifecyle.

```
NuSMV > check_ltlspec
- - specification G (SLA.state = succ -> (( H V1.state = IaaSbpl  & X Stage.state = ViolFixing) | G V1.state = IaaSpl)) is true
 NuSMV > check_ctlspec
- - specification AG ((Stage.state = Monitoring & V1.state = IaaSbpl) -> AX Stage.state = ViolDetection) is true
- - specification AG ((V1.availability < 99 | V1. security != high) -> V1.state = IaaSbpl)  is true
- - specification AG (Stage.state = Establishment -> AX Stage.state = Monitoring) is true
- - specification AG (Stage.state = Monitoring -> (V1.state = IaaSbpl  | V1.state = IaaSpl))  is true
- - specification AG ((V1.numbviolav = SLA. maxviolav | V1.numbviols = SLA. maxviols) -> AX Stage.state = Ptermination) is true
- - specification AG (Stage.state = Establishment -> !(V1.state = IaaSbpl  | V1.state = IaaSreq)) is true
- - specification AG !(V1.numbviolav > SLA. maxviolav | V1.numbviols > SLA. maxviols) is true
- - specification AG !(SLA.state = viol & V1.state = IaaSpl) is true
- - specification AG !(V1. security = low & EX Stage.state = ViolFixing) is true
- - specification AG !(V1. security = high & Stage.state = ViolDetection) is false
- - as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
   -> State: 1.1 <-             -> State: 1.2 <-              -> State: 1.3 <-            -> State: 1.4 <-
    S1.state = SaaSreq           S1.state = SaaSpl             V1.state = IaaSbpl              S1.state = SaaSbpl
    V1.state = IaaSpl            SLA.state = resp              V1.availability = 98            V1.numbviolav = 1
    V1.availability= 99          Stage.state = Establishment   Stage.state = Monitoring        V1.PenAmount = 20
    V1.numbviolav = 0                                                                          SLA.state = viol
    V1.numbviols = 0                                                                           Stage.state = ViolDetection
    V1security = high
    V1.PenAmount = 0
    SLA.state = neg
    Stage.state = Negotiation
    S1.provider = TrustUs
    V1.provider = IaaS4All
    SLA.availability = 99
    SLA. security = high
    SLA. maxviolav = 3
    SLA. maxviols = 1
    SLA.PenUnit = 20

- - specification AG (Stage.state = ViolDetection -> V1.PenAmount  > 0) is true
```

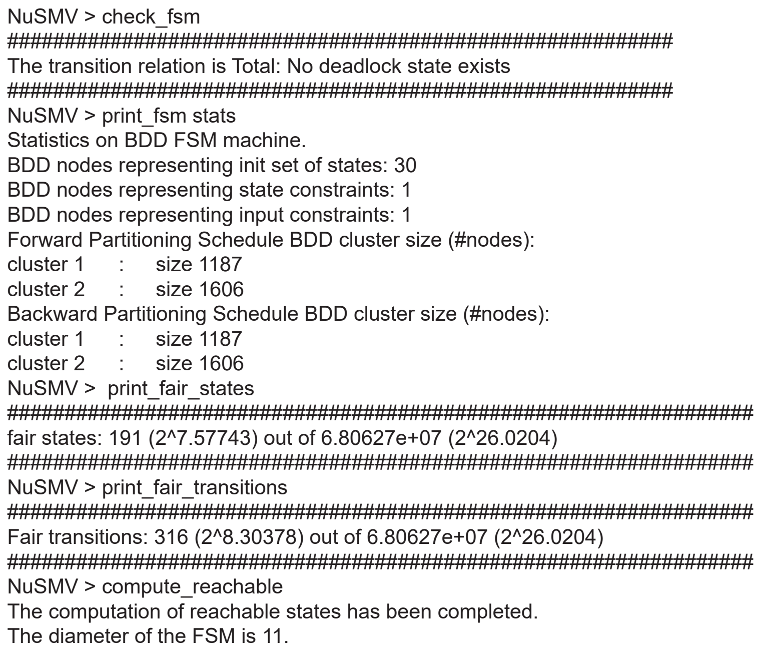
Figure 19: Results of the formal verification

8.1. General approaches

Nguyen et al. [57] proposed a blueprinting approach for the description of cloud service offerings. The defined blueprint, as a uniform description format, assists developers to select, customize, and compose the different types of cloud services (i.e., IaaS, PaaS, and SaaS). Moreover, the authors detailed a six stages life cycle of blueprints starting from their specification to their deployment to the cloud. In addition, they defined a case study and developed a prototype based on the blueprint concept in the EC's 4caaSt FP7 project.

Authors in [58] proposed a generic autonomic manager (AM) that deals with the management of cloud services. The AM allows the specification, (re)configuration, and monitoring of any XaaS layer. In addition to the autonomic computing, the CoMe4ACloud project applied the Model-driven engineering (MDE) principals and used the constraint programming. The authors defined two complementary metamodels. The first one is used for the specification (at design time) of cloud systems' topologies. The second one is used to represent the actual running systems' configurations. To determine the optimal configuration of the modeled systems with the imposed constraints, the proposed AM relies on a constraint solver. The authors applied the proposed approach to a real Openstack IaaS infrastructure.

