# Modeling and Performance Analysis of Resource Provisioning in Cloud Computing using Probabilistic Model Checking

Hichem Debbi
Department of Computer Science, University of M'sila, M'sila, Algeria
E-mail: hichem.debbi@univ-msila.dz

**Keywords:** cloud computing, IaaS, performance analysis, resource provisioning, probabilistic model checking, PRISM

**Received:** September 14, 2020

*Cloud computing consists of an advanced set of technologies that allow cloud providers to offer computing resources such as infrastructure, platforms and applications to be accessible over the Internet as services. Cloud computing relies on virtualization of resources in the cloud data centers, where a set of Virtual Machines (VMs) are deployed on Physical Machines (PMs) to provision and serve user requests. Due to the dynamic nature of cloud environments and complexity of resources virtualization, as well as the diversity of user's requests, developing effective techniques to evaluate and analyze the performance of cloud centers has become highly required. In this paper, we propose the use of probabilistic model checking as an effective framework for the evaluation and the performance analysis of resource provisioning in the cloud. Based on an analytical model for resource provisioning in Infrastructure-as-a-Service (IaaS) cloud, we build a stochastic model using the probabilistic model checker PRISM and analyze it against a useful set of probabilistic and reward properties that help to measure and analyze cloud performance in an efficient way.*

*Povzetek: Analizirane so razne komponente računanja v oblaku, npr. modeliranje in performance virov.*

## 1 Introduction

Cloud computing is a novel information technology that provides access to different IT services on demand over the Internet. The services provided through the cloud range into three main categories: Infrastructure as a Service (Iaas), where infrastructure resources such as: servers, storage, network components are provisioned. Platform as a Service (PaaS), which provides an environment for developing, running and managing applications efficiently by reducing the complexity related to infrastructure. Software as a Service (SaaS), which represents the largest cloud market, in which the task of managing software is moved to third-party services. Cloud computing has been treated from different aspects such as: security [22], load balancing [24], storage[7] and consistency [21].

In cloud computing literature, we refer usually to service providing by the technical term, provisioning. In this regard, Vaquero et al. [23] defined cloud as: the provision of computing infrastructure, which aims to shift the location of the computing infrastructure to the network in order to reduce the costs associated to management and maintenance of hardware and software resources. These resources are offered to the customer by cloud providers based on specific legally binding contracts called Service Level Agreements (SLAs), which state Quality of Service (QoS) parameters, such as time, cost, availability and security that should be guaranteed by service providers in order to meet customer's needs and execute service requests. Buyya et al. [5] defined the cloud as: "A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service level agreements (SLA) established through negotiation between the service provider and the customers".

In IaaS cloud, virtualization plays a crucial role in enabling cloud computing services, in fact, it is a principal mechanism that enables cloud providers to cope with multiple requests of users through virtualization of physical machines(PMs). Virtualization refers to the abstraction of computing resources in a way that a single physical machine can run a set of virtual machines(VMs)[3].

However, due to dynamic nature of cloud computing environments and the complexity related to managing infrastructure resources from a side, and the diversity in customers requests from another, addressing the effective ways to instantiate, provision and deploy infrastructure resources to handle user requests and meet QoS requirements is considered as a big challenge and very critical issue in cloud computing. Therefore, performance analysis and evaluation of cloud computing environments have attracted recently much attention and formed an active research area.

Cloud performance analysis is beneficial for both cloud providers and consumers because it helps to get a deep insight on the infrastructure resources and how they should

be provisioned and scaled to execute various customers requests. Among various performance and evaluation methods, analytical modeling-based methods represent the major research that has been done in this area [8]. Since resources provisioning and usage is highly variable and uncertain, and since the arrival of customer requests is stochastic, these methods are stochastic in general, and employ queuing theory with different buffers to cope with a large number of requests given the available resources, and thus, performance measures are quantified using probabilistic methods. These stochastic methods can effectively capture the uncertainty beyond cloud provisioning behavior and estimate perfectly cloud metrics. Hence, SLA can be maintained and an overall optimization can be achieved. Continuous-time Markov Chains (CTMC), Stochastic Reward Net (SRN) and Stochastic Petri Nets (SPN) are all stochastic models that have been used for modeling and analyzing cloud services performance, and they showed promising results.

Performance behavior in the cloud is affected by a large set of parameters, and thus many system variables must be introduced to capture every modeling detail. CTMC models represent a good candidate to model every detail of the system [12, 17]. However, as the system variables under modeling grow up, the analysis could be intractable, since it results in a very large state space, which is known as the state explosion problem. To cope with such a problem, a solution based on decomposing the entire model into small interacting sub-models is proposed to facilitate and speed-up the model generation[12].

Bradley et al.[4] stated that symbolic approaches are very useful for performance and resilience modeling and analysis of massive stochastic systems, and thus they are very suitable for state space representation for cloud computing systems. Symbolic techniques such as Multi-terminal Binary Decision Diagrams are efficiently used to encode CTMCs, and enable steady-state and transient analysis. These techniques are efficiently employed by the probabilistic model checker PRISM[14], whose language features synchronization between modules. These advantages make PRISM a suitable tool for the performance analysis of cloud computing systems.

In this paper, we aim to show how probabilistic model checking can be used for the performance analysis and evaluation of IaaS cloud based on analytical modeling methods using CTMCs. Probabilistic model checking has appeared as an extension of model checking for analyzing systems that exhibit stochastic behavior. These systems are described usually using Discrete-Time Markov Chains (DTMC), Continuous Time Markov Chains (CTMC) or Markov Decision Processes (MDP), and verified against properties specified in Probabilistic Computation Tree Logic (PCTL)[13] or Continuous Stochastic Logic (CSL) [1, 2].

Using the probabilistic model checker PRISM [14], we show that analytical models, even if they are composed of many interacting sub-models, can be easily expressed in PRISM language and analyzed in an efficient way. The entire model can be generated from interacting sub-models in reasonable time thanks to many numerical solution methods employed by PRISM that can deal perfectly with the state explosion problem. In this paper, we chose the model proposed by [12] as a case study. With probabilistic model checking, we will not be able only to compute probabilities related to QoS metrics, but also we can verify such safety properties and analyze reward-based properties.

The rest of this paper is organized as follows. In Section 2 we present some related works to cloud performance analysis. Section 3 presents some preliminaries and definitions on PRISM language. In section 4, we present the analytical model and its implementation in PRISM with detailed analysis of probabilistic and reward properties. Finally, we conclude the paper in Section 5.

# 2 Related work

The performance analysis and evaluation of cloud computing services can be performed through two ways: measurement-based methods and analytical modeling-based methods. In measurement-based methods [19], both cloud services and performance metrics to be evaluated should be known in prior, and then the benchmark to be tested should be chosen accordingly. After that, the testing experiments can be executed. Actually, this technique suffers from extensive experiments that should be executed with different workloads and system configurations, which may make the construction of appropriate testbeds that can represent realistic cloud services scenarios a costly task. Despite that, some measurements become invalid when cloud service providers upgrade their software and hardware to enhance their services. Therefore, analytical modeling-based methods are considered as a good alternative since they are of low cost, and can cover large parameters of cloud services, especially that these methods can analyze features of services even in early stages of design.

