Pós-Graduação em Ciência da Computação

Danilo Mendonça Oliveira

A modeling framework for infrastructure planning of Workflow-as-a-Service environments

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Danilo Mendonça Oliveira

A modeling framework for infrastructure planning of Workflow-as-a-Service environments

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

Advisor: Paulo Romero Martins Maciel
Co-advisor: Nelson Souto Rosa

Recife
2019

FICHA

Tese de Doutorado apresentada por **Danilo Mendonça Oliveira** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**A modeling framework for infrastructure planning of Workflow-as-a-Service environments**" **Advisor: Paulo Romero Martins Maciel** e aprovada pela Banca Examinadora formada pelos professores:

Prof. XXCentro de Informática/ UFPE

Profa. YYGerência Educacional de Informática / IFPB

Profa. TTCentro de Informática / UFPE

Prof. Advisor: Paulo Romero Martins Maciel
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 21 de Janeiro de 2019.

Prof. Aluizio Fausto Ribeiro Araújo
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

ACKNOWLEDGEMENTS

I thank and give all credit of this work to my God and Lord. Thank you, Celestial Father, for the gift of life and for providing me strength and wisdom to accomplish this work.

I want to thank my advisors, Paulo Maciel and Nelson Rosa, for guiding me in this research and giving me invaluable counseling needed for conducting it successfully. My sincere thanks also go to Andre Brinkmann and Tim Suess, for accepting me in your research group at the Johannes Gutenberg University and for sharing your knowledge with me.

I thank my family for supporting me in this journey. Thank you, mom and dad, for being the absolute best parents in the world. I thank my wife Leilane for nourishing me with your love, and for enabling me to endure these difficult days. I thank my sister and my brother-in-law for all you did for me during my academic life.

My gratitude also goes to all my friends and fellow researchers from the Center of Informatics of UFPE, from the Center for Data Processing of the Johannes Gutenberg University, and the Big Storage Project.

“If you optimize everything, you will always be unhappy."
(Donald Knuth)

RESUMO

Considerando as características de provisionamento dinâmico e a ilusão de recursos ilimitados, nuvens computacionais estão se tornando uma alternativa popular para executar workflows científicos. Numa nuvem computacional para processamento de workflows, o desempenho do sistema é altamente influenciado por dois fatores: a estratégia de escalonamento e falhas de componentes. Falhas numa nuvem podem afetar vários usuários simultaneamente e diminuir o número de recursos computacionais disponíveis. Uma estratégia de escalonamento ruim pode aumentar o makespan e diminuir a utilização das máquinas físicas. Neste trabalho, nós propomos um framework de modelagem e um conjunto de modelos formais e métodos para auxiliar o planejamento de infraestrutura de nuvens do tipo Workflow-as-a-Service. Este framework de modelagem provê auxílio às tarefas de: i) planejar a implantação de aplicações de workflow em nuvens computacionais a fim de maximizar métricas de desempenho e confiabilidade; ii) planejar os arranjos de redundância na infraestrutura de nuvem a fim de reduzir o gasto de aquisição e, ao mesmo tempo, garantir requisitos de disponibilidade; iii) identificar gargalos de disponibilidade e habilitar a priorização dos componentes mais críticos. Nós conduzimos três estudos de caso a fim de ilustrar e validar o framework de modelagem proposto. O primeiro estudo de caso emprega um modelo hierárquico de disponibilidade usando modelos RBD e DRBD e aplica métodos de análise de sensibilidade para detectar os parâmetros mais influentes na métrica considerada. O segundo estudo de caso extende o anterior, ao modelar a infraestrutura de nuvem como um problema de alocação de redundância (RAP - redundancy allocation problem). Para minimizar o custo de aquisição enquanto a disponibilidade é maximizada, nós combinamos o método de busca local com o algoritmo de bisecção. No último estudo de caso, nós propomos um método para a otimização do escalonamento de workflows em nuvem. Este estudo de caso adota um algoritmo meta-heurístico acoplado a um modelo de performabilidade que provê a função fitness das soluções exploradas. Os resultados experimentais obtidos nos estudos de caso demonstram a eficácia do framework em auxiliar tarefas de planejamento de infraestrutura, permitindo que provedores de nuvem maximizem a utilização de recursos, reduzam custos operacionais, e garantam requisitos de SLA.

Palavras-chaves: planejamento de infraestrutura, computação em nuvem, otimização combinatória, modelagem estocástica, simulação de eventos discretos, redes de Petri, diagramas de bloco de confiabilidade

ABSTRACT

Given the characteristics of dynamic provisioning and the illusion of unlimited resources, "the cloud" is becoming a popular alternative for running scientific workflows. In a cloud system for processing workflow applications, the performance of the system is heavily influenced by two factors: the scheduling strategy and failure of components. Failures in a cloud system can simultaneously affect several users and depreciate the number of available computing resources. A bad scheduling strategy can increase the expected makespan and the idle time of physical machines. In this work, we propose a modeling framework, and a set of formal models and methods for supporting the infrastructure planning of Workflow-as-a-Service clouds. This modeling framework supports the tasks of: i) planning the deployment of workflow applications in computational clouds in order to maximize performance and reliability metrics; ii) planning the redundancy arrangements in the cloud infrastructure in order to reduce the acquisition cost while satisfying availability requirements; iii) identifying availability bottlenecks and enabling the prioritization of critical components for improvement. We conducted three case studies in order to illustrate and validate the proposed modeling framework. The first case study employs a comprehensive hierarchical availability model using RBD and DRBD models and applied sensitivity analysis methods in order to find the most influential parameters. The second case study extends the previous one by modeling a cloud infrastructure as an instance of the redundancy allocation problem (RAP). To minimize the acquisition cost while maximizing the availability of the system, we proposed the combined use of a local-search algorithm and the bisection method. In the last case study, we optimize the scheduling of scientific cloud workflows. This case study comprises the use of a metaheuristic algorithm coupled with a performability model that provides the fitnesses of the explored solutions. The experimental results obtained in all case studies have proven the framework effectiveness on aiding planning infrastructures tasks, allowing cloud providers to maximize resource utilization, reduce operational costs, and ensure SLA requirements.

Key-words: infrastructure planning, cloud computing, combinatorial optimization, stochastic modeling, discrete event simulation, Petri nets, reliability block diagrams

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

API	Application Program Interface
CC	Cluster Controller
CLC	Cloud Controller
CTMC	Capacity Oriented Availability
CPN	Colored Petri Net
CTMC	Continous Time Markov Chain
DaaS	Data-as-a-Service
DoE	Design of Experiments
EC2	Amazon Elastic Compute Cloud
DAG	Directed Acyclic Graph
DFT	Dynamic Fault Tree
DRBD	Dynamic Reliability Block Diagram
DTMC	Discrete Time Markov Chain
EDSPN	Extended Deterministic Stochastic Petri nets
FT	Fault Tree
HEFT	Heterogeneous Earliest Finish Time
IaaS	Infrastructure-as-a-Service
JVM	Java Virtual Machine
PaaS	Platform-as-a-Service
MSL	Mercury Scripting Language
MTBF	Mean Time Between Failures
MTTR	Mean Time to Failure
MTTF	Mean Time to Repair
NC	Node Controller
PMX	Partially Mapped Crossover
PN	Petri Net
OS	Operating System
RAP	Redundancy Allocation Problem
RBD	Reliability Block Diagram
S3	Amazon Simple Storage

SaaS Software-as-a-Service

SAN Storage Area Network

SC Storage Controller

SLA Service Level Agreement

SPN Stochastic Petri Net

SRIP Single Replication in Parallel

VM Virtual Machine

WaaS Workflow-as-a-Service

CONTENTS

1 INTRODUCTION

In the past decades, computers become an invaluable asset for scientists in many fields of human knowledge. Simulation models are useful when experiments in the real world are too difficult or costly to execute, or when the phenomenon of interest is impossible to reproduce (for instance, in studies about the origin of the universe). Such models are often computationally intensive and require an execution environment composed of many processing units. Many computational scientific applications can be expressed as workflows, i.e., a set of subtasks with data and control-flow dependencies between them. In such applications, the scheduling of tasks to the processing units plays a vital role in the performance of the system, but finding an optimal schedule is an NP-Hard problem (BOOK et al., 1980).

Nowadays, cloud computing has been attracting attention as a platform for running scientific applications (HOFFA et al., 2008; VÖCKLER et al., 2011; JUVE et al., 2009). The pay-per-use model eliminates the need for upfront investment on a cluster/supercomputer. Moreover, cloud users do not need to worry about managing the underlying hardware infrastructure. While this model makes things more convenient for the user, this task becomes a severe issue for cloud providers, who need to guarantee reliability and performance levels specified by a Service Level Agreement (SLA).

The use of formal models from the earliest stages of development helps to ensure that non-functional requirements such as performance, reliability, availability, and security are met. (MENASCE et al., 2004). Formal models also can support the activities of infrastructure and capacity planning of cloud environments (SOUSA; MACIEL; ARAÚJO, 2009; CALLOU et al., 2012; DANTAS et al., 2015). More specifically, modeling techniques support the tasks of: i) comparing multiple alternative designs concerning operational costs and SLA-related quantitative metrics; ii) testing the system's behavior under different combinations of workload and configuration parameters as means of fine-tuning and identification of performance and availability bottlenecks. By contrast, assessing cloud performance and reliability metrics via measurement-based evaluation is often prohibitive in practice. Furthermore, model-based evaluation is the only option in the early stages of development, when a minimum viable deployment of the system is not available (JAIN, 1990).

Nevertheless, many technical challenges arise when carrying out model-based infrastructure planning of computational clouds, which are designed for workflow applications. In this thesis, we identify those challenges and propose a modeling framework for the infrastructure planning of Workflow-as-a-Service clouds (WaaS) (WANG et al., 2014).