Amato and Moscato [59] exploited workflow patterns and cloud to deal with the analysis and verification of cloud service composition problem. They proposed a methodology that tackles this problem and facilitates the description of services and their relationships with other resources required by the composition. Model transformation techniques were used and the Pattern Grounding Language (PGL) was defined in order to translate patterns declarations in workflow-based descriptions. The transformation creates a graph in the Operational Flow Language (OFL) that enables the analysis and evaluation of QoS properties of composite cloud services.



```
NuSMV > check_fsm
############################################################
The transition relation is Total: No deadlock state exists
############################################################
NuSMV > print_fsm stats
Statistics on BDD FSM machine.
BDD nodes representing init set of states: 30
BDD nodes representing state constraints: 1
BDD nodes representing input constraints: 1
Forward Partitioning Schedule BDD cluster size (#nodes):
cluster 1     :     size 1187
cluster 2     :     size 1606
Backward Partitioning Schedule BDD cluster size (#nodes):
cluster 1     :     size 1187
cluster 2     :     size 1606
NuSMV >  print_fair_states
###################################################################
fair states: 191 (2^7.57743) out of 6.80627e+07 (2^26.0204)
###################################################################
NuSMV > print_fair_transitions
###################################################################
Fair transitions: 316 (2^8.30378) out of 6.80627e+07 (2^26.0204)
###################################################################
NuSMV > compute_reachable
The computation of reachable states has been completed.
The diameter of the FSM is 11.
```

Figure 20: Deadlock and reachability properties

Le Sun et al. [60] summarized the specific requirements needed for cloud service description languages/models. Then, based on these Cloud-specific requirements, they extended the Unified Service Description Language (USDL) to model cloud services. The proposed cloud service description model (CSDM) defines a new module (called transaction) and updates the nine modules defined in USDL (adding new classes and attributes in these modules). Moreover, CSDM permits the description of cloud services and this description supports different aspects such as the delivery and deployment models, the participating roles, and the measurement of service qualities.

In the same direction, Ghazouani and Slimani [61] adopted and extended USDL to be suitable for the description of cloud services. The authors focused on the semantic aspect not covered by USDL. Therefore, the authors defined a WSMO ontology to semantically describe services in the cloud. The authors defined nine concepts in the WSMO ontology, each concept represents a module in USDL. In addition, classes, properties and data from USDL are represented, respectively, by sub-concept, attributes and instances in the defined ontology. The authors used the proposed ontology to describe three services from different layers (SaaS, PaaS and IaaS).

Authors in [62] proposed a formal approach to model, verify, and provision elastic component-based applications. The proposed Event-B based approach starts by modeling the component artifacts. Then, it uses the refinement concept defined in Event-B to introduce the elasticity mechanisms to the previous component artifacts. In fact, the authors proposed two Event-B events, namely duplicate and consolidate, to scale up or scale down applications. The PROB animator/model checker was used to validate the proposed model, and implemented their approach as an Eclipse plug-in.

Jlassi et al. [63] proposed an Event-B-based modeling approach for the management of the Free and Open Source Software (FOSS) applications, their composition, and their horizontal elasticity. The authors dealt with the cloud resource allocation problem for this kind of applications. In fact, they applied different refinements in order to model the FOSS' required cloud resources (computing, networking and storage resources), their allocation process, and also the vertical elasticity and shareability properties of these resources. The authors verified the proposed model using the PROB model checker and the proof correctness.

8.2. Cloud SLA management

Cloud SLA management has been the focus of several approaches [7]. García and Blanquer [64] proposed a generic SLA-based methodology for describing cloud services. This methodology generates different SLA fragments that include static and dynamic features of services. In order to offer complete SLA templates, García and Blanquer introduced an algorithm that composes, on-the-fly, the SLA-based representations of cloud services (SLA fragments). The authors described optimization techniques used to improve the introduced algorithm's performance. The proposed methodology was developed in the Cloudcompaas framework project [53].

Authors in [65] proposed a modeling approach based on the model-driven architecture principles. This approach defines three metamodels: the CloudCustomer, the CloudProvider, and the CWSLA metamodels. They are based on the standard Business Process Modeling Notation (BPMN), the UML QoS framework, and the Web Service Level Agreement (WSLA). The three proposed languages enable the description of the functional and non-functional requirements of composite SaaS applications, the required IT resources (IaaS) to satisfy these requirements, and the SLA established between providers and their customers, respectively. In addition, the authors detailed the generation process of the SLA document using the Atlas Transformation Language (ATL).

Uriarte et al. [66] dealt with the dynamic SLAs in the domain of cloud computing. They defined a new SLA language, called SLAC, that takes into consideration the specific features of cloud services and the dynamic aspects of cloud environments. In fact, in order to reduce the renegotiation of SLA terms, this language enables the definition of agreements with new features such as the expression of changing requirements of the involved parties and the dynamic service levels. An open source framework supporting the SLAC language was developed and used in the different SLA lifecycle.

Authors in [49] defined a new model called SLA aware Service (SLAaaS). This model was integrated with the different levels of the cloud architecture (i.e., end-users, IaaS, PaaS and SaaS). Cloud Service Level Agreement (CSLA), a novel QoS and SLA language was proposed for any cloud service. The CSLA language allows the management of cloud elasticity by introducing new properties such as QoS uncertainty, functionality degradation and advanced penalty model. Moreover, the authors adopted the control theory to manage and ensure SLA in the cloud. They illustrated, through different use cases, how to use the SLAaaS model to establish cross-layer SLAs.