Li et al. [20] addressed the analysis of cloud services by modeling the service as a queuing network consisting of two tandem servers, web server and service server. After service completion at the level of the web server, the request either exits the network or continues to be executed at the service server. Both servers are modelled as M/M/1 queue with an exponential distribution of arrival and service times. The main metric under evaluation in this paper was response time. Based on this measure, a relationship between the number of customers, the minimal service resources and the highest level of services can be easily derived. However, this work lakes an important feature in cloud computing modeling, which is virtualization.

Chen et al. [6] have also considered queuing network to estimate two different performance metrics, which are practically needed more in the context of cloud computing, request completion time (ECT) and rejection probability (RP). The authors in this work consider also two

queues, admission queue, and PM queue. Visualization is addressed considering many parameters, such as buffer size of queues, number of virtual machines, number of physical machines and error/recovery rates. In this work, each job is denoting a VM instance, and each VM is deployed on a single PM.

While previous works assume an exponential distribution of requests, considering heterogeneity in cloud modeling is actually more appropriate to better analyze some dynamic properties. In this regard, Khazaei et all. [16] introduced an embedded Markov model as an approximate analytical model based on M/G/m/m+r queuing system with single task arrivals and a task buffer of finite capacity. By solving the approximate model, complete probability distribution of the request response time, and important performance indicators such as the mean number of tasks in the system, the blocking probability, and the probability of immediate service can be easily estimated.

Covering more cloud services parameters by performance evaluation model is highly needed. However, sometimes the analysis of such a model tends to be intractable. To deal with this issue, some works [12, 17] proposed a solution based on interacting stochastic sub-models of Continuous-time Markov Chain (CTMC), thus, quantifying performance metrics can be realized in a scalable manner. In [12], the main QoS addressed was service availability and provisioning response delays. The requests or jobs submitted by users can be served in different pools named (hot, warm and cold), the decision in which pool the request should be served is made by a module called resource provisioning decision model, which is a CTMC model consisting of a queue with finite length. Another queue is found at each PM, where some requests/jobs can wait for VM provisioning. While this model is limited to service requests with a single task, Kazai et all. [17] proposed a similar solution, but capable of dealing with batch arrival of requests, where multiple VMs can be provisioned to handle multi-tasks based on a single service request, thus realizing a high degree of visualization.

Probabilistic model checking has already been used for modeling and analysis of cloud computing. Kikuchi and Matsumoto [18] have used PRISM for the performance modeling and analysis of concurrent live migration operations in cloud computing systems. Live migration plays a crucial role in cloud virtualization since it guarantees transporting VMs from a host to another without affecting the performance of the services. The authors described the performance model of concurrent VM live migration operations as a CTMC in PRISM language, and it has been verified against two main quantitative properties regarding the operations that can be stacked in waiting state at sender side, and the operations that are executed at server side.

In [15], the authors defined an interesting set of resource usage patterns in PRISM language as an MDP, and then introduced a set of reward-based properties for analyzing cost variation, and min/max probabilistic properties to analyze deployment's resource usage. These probabilistic patterns before being generated as MDPs, are first expressed in a higher language called probabilistic pattern modeling language (PPM).

Evangledis et al. [9] addressed performance modeling and formal verification of auto-scaling policies in PaaS and IaaS to provide performance guarantees to reduce SLAs violations, where two cloud services providers Amazon EC2 and Azure have been considered. The authors considered rule-based auto-scaling policies, where upper and/or lower bound on performance metrics such as CPU are expressed. The dynamics of auto-scaling process are expressed in PRISM as DTMC, and verified against probabilistic properties to estimate CPU utilization and response time violation for each auto-scaling policy, thus refining QoS violation thresholds for the policies.

We summarize the existing related work in Table 1

# 3 PRISM

PRISM is a tool used for formal modeling and analyzing systems that exhibit random or probabilistic behavior [14]. It supports several types of probabilistic models such as DTMCs, CTMCs and MDPs. The analysis is performed on these models against properties specified in PCTL logic [13] for DTMCs and MDPs and Continuous Stochastic Logic (CSL) [1, 2]for CTMCs. PRISM uses several numeric methods for model analysis such as Gauss-Seidel method, Backwards Gauss-Seidel method and Jacobi method. For MDPs and CTMCs, PRISM uses value iteration and uniformisation, respectively. As additional features, PRISM offers a simulation framework for reasoning about probabilities and rewards.

A model in PRISM consists of one or several modules that interact with each other. The module is specified using PRISM language as a set of guarded commands.

$[< action >] < guard > \longrightarrow < updates >$

Where the guard is a predicate over the variables of the system and the updates describe probabilistic transitions that the module can make if the guard is true. These updates are defined as follows:

$< prob >:< atomicupdate > + ..... + < prob >:< atomicupdate >$

When representing CTMCs, $< prob >$ will refer to transition rates instead of discrete probabilities. PRISM also supports rewards which are real values associated with states or transitions of the model. Where state rewards can be specified as: $g : r$, and transition rewards are represented as: $[a]g : r$.

The properties for a CTMC model can be specified in CSL logic that allows the specification of both transient behavior and steady state behavior. We use the P operator for specifying transient properties and S operator for specifying steady state properties. Another interesting operator employed is the R operator that is used to reason on the expected value of rewards.

**Example**

| | Analytical models | Analytical Tools | Properties | Scope | Experimental setting |
|---|---|---|---|---|---|
| Ghosh et al. [12] | CTMCs | SPHERE | service availability, provisioning response delays | Infrastructure | IBM SmartCloud |
| Khazaei et al.[17] | CTMCs | Maplesoft | rejection probability, response delays | Infrastructure | Artifex engine |
| Li et al.[20] | queuing networks | Matlab | completion time, rejection probability, system overhead rate | Infrastructure | – |
| Chen et al. [6] | queuing networks | – | Rejection probability, task completion time | Infrastructure | XenServer and OpenStack |
| Kikuchi et al.[18] | CTMCs | PRISM | stacked operations, executed operations | Infrastructure | XenServer |
| Jhonson et al.[15] | MDPs | PRISM | Cost variation, deployment's resource usage | Infrastructure | – |
| Evangledis et al.[9] | DTMCs | PRISM | CPU utilization, response time violation | Infrastructure, Platform | Amazon EC2 and Azure |

Table 1: Main related work on cloud performance analysis.

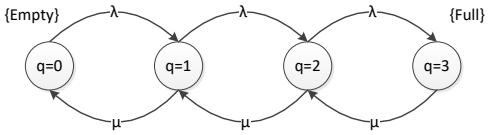


Figure 1: Queue model.

```
ctmc
const int lq = 3;//queue length
const double lambda = 1/10;//arrival rate
const double mu = 1/2;//service rate
module Queue
q: [0..lq] init 0;
[Arive] (q<lq) -> lambda: (q'=q+1);
[Serve] (lq>0) -> mu : (q'=q-1);
endmodule
label "full"= q=3;
```

Figure 2: Prism model for the queuing system.