1.1 MOTIVATION AND JUSTIFICATION

Creating availability models of complex systems such as computational clouds requires a proper set of techniques and methods. Using only combinatorial models such as Reliability Block Diagrams (RBD) (XU; XING, 2007) and Fault Trees (FT) (VESELY et al., 1981) may introduce unrealistic assumptions and does not allow to represent interactions between components, such as failure dependencies and redundancy arrangements. On the other hand, state-space based models such as Continuous Time Markov Chains (CTMC) (MARKOV, 1906) and Stochastic Petri Nets (SPN) (GERMAN, 2000) enable representing such dependencies but suffer from state-explosion, and simplifications are often made to avoid complex models. The literature often suggests hierarchical modeling as a means to obtain the advantages of both categories.

The complexity of cloud infrastructures (i.e., the large number of hardware and software components and their interdependence relationships) also raises the need for performance evaluation methods that consider the failure of components. A performance study that disregards reliability aspects may not give accurate results since failures of physical and virtual machines can increase waiting times and decrease throughput (QIU et al., 2015). Given the difficulties of assessing the performance degradation caused by failures in a cloud environment via measurement-based evaluation, state-space based models are the most commonly used technique in the literature for performability evaluation of cloud and grid systems (GHOSH et al., 2010; QIU et al., 2015; WANG; CHANG; LIU, 2016; RAEI; YAZDANI, 2017).

Evaluating the effects of both scheduling and hardware/virtual machine failures on the performability of workflow applications in cloud environments is a challenging task for space-state based models due to the high number of states to be considered. Moreover, the exponential distribution may not be a good fit for computation times in workflows. Given this intrinsic limitation of space-based models, many existing research efforts towards the modeling of cloud applications employ discrete event simulators. CloudSim (CALHEIROS et al., 2011) is the most widely adopted simulation framework in the cloud computing literature. Thanks to its flexible architecture, many extensions were proposed to address aspects not initially implemented by CloudSim. Some of the extensions feature auto-scaling (CAI; LI; LI, 2017), federated clouds (KOHNE et al., 2014), fault tolerance mechanisms (DAVIS et al., 2017; ZHOU et al., 2013; ALWABEL; WALTERS; WILLS, 2015), and workflow applications (CHEN; DEELMAN, 2012; BUX; LESER, 2015; MALAWSKI et al., 2015).

Besides the intricacies of performance and availability modeling of cloud workflow applications, there is a need for integrating formal model evaluation to infrastructure planning methods and tools. To be more specific, a software modeling tool to be employed in such study may offer the following features: i) multiple modeling formalisms and hierarchical modeling; ii) high-level formalisms, such as high-level Petri nets, Dynamic Reliability Block Diagrams (DRBD), and so on.; iii) support to symbolic evaluation (i.e., the

ability to represent input parameters and output metrics as variables and expressions); iv) and exporting the evaluator engine as a programmable interface. Hierarchical modeling and high-level models offer means to alleviate the difficulties faced when modeling WaaS clouds. Symbolic model evaluation and accessing the model evaluator engine via a programming language allow to couple the modeling strategy to infrastructure planning methods and tools.

1.2 OBJECTIVES

Reducing acquisition and operational costs while maintaining user's agreements is a goal pursued by cloud providers. The primary goal of this work is to propose a modeling framework and a cohesive set of formal models and methods for assisting the infrastructure planning of Workflow-as-a-Service clouds. The proposed modeling framework will support the tasks of: i) planning the deployment of workflow applications in computational clouds in order to maximize performability metrics; ii) planning the redundancy arrangements in the cloud infrastructure to reduce the acquisition cost while satisfying availability requirements; iii) identifying availability bottlenecks and enabling the prioritization of critical components for improvement.

In order to accomplish the stated primary goal, the following list of specific goals is set:

- To propose a novel modeling runtime for the Mercury tool aimed at flexible hierarchical modeling and higher-level modeling formalisms;
- To develop a method for identifying performance and reliability bottlenecks of cloud infrastructures supported by formal modeling and sensitivity analysis;
- To study and devise new modeling and meta-modeling techniques for performance and availability modeling of cloud workflow applications;
- To investigate optimization methods for scientific workflows applications running in a cloud environment;

For making the best use of the proposed framework, models and methods, knowledge in both computer programming and formal modeling is required. Nonetheless, a cloud engineer equipped with such a skillset can build software tools targeted at non-technical cloud users and stakeholders. Such tools may provide a user-friendly interface and allow users to modify specific model parameters and verify the expected changes in the response metrics, without exposing the details about the adopted formal models.

1.3 ORGANIZATION OF THIS WORK

The remaining of this thesis is organized as follows. Chapter 2 provides an overview of the concepts and theory covered in this work. Chapter 3 presents a literature review concerning

the proposed framework, methodology, and conducted case studies. Chapter 4 describes the proposed framework and its supporting methodology for infrastructure planning of WaaS clouds.

Chapter 5 presents an overview of the WaaS cloud architecture and suggests formal models for availability and performance evaluation of such environments. In Chapter 6, we conduct a series of case studies for validating and illustrating the proposed approach. Finally, in Chapter 7, we conclude this work and suggest work that can be done based on this approach.

2 BACKGROUND

This chapter presents the basic theory and concepts related to the contents of this work. Section 2.1 gives an overview of cloud computing. Scientific workflows applications are described in Section 2.2. Section 2.3 gives an introduction to dependability concepts. Finally, sections 2.4, 2.5, and 2.6 introduce important formalisms for availability and performance modeling, namely, Reliability Block Diagrams, Continuous Time Markov chains, and Petri nets.

2.1 CLOUD COMPUTING

Cloud computing is a computational model in which resources such as processing, network, storage, and softwares are offered via the Internet and can be accessed remotely (ARMBRUST et al., 2010). This model allows users to obtain resources by demand, with elasticity, at a low cost and delivered as traditional services such as water, gas, electricity, and telephony (DINH et al., 2013). Among the many definitions of cloud computing existing in the literature, we present the definition proposed by (VAQUERO et al., 2008):

> *Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms, or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), also allowing for optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model, in which guarantees are offered by the Infrastructure Provider by means of a customized Service Level Agreement.*

Cloud computing is a recent computational paradigm, but it was created using existent technologies as its foundations: virtualization, grid computing, and utility computing. The computational resources of a grid are provisioned to clients as virtual machines so each customer only pays for the resources that he/she consume, instead of paying a fixed fee.

This computational model has some characteristics that turn it attractive to a significant number of corporations. If a business wants to launch a service on the Internet, it can rent some set virtual machines and deploy the service on them, without having to own the server infrastructure. This provisioning model helps to reduce the time-to-market of services and products and to make them more competitive. Moreover, the elastic provisioning of resources by a cloud allows a system to react to changes in the workload, without the need of an infrastructure oversizing. For instance, consider as an example the Amazon Data Center before becoming a cloud provider. To support infrequent workload peaks, the infrastructure was oversized in such a way that the average utilization was only 10% (JEFF...,). By relying on cloud elasticity, a system can allocate more virtualized resources

on demand to handle a high workload and relinquish them when they are not necessary anymore.

We can characterize cloud systems regarding two aspects. The first aspect defines if the system is maintained by its user, by a third party agent, or by a combination of both. By this point of view, we have three classes of clouds:

- A **private cloud** is designed to be used by a single organization. An external agent can provide a private cloud infrastructure, or the organization itself can manage it. The latter option is preferred to run critical data systems (e.g., banks, government, and military agencies), in which data leakage is not tolerated. The disadvantage of this approach is the significant monetary cost involved in purchasing the data-center components. However, this option enables better management and consolidation of resources, allowing more efficient usage of the available hardware (ZHAO; FIGUEIREDO, 2007; KIM; MACHIDA; TRIVEDI, 2009);
- **Public clouds** are clouds maintained by service providers that offer their computational resources to other organizations. The usage of public clouds removes the need for upfront costs on physical servers and provides elasticity (the ability to acquire more virtualized servers on demand to adapt to an increasing workload). An additional advantage is that it is up to the cloud provider to maintain the system and ensuring that SLAs are satisfied (CAROLAN et al., 2009);
- A **hybrid cloud** is a combination of both previous schemes to get the best of each approach. Critical data can be stored in the private part of the cloud, at the same time that it is possible to scale the service beyond the limits of a typical private cloud. On the other hand, greater complexity in cloud management accompanies those benefits and finding the better partitioning between public and private components is challenging (ZHANG; CHENG; BOUTABA, 2010).

The other aspect in which clouds can be classified is regarding their business model:

- **Infrastructure as a Service (IaaS)** - This business model partitions the resources of a data center among users as virtualized resources. The users of IaaS clouds get raw virtual machines and are responsible for maintaining the whole software stack of the services that will run on them;
- **Platform as a Service (PaaS)** - In this service model, the cloud provider abstracts the virtual machines and the operating systems, and offers to its users a high-level programming platform. The advantage of this design is the transparent scaling of the underlying infrastructure at runtime (VAQUERO et al., 2008);
- **Software as a Service (SaaS)** - This business model targets everyday Internet users, instead of system administrators and programmers. In this model, standard

software and data are stored on the cloud and can be accessed by any computer connected to the Internet via a Web interface.

Besides the models mentioned above, there is the **Data as a Service (DaaS)** model (TRUONG; DUSTDAR, 2009), in which the cloud provides a secondary storage service, ensuring high reliability and data integrity. A DaaS service can be viewed as IaaS or SaaS, depending on the context in which it is used. For instance, consider the storage and backup service of Dropbox [1]. Its clients perceive this service as a SaaS cloud. However, the Dropbox itself uses the Amazon S3 (Simple Storage) service [2] to store client data. From the Dropbox viewpoint, the Amazon storage is an IaaS service.