Lu et al. [9] adopted the actor model for the management of the SLA lifecycle in cloud environments. They defined four types of actors: customer, SLA manager, SLA and QoS actors. Finite state machines are used to model these actors. A framework consists of different layers of actors related in a parent-child relationship was proposed. This hierarchy between actors permits escalation of the failure upwards until its resolution. The authors presented a fault tolerance scenario and illustrated failure handling in the different framework's layers. The proposed approach was simulated using CloudSim framework.

In addition, to achieve the expected SLA requirements, the elasticity and scalability of cloud resources has been addressed in several works [67, 68, 69, 70]. Authors in [67] tackled the problem of scaling fog/edge computing nodes to fulfill the dynamic IoT workload and avoid any SLA violation. They proposed a queuing theory-based analytical model that, depending on the IoT workload, determines to scale down or up fog computing resources in order to meet the agreed-on SLA and the expected QoS parameters. The analytical model involves three queuing sub-systems, namely edge computing, cloud gateway, and cloud data center models. Different mathematical formulas are obtained and used for performance analysis considering different key parameters such as the system loss rate, the CPU utilization, the system throughput, and the system response time. Authors validated the proposed models using discrete event simulations based on the Java Modeling Tool.

Al-Haidari et al. [68] studied the autoscaling of cloud resources regarding two factors: CPU thresholds and the scaling size. They conducted several simulations experiments to analyze the impact of configuring these factors on the performance of cloud services. The performance is evaluated in terms of several metrics such as resource utilization, response time, and the cost of the allocated instances. To minimize the cost and offer a low response time, thus ensuring SLA objectives, authors proposed a method that determines the optimal value of the upper CPU thresholds and the scaling size.

Calyam et al. [69] tackled the problem of designing and verifying resource allocation schemes for virtual desktop clouds (VDCs). They developed a tool called *VDC-Analyst* allowing the estimation of two quality of experiments (QoE) metrics *Service Response Time* and *Net Utility*. The authors developed the *Multi-stage Queuing Model* and the *Cost-Aware Utility-Maximal Resource Allocation Algorithm* and integrated them into the *VDC-Analyst* tool to be used in the QoE metrics estimation. In addition, the VDC-Analyst can be used for the simulation and emulation of VDC systems in two different modes *run simulation* and *run experiment*.

El Kafhali and Salah [70] adopted the queuing theory to propose a stochastic model for analyzing performance in cloud data centers. The performance evaluation considers several QoS metrics including drop rate, response time, and CPU utilization. The proposed model allows to estimate the proper number of VM instances required to meet the expected QoS. Authors used Java Modeling Tool simulator for the validation of this model and illustrated its usefulness through different scenarios.

8.3. BRS_based approaches

Several research studies have adopted the bigraph theory in the cloud computing area [30, 51, 52, 71, 72]. These studies have addressed cloud systems modeling focusing on the elasticity mechanisms. However, in addition to the cloud systems modeling, we have addressed more and important aspects that have not been the focus of them: the SLA and its lifecycle. Furthermore, different from these studies, ours considers the dynamic change of cloud entities states, and the different specific features of cloud services such as delivery models, and QoS. Regarding the verification step, we check more other important properties that are safety, liveness, and reachability. Compared to our earlier works [31, 73], this research proposes a more general definition that can represent any type of cloud services or resources, specify different QoS and properties, and defines several types of interactions. Also, it proposes a four-stages SLA lifecycle and uses different rules to model it. Moreover, it proceeds with the verification step by using the NuSMV model checker and checking different properties.

Benzadri et al. [30] presented the first attempt to adopt bigraphs for modeling cloud computing systems. The proposed model consists of two bigraphs that describe the front-end part (Cloud Customers Bigraph CCB) and back-end part (Cloud Services Bigraph CSB) of cloud systems. While the former describes cloud customers, the latter represents the three service delivery models. The authors defined three different ports assigned to cloud services nodes to represent the different deployment models. The dynamic structure of the cloud model was specified using a reaction rules-set that allows the migration, allocation, and des-allocation of services.

The last work has been extended in [71]. Regarding the static structure, the authors defined a third bigraph for the virtualisation layer (data center and servers) (CVB). Regarding the dynamic evolution, Benzadri et al. introduced different rules according to the three bigraphs such as user changing location, service migration, and server redundancy rules. In order to design and analysis cloud architectures, the authors presented a mapping of the BRS-based model to the Maude language. Three cloud-related properties was introduced (concurrency, mobility, and auto scaling), specified in LTL, and checked with the Maude model checker.

The authors in [51] dealt with the elasticity of cloud systems. They proposed a BRS-based model that consists of two bigraphs encoding the front-end and the back-end of cloud systems. A third bigraph was used to model the elasticity controller of these systems. This latter adopted the IBM's [74] Monitor, Analyze, Plan, and Execute (MAPE) autonomic control loop. Furthermore, two categories of bigraphical reaction rules are introduced. The first one is used to model clients/applications interactions. The second one expresses the elementary elastic methods (vertical scale, horizontal scale, and migration). These rules are then used to simulate four scenarios in which the different elasticity methods are applied.