Let us consider the CTMC presented in Figure 1. It represents a queuing system with maximum length 3. The system can move from an empty state, where there is no job to a new state with $(q = 1)$ by the arriving of a new job wit rate $\lambda$, and can return to the previous state by serving the job with a rate $\mu$. The same thing applies for the rest of states. The corresponding module for this model is described in Figure 2. We have to declare 3 variables, the integer variable $lq$ that refers to queue length and two double variables representing the arrival and services rates. The main variable is $q$, which represents the possible states of the system through raising two main transitions, *Arrive* and *Service*, with their corresponding rates. We can express probabilistic properties based on the value of time variable $T$. For instance, we can express a CSL property that states that the probability of the queue being full with time $T$ should not exceed the probability 0.5: $P <= 0.5[trueU <= T"full"]$. The property can be rewritten in a different way to estimate the probability of the property being true within time unit $T$ as $P =?[trueU <= T"full"]$.

# 4 Case study

The model that we are going to study concerns data centers that consist of a number of Physical Machines (PMs) [12]. When user requests arrive at a cloud center, a virtual machine or many VMs are deployed on PMs to serve this request. A single VM can be provisioned to serve a single request, however, in reality, multiple VMs can be provisioned on a single or multiple PMs to serve such complex request or super-task [17]. The PMs are grouped into three servers: hot (i.e., running VMs), warm (turned on, but without running VMs) and cold (turned off). It is tried first to provision the request on a hot pool if there is enough capacity, if there is no a hot PM available, there will be a

look-up for a warm PM, if all warm PMs are busy, a PM in the cold pool is used. In the case where no PM is available in all pools, the request will be simply rejected. The strategy of regrouping the PMs into multiple pools results in a good performance by reducing VMs provisioning delay and operational costs. The model proposed consists of three sub-models that refer to the three main steps of cloud servicing, which are resource provisioning decision, VM provisioning, and run-time execution. The overall solution is obtained by interacting over these three sub-models. The steps of provisioning and servicing are presented in Figure 3. In the following, we will describe each of the CTMCs models.

## 4.1 Resource provisioning decision model (RPDM)

This module is responsible for choosing the PM that can accept the request and in which pool. A finite decision queue is employed, where decisions are made on FIFO basis. The arrival to RPDM is modeled as Poisson process with arrival rate $\lambda$. The related CTMC model is shown in Figure 5.

The states in the model are presented as pairs $(i, j)$, where $i$ denotes the number of requests being waiting in the global queue, and $j$ denotes the pool on which the request is under provisioning. The initial state $(0, 0)$ means that the system is in an empty state, where there is no request, neither in the queue nor under provisioning. $j$ is set to 'h' if there is at least one hot PM that can accept the job for provisioning. Similarly, when $j$ is set to 'w' (or 'c'), that means that a warm (or cold) PM can accept the job for provisioning. The waiting queue for this model has a maximum number $N$.

From the initial state, by arriving of a new request, the system moves to the state $(0, h)$ with rate $\lambda$, since it tries to find a hot PM first. From this state the following three possible transitions can occur:

- A request is accepted for provisioning in a hot PM, and thus the module returns to the state $(0, 0)$ with rate $P_h\delta_h$.
- Another request arrives, and the system moves to state $(1, h)$ with rate $\lambda$.
- No hot PM can accept the request for provisioning due to insufficient capacity, and thus the system tries to find a warm PM and transits to state $(0, w)$ with rate $\delta_h(1 - P_h)$

Now, from the state $(0, w)$, the model tries to find an available warm PM to provision the request, if one warm PM is available, the model moves back to the initial state with rate $P_w\delta_w$, otherwise, the module tries to find a PM in cold pool by making a transition to $(0, c)$ with rate $\delta_w(1 - P_w)$. Then, from the state $(0, c)$, the request can be either accepted in the cold pool, and thus the model moves back to the initial state with rate $P_c\delta_c$, or the request is rejected when there is no available cold PM, and thus the model moves to the same state with a rate $\delta_c(1 - P_c)$. The state where $i >= 1$ means that $i$ request is waiting in the queue.

The related prism module of this model is depicted in Figure 6. We use two main variables, $i$ and $j$, where $i$ refers to the number of jobs waiting in the queue, and j denotes the type of pool (j=1 for hot, j=2 for warm and j=3 for cold). The commands with *wait* action and rate $\lambda$ (lines 6, 11 and 16) refer to a new request waiting and staying at the same pool, hot, warm and cold respectively. The other commands of provision refer to the provision in hot, warm and cold respectively with the appropriate rates. The rest of the commands where no action is defined refer to searching for a PM in the next pool. While *wait* actions have no control on the entire model, and they are just used for indication, the other actions (Provision_hot, Provision_warm and Provision_cold) are used for synchronization with the rest of provisioning models (hot, warm and cold).

To build our model we need global as well as local variables. While local variables are defined at each module, global variables are defined at the top of the global model, thus they can be used by all modules. The rates in CTMC models are usually defined as global variables. In addition, we can define some variables that play an important role in defining properties such as the time variable $T$. The set of variables with their values are presented in Figure 4. These values are basically adapted from [12, 10].

## 4.2 VM provisioning models

These models capture instantiation, configuration and provisioning of a VM on a PM. The model for provisioning a hot PM is described as a CTMC in Figure 10. In this model, requests, PMs and VMs are all assumed to be homogeneous, and each request is for one VM instance. We also assume that inter-arrival time, service time and VM provisioning time are all exponentially distributed.

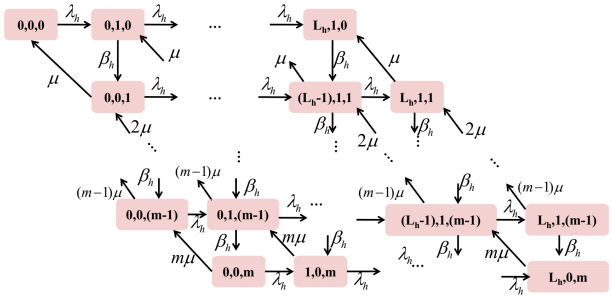


Figure 10: VM provisioning model for each hot PM[12, 11].

States of provisioning model are controlled by three main variables $i$, $j$ and $k$, where $i$ presents the number of requests in PM's queue, $j$ presents the number of VMs currently being provisioned, and $k$ presents the number of VMs which have already been deployed. There are also input parameters that control the model, $L_h$ that represents the size of PM's queue, $j$ can be 0 or 1 if the VMs are