2.2 SCIENTIFIC WORKFLOW APPLICATIONS

A scientific workflow consists of a set of computing and I/O intensive tasks with precedence constraints between them. Scientific workflows can be represented by a directed acyclic graph (DAG). A DAG G is defined by a tuple $\{T, E\}$, where $T = \{T_1, T_2, \ldots, T_n\}$ is the set of tasks and $E = \{e_1, e_2, \ldots, e_m\}$ is the set of precedence constraints. Each $e_i = (T_a, T_b)$ tuple denotes that task T_b starts to execute after T_a finishes and have sent some input data to T_b.

A scheduler should map tasks to a set of processors according to some predefined goals such as utilization of resources, makespan (the total length of a schedule), throughput, meeting of deadlines, etc. A scheduling algorithm can either require as input the number of processing units or try to find an optimal/near-optimal number of processing units in conjunction with the mapping of tasks. The latter case is harder since it leads to an increased search space. Additionally, while increasing the number of processors can shorten the makespan of an individual job, it also may increase the idle time of processors due to the precedence constraints between tasks (RODRIGUEZ; BUYYA, 2017).

Figure 1 shows an example of an application modeled as a DAG (obtained from (XU et al., 2015)). The node weights are the computing times, and the edge weights are the communication times, i.e., the time for sending the results needed by a dependent task running on a different processing node. Figures 2 a) and b) show two different schedulings for the DAG of Figure 1.

Cloud computing provides computational resources such as servers, networks, storage, applications, and services, with little management effort and few interactions with the service provider (MELL; GRANCE et al., 2011). In this way, researchers without access to a supercomputer or cluster can run scientific workflows by allocating virtualized resources in a cloud for a given period (WANG et al., 2008). Since many practical scheduling problems are either NP-Hard or NP-Complete, much effort is necessary to apply and adapt meta-

[1] <https://dropbox.com>
[2] <http://aws.amazon.com/s3/>

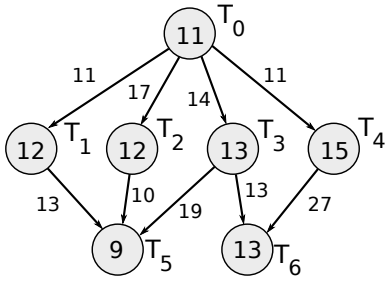


Figure 1 – Example of Directed Acyclic Graph (XU et al., 2015)

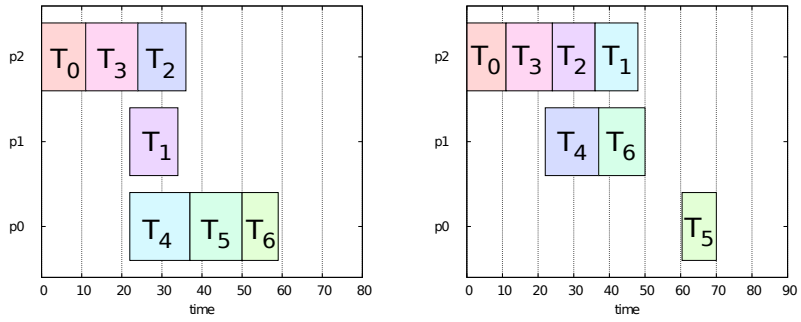


Figure 2 – Schedulings of the DAG shown in Figure 1

heuristic algorithms to schedule tasks in cloud computing environments. The contributions made in this field include the following:

- Devising heuristics algorithms able to provide near-optimal solutions under certain constraints (LIN; WU; WANG, 2016);
- Adapting nature inspired and evolutionary algorithms for this problem, such as Genetic algorithm (GU et al., 2012) (ZHENG et al., 2011) (ZHAO et al., 2009), Honey bee colony (LD; KRISHNA, 2013) (BITAM, 2012), ant colony (CHEN; ZHANG, 2013) (TAWFEEK et al., 2013), and Fish Swarm (ZHAO; TIAN, 2014);
- Dealing with a heterogeneous system (processors with different computing power) (PANDA; JANA, 2015);
- Dealing with conflicting aspects of a multi-cost objective function (e.g., energy versus makespan) (MEZMAZ et al., 2011).

2.3 DEPENDABILITY

The idea of dependability is defined by a set of measures that indicate the capability of a system to continue working correctly, despite the presence of *failures* and *errors*. We consider that a system is free of failures when it delivers its intended, service regarding functional and non-functional requirements, as perceived by the users. An *error* is a portion

of the system state that can lead the system to the failure, and a *fault* is the hypothesized cause of an error (RANDELL, 1998).

We can divide the dependability related concepts into three groups (AVIZIENIS et al., 2001): *attributes*, *means*, and *threats*. The dependability attributes are the set of metrics which enable obtaining quantitative measures that are fundamental for analyzing the offered services. *"Dependability means"* are the techniques available for increasing the dependability of a system, i.e., by removing/preventing/forecasting failures or making the system tolerant to their presence. Fault tolerance mechanisms rely on redundancy techniques since identical components can be used to replace failed ones at runtime (STORM, 2012). Finally, the threats correspond to the failures, errors, and faults of a system.

The dependability attributes are reliability, availability, maintainability, performability, security, testability, confidentiality, and integrity (LAPRIE, 1992). Among these attributes, the most commonly considered in dependability studies are reliability and availability. The reliability metric measures the probability that a system will deliver its function correctly during a specified time interval, without the occurrence of failures in this interval. The following formula provides the reliability of a system:

$$R(t) = P\{T > t\} = e^{-\int_0^t \lambda(t)dt}, t \geq 0 \ , \tag{2.1}$$

where T is a random variable that represents the time to failure of the system and $\lambda(t)$ is known as the hazard function. We define the availability metric as (MACIEL et al., 2012a):

$$A = \frac{MTTF}{MTTR + MTTF} \cong \frac{MTBF}{MTTR + MTBF} \tag{2.2}$$

where MTTF is the mean time to failure, MTBF is the mean time between failures, and MTTR is the mean time to repair.

2.3.1 Dependability modeling

Formal models for dependability evaluation are classified into two categories: combinatorial models and state-space based models (MACIEL et al., 2012b). Combinatorial models represent the conditions that may result in an operational/failed state, based on the structural relationships of its subcomponents. State-space based models describe the system behavior by means of discrete states and the events that provoke transition between states. Some examples of combinatorial models include Reliability Block Diagrams and Fault Trees. Continuous Time Markov Chains, Queueing Networks, and Stochastic Petri nets are examples of state-space based models.

There are tradeoffs between the mentioned modeling categories (MALHOTRA; TRIVEDI, 1994). Combinatorial models are more straightforward to create, and the analysis methods are (in general) faster. However, their modeling capacity is more limited. When using such models, the modeler should assume that components are stochastically independent.

State-space based models are more complicated to create and evaluate, but they possess more expressiveness power than combinatorial models. With state-space based models, a modeler can represent more complex scenarios such as redundancy arrangements (in cold and warm standby modes) (LOGOTHETIS; TRIVEDI, 1995).

By employing **hierarchical modeling** techniques, it is possible to combine combinatorial and state-space models and achieve the best of both worlds (MALHOTRA; TRIVEDI, 1993). Different modeling formalisms, or different models within the same formalism, can be combined in different abstraction levels, resulting in bigger hierarchical models. The top-level model represents the system at an architectural level (the most significant modules in the highest level), while lower level models describe the detailed behavior of each subsystem. Hierarchical models have been used to deal with the complexity of many application areas, such as sensor networks (KIM; GHOSH; TRIVEDI, 2010), telecommunication infrastructure (TRIVEDI et al., 2006), and private clouds (DANTAS et al., 2012).

2.4 RELIABILITY BLOCK DIAGRAM

An RBD model specifies which combinations of failed and active components can yield a proper system operation (i.e.: it specifies the *operational mode* of the system). Visually, it is represented by a graph with a *start* vertex and an *end* vertex, connected to the component vertices, which are represented by a rectangular shape and account for the system components or subsystems. The nodes can be connected following specific structures, such as series, parallel, bridge, k-out-of-n, or a combination of the previous structures (KUO; ZUO, 2003). In an RBD model, the system is considered functional (available or reliable) if there is at least one path composed of functional components, from the start vertex to the end vertex.

Figures 3 a) to c) show some examples. Figure 3 a) show an RBD model with a series arrangement of the components C1, C2 and C3, Figure 3 b) shows a parallel arrangement, and Figure 3 c) shows a combination of series/parallel structures. It is important to stress that the organization of the components in the RBD model does not necessarily represent the physical topology of the real system. For a series structure composed of n independent components, the probability for the system to be operational is:

$$P\{\phi(\boldsymbol{x}) = 1\} = P\{\phi(x_1, x_2, \ldots, x_i, \ldots, x_n) = 1\}$$

$$= \prod_{i=1}^{n} P\{x_i = 1\} = \prod_{i=1}^{n} p_i \tag{2.3}$$

where $p_i = P\{x_i = 1\}$ are the functioning probabilities of blocks b_i. For a parallel structure composed of n identical and independent components, the probability for the system to be operational is:

$$P\{\phi(\boldsymbol{x}) = 1\} = P\{\phi(x_1, x_2, \ldots, x_i, \ldots, x_n) = 1\}$$