The authors in [52] delimited their research to the back-end and the elasticity controller parts presented in [51]. The BRS-based approach in this work introduced two cross-layer elasticity strategies (provisioning and de-provisioning host environment strategies) and focused on the horizontal scale of resources in the cloud. The authors proceeded with the verification step to check the correctness of the elastic behaviors of cloud systems, and therefore they proposed an executable solution based on the encoding of the bigraphical specification into Maude language. They also presented a quantitative analysis of the proposed elasticity strategies based on a queuing approach and through different simulated scenarios.

Moudjari et al. [72] dealt with the management of cloud elasticity at two levels: the intra-cloud and the inter-cloud. They proposed a *Multi Agent System for Cloud of Clouds Elasticity Management (MAS-C2EM)*. The clouds cooperation process, offered by MAS-C2EM, uses a fuzzy dominance approach to ensure the management of inter-cloud elasticity. The BRS formalism was used to specify the proposed MAS-C2M. The obtained bigraphical MAS-C2M (BigMAS-C2EM) defines five classes of reaction rules to describe the clouds cooperation process. In order to formally analyze some properties of BigMAS-C2EM, the authors used the model checker BigMC.

The present work builds upon our previous works [31, 73]. In the first work [31], we tackled the service composition problem in the cloud. We defined a cloud service composition signature considering cloud architectures composed of three layers: end-users, SaaS and UaaS. We introduced different reaction rules to model the dynamic behavior of cloud actors and services involving in such composition. We presented a case study to illustrate both the direct and indirect dependencies among the different cloud entities. In the best of our knowledge, it was the first attempt to consider SLA in BRS-based approaches.

In the second work [73], we used BRS to model a general provider-customer interaction in the cloud. We defined different states for customers, services and also SLAs. In addition, we proposed several reaction rules to model the changing states of cloud entities during the SLA lifecycle. We applied the introduced models on a four-layers cloud architecture. A case study with different scenarios was presented. Even if this work was our first attempt to model the SLA lifecycle, only two stages were considered.

Table 5 summarize the BRS-based related approaches and compares them to our work regarding several service and SLA concepts:

- SM : indicates whether the proposed solution supports all service delivery models (XaaS) or only some cloud services types (SaaS, PaaS, or IaaS). From this table, we notice that all the above BRS-based approaches support XaaS services (except [72]). However the present work benefits from the sorting mechanism and proposes a generic definition for cloud services, and therefore allowing not only modeling the well-known types but any other cloud services or resources.
- DM: refers whether the different deployment models are taken into consideration (✓) or not (empty cell). Authors in [30, 71] proposed to use ports to describe the different models. In our work, we have used parametric controls to model them.
- QoS: shows the capacity of the proposed solutions to model QoS. Authors in [72] cited some examples of QoS in the proposed MAS-based model (MAS-C2EM) but any of them was considered in the corresponding BRS-based model (BigMAS-C2EM). Authors in [51] abstracted resources properties (such as CPU, memory and storage) as simple controls. These latter are used only to show these properties but not to carry data and show their real values. However, in our work we have introduced different parametric controls to model QoS and the different properties of services.
- SLA: determines whether the SLA was considered (✓) or not (empty cell). Only two works [73, 31] considered cloud SLAs. However, these works proposed only two different states of the SLA. In the present paper, we have proposed different controls to model these agreements and describe their different states (five states).
- SLAL: indicates whether the BRS-based approaches considered the whole SLA lifecycle (✓), only some of its stages (~), or not at all (empty cell). Our work [73] was the first attempt to model the SLA lifecycle using BRS. However, this study [73] did not cover all the SLA lifecycle and only two stages were considered (establishment and monitoring). None of the other works models SLA lifecycle. In the present paper, the authors have proposed different bigraphical reaction rules modeling the different phases of the SLA lifecycle. Compared to the above BRS-based approaches, almost of these works [51, 52, 72] focused on the problem of elasticity in the cloud and proposed different reaction rules to model elasticity strategies. In our work, these proposed rules can be adopted as appropriate solutions in the "service adjustment" scenario during the monitoring phase.
- INT: we use the symbol ✓ to indicate that the proposed approach supports different types of interactions (such as customer-provider interaction, vertical and horizontal compositions) and allows the definition of other relationships. The symbol ~ is used to indicate that the proposed solution does not consider different relationships and does not discuss its extension to support other types of interactions. Almost of the proposed BRS-based approaches supported the interaction between customers and cloud systems, and considered the vertical service composition in terms of deployment relationships. However, we do not find any work that discusses the possibility of extension to support other types. In our work we tackled this problem by using ports and the link sorting mechanism. Different ports and link sorts are defined to describe customer-provider interaction, vertical and horizontal compositions.

The BRS-based related approaches are summarized and compared to our work regarding verification concepts (see Table 6):

- VER: in table 6, we keep only works that proceeded with the verification step (✓). Authors in [71, 51, 52, 72] discussed the verification of the proposed models using two different model checkers. No verification was provided in [31, 73, 30].