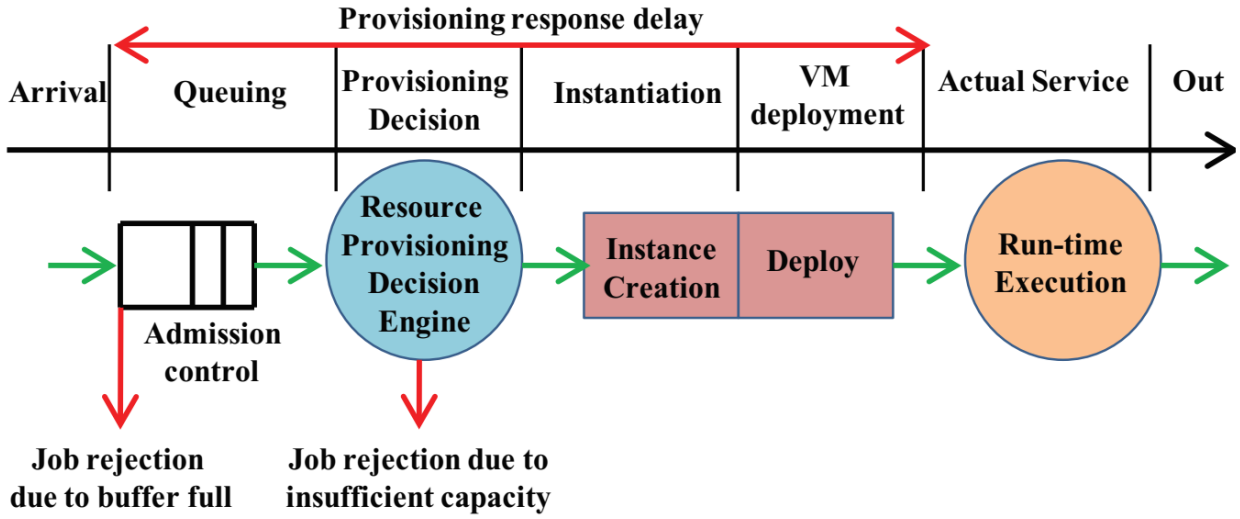


Figure 3: Request provisioning and servicing steps [12].

assumed to be provisioned one at a time, otherwise, a parameter can be introduced to refer to the number of VMs under provisioning. Finally, we have the maximum value of $k$, which refers to the maximum number of VMs that can run in parallel ($m$). The other parameters concern rates: effective job arrival rate ($\lambda_h$), VM provisioning rate ($\beta_h$) and service rate ($\mu$).

As we see in Figure 10, when a request arrives, the model moves from state (0,0,0) to state (0,0,1) with a rate $\lambda_h$, which means that the current request is under provisioning, then it moves to the state (0,0,1), with a rate $\beta_h$, the last state indicates that one VM is deployed, upon service completion, VM instance is removed and the model goes back state $(0,0,0)$ with service rate $\mu$.

The related PRISM mode for a hot PM is presented in Figure 7. We use here three variables: $xh$ that refers to the state of PM's queue, $yh$ that refers to the provisioning state and $zh$ that refers to the number of VMs being deployed. The commands refer in order to provisioning, deployment and service respectively. An additional action has been added just to use it in properties that specify the number of requests being rejected.

The two other CTMCs for warm and cold PMs are similar, though, they can define different arrival and instantiation rates (see Figure 11 and Figure 12). The most important difference concerns provisioning step, where in both warm and cold pools, PMs are turned on but not ready to use, thus, they require additional startup time. Time to make a warm/cold PM ready for use is exponentially distributed with a rate $\gamma_w/\gamma_c$.

The related PRISM modules describing warm and cold provisioning models are shown in Figures 8 and 9 respectively. We notice that warm and cold models define the same variables and steps as the hot model does, except with the provisioning step, where the values $yw$ and $yc$ have a larger range, $yw = 2/yc = 2$ refer to $1^*$ and $yw = 3/yc = 3$ refers to $1^{**}$. Thus, compared to the hot model, an additional section has to be added starting from line 11.

For the following values: ($lq = 6, lh = 1, lw = 1, lc = 1$ and $m = 2$), the model generated by PRISM consists of 72859 states and 289147 transitions. The size of the model may mainly vary to the number of variables used in the module, as well as the range of their values. For instance, if we let the value of $j$ of warm and cold pools as the same as hot (i.e [0..1]), we had to introduce two new Boolean variables to replace the values 2 and 3 of $j$. Such a solution could result in an additional large set of states.

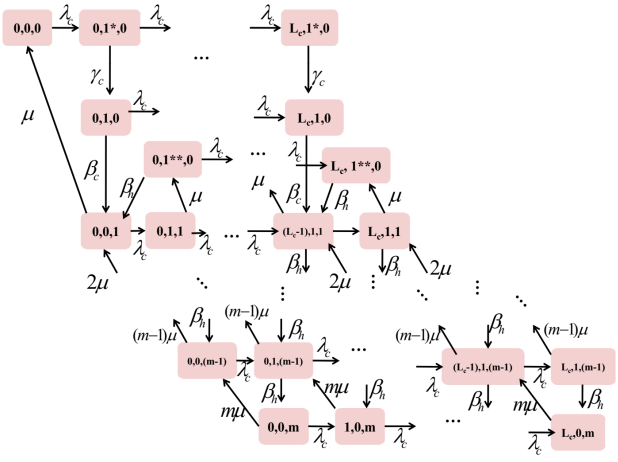


Figure 12: VM provisioning model for each cold PM[11].

## 4.3 Specification

In this section, we will show how we can specify quantitative properties in PRISM to reason about many measures of cloud performance through various operators employed by PRISM, which are the transient operator $P$, the steady

```
1  //arrival and service rates
2  const double lambda;
3  const double mu;
4  // provisioning rates
5  const double betaH=1;
6  const double betaW=1/2;
7  const double betaC=1/3;
8  //max VMs deployed, time and global queu size
9  const m=2 ;
10 const double T;
11 const  lq =6;
12 // buffer size hot, warm,cold
13 const lh=1;
14 const lw=1;
15 const lc=1;
16
17 const  double deltaH=3;
18 const  double deltaW=3;
19 const  double deltaC=3;
20 // prob. off succes in hot, warm and cold
21 const double ph=0.9 ;
22 const double pw =0.8 ;
23 const double pc=0.7 ;
24
25 const double lamH =deltaH* (1-ph);
26 const double lamW=deltaW * (1-pw);
27 const double lamC =deltaC* (1-pc);
28
29 const double muH=deltaH*ph;
30 const double muW=deltaW*pw;
31 const double muC=deltaC*pc;
32
33 const double lambdaH = lambda/2;
34 const double lambdaW =lambda/4;
35 const double lambdaC =lambda/5;
36
37 const double gammaW =1;
38 const double gammaC=1;
```

Figure 4: Global variables.

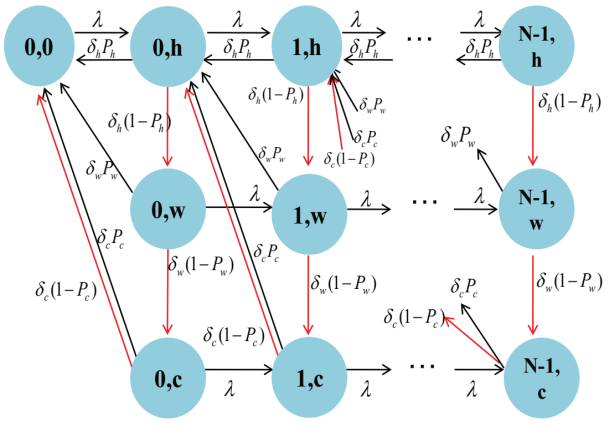


Figure 5: Resource provisioning decision model[12].

operator $S$ and the reward operator $R$. The model checking algorithm used during the analysis phase is Jacobi method, though, we can use other methods such as Gauss-Seidel method. We can use many variations in the model parameters during the analysis, such as the number of PM's, the number of VMs, arrival and service rates, etc. Since each PM is represented by a complete module, to ease the analysis, we are going to fix the number of PMs, so no module duplication will be used.