$$= 1 - \prod_{i=1}^{n} P\{x_i = 0\} = 1 - \prod_{i=1}^{n}(1 - P\{x_i = 1\})$$

$$= P\{\phi(\boldsymbol{x}) = 1\} = 1 - \prod_{i=1}^{n}(1 - p_i) \tag{2.4}$$

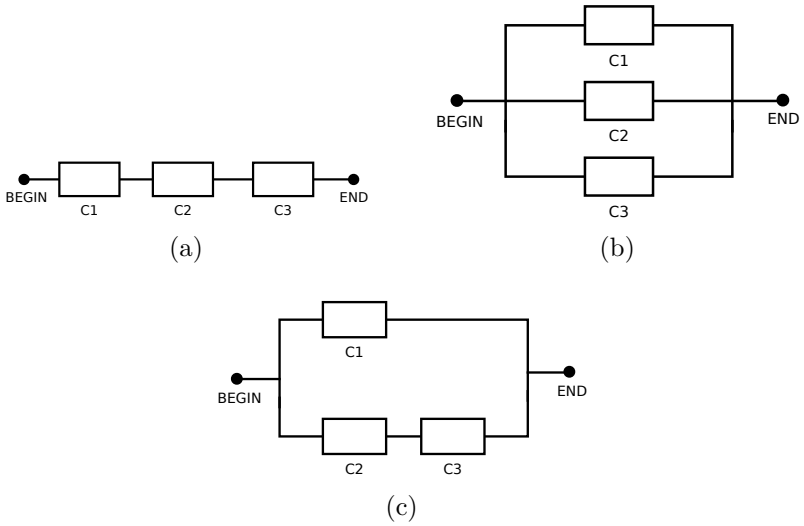


Figure 3 – RBD examples

In an RBD model, it is possible that one component appears in more than one place. However, when there is a repeated component, the evaluation of the RBD will not be straightforward. Instead, we must perform a pivotal decomposition in the block (KUO; ZUO, 2003). Consider the RBD of Figure 4. It can be noticed that the C component appears in two places. If we solve this model in a straightforward way, it will be as if the two C blocks are different components with the same failure and repair rates. To solve this model, we must factor the C component, generating two submodels. The first submodel represents the case when C is unavailable, and the second represents the case when C is functional. We obtain the availability of the original model using the expression:

$$A = A_C \times A_1 + (1 - A_C) \times A_2,$$

where A_C is the availability of the component, A_1 is the availability of the first submodel, and A_2 is the availability of the second submodel.

2.4.1 Dynamic Reliability Block Diagram

Dynamic Reliability Block Diagrams are state-space based models used to evaluate availability and reliability metrics, allowing the system to be specified using the RBD

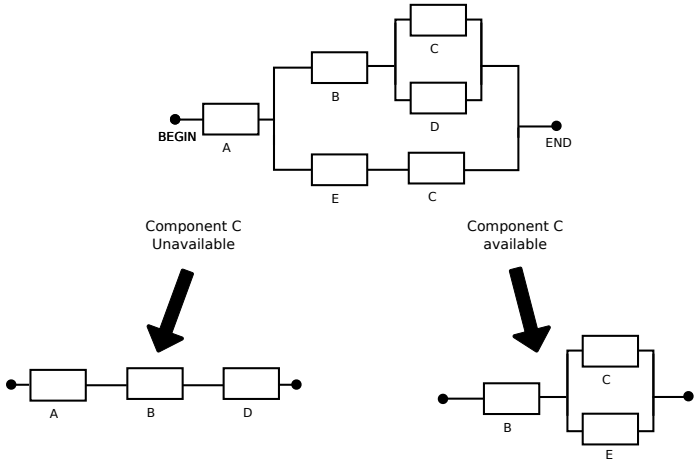


Figure 4 – Pivotal decomposition of an RBD

notation (DISTEFANO; XING, 2006). At the same time, DRBDs allows specifying the dependency relationships between components according to failure and repair events. Such relationships cannot be represented in pure combinatorial models like RBDs or Fault Trees (XU; XING, 2007). The DRBD notation offers two special control blocks for representing component dependencies: SDEP (state-based dependency) and SPARE (spare-part relationship). The first block is used to specify that the failure of a component causes the immediate failure of one or more components. The SPARE control block is used to describe redundancy arrangements in hot, warm or cold standby. Those redundancy strategies will be explained later in Section 4.2.2.

Figure 5 illustrates a DRBD model similar to a traditional RBD, in the sense that it contains groupings of blocks in series and parallel. However, we can use the SDEP and SPARE control blocks to make explicit the relationships between the blocks. In this model, there is a dependency of failure between blocks A and B, which implies that the failure of block A leads to the immediate failure of block B. The opposite is not true as the failure of block B does not affect block A. Blocks C and D are configured to work in warm standby redundancy. This indicates that block D, despite being functional, is not able to receive a workload while block C is functioning. When block C fails, the system detects this event and activates block D to receive the workload.

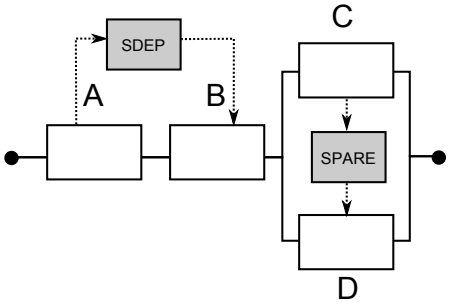


Figure 5 – Example of DRBD model

2.5 CONTINUOUS TIME MARKOV CHAINS

Markov chains are a special class of stochastic process that were first introduced by Andrey Markov in 1906 (MARKOV, 1906). This formalism represents a convenient way to represent dynamic properties of computational systems (BOLCH et al., 2006a). Markov chains are a foundation where many performance and dependability evaluation techniques are built (MENASCE et al., 2004). As it will be shown in Section 2.6, some stochastic Petri nets can be evaluated by the analysis of the embedded Markov chain. Similarly, many properties and theorems on the field of queueing theory are proven through Markovian models (KLEINROCK, 1976).

Before introducing the definition of a Markov chain, we present the basic definition of stochastic processes:

Definition 1 *A stochastic process is defined by a family of random variables $X_t : t \in T$, where each random variable X_t is indexed by a parameter t, and the set of all possible values for X_t is the state-space of the stochastic process.*

In general, the t parameter is defined as a temporal parameter, if the following condition is satisfied: $T \subseteq \mathbb{R}^+ = [0, \infty)$. If the set T is discrete, the process is labeled as a discrete-time stochastic process, otherwise it is a continuous-time process. Similarly, a stochastic process can be labeled as continuous or discrete state process depending on its state-space set S. A discrete-space stochastic process is classified as a **chain**.

A stochastic process is classified as a **Markovian process** if the following property holds:

Definition 2 *A stochastic process $X_t : t \in T$ is a Markovian process if, for each $0 = t_0 < t_1 < \ldots t_n < t_{n+1}$ and, for each $s_i \in S$, the probability $P(X_{t_{n+1}} \leq s_{n+1})$ depends only on the last value for X_{t_n}, and does not depends on the previous values $X_{t_0}, X_{t_1}, \ldots X_{t_{n-1}}$. That is:*

$$P(X_{t_{n+1}} \leq s_{n+1} \mid X_{t_n} = s_n, X_{t_{n-1}} = s_{n-1}, \ldots, X_{t_0} = s_0) =$$
$$P(X_{t_{n+1}} \leq s_{n+1} \mid X_{t_n} = s_n)$$

In other words, this property defines that, once a system is in a specific state, the transition to the next state depends only on this state. The history of previous states the system traversed to reach the current state is completely forgoten. For this reason, this fact is referred to as the **memoryless property**. Markovian process with a discrete state-space S are named **Markov chains**, and they are further classified into Discrete Time Markov Chains (DTMC) and Continuous Time Markov Chains, according to the time set T.

CTMCs can be represented by means of a state diagram. The graph nodes define the states of the set S and the directed arcs define the transition events for changing the current state. In a CTMC, the weight w of an arc directed from a state s_j to a state s_i defines the **migration rate** from s_j to s_i. The inverse of w represents the expected **sojourn time** in s_j. The only random variable that holds the memoryless property, and that is therefore suitable for the analysis of continuous time Markov chains, is the **exponential distribution** (BOLCH et al., 2006a).

A CTMC can be represented also by a **transition rate matrix**, also named the CTMC's Q matrix. A q_{ij} $(i \neq j)$ element from Q corresponds to the transition rate from the state s_i to the state s_j. If this value is equal to zero, this means that the corresponding state diagram has no arc from s_i to s_j. The diagonal elements q_{ii} are defined as the inverse of the sum of all elements in the row i.

A CTMC is said to be **ergodic** if it is possible to reach every state from another state in a finite number of steps. The **stationary analysis** of an ergodic continuous time Markov chain consists of finding the vector of probabilities: $\pi = \{pi_1, pi_2, \ldots, pi_n\}$, where π_i is the **stationary probability** of s_i, i.e., the probability of finding the system in the state s_i after a long execution time. This probability does not depend on the system's initial state.

The method for computing the stationary probabilities of a CTMC is described as follows. Consider the Q matrix for the ergodic CTMC depicted in Figure 6:

$$Q = \begin{pmatrix} -3*\lambda & 3*\lambda & 0 & 0 \\ mu & -mu-2*\lambda & 2*lambda & 0 \\ 0 & 2*\mu & -2*\mu-\lambda & \lambda \\ 0 & 0 & 3*\mu & -3*\mu \end{pmatrix}$$

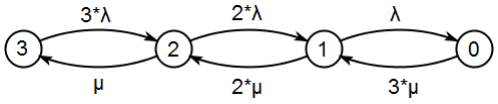


Figure 6 – CTMC example

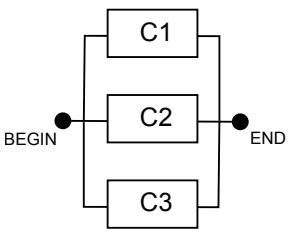


Figure 7 – CTMC example

The CTMC of Figure 6 is equivalent to the RBD model of Figure 7. The λ parameter represents the failure rate of a component and the μ parameter represents the repair

rate. The numeric value inside each state represents the number of properly functioning components.