- LOG: refers to the temporal logic formulas used in the verification process. Almost of these studies specified their properties in the form of LTL formulas. In our work, we have used the two formulats LTL and CTL.
- VT: indicates which tools were used in the verification process. Maude LTL model checker was used in [51, 52, 71]. The BigMC model checker was used in [72]. In this work, we have used NuSMV model checker.
- VP: specifies the different properties that are checked. The author in [71] defined three cloud computing-related properties (concurrency (C), mobility (M), and auto scaling (AS)) and check them. Authors in [52] defined different elasticity strategies (ES) and verify their correctness.The reachability property (R) was evaluated in [72]. In our work, we have analyzed different properties including: liveness (L), safety (S), and reachability (R).

Approach	SM	DM	QoS	SLA	SLAL	Int
Benzadri et al. [30]	XaaS	✓				~
Benzadri et al. [71]	XaaS	✓				~
Sahli et al. [51]	XaaS		~			~
Khebbeb et al. [52]	XaaS					~
Moudjari et al. [72]	SaaS/IaaS		~			~
Kamel et al. [31]	XaaS			✓		✓
Kamel et al. [73]	XaaS			✓	~	✓
Our work	XaaS	✓	✓	✓	✓	✓

Table 5: Comparison of BRS-based approaches regarding service and SLA-related concepts

Approach	VER	LOG	VT	VP
Benzadri et al. [71]	✓	LTL	Maude	C, M and AS
Sahli et al. [51]	✓	LTL	Maude	
Khebbeb et al. [52]	✓	LTL	Maude	ES
Moudjari et al. [72]	✓	CTL alike	BigMC	R
Our work	✓	LTL/CTL	NuSMV	L, S, and R

Table 6: Comparison of BRS-based approaches regarding verification-related concepts

9. Conclusion

The present work proposes a formal approach that adopts the BRS formalism and the NuSMV model checker for modeling and verifying cloud systems. In the modeling phase of this approach, We extend bigraphs to cope with the structure, the dynamic behavior, and also the distinctive aspects of cloud systems. We illustrate that BRS is a well suited and powerful formalism for modeling these systems. Regarding the static structure of cloud systems, the proposed model uses BRS sorts and controls to describe different cloud entities such as services (considering their specific features), SLAs, end-users, and providers. Regarding the dynamic behavior, the approach focuses on the SLA lifecycle. More precisely, it introduces different bigraphical reaction rules that details the interaction and changing states of the different cloud entities during a defined four-stages SLA lifecycle. To the best of our knowledge, this is the first work to address the SLA and its lifecycle management using BRS. In the second phase of this approach, we verify the correctness of interaction behaviors of the different cloud entities. This phase consists of three steps. First, the obtained BRS models are translated to SMV descriptions. Then, we define a set of liveness and safety properties representing the interaction behaviors and we use LTL and CTL formulas to specify them. Finally, the NuSMV model checker verifies the resulting SMV descriptions against the defined properties. The feasibility of our approach is illustrated using a case study.

Although promising, the proposed approach can be enhanced regarding different limitations that we will address in future researches. The number and complexity of aspects addressed in both phases (modeling and verification) depend on the type and complexity of the properties we want to check. The use of sites allows us to focus on our necessities alleviating the model complexity. However, the increasing number of cloud platforms and their heterogeneity (different service models, QoS, properties, SLAs, etc.) lead to an increasing number of aspects to be considered during these phases. Therefore, we intend to hide this complexity by developing different tools offering Graphical User Interface (GUI) to simplify, for instance, the specification of requirements and offerings (including services, their properties and also the different scalability and elasticity rules), and also the generation of different reaction rules used during the monitoring stage of SLA. Moreover, the BRS-SMV mapping is currently done manually. Automating this task is very important since such task of mapping and transformation is error prone and time consuming.

We believe that, relying on the matching techniques developed in this theory, the proposed BRS models, introduced in this work to describe services and their properties, would play an important role in the area of services discovery and composition. In addition, adopting the models@runtime approach [75] would be also helpful especially for the management of SLA and its lifecycle. Finally, we are working on the transformation of the proposed BRS-based descriptions to different description standards including TOSCA [41] and OCCI [42] with the aim that these standards were accepted by the industry to enable features such as the automatic SLA enforcement or, at least, to offer enough information so users may enable this enforcement automatically.

Acknowledgements

This study was funded in part by the Spanish Ministry of Science and Innovation under Grants RTI2018-093608-B-C32, and also by the JCCM regional project SBPLY/17/180501/ 000276/1. All these projects are co-financed by the European Union FEDER Funds.

Data Availability

We have deposited the obtained data, supporting the findings of this research, in the Mendeley repository (`https://data.mendeley.com/datasets/vz7v9cgsrr/draft?a=ad29394f-d170-4321-8948-43f9e2acaca0`). A DOI reference will be provided over acceptance of the paper.

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (2009) 599 – 616.

[2] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, Journal of Internet Services and Applications 1 (2010) 7–18.

[3] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, B. Hu, Everything as a service (xaas) on the cloud: Origins, current and future trends, in: 2015 IEEE 8th International Conference on Cloud Computing, pp. 621–628.

[4] P. M. Mell, T. Grance, Sp 800-145. the nist definition of cloud computing (2011).