We can use the simulation framework of PRISM to provide a detailed analysis based on these values. The different graphs to plot will be based on the variation of three main values, arrival rate $\lambda$, service rate $\mu$ and time variable $T$. All the values are considered in minutes. We will show that the main measures, job rejection probability and waiting time can be easily computed using probabilistic and reward properties. We will also show how to obtain additional important measures. Before presenting these properties, we should introduce some labels that are employed to express in a better way these properties. The set of labels that we are going to use are presented in Figure 13, and the set of rewards are presented in Figure 14.

**Job rejection probability** As explained before, job rejection could be at the level of the global queue, where the buffer size reaches its limit, or on the level of the provisioning module, where there is no sufficient resources, which means that all PMs queues are full (xh=lh & xw=lw &xc=lc). To obtain the rejection probability due to insufficient capacity we use the following steady-state property :

$S =?[\text{"}all\_Pools\_Full\text{"}]$

We fix the arrival rate by $\lambda = 8$, and we use different values of the mean service time $\mu$ to obtain the results presented in Figure 15. From the Figure, we see that increasing the mean service time results in increasing job rejection probability. It's evident that taking more time to serve a request could result in rejecting new arriving requests. We can also use a steady property to estimate the long-run probability of the queue being more than 75% full, this interesting property results in 0.99 and can be expressed as the following: $S =?[i/lq > 0.75]$.

Another important measure can be estimated using steady-state operator is the steady probability that the system is in full provision state $(yh = 1\&yw >= 1\&yc >= 1)$, which means that in all pools, there is a request being provisioned. This property is expressed as $S = ?[\text{"}all\_Provision\text{"}]$ and returns a value of 0.94.

We can reason on minimum and maximum delay time taken for provisioned requests before being served. To do so we use the following reachability reward properties that estimate the reward accumulated along a path until a certain state is reached. The time reward is denoted by "time" in Figure 14 (line 9), where every transition is counted.

$R\text{"}time\text{"} =?[F(j = 1\&zh = 1)\{j = 1\&zh = 0\}\{max\}]$
$R\text{"}time\text{"} =?[F(j = 2\&zw = 1)\{j = 2\&zw = 0\}\{max\}]$

```
module rpdm
i: [0..lq];
j:[0..3] init 0;

[] (i=0) &(j=0)-> lambda :(i'=i)&(j'=1);
[wait] (i<lq-1) &(j=1)-> lambda :(i'=i+1)&(j'=j);
[] (i<lq) &(j=1)-> lamh :(i'=i)&(j'=2);
[Provision_hot] (i>0 )& (i<lq)&(j=1) -> muH :(i'=i-1)&(j'=j);
[Provision_hot] (i=0)&(j=1) -> muH :(i'=0)&(j'=0);

[wait] (i<lq-1)&(j=2) -> lambda: (i'=i+1)&(j'=j);
[] (i<lq)&(j=2) -> lamw:(i'=i)&(j'=3);
[Provision_warm] (i>0 )&(i<lq)&(j=2) -> muW :(i'=i-1)&(j'=j-1);
[Provision_warm] (i=0)&(j=2) -> muW :(i'=0)&(j'=0);

[wait] (i<lq-1)&(j=3) -> lambda :(i'=i+1)&(j'=j);
[] (i>0 )&(i<lq)&(j=3)-> lamc :(i'=i-1)&(j'=j-2);
[Provision_cold] (i>0 )&(i<lq)&(j=3)-> muC :(i'=i-1)&(j'=j-2);
[Provision_cold] (i=0)&(j=3) -> muC :(i'=0)&(j'=0);
endmodule
```

Figure 6: PRISM model for The RPDM module.

```
module vmpsm_hot
xh: [0..lh] init 0;// PM queue
yh: [0..1] init 0;// provision
zh: [0..m] init 0;// deployement

[Provision_hot] (xh=0) &(yh=0)&(j>=0)&(j<2)-> lambdaH :(yh'=1);
[Provision_hot] (xh<lh) & (yh=1) &(j=1)-> lambdaH :(xh'=xh+1);
[] (xh>0) & (yh<1)&(j=1)-> lambdaH :(yh'=yh+1)& (xh'=xh-1);
[] (yh>0)& (zh<m)&(j=1)-> betaH:(yh'=yh-1)&(zh'=zh+1);
[serve_hot] (zh>0)& (zh<=m)&(j=1)-> zh*mu:(zh'=zh-1);
[Reject_hot] (xh=lh) & (j=1) ->true;
endmodule
```

Figure 7: PRISM model for hot PM.

```
module vmpsm_warm
xw: [0..lw] init 0;// PM queue
yw: [0..3] init 0;// provision
zw: [0..m] init 0;// deployement

[Provision_warm] (xw<lw) & (yw=1) &(j=2)-> lambdaW :(xw'=xw+1);
[] (xw>0) & (yw<1)&(j=2)-> lambdaW :(yw'=yw+1)& (xw'=xw-1);
[] (yw>0)& (zw<m)&(j=2)-> betaW:(yw'=yw-1)&(zw'=zw+1);
[serve_warm] (zw>0)& (zw<=m)&(j=2)-> zw*mu:(zw'=zw-1);
// provision steps different than hot
[Provision_warm] (xw=0) &(yw=0) &(zw=0)&(j=2)-> lambdaW :(yw'=2);
[Provision_warm] (xw<lw) &(zw=0) & (yw=2)&(j=2) -> lambdaW :(xw'=xw+1) & (yw'=2);
[] (xw<lw) &(yw=2)& (zw=0) &(j=2) -> deltaW :(xw'=xw) & (yw'=1);
[] (zw=0)& (yw=3)&(j=2)-> betaH:(yw'=yw-1)&(zw'=zw+1);
[serve_warm] (zw=1)& (yw=1)&(j=2)-> zw*mu:(zw'=zw-1) & (yw'=3);

[Reject_warm] (xw=lw) & (j=2)->true;
endmodule
```

Figure 8: PRISM model for warm PM.

```
module vmpsm_cold
xc: [0..lc] init 0;// PM queue
yc: [0..3] init 0;// provision
zc: [0..m] init 0;// deployment

[Provision_cold] (xc<lc) & (yc=1) &(j=3)-> lambdaC :(xc'=xc+1);
[] (xc>0) & (yc<1)&(j=3)-> lambdaC :(yc'=yc+1)& (xc'=xc-1);
[] (yc>0)& (zc<m)&(j=3)-> betaC:(yc'=yc-1)&(zc'=zc+1);
[serve_cold] (zc>0)& (zc<=m)&(j=3)-> zc*mu:(zc'=zc-1);
// provision steps different than hot
[Provision_cold] (xc=0) &(yc=0) &(zc=0)&(j=3)-> lambdaC :(yc'=2);
[Provision_cold] (xc<lc)&(zc=0) & (yc=2)&(j=3) -> lambdaC :(xc'=xc+1) & (yc'=2);
[] (xw<lw) &(yc=2)& (zw=0) &(j=3) -> deltaC :(xc'=xc) & (yc'=1);
[] (zc=0)& (yc=3)&(j=3)-> betaH:(yc'=yc-1)&(zc'=zc+1);
[serve_cold] (zc=1)& (yc=1)&(j=3)-> zc*mu:(zc'=zc-1) & (yc'=3);

[Reject_cold] (xc=lc) & (j=3)->true;
endmodule
```

Figure 9: PRISM model for cold PM.