The **balance equations** of Equation 2.5 means that the sum of all input rates for every state is equal to the sum of all output rates for that state. The Equation 2.6 states that the all stationary probabilities pi_i are mutually exhaustive and exclusive.

$$\pi Q = 0 \tag{2.5}$$

$$\sum_{i=1}^{n} \pi_i = 1 \tag{2.6}$$

For the matrix Q displayed above, equations 2.5 and 2.6 combined produce the following system of linear equations:

$$\begin{aligned} -3\lambda\pi_0 + \mu\pi_1 &= 0 \\ 3\lambda\pi_0 + (-\mu - 2 * \lambda)\pi_1 + 2 * \mu\pi_2 &= 0 \\ 2\lambda\pi_1 + (-2 * \mu - \lambda)pi2 + 3 * \mu\pi_3 &= 0 \\ \lambda\pi_2 + -3 * \mu\pi_3 &= 0 \\ \pi_0 + \pi_1 + \pi_2 + \pi_3 &= 1 \end{aligned} \tag{2.7}$$

By solving the system of linear equations above and fixing the CTMC parameters as $\lambda = 1/1200$ and $\mu = 1/24$, we can obtain the steady state availability by the following expression:

$$A = 1 - \pi_3 = 0,99999246$$

, which is the same value obtained by solving the corresponding RBD model.

2.6 PETRI NETS

Petri nets are a family of mathematical formalisms suitable for representation and analysis of concurrent/distributed systems. Petri nets enable the representation of characteristics such as choice, parallel execution of activities, and resources sharing. It was proposed initially in 1962 by Carl Adam Petri in his Ph.D. thesis, and since then the initial concept has been expanding with many extensions, verification algorithms, and applications in various fields of knowledge.

The most basic type of Petri nets is named *Place/Transition nets* (P/T nets). It also is named *elementary Petri nets* (REISIG; ROZENBERG, 1998), or simply *Petri nets* (when the specific type is omitted, we can consider that it is a Place/Transition net). Figure 8 shows some important extensions to the basic Petri net class. Some extensions add the notion of time to the net events. Other extensions include high-level features such as colored tokens,

objects, or modularity concepts. In this section, we introduce the elementary, stochastic, and object-oriented Petri nets.

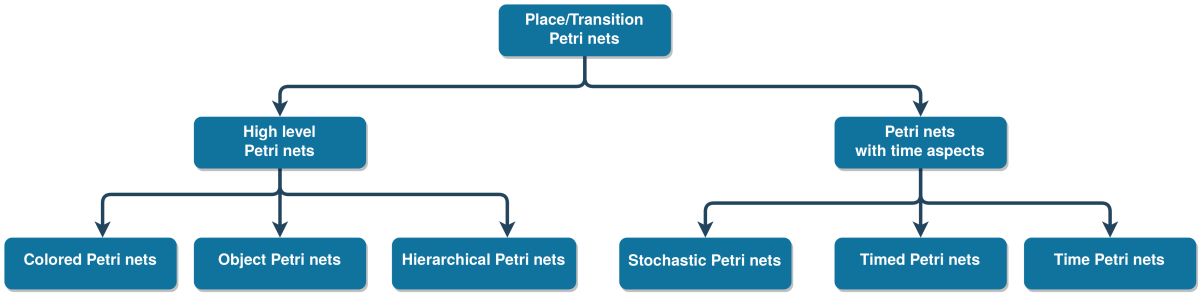


Figure 8 – Petri nets variants

2.6.1 Place/Transition nets

A P/T net is represented by a bipartite digraph composed of two set of vertices: places and transitions. Since it is a bipartite digraph, no arcs connects two vertices of the same type: a place connected to another place, or a transition connected to other transition. In Figure 9, we show an example of a Petri net, where this model represents a production line. A machine manufactures a product, and it is composed of three nuts and three bolts. After being assembled, it is stored into a deposit. The elements of a Petri net are described as follows.

- **Places** are the circle-shaped vertices. They represent a passive element in the net and, generally, they are used to store or accumulate things (nuts, bolts, products, etc.), or to indicate a state in the model (like the availability of a resource, for instance);
- **Transitions** are the rectangle-shaped vertices. They represent the active elements in the net. Transitions can fire actions that lead to a change in the system state. The firing of a transition changes the net **marking** according to the arcs connected to the transition;
- **Arcs** impose relationships between places and transitions. Arcs that has a place as source and a transition as target are classified as **input arcs**, otherwise, they are classified as **output arcs**.

A transition t relates to two sets of places. The **pre-set** of t is composed of all places connected to t via output arcs. It is denoted by the notation ${}^\bullet t$. The **post-set** of t, denoted by $t^\bullet$), is composed of all places connected to t via input arcs. Figure 10 illustrates the **assembly-package** transition with its pre/post-sets. We say that a transition t is **enabled** if each place of its pre-set has a number of tokens equal to or greater than the weight of the arc connected to the transition. The **firing** of a transition is an event that

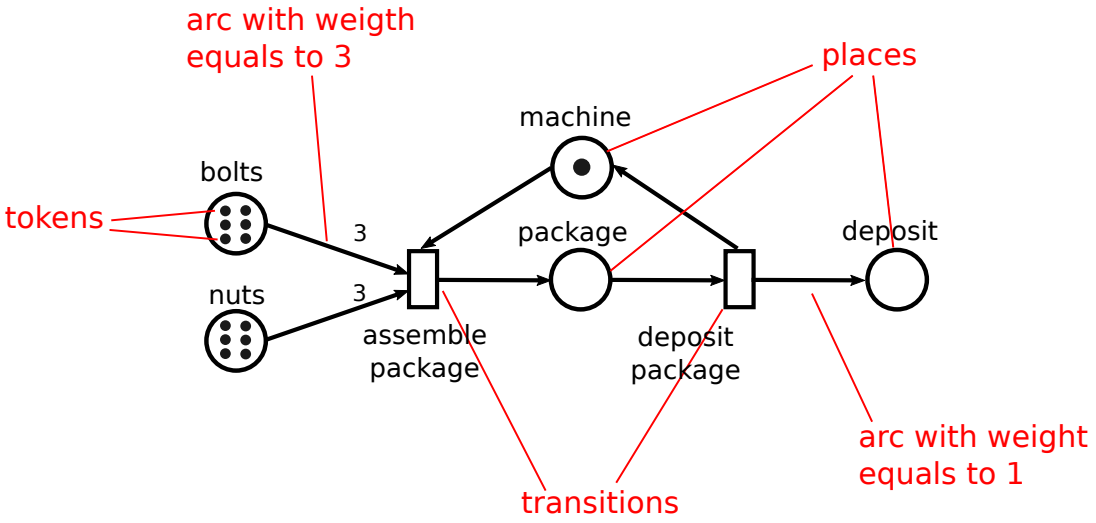


Figure 9 – Petri net - example

changes the **marking** of the Petri net, i.e., it provokes a change in the system state. A transition is allowed to fire if it is enabled. The firing of a transition decreases the number of tokens in every place in its pre-set and adds tokens in every place in its post-set. The number of tokens added/removed is given by the weight of the arc. Figure 11 illustrates the state of the production line Petri net after and before the firing of **assembly-package**.

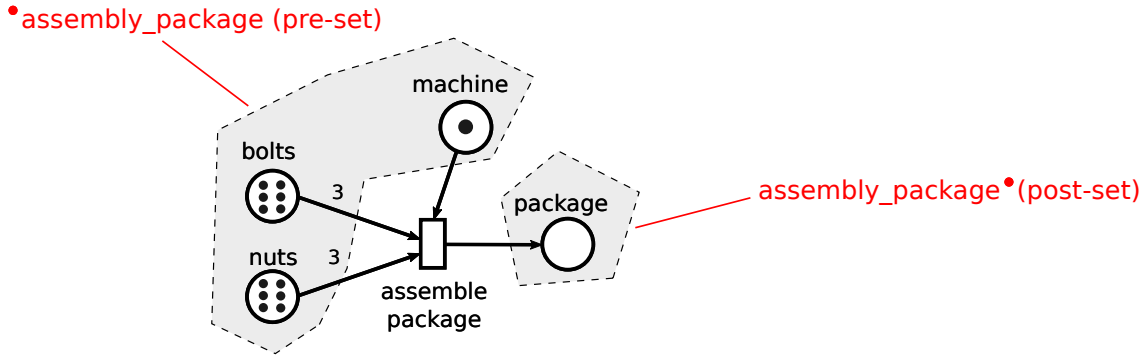


Figure 10 – *Pre* and *post-sets*

Figure 11 – Firing of a transition - example

We explained the basic concepts of a P/T through of its visual mechanisms. The formal definition of a Petri net is presented below.

Definition 3 *A Place/Transition Petri net is a bipartite graph represented by a tuple* $R = (P, T, F, W, m_0)$*, such that:*

- $P \cup T$ *are the vertices of the graph.* P *is the set of places and* T *is the set of transitions of the net;*
- P *and* T *are disjoint sets, i.e.,* $P \cap T = \varnothing$*;*
- F *is the set of edges of* R*, such that* $F \subseteq A = (P \times T) \cup (T \times P)$*;*
- $W : A \to \mathbb{N}$ *is the set that represents the weight of the arcs;*
- m_i *represents a marking in the Petri net. It is represented by a function* $m_i : P \to \mathbb{N}$*.* m_0 *represents the initial marking of the Petri net.*

The (P, T, F, W) tuple is defined as the **structure** of the Petri net and the tuple $R = (P, T, F, W, m_0)$ is referred as a **marked Petri net**. The **reachability graph** of a Petri net is a directed graph with vertex set composed of the set of reachable markings of a Petri net. The edge set of a reachability graph comprises the transitions that provoked the marking change. This graph enables the modeler to assess properties and metrics of a Petri net. The next subsection shows how the reachability graph of a Stochastic Petri net can be used as a basis for obtaining an equivalent Continuous Time Markov Chain. The Figure 12 shows the reachability graph for the production line Petri net of Figure 9. The marking of each state is represented in the vertex label. The numbers displayed correspond to the number of tokens in the places **bolts**, **nuts**, **package**, **deposit**, and **machine**, respectively.