[5] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann, A systematic review of cloud modeling languages, ACM Computing Surveys (CSUR) 51 (2018) 1–38.

[6] P. Patel, A. H. Ranabahu, A. P. Sheth, Service level agreement in cloud computing (2009).

[7] F. Faniyi, R. Bahsoon, A systematic review of service level management in the cloud, ACM Computing Surveys (CSUR) 48 (2016) 43.

[8] L. Wu, R. Buyya, Service level agreement (SLA) in utility computing systems, CoRR abs/1010.2881 (2010).

[9] K. Lu, R. Yahyapour, P. Wieder, E. Yaqub, M. Abdullah, B. Schloer, C. Kotsokalis, Fault-tolerant service level agreement lifecycle management in clouds using actor system, Future Generation Computer Systems 54 (2016) 247 – 259.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the Clouds: A Berkeley View of Cloud Computing, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[11] Z. Ye, X. Zhou, A. Bouguettaya, Genetic algorithm based qos-aware service compositions in cloud computing, in: International Conference on Database Systems for Advanced Applications, Springer, pp. 321–334.

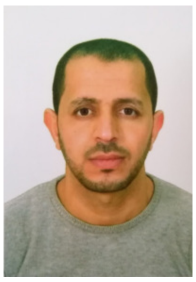
[12] Z. Ye, S. Mistry, A. Bouguettaya, H. Dong, Long-term qos-aware cloud service composition using multivariate time series analysis, IEEE Transactions on Services Computing 9 (2016) 382–393.

[13] R. Karim, C. Ding, A. Miri, M. S. Rahman, Incorporating service and user information and latent features to predict qos for selecting and recommending cloud service compositions, Cluster Computing 19 (2016) 1227–1242.

[14] R. Retter, C. Fehling, D. Karastoyanova, F. Leymann, D. Schleicher, Combining horizontal and vertical composition of services, Service Oriented Computing and Applications 6 (2012) 117–130.
[15] J. O. Gutierrez-Garcia, K. M. Sim, Agent-based cloud service composition, Applied intelligence 38 (2013) 436–464.
[16] D. Weerasiri, M. C. Barukh, B. Benatallah, Q. Z. Sheng, R. Ranjan, A taxonomy and survey of cloud resource orchestration techniques, ACM Computing Surveys (CSUR) 50 (2017) 1–41.
[17] A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, B. Blake, S. Dustdar, F. Leymann, M. Papazoglou, A service computing manifesto: The next 10 years, Commun. ACM 60 (2017) 64–72.
[18] L. Lamport, Proving the correctness of multiprocess programs, IEEE transactions on software engineering (1977) 125–143.
[19] R. Milner, The Space and Motion of Communicating Agents, Cambridge University Press, New York, NY, USA, 1st edition, 2009.
[20] R. Milner, Pure bigraphs: Structure and dynamics, Information and Computation 204 (2006) 60 – 122.
[21] E. Clarke, O. Grumberg, D. A. Peled, Model checking the mit press, Cambridge, Massachusetts, London, UK (1999).
[22] C. Baier, J.-P. Katoen, Principles of model checking, MIT press, 2008.
[23] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, Nusmv: a new symbolic model checker, International Journal on Software Tools for Technology Transfer 2 (2000) 410–425.
[24] C. Tsigkanos, T. Kehrer, C. Ghezzi, Modeling and verification of evolving cyber-physical spaces, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp. 38–48.
[25] Y. Cao, Z. Huang, C. Ke, J. Xie, J. Wang, A topology-aware access control model for collaborative cyber-physical spaces: Specification and verification, Computers & Security (2019).
[26] L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, H. Niss, Bigraphical models of context-aware systems, in: International Conference on Foundations of Software Science and Computation Structures, Springer, pp. 187–201.
[27] L. Yu, W.-T. Tsai, G. Perrone, Testing context-aware applications based on bigraphical modeling, IEEE Transactions on Reliability 65 (2016) 1584–1611.
[28] M. Calder, A. Koliousis, M. Sevegnani, J. Sventek, Real-time verification of wireless home networks using bigraphs with sharing, Science of Computer Programming 80 (2014) 288–310.
[29] M. Calder, M. Sevegnani, Modelling ieee 802.11 csma/ca rts/cts with stochastic bigraphs with sharing, Formal Aspects of Computing 26 (2014) 537–561.
[30] Z. Benzadri, F. Belala, C. Bouanaka, Towards a formal model for cloud computing, in: International Conference on Service-Oriented Computing, Springer, pp. 381–393.
[31] O. Kamel, A. Chaoui, M. Gharzouli, Cloud service composition modeling using bigraphical reactive systems, in: Proceedings of the 21st International Database Engineering & Applications Symposium, ACM, pp. 40–48.
[32] E. Elsborg, T. T. Hildebrandt, D. Sangiorgi, Type systems for bigraphs, in: C. Kaklamanis, F. Nielson (Eds.), Trustworthy Global Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 126–140.
[33] J. J. Leifer, R. Milner, Transition systems, link graphs and petri nets, Mathematical Structures in Computer Science 16 (2006) 989–1047.
[34] S. Benford, M. Calder, T. Rodden, M. Sevegnani, On lions, impala, and bigraphs: modelling interactions in physical/virtual spaces, ACM Transactions on Computer-Human Interaction 23 (2016).
[35] R. Milner, Bigraphs and their algebra, Electronic Notes in Theoretical Computer Science 209 (2008) 5–19.
[36] A. J. Glenstrup, T. C. Damgaard, L. Birkedal, E. Højsgaard, An implementation of bigraph matching, IT University of Copenhagen (2007) 22.
[37] G. Perrone, S. Debois, T. T. Hildebrandt, A model checker for bigraphs, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, ACM, New York, NY, USA, 2012, pp. 1320–1325.
[38] A. J. Faithfull, G. Perrone, T. T. Hildebrandt, Big red: A development environment for bigraphs, Electronic Communications of the EASST 61 (2013).
[39] M. Sevegnani, M. Calder, Bigrapher: rewriting and analysis engine for bigraphs, in: International Conference on Computer Aided Verification, Springer, pp. 494–501.
[40] A. Souri, N. J. Navimipour, A. M. Rahmani, Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review, Computer Standards & Interfaces 58 (2018) 1–22.
[41] Oasis standard: Topology and orchestration specification for cloud applications version 1.0, `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`, 2013.
[42] R. Nyren, A. Edmonds, A. Papaspyrou, T. Metsch, B. Parák, Open cloud computing interface - core, `https://occi-wg.org/`, 2016.
[43] M. Behrendt, B. Glasner, P. Kopp, R. Dieckmann, G. Breiter, S. Pappe, H. Kreger, A. Arsanjani, Introduction and architecture overview ibm cloud computing reference architecture 2.0, Draft Version V 1 (2011).
[44] F. Moscato, R. Aversa, B. Di Martino, T.-F. Fortiş, V. Munteanu, An analysis of mosaic ontology for cloud resources annotation, in: 2011 federated conference on computer science and information systems (FedCSIS), IEEE, pp. 973–980.
[45] W.-T. Tsai, X. Sun, J. Balasooriya, Service-oriented cloud computing architecture, in: 2010 seventh international conference on information technology: new generations, IEEE, pp. 684–689.
[46] Y. Kouki, T. Ledoux, R. Sharrock, Cross-layer sla selection for cloud services, in: 2011 First International Symposium on Network Cloud Computing and Applications, IEEE, pp. 143–147.
[47] Z. Ye, A. Bouguettaya, X. Zhou, Economic model-driven cloud service composition, ACM Trans. Internet Techn. 14 (2014) 20:1–20:19.
[48] R. Ranjan, B. Benatallah, S. Dustdar, M. P. Papazoglou, Cloud resource orchestration programming: overview, issues, and directions, IEEE Internet Computing 19 (2015) 46–56.
[49] D. Serrano, S. Bouchenak, Y. Kouki, F. A. de Oliveira Jr, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, P. Sens, Sla guarantees for cloud services, Future Generation Computer Systems 54 (2016) 233–246.
[50] B. Shojaiemehr, A. M. Rahmani, N. N. Qader, Cloud computing service negotiation: A systematic review, Computer Standards & Interfaces 55 (2018) 196–206.