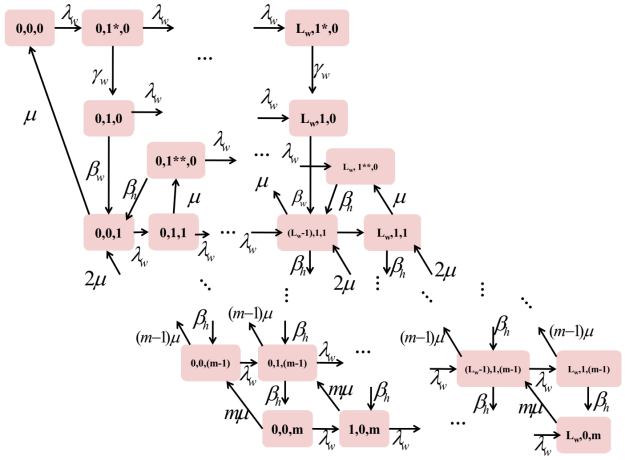


Figure 11: VM provisioning model for each warm PM[11].

```
label  "deployed_Max_In_hot" = (zh =m & j=1);
label  "deployed_Max_In_warm" = (zw =m & j=2);
label  "deployed_Max_In_cold" = (zc =m & j=3);
label "all_Provision"= (yh=1&yw>=1&yc>=1);
label "all_Pools_Full" = (xh=lh &xw=lw
& xc=lc);
label "maximum_Deployment" = (zh=m & zw=m
& zc=m);
```

Figure 13: Labels.

```
rewards "queue_size"
true : i;
endrewards
rewards "Provision_queue_full"
[Provision_hot](i=lq-1) : 1;
[Provision_warm](i=lq-1) : 1;
[Provision_cold](i=lq-1) : 1;
endrewards
rewards "time"
true : 1;
endrewards
rewards "Waiting_Pools"
true : xh + xw + xc;
endrewards
rewards "VMs_Deployed"
true : zh + zw + zc;
endrewards
rewards "VMs_Deployed_Hot"
true : zh;
endrewards
rewards "VMs_Deployed_Warm"
true :zw;
endrewards
rewards "VMs_Deployed_Cold"
true : zc;
endrewards
rewards "request_Reject_Warm"
[Reject_warm]true: 1;
endrewards
```

Figure 14: Rewards.

$R"time" =?[F(j=3\&zc=1)\{j=3\&zc=0\}\{max\}]$

The properties estimate the reward that a state where a request is served can be reached starting from a state where the request is provisioned before being served. Roughly speaking, it computes the complete time between provisioning and service. For the hot pool, PRISM returns a

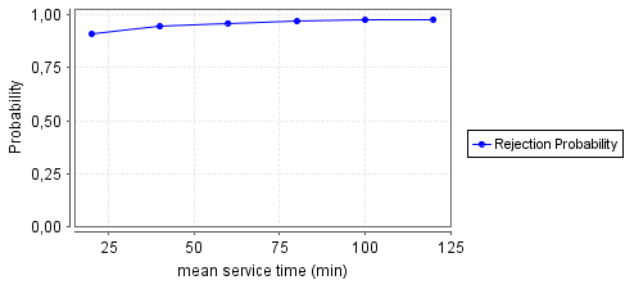


Figure 15: Job rejection probability.

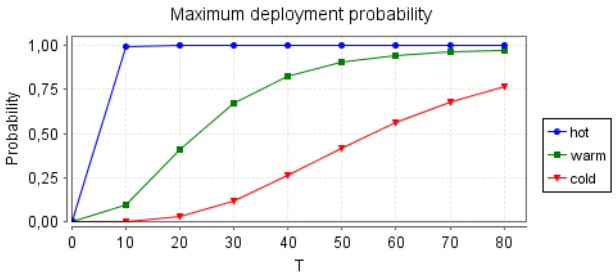


Figure 16: Probability of max deployment over time T.

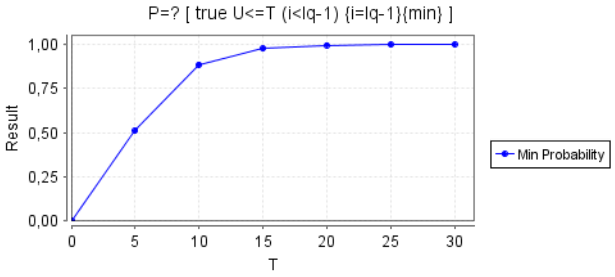


Figure 17: Min probability using filter of full queue.

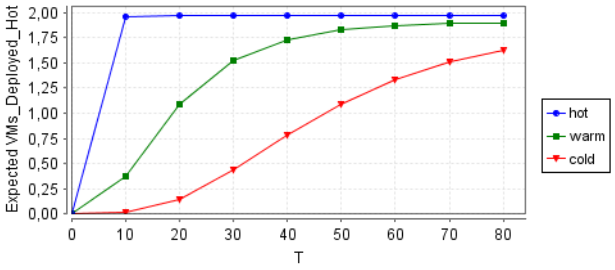


Figure 18: Number of VMs deployed.

result of 3 time units, 15 for the warm pool and 37 for the cold pool. It is evident that the time taken for the request to be served in hot pool is much less than warm and cold, since the last pools require additional provisioning time, and the resource provisioning decision model tries to provision the request in hot pool first. As we can compute maximum reward/probability, we can also compute minimum reward/probability using the feature of filter. For instance, we can compute the minimum probability of the global queue being not full, starting from the state where the queue has been already full. The property is presented as :

$P =?[trueU <= T(i < lq-1)\{i = lq-1\}\{min\}]$

The results of this property for different values of $T$ are presented in Figure 17. A simple reachability property without a filter can be used to compute the probability of reaching the maximum deployment in each pool as follows :

$P =?[trueU <= T"deployed\_Max\_In\_hot"]$
$P =?[trueU <= T"deployed\_Max\_In\_warm"]$
$P =?[trueU <= T"deployed\_Max\_In\_cold"]$

The results of these properties are depicted in Figure 16. We see that the probability of reaching maximum deployment ($zh = 2$) in hot increases faster than warm and cold. For warm pool, the maximum probability value is not reached until approximately $T = 70$.

We can also use *Instantaneous reward* properties to reason on the reward of a model at a particular instant of time. This type of properties associates with a path the reward in the state of that path when exactly T time units have elapsed. We can use it to estimate for instance the exact number of requests waiting in the global queue in an instance T as follows:

$R"queue\_size" =?[I = T]$

As time elapses, the reward will increase until it reaches its limit, which will be at most $lq - 1$. Similarly, we can use the property $R"Waiting\_Pools" =?[I = T]$ to compute the number of requests being waiting. We can use these *Instantaneous reward* properties also to reason about the number of VMs deployed globally at an instance T: $R"VMs\_Deployed" =?[I = T]$. For instance, given a value of ($T = 60$), the value returned is 5. For different values of $T$, we can use the following properties to estimate the number of VMs deployed at each pool.