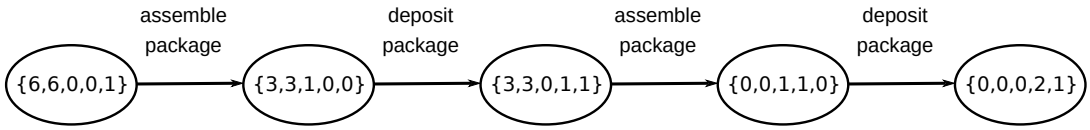


Figure 12 – Reachability graph

2.6.2 Stochastic Petri nets

Stochastic Petri nets are a timed extension to the elementary Petri nets in which transitions can be classified as timed or immediate (CHIOLA et al., 1993). Differently from *Timed Petri nets* or *Time Petri nets*, that have deterministic firing times, the firing times of timed transitions in an SPN are determined by random variables. Immediate transitions fire automatically when they become enabled, without delay. If an SPN holds the following properties, namely, reversibility [3], absence of deadlocks [4], the firing delays are exponentially

[3] A Petri net is said to be reversible if, for any marking M_k reachable from the initial marking M_0, M_0 is reachable from M_K

[4] A deadlock is a marking with no enabled transitions

distributed, and if it has a finite marking reachability graph, then it has an **embedded Continuous Time Markov Chain** associated with it.

The following list presents some concepts introduced by the SPN notation:

- **Inhibitor arc** is a special arc type. It is represented in the graphic notation by a white circle in the end of the arc. An inhibitor arc is always directed from a place towards a transition, and it can be weighted. Inhibitor arcs differ from regular arcs since it does not cause changes in the net marking. Its function is to disable the connected transition if the connected place has a number of tokens equals or greater than the arc's weight;

- **Guard expressions** are boolean expressions that can be associated with transitions. A transition will not be enabled if its guard expression (if any) does not evaluate to logical truth (according to the current marking);

- **Priority associated with immediate transitions.** Priorities are integer numbers assigned to immediate transitions. When multiple immediate transitions are enabled (according to its pre-set, connected inhibitor arcs, and guard expressions), only the transitions with higher priority are eligible to fire;

- **Weights associated with immediate transitions.** When two or more immediate transitions with the same priority are enabled, one transition should be non-deterministically selected for firing. The probability of choosing a transition is given by its weight (a real number), and the weight of the other transitions enabled at the same time, with the same priority, according to the formula:

$$Prob = \frac{\text{Transition Weight}}{\text{Sum of weights for all enabled immediate transitions}}$$

  For example, if there are three enabled immediate transitions with the same priority, two of them with weight equals to 1, and the third with weight equal to 2, the firing probabilities are equal to $0.25, 0.25,$ and 0.5, respectively;

- **Server semantics.** A timed transition can be able to have multiple firings being processed in parallel. The server semantics of the transition determines this behavior. A single-server transition does not present any parallelism, and all firings occur sequentially. K server or multiple server transitions allow up to k parallel firings. An infinite server transition allows as many firings as possible, according to the net's marking.

To illustrate the concepts mentioned above, we present SPN example in Figure 13. This example, extracted from (GERMAN, 1996), represents a M/M/1/K queue. The **generate** transition creates tokens that correspond to incoming requisitions. The **generated** place

represents the waiting queue. This place is connected to two immediate transitions. The **enter** transition moves the token from **generated** to the **buffer** place. The **drop** transition represents the rejection of a request. The number of tokens in the **free** place dictates the choice between the **enter** and the **drop** transitions. Each time a token is moved from **generated** to **buffer**, a token is removed from **free**. When there are no tokens in this place, the inhibitor arc no longer prevents the **drop** transition of being enabled. Also, the **enter** transition is no longer active since there are no tokens in one of its input places. The transition **service** represents the processing of requests by the server.

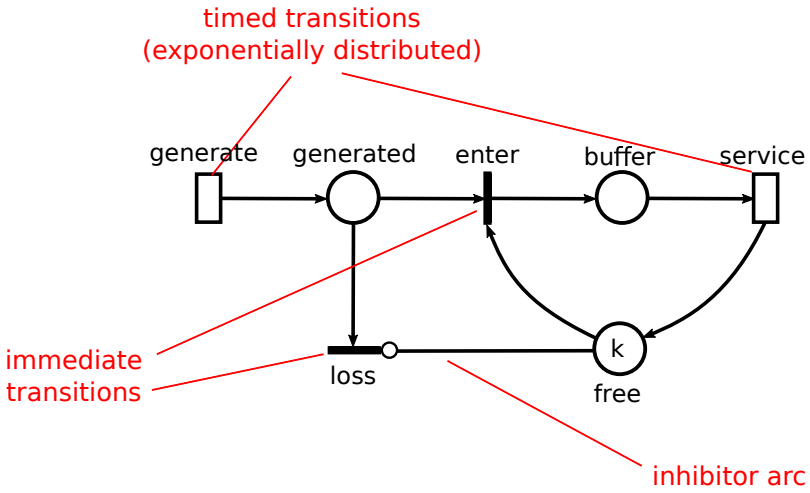


Figure 13 – Stochastic Petri net example (GERMAN, 1996)

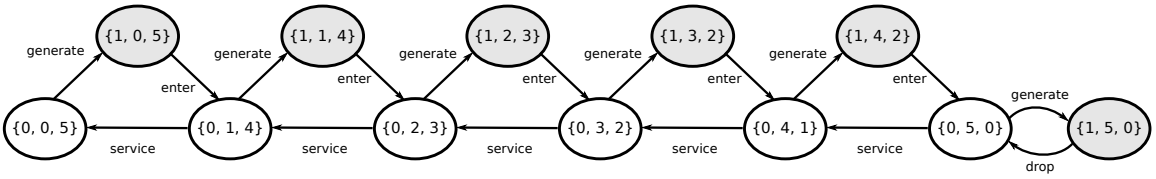
If an SPN does not have immediate transitions, and all transitions follow the single server semantics, the embedded CTMC can be obtained by the following rules (MARSAN et al., 1994):

1. The CTMC state-space is equal to the SPN reachability graph. Each M_i marking is equivalent to the s_i state of the embedded CTMC;

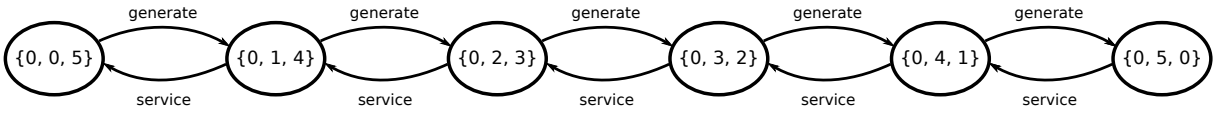
2. The transition rate from a s_i to a s_j state is equal to the sum of all (exponentially distributed) firing rates of timed transitions that are enabled in the M_i marking, that lead to M_j.

For general SPNs, the process of obtaining the embedded CTMC is similar, but the presence of immediate transitions makes it necessary to apply an additional step. In the reachability graph of an SPN with immediate transitions, the nodes can be classified into two types. If a marking has one or more immediate transitions enabled, it is labeled as a **vanishing marking**. Otherwise, the marking is labeled as a **tangible marking**. The sojourn time in a vanishing state is equal to zero and, therefore, such states must be removed in order to obtain the embedded CTMC of an SPN. When removing a vanishing state, the transitions point to/coming from it should be adjusted. The reader can refer to the work done by (MARSAN; CONTE; BALBO, 1984) for the exact algorithm. In Figure 14a,

we show a reachability graph (with vanishing states included) for the SPN of Figure 13. The embedded CTMC obtained after removing the vanishing states is displayed in Figure 14b.



(a) Reachability graph with vanishing markings



(b) Embedded CTMC

Figure 14 – Reachability graph and embedded CTMC of an SPN

2.6.3 Object-Oriented Petri nets

Object Petri nets (OPN) are Petri net variants that borrow concepts from object-oriented programming, namely, encapsulation, inheritance, and polymorphism. The Object Petri Net definition from Valks (VALK, 2003) - the *nets within nets paradigm* - specifies Petri nets with tokens that can contain nested subnets. This representation can be useful to model the movement of elements with their internal events. Transitions of an object token can fire independently from the upper-level net state, or they can fire synchronously with a transition from the upper level. Object Petri Nets also introduce the concepts of *reference semantics* and *value semantics* (VALK, 1999), analogous to the programming concept. In the reference semantics, the same subnet can be referenced by two or more tokens in different places, and the object token state will the same for all of them. In the value semantics, it is created multiple copies of the same object token, and changing the state of one of them does not change the others.

The Object Petri Net notation discriminates two categories of places, according to the tokens they can contain: black tokens and object tokens. It also does not consider object-oriented concepts such as polymorphism, inheritance, and encapsulation. In another direction, the Object-Oriented Petri Net introduced by Lakos (LAKOS; KEEN, 1991) is an extension of the Colored Petri Net (CPN) formalism in which the tokens can contain not only data attributes but behavior as well. The token colors are now classes in an object-oriented context, and the notions of inheritance and dynamic binding are now present. This work presents the LOOPN language as a concrete implementation of this paradigm.

The Object Petri Net notation from Lakos meets the nets within nets paradigm in an extension to the LOOPN language called LOOPN++ (LAKOS; KEEN, 1994). The author

presents the Russian Philosophers problem to illustrate the language concepts. This is a variation of the dining philosophers problem in which each philosopher can be eating, or thinking about a dining philosophers problem. It is possible to have many levels in this problem, such as the Russian *matryoshka* dolls.