[51] H. Sahli, N. Hameurlain, F. Belala, A bigraphical model for specifying cloud-based elastic systems and their behaviour, International Journal of Parallel, Emergent and Distributed Systems 32 (2017) 593–616.
[52] K. Khebbeb, N. Hameurlain, F. Belala, H. Sahli, Formal modelling and verifying elasticity strategies in cloud systems, IET Software 13 (2018) 25–35.
[53] A. G. García, I. B. Espert, V. H. García, Sla-driven dynamic cloud resource management, Future Generation Computer Systems 31 (2014) 1–11.
[54] I. V. Paputungan, A. F. M. Hani, M. F. Hassan, V. S. Asirvadam, Real-time and proactive sla renegotiation for a cloud-based system, IEEE Systems Journal 13 (2018) 400–411.
[55] T. Labidi, A. Mtibaa, H. Brabra, Cslaonto: a comprehensive ontological sla model in cloud computing, Journal on Data Semantics 5 (2016) 179–193.
[56] R. Trapero, J. Modic, M. Stopar, A. Taha, N. Suri, A novel approach to manage cloud security sla incidents, Future Generation Computer Systems 72 (2017) 193–205.
[57] D. K. Nguyen, F. Lelli, M. P. Papazoglou, W.-J. Van Den Heuvel, Blueprinting approach in support of cloud computing, Future Internet 4 (2012) 322–346.
[58] Z. Al-Shara, F. Alvares, H. Bruneliere, J. Lejeune, C. PrudHomme, T. Ledoux, Come4acloud: An end-to-end framework for autonomic cloud systems, Future Generation Computer Systems 86 (2018) 339–354.
[59] F. Amato, F. Moscato, Exploiting cloud and workflow patterns for the analysis of composite cloud services, Future Generation Computer Systems 67 (2017) 255–265.
[60] L. Sun, J. Ma, H. Wang, Y. Zhang, J. Yong, Cloud service description model: an extension of usdl for cloud services, IEEE Transactions on Services Computing 11 (2015) 354–368.
[61] S. Ghazouani, Y. Slimani, Towards a standardized cloud service description based on usdl, Journal of Systems and Software 132 (2017) 1–20.
[62] M. Graiet, L. Hamel, A. Mammar, S. Tata, A verification and deployment approach for elastic component-based applications, Formal Aspects of Computing 29 (2017) 987–1011.
[63] S. Jlassi, A. Mammar, I. Abbassi, M. Graiet, Towards correct cloud resource allocation in foss applications, Future Generation Computer Systems 91 (2019) 392–406.
[64] A. G. García, I. Blanquer, Cloud services representation using sla composition, Journal of Grid Computing 13 (2015) 35–51.
[65] K. Boukadi, R. Grati, H. Ben-Abdallah, Toward the automation of a qos-driven sla establishment in the cloud, Service Oriented Computing and Applications 10 (2016) 279–302.
[66] R. B. Uriarte, R. De Nicola, V. Scoca, F. Tiezzi, Defining and guaranteeing dynamic service levels in clouds, Future Generation Computer Systems 99 (2019) 27–40.
[67] S. El Kafhali, K. Salah, Efficient and dynamic scaling of fog nodes for iot devices, The Journal of Supercomputing 73 (2017) 5261–5284.
[68] F. Al-Haidari, M. Sqalli, K. Salah, Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources, in: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, volume 2, IEEE, pp. 256–261.
[69] P. Calyam, S. Rajagopalan, S. Seetharam, A. Selvadhurai, K. Salah, R. Ramnath, Vdc-analyst: Design and verification of virtual desktop cloud resource allocations, Computer Networks 68 (2014) 110–122.
[70] S. El Kafhali, K. Salah, Stochastic modelling and analysis of cloud computing data center, in: 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), IEEE, pp. 122–126.
[71] Z. Benzadri, C. Bouanaka, F. Belala, Big-caf: a bigraphical-generic cloud architecture framework, International Journal of Grid and Utility Computing 8 (2017) 222–240.
[72] R. Moudjari, Z. Sahnoun, F. Belala, Towards a fuzzy bigraphical multi agent system for cloud of clouds elasticity management, International Journal of Approximate Reasoning 102 (2018) 86–107.
[73] O. Kamel, A. Chaoui, M. Gharzouli, Towards a formal modeling of cloud services during the life-cycle of service level agreement, in: Proceedings of the International Conference on Big Data and Internet of Thing, ACM, pp. 115–119.
[74] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (2003) 41–50.
[75] G. Blair, N. Bencomo, R. B. France, Models@ run. time, Computer 42 (2009) 22–27.