$R"VMs\_Deployed\_Hot" =?[I = T]$
$R"VMs\_Deployed\_Warm" =?[I = T]$
$R"VMs\_Deployed\_Cold" =?[I = T]$

The graph presented in Figure 18 shows the expected number of VMs being deployed for different values of T. It is evident that always the number of VMs in hot pool is greater, where $zh$ reaches its limit rapidly before $zw$ and $zc$ respectively, because the RPDM tries always to find a hot PM first. The mean service time has a great impact on the results, by increasing its value, it could result in higher values of (zh, zw and zc), since each request takes much time to be served.

Unlike the *Instantaneous reward* properties, we can use *Steady-state reward* properties to compute reward in the long-run. To do so, the previous property of queue size can be written as follows: $R"queue\_size" =?[S]$ and it results in the value of $lq - 1$.

The last type of reward properties that can be used by PRISM, is the cumulative reward that associates a reward that is accumulated along the path until a bound T. For in-

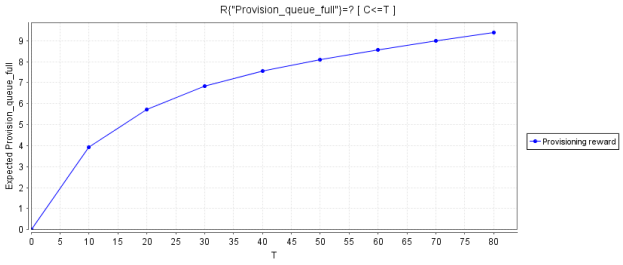


Figure 19: Expected number of requests provisioned after full queue.

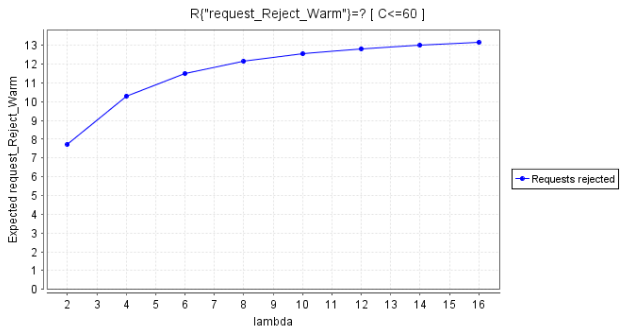


Figure 20: Expected number of requests rejected in warm.

stance, using the property $R$"$Provision\_queue\_full$" $=?[C <= T]$, we will be able to compute the expected number of requests being provisioned from the state where the global queue is full. The results over time T are presented in Figure 19.

Now, we want to use cumulative reward properties to reason on the requests being rejected at the level of each pool with respect to the value of the arrival rate $\lambda$. From the previous results, we choose the warm pool for this property, because it knows a medium number of requests being provisioned and served compared to hot and cold. For a fixed period of time of 60 minutes, we try to estimate the number of requests being rejected for different values of $\lambda$ using the following property:

$R$"$request\_Reject\_Warm$" $=?[C <= 60]$

The results of this property as returned by PRISM are depicted in Figure 20. We notice that as the value of $\lambda$ increases, the expected number of requests rejected in warm increases as well, because as much as requests arrive, more hot PMs start accepting requests and subsequently warm and cold PMs.

### 4.4 Power consumption analysis

The importance of cumulative reward properties can be clearly shown in the context of power consumption. We use it here for estimating power consumption of the three pools. It is assumed that a hot PM consumes an idle power $h_l$ when no VM is running, and the power consumption of a VM is assumed to be $v_a$. For the hot pool, a reward of $r_{(i,j,k)} = h_l + Kv_a$ is assigned to each state of the hot pool, where $k$ represents the number of VMs being deployed. The rest of rewards rates for warm and cold pools can be found at [11], and their rewards as interpreted in PRISM are presented in Figure 21.

We notice that power consumption rates for both warm and cold have much details, since they require much additional startup time to be ready for use. This is represented in the variables $yw$ and $yc$ that have three possible values. It is assumed here that $w_{l1} \leq w_{l2} \leq w_{l3} \leq h_1$, and it is the same case for cold pool: $c_{l1} \leq c_{l2} \leq c_{l3} \leq h_1$. The values as adapted from [11] are declared as global variables in PRISM (see Figure 22). The values ($w_{l1}, w_{l2}$, $w_{l3}$) are assumed to be within 20 - 50% of $h_l$, and the values($c_{l1}, c_{l2}$,$c_{l3}$) 0-40% of $h_l$. Given these values, we can estimate the power consumption at each pool using the following cumulative reward properties :

$R$"$Power\_hot\_PM$" $=?[C <= T]$

$R$"$Power\_warm\_PM$" $=?[C <= T]$

$R$"$Power\_cold\_PM$" $=?[C <= T]$

The results of these properties over time are presented in Figure 23. We see that power consumption in hot pool is much higher than warm and cold pools, due to the higher rates in hot. In addition, requests are provisioned more in hot, then warm and finally cold. These results can be also explained based on the previous graph (see Figure 18). We notice that after 10 time units, the power consumption in warm starts getting higher, due to a request being provisioned in warm. While warm power consumption could exceed 10% of hot power consumption by time $T = 80$, the cold power consumption stays in a low level.

## 5 Conclusion

In this paper we illustrated the use of probabilistic model checking as an effective framework for the evaluation and performance analysis of IaaS clouds. Using PRISM model checker, we implemented an analytical model that consists of many interactive sub-models. The model describes and quantifies the steps of provisioning and serving user requests on virtual machines (VMs), which are deployed on physical machines (PMs) regrouped in different pools. Using transient and steady properties, we were able to compute many important performance measures, such as rejection probability and time delay. In addition, using different types of reward properties, we were able to estimate many reward-based measures, especially the power performance trade-off of the IaaS cloud. The reliable estimations obtained can help cloud providers to get a better insight on cloud performance, thus avoiding SLA violation.

## References

[1] Aziz, A., Sanwal, K., Singhal, V., and Brayton, R. Model-checking continuous-time markov chains. *ACM Transactions on Computational Logic 01*, 1 (2000), 162–170.

```
rewards "Power_hot_PM"
(j=1 & zh>0) : h_l + zh*v_a;
endrewards
rewards "Power_warm_PM"
(j=2 & xw=0 & yw=0 & zw=0) : w_l1;
(j=2 & (xw>=0 & xw<=lw) & yw=1 & zw=0) : w_l3;
(j=2 & (xw>=0 & xw<=lw) & yw=2 & zw=0) : w_l2;
(j=2 & (xw>=0 & xw<=lw) & yw=3 & zw=0) : h_l;
(j=2 & xw=0 & yw=0 &(zw>=1 & zw<=m) ) : h_l + zw*v_a;
(j=2 & (xw>=0 & xw<=lw) & yw=1 &(zw>=1 & zw<=m-1) ) : h_l + zw*v_a;
(j=2 & (xw>=0 & xw<=lw) & yw=0 & zw=m) : h_l + m*v_a;
endrewards
rewards "Power_cold_PM"
(j=3 & xc=0 & yc=0 & zw=0) : c_l1;
(j=3 & (xc>=0 & xc<=lc) & yc=1 & zc=0) : c_l3;
(j=3 & (xc>=0 & xc<=lc) & yc=2 & zc=0) : c_l2;
(j=3 & (xc>=0 & xc<=lc) & yc=3 & zc=0) : h_l;
(j=3 & xc=0 & yc=0 &(zc>=1 & zc<=m) ) : h_l + zc*v_a;
(j=3 & (xc>=0 & xc<=lc) & yc=1 &(zc>=1 & zc<=m-1) ) : h_l + zc*v_a;
(j=3 & (xc>=0 & xc<=lc) & yc=0 & zc=m) : h_l + m*v_a;
endrewards
```