The PNTalk project (JANOUSEK, 1995) defines a modeling framework for the Smalltalk language based on object-oriented Petri Nets. It defines a Petri net as a set of active objects (LAVENDER; SCHMIDT, 1995) that can interact by exchanging messages. Those objects can be object nets containing transitions, places, and tokens, or method nets, that are similar to object nets, but they are dynamically created in response to a method invocation. The Renew tool (KUMMER et al., 1999) provides the same level language level integration for the Java language. This tool is a concrete implementation of the Reference nets paradigm and provides access to all features existent on the Java language to the reference net models.

2.7 FINAL REMARKS

This chapter presented the theoretical background needed for a better comprehension of this work. First, it described the targeted study subject of this work - cloud computing and scientific workflow applications. Then, it provided an introduction about dependability evaluation and modeling. The joint analysis of dependability and performance aspects of computational systems, namely, performability evaluation, is covered later in Section 5.4. Finally, the presented chapter introduced the modeling formalisms adopted in this work for performance and dependability models: Continous Time Markof Chains, Reliability Block Diagrams, and Petri nets.

3 RELATED WORK

This chapter presents a list of related works in the main topics covered in this work. Modeling tools and frameworks proposed by the literature are discussed in Section 3.1. Section 3.2 presents a list of works about cloud availability modeling and the redundancy allocation problem. Section 3.3 presents works about performability modeling of cloud and grid systems. In Section 3.4, we make a summary of existing simulation models for representing workflow applications and the failure of workflow tasks and cloud components. Finally, in Section 3.5, we describe the works related to the optimization of cloud workflow applications.

3.1 MODELING TOOLS AND FRAMEWORKS

The TimeNET software package (GERMAN et al., 1995) is a specialized graphical tool to model Petri nets. It provides support for two classes of Petri nets: Colored Petri nets, and Extended Deterministic and Stochastic Petri nets (EDSPN). Colored Petri nets can be solved only by simulation, and EDSPN models can be solved either by simulation or numerical methods. CPN Tools (JENSEN; KRISTENSEN; WELLS, 2007) is a graphical modeling framework focused on Colored Petri nets that provides support for space-state based analysis methods and discrete-event simulation. It allows the use of the Standard ML programming language for creating user-defined functions that can be used by the models. The PIPE tool (BONET et al., 2007) implements more methods for structural analysis (invariants, traps, siphons, etc.) of GSPNs than the TimeNET and Mercury tools, but provides less support for obtaining metrics from the stationary and transient analysis. The GreatSPN tool (CHIOLA et al., 1995) is another tool specialized on SPN and CPN models that implements methods for structural analysis, Markov chain generation, and discrete event simulation. Additionally, it allows the composition of two or more interacting models.

The SHARPE tool (SAHNER; TRIVEDI, 1987) is a well-established software for performance and dependability modeling that dates back to the year of 1987. It provides support for many formalisms, namely: fault trees, reliability block diagrams, acyclic series-parallel graphs, acyclic and cyclic Markov and semi-Markov models, Stochastic Petri nets, and finally product-form queueing networks. The SHARPE software package provides a graphical interface and a scripting language for creating and evaluating the models. The Möbius tool (DALY et al., 2000) is a modeling framework enabled for multi formalism hierarchical models. Its extensible architecture allows the incorporation of new formalisms that can be solved by Markov chain generation or by discrete event simulation. The PRISM (KWIATKOWSKA; NORMAN; PARKER, 2002) tool provides a modeling language for model

verification of discrete and continuous time Markov chains.

Comparing the Mercury script language (MSL) with the SHARPE language, MSL is more verbose but has a more intuitive syntax. While both languages have similar features (e.g.: loops, symbolic evaluation, support to hierarchical models, etc.), both also have unique features. The SHARPE tool provides more formalisms and solution algorithms and it is especially useful when dealing with Markov chains, reliability graphs, and fault trees. On the other hand, the Mercury tool is particularly useful when one must deal with SPNs, providing some features that are not found in other SPN/CPN tools such as phase-type distributions, meta-programming, and event-based programming. The TimeNet and CPN tools offer support for hierarchical Petri net models, but only for Colored Petri nets. While it can be argued that CPN is a superclass of P/T nets, both tools do not provide support for using hierarchical transitions on models that can be solved by Markov chain generation. The MSL language also offers more statistical distributions than CPN Tools, TimeNET, and PIPE tools.

3.2 CLOUD AVAILABILITY INFRASTRUCTURE PLANNING

High availability is critical on cloud systems since downtime periods can lead to significant losses of revenue and user dissatisfaction. Due to the scale of such systems, a series of studies on literature employs *hierarchical* modeling for evaluating dependability attributes (CHUOB; POKHAREL; PARK, 2011; DANTAS et al., 2012; WEI; LIN; KONG, 2011). In (CHUOB; POKHAREL; PARK, 2011), Markov chains were used to create hierarchical availability models for private cloud systems. A set of measurements was performed on a Eucalyptus cloud to obtain the failure and repair rates for the model. A similar work, presented in (DANTAS et al., 2012), proposes a hierarchical model that combines Reliability Block Diagrams and Continuous Time Markov Chains to measure the availability of a Eucalyptus private cloud with warm-standby redundancy support in the cloud controller. Wei et al. (WEI; LIN; KONG, 2011) created a hierarchical model composed by Reliability Block Diagrams with Stochastic Petri nets for evaluating the dependability of Virtual Data Centers. The models were used to study how the consolidation backup and live migration affect the reliability and consolidation ratio metrics of the virtualized data center.

The tradeoff between the complexity of state-space-based models and the inherent limitations of combinatorial models have motivated the need for techniques that exploit the best of both categories. Dynamic Fault Trees (DFT) were the first solution proposed which extended combinatorial Fault Trees with dynamic gate elements (BOYD, 1992; DUGAN; DOYLE, 1996). However, due to certain limitations of the DFT formalism, as well some difficulties in using this technique, the Dynamic Reliability Block Diagram formalism was proposed (DISTEFANO; PULIAFITO, 2007). To obtain the metrics of a DRBD model, the author used continuous time Markov chains for models with exponential failure and repair rates, and Monte Carlo simulation for models that use general distributions.

Robidoux *et al.* (ROBIDOUX et al., 2010) proposed the use of colored Petri nets (CPN) as a means of solving DRBDs models and verifying structural properties such as liveness and deadlocks. A disadvantage of the adopted CPN model is that it can only be solved by simulation. Following a similar strategy, Signoret (SIGNORET et al., 2013) shows how to map an RBD to a Petri net, while introducing constructs for shared repair teams and preventive maintenance. It also introduces the concept of a "*RBD-driven Petri Net*", which has the same objective as DRBD. This work used a modular high-level Petri net notation that is solved only by simulation.

Redundancy is a widely used technique for achieving high availability. However, using redundant components increases the cost of the system. Optimizing the number of redundant elements of a fault tolerant system is known in the literature as the *redundancy allocation problem* (RAP), and it was proven to be NP-hard (COIT; SMITH, 1996b). This fact led to the investigation of several meta-heuristic approaches to solve this problem such as particle swarm (COELHO, 2009; GARG; SHARMA, 2013), bee colony (YEH; HSIEH, 2011; GARG; RANI; SHARMA, 2013), and genetic algorithms (COIT; SMITH, 1996a; GUPTA; BHUNIA; ROY, 2009). In (HE et al., 2015), a novel fish swarm algorithm was proposed to solve large-scale RAP problems. (BOSSE; SPLIETH; TUROWSKI, 2016) et al. studied the RAP problem on IT Services with inter-component dependencies. They adopted Petri net models because such dependencies cannot be represented with combinatorial models. The authors employed Monte Carlo simulation to evaluate the Petri net models.

Our work brings the RAP problem to the topic of cloud dependability modeling. We tried to take into account two crucial characteristics: being able to solve large-scale problems (HE et al., 2015), and representing inter-component dependencies (BOSSE; SPLIETH; TUROWSKI, 2016) existent on cloud systems. For steady-state availability optimization, we used the pivot decomposition instead space state-based models to avoid the use of simulation and the bisection algorithm to treat the optimization problem as a single objective problem and to reduce the search space.

3.3 PERFORMABILITY MODELING OF CLOUD AND GRID ENVIRONMENTS

Performability is the study of systems performance when subjected to the effect of failures on its subcomponents (MEYER, 1980). The performance of a system is said to be *degradable* if failure events may affect it negatively. For instance, a mesh network of routers can tolerate a certain number of failures, but the overall performance will be affected as some routers may be subject to overheads. Similarly, failures on worker nodes in cloud and grid environments can diminish the number of available processing resources, and therefore increasing queueing times and decreasing throughput of jobs.

Due to a large number of components of cloud and grid environments, we can expect a significant failure rate even if the mean time to failure of individual components is high. Thus, neglecting the impact of failures in performance studies of such systems can lead to

misleading results. Space stated based models (Stochastic Petri Nets and Markov Chains) is the most adopted method for joint performance/availability evaluation of clouds and grids systems. Dealing with space state explosion is a recurrent problem handled by every work in this category.

Ramakrishnan and Reed (RAMAKRISHNAN; REED, 2008) propose a qualitative framework for the performability evaluation of scientific workflows running on grid systems. The framework encompasses a Markov Reward Chain model (BOLCH et al., 2006b) and simulations calibrated with data from real grid applications. Xia et al. (XIA et al., 2015) describe a queuing network model for evaluating estimated service time and request rejection probability of an Infrastructure-as-a-Service cloud. This model represents features such as request handling, job creation, job execution, job rejection due to insufficient queue capacity, and failure and repair events of physical machines configured in hot/warm standby mode. Ever et al. (EVER, 2017) propose a set of equations obtained from queuing theory for evaluating the performability of clouds with large numbers of servers. Since the underlying space-state model does not need to be generated, this model can represent a large number of servers and simultaneous requests.

A strategy to avoid space state explosion is to adopt small models rather than use a big monolithic one. For combining the results of the submodels, iteration methods can be used (MAINKAR; TRIVEDI, 1996). Ghosh (GHOSH et al., 2010) uses interacting homogeneous time Markov chains to perform end-to-end performability analysis of cloud services. The proposed model is used to evaluate two essential metrics: service availability and response time. Raei et al. (RAEI; YAZDANI, 2017) developed Stochastic Reward Net models for representing a public cloud and a cloudlet providing virtual machines for mobile applications. For avoiding space state explosion when modeling both performance and availability aspects of the considered system, the authors divided the public cloud and cloudlet parts into two separated models and used the fixed point iteration method to obtain a joint result.

The performability models cited in this subsection are based on Markov Chain (RAMAKRISHNAN; REED, 2008) (BOLCH et al., 2006b), Stochastic Petri nets (with Markov chain generation) (MAINKAR; TRIVEDI, 1996) (GHOSH et al., 2010) (RAEI; YAZDANI, 2017), and Queuing Theory (XIA et al., 2015) (EVER, 2017). By contrast, our work adopts a discrete simulation approach based on Stochastic Petri nets components and automatic generation of models. The advantage of our model over the works mentioned above is the ability to model DAGs as job requests and the relationship between VM and hardware failures. Incorporating these features into space-state based models would lead to a space-state explosion problem. Also, using the exponential distribution for representing job times can introduce distortions when modeling the makespan of a stochastic DAG (as we demonstrate in Subsection 6.3.3.2).

3.4 SIMULATION OF WORKFLOW EXECUTION IN CLOUD ENVIRONMENTS

Simulation is a commonly used approach for evaluating the performance of load-balancing algorithms, allocation policies, and scheduling strategies in cloud systems, considering dynamic workload patterns. CloudSim (CALHEIROS et al., 2011) is the most adopted cloud simulation software in the literature. The CloudSim simulator allows the representation of data-center infrastructures, VM allocation policies, user level workloads, and coordination between multiple cloud environments through a cloud broker service. After being released as open source software, CloudSim was extended in many different ways by the research community. Fault tolerance capabilities were introduced in (ZHOU et al., 2013). Federated-CloudSim (KOHNE et al., 2014) extended CloudSim to represent SLA policies in federated clouds. FailureSim (DAVIS et al., 2017) introduced failure prediction of cloud nodes based on ANNs. Performance and usage levels (bandwidth, number of tasks running, the quantity of available million of instructions per second per node) are used as predictors for training the network. Alwabel et al. proposed DesktopCloudSim (ALWABEL; WALTERS; WILLS, 2015), a CloudSim extension with a layer of failure injection for the physical nodes. Like our work, DesktopCloudSim allows the investigation of failure events on system throughput.

CloudSim does not offer, by default, classes for representing workflows modeled as DAGs. Given the importance of this application category, some extensions to CloudSim were proposed for representing workflows. WorkflowSim (CHEN; DEELMAN, 2012) is a CloudSim extension that includes support for workflow representation and management. It also provides task aggregation capabilities and a fault generator at a job/task level. It can generate recoverable transient failures that can be handled by task re-execution and permanent job failures that cannot be recovered. DynamicCloudSim (BUX; LESER, 2015) is a CloudSim based simulator that includes workflow execution considering VM inhomogeneity and failure of tasks at runtime. Malawski et al. (MALAWSKI et al., 2015) developed a cloud workflow simulator for evaluating task scheduling and resource provisioning algorithms for optimizing the execution of workflow ensembles under deadline constraints in IaaS clouds. ElasticCloudSim (CAI; LI; LI, 2017) is a CloudSim extension for evaluating workflow applications, that supports auto-scaling capabilities and considers non-deterministic (stochastic) workflows.

Our work differs from WorkflowSim and DynamicCloudSim by modeling failures at infrastructure level instead of representing transient/permanent failure on tasks. Failures on node level are found on DesktopCloudSim and FailureSim, but they cannot represent workflows. We opted to not creating another CloudSim extension as the employed SPN based simulator presents some advantages. The proposed SPN models can be used separately for obtaining other metrics than performability (e.g., availability, reliability, and expected makespan). Using this simulation environment also allows us to use existing SPN models in the literature for representing reliability and performance aspects of our system.

3.5 CLOUD WORKFLOW OPTIMIZATION

The execution of workflow applications on clouds brings the need for new modeling strategies, scheduling algorithms, and optimization metrics. The reason for this need is the particular aspects of cloud systems when contrasted to traditional grid/cloud environments. Kliazovich et al. (KLIAZOVICH et al., 2016) demonstrated how existent workflow models fail to address the communication patterns typically found in cloud workflow applications. They proposed CA-DAG (Communication-Aware DAG), a workflow model which represents communication processes as vertices instead of edges. Arabnejad and Barbosa (ARABNEJAD; BARBOSA, 2014) developed a Heterogeneous Budget Constrained Scheduling (HBCS) for minimizing makespan and rental cost of cloud workflow applications. The HBCS algorithm can reduce up to 30% of the execution time while maintaining the same budget level.

Many works in the scheduling literature consider deterministic computation and communication times. However, using a deterministic objective function does not match the non-deterministic nature of real-world applications (ANDO; NAKATA; YAMASHITA, 2009). In this sense, Zheng et al. (ZHENG; SAKELLARIOU, 2013) proposed a Monte Carlo based scheduling method for cloud/grid workflows which consider non-deterministic computing and communication time. The method is not dependent on a particular heuristic algorithm, and the HEFT (Heterogeneous Earliest Finish Time) is adopted in the evaluation. In (ZHENG; WANG; ZHANG, 2016), a randomized version of HEFT was proposed. The algorithm consists in running a deterministic HEFT for random predictions of the stochastic DAG, generating a list of potential candidates for best scheduling. The scheduling from the list with the smaller expected makespan is selected. Cai et al. (CAI et al., 2017) present a dynamic algorithm for minimizing the rental cost (of VMs in a cloud) of bag-of-tasks workflows with non-deterministic times. The Cloud Workflow Scheduling Algorithm (CWSA) (RIMAL; MAIER, 2017) aims to optimize the scheduling of workflows in a multi-tenant cloud environment. This algorithm considers non-deterministic times for task computation times.

The presence of failures in data centers can pose a threat to workflow applications with strict deadlines. In (WANG et al., 2015), an original fault-tolerant scheduling algorithm name FESTAL was proposed. It employs a primary/backup redundancy model and VM migration to achieve high-availability and load balancing into a cloud workflow application. Vinay et al. (VINAY; KUMAR, 2017) present a new heuristic for cloud scheduling named CHEFT (Cluster-based Heterogeneous Earliest Finish Time). It uses the idle time of the processors for resubmitting the failed tasks as a mean to achieve fault tolerance. FASTER (ZHU et al., 2016) is another algorithm that employs the primary/backup redundancy model for providing a fault tolerant scheduling mechanism for cloud applications. Performability was first considered an objective function in (ENTEZARI-MALEKI et al., 2018). The authors developed a performability model of a grid resource based on a stochastic reward net model and the universal generating function. The proposed model is connected to a genetic algorithm which aims to optimize the scheduling of a DAG into a set of grid resources.

Our work aims to contribute to the research line opened by Entezari et al. (ENTEZARI-MALEKI et al., 2018) - workflow scheduling optimization from a performability viewpoint. To the best of our knowledge, no existing method covers simultaneously performability as the objective function, multi-tenancy, non-deterministic computing/communication times, and failures of hosts and VMs. Table 1 shows a comparison of our work to the state of the art.

It is worthwhile to mention that ensuring an expected Service Level Agreement may be done by the actual measurement of the system. Niehoster et al. (NIEHORSTER et al., 2010) proposed an architecture for a scientific SaaS cloud that can enforce an expected SLA level. By employing a set of sensors coupled to the running workflows applications, a reasoning engine, and a set of actuators, the cloud is able to adjust its provisioning level aiming to avoid SLAs violations while avoiding excessive overprovisioning. In (NIEHOERSTER; BRINKMANN, 2011), Niehoster and Brinkmann extended the mentioned work by adding predictive Support Machine Vector models in order to extend the accuracy of the actuators further.

Table 1 – Comparison of the state of the art for cloud workflow scheduling

Work	Metric	Multi-tenancy	Non-deterministic times	Failure model
(KLIAZOVICH et al., 2016)	Makespan	No	No	Not considered
(ARABNEJAD; BARBOSA, 2014)	Makespan and rental cost	No	No	Not considered
(ZHENG; SAKELLARIOU, 2013)	Makespan	No	Yes	Not considered
(CAI et al., 2017)	Rental cost	Yes	Yes	Not considered
(RIMAL; MAIER, 2017)	Makespan and resource usage	Yes	Yes	Not considered
(WANG et al., 2015)	Job reliability and resource usage	Yes	No	Host failures
(ZHU et al., 2016)	Job reliability and resource usage	Yes	No	Host failures
(VINAY; KUMAR, 2017)	Makespan and cost	No	No	Task failures
(ENTEZARI-MALEKI et al., 2018)	Performability	Yes	Yes (exponential times)	Processor failures
Current work	Performability	Yes	Yes	Host, VM, and cloud manager

4 MODELING FRAMEWORK FOR INFRASTRUCTURE PLANNING OF WORKFLOW-AS-A-SERVICE CLOUDS

In this work, we developed a modeling framework for model-based infrastructure planning of Workflow-as-a-Service clouds. The main goals of this framework are: i) to facilitate the composition of performance and availability models of WaaS clouds; and ii) to enable the integration of the modeling approach into sensitivity analisys and discrete optimization methods for infrastructure planning. This chapter presents the framework and a supporting methodology for conducting a infrastructure planning project by making use of our framework.

4.1 SUPPORTING METHODOLOGY

The fluxogram shown in Figure 15 describes the supporting methodology for