Author biographies



Oussama Kamel is an Assistant Professor in the faculty of medicine, University Salah boubnider Constantine 3, Algeria, since 2012. He is currently a Ph.D. candidate at the Modelling and Implementation of Complex Systems laboratory (MISC) in Department of Computer Science and its Applications (DIFA) of New Information and Communication technologies Faculty, University Constantine 2 Abdelhamid Mehri, Algeria. His research focuses on cloud computing, Service-oriented Architecture, service composition, formal modeling and verification.

Allaoua Chaoui is a full Professor of Computer science at the University Constantine 2 Abdelhamid Mehri, Algeria. Actually, he is the Head of the Department of Computer Science and its Applications, Faculty of NTIC. He is also the head of the research Team Software Engineering and Formal Methods, MISC Laboratory. He received his Master degree in Computer science from the University of Constantine (in cooperation with the University of Glasgow, Scotland) in 1992 and his PhD degree from the University of Constantine (in cooperation with the CEDRIC Laboratory of CNAM in Paris, France) in 1998. He has served as associate Professor in Philadelphia University in Jordan for five years and University Mentoury Constantine for many years. During his career he has designed and taught courses in Software Engineering and Formal Methods. Prof Allaoua Chaoui has published many peer-reviewed scientific papers in International Journals and Conferences. He supervised many Master and PhD students. His research interests include Model Driven Engineering, Mobile Computing, formal specification and verification of distributed systems, and graph transformations and their correctness.

Gregorio Díaz is an associated Professor at the University of Castilla-La Mancha within the ReTiCS research group with tenure distinction since 2011, published more than 17 journal papers, from which 15 are indexed by the JCR index, participated in 38 international and national conferences, main researcher of 3 FEDER projects. His research goals are aimed to make software more reliable, secure, and easier to design. He has supervised more than 24 master theses, including 4 in research areas and 2 PhD thesis. He has taught in several courses related to undergraduate and postgraduate studies awarded with the quality award Euro-Inf Bachelor by EQANIE. (http://orcid.org/0000-0002-9116-9535)

Mohamed Gharzouli was born in Constantine (Algeria). He received his BS degree in Computer Science from Mentouri University of Constantine in 2002 and Magister degree in Computer Science from Larbi Tebessi University of Tebessa (Algeria) in 2004. He has completed his PhD degree in Advanced Information systems from Mentouri University in September 2011. He received his habilitation qualifications (Accreditation to supervise research) from Abdelhamid Mehri University in 2015. Currently, he is an Associate Professor in faculty of new technologies of information and communication, university of Abdelhamid Mehri, Algeria. Also, he is a member of Formal Methods for Software Engineering team of MISC Laboratory, in the same university. He published many articles in international conferences and journals. His researches interests include service-oriented architectures, Web applications, and use of information technologies in different fields like Business and Education.