Figure 21: Power consumption rewards definition.

```
// Power hot pm
//power consumption in a hot PM (hl)
const double h_l=270;
//power consumption per VM
const double v_a = 16;
// Power warm pm
const double w_l1=54;
const double w_l2=100;
const double w_l3=135;
// Power cold pm
const double c_l1=0;
const double c_l2=50;
const double c_l3=108;
```

Figure 22: Power consumption rates.

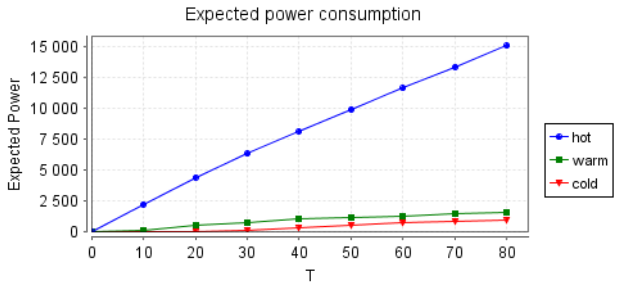


Figure 23: Power Consumption.

[2] Baier, C., Haverkort, B., Hermanns, H., and Katoen, J.-P. Model checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering 29*, 07 (2003), 524–541. https://doi.org/10.1109/tse.2003.1205180.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 05 (2003), 164–177.

[4] Bradley, J. T., Cloth, L., Hayden, R. A., Kloul, L., Reinecke, P., Siegle, M., Thomas, N., and Wolter, K. *Resilience Assessment and Evaluation of Computing Systems*. Springer, 2012, ch. Scalable Stochastic Modelling for Resilience. https://doi.org/10.1007/978-3-642-29032-9.

[5] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems 25*, 06 (2009), 599–616. https://doi.org/10.1016/j.future.2008.12.001.

[6] Chen, P., Xia, Y., Pang, S., and Li, J. A probabilistic model for performance analysis of cloud infrastructures. concurrency and computation. *Practice and Experience 27*, 17 (2015), 4784–4796. https://doi.org/10.1002/cpe.3462.

[7] DESHPANDE, P., SHARMA, S., AND PEDDOJU, S. Efficient multimedia data storage in cloud environment. *Informatica 39*, 04 (2014), 431–442.

[8] DUAN, Q. Cloud service performance evaluation: status, challenges, and opportunities – a survey from the system modeling perspective. *Digital Communications and Networks 03*, 02 (2017), 101–111. https://doi.org/10.1016/j.dcan.2016.12.002.

[9] EVANGELIDIS, A., D.PARKER, AND BAHSOON, R. Performance modelling and verification of cloud-based auto-scaling policies. *Future Generation Computer Systems 87*, 04 (2018), 629–638. https://doi.org/10.1109/ccgrid.2017.39.

[10] GHOSH, R. *Scalable Stochastic Models for Cloud Services*. PhD thesis, Duke University, 2012.

[11] GHOSH, R., NAIKY, V., AND TRIVEDI, K. Power-performance trade-offs in iaas cloud: A scalable analytic approach. In *IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)* (2011), pp. 152–157. https://doi.org/10.1109/dsnw.2011.5958802.

[12] GHOSH, R., TRIVEDI, K., NAIK, V. K., AND KIM, D. End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach. In *2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing* (2010), pp. pp. 125–132. https://doi.org/10.1109/prdc.2010.30.

[13] HANSSON, H., AND JONSSON, B. logic for reasoning about time and reliability. *Formal aspects of Computing 6*, 5 (1994), 512–535.

[14] HINTON, A., KWIATKOWSKA, M., NORMAN, G., AND PARKER, D. Prism: A tool for automatic verification of probabilistic systems. In *TACAS* (2006), LNCS, vol. 3920, Springer, Berlin, Heidelberg, pp. 441–444. https://doi.org/10.1007/11691372_29.

[15] JOHNSON, K., REED, S., AND CALINESCU, R. Specification and quantitative analysis of probabilistic cloud deployment patterns. In *HVC* (2012), LNCS, vol. 7261, Springer, Berlin, Heidelberg, pp. 145–159. https://doi.org/10.1007/978-3-642-34188-5_14.

[16] KHAZAEI, H., MISIC, J., AND MISIC, B. Performance analysis of cloud computing centers using m/g/m/m+r queuing systems. *IEEE Transactions on Parallel and Distributed System 23*, 05 (2012), 936–943. https://doi.org/10.1109/tpds.2011.199.

[17] KHAZAEI, H., MISIC, J., AND MISIC, V. A fine-grained performance model of cloud computing centers. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS 24*, 11 (2013), 2138–2147. https://doi.org/10.1109/tpds.2012.280.

[18] KIKUCHI, S., AND MATSUMOTO, Y. Performance modeling of concurrent live migration operations in cloud computing systems using prism probabilistic model checker. In *IEEE 4th International Conference on Cloud Computing* (2011), pp. 49–56. https://doi.org/10.1109/cloud.2011.48.

[19] LI, Z., ZHANG, H., OBRIEN, L., CAI, R., AND FLIN, S. On evaluating commercial cloud services: A systematic review. *Journal of Systems and Software 86*, 09 (2013), 2371–2393. https://doi.org/10.1016/j.jss.2013.04.021.

[20] LI, Z., ZHANG, H., OBRIEN, L., CAI, R., AND FLIN, S. Stochastic modeling and quality evaluation of infrastructure-as-a-service clouds. *IEEE Transactions on Automation Science and Engineering 12*, 01 (2015), 162–170. https://doi.org/10.1109/tase.2013.2276477.

[21] MAHFOUD, Z., AND NOUALI-TABOUDJEMAT, N. Consistency in cloud-based database systems. *Informatica 43*, 03 (2019), 313–320. https://doi.org/10.31449/inf.v43i1.2650.

[22] PONNURAMU, V., AND TAMILSELVAN, L. Secured storage for dynamic data in cloud. *Informatica 40*, 01 (2016), 53–62.

[23] VAQUERO, L., RODERO-MERINO, L., CACERES, J., AND LINDNERS, M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review 39*, 01 (2009), 50–55. https://doi.org/10.1145/1496091.1496100.

[24] WIDED, A., AND OKBA, K. A novel agent based load balancing model for maximizing resource utilization in grid computing. *Informatica 43*, 03 (2019), 355–262. https://doi.org/10.31449/inf.v43i3.2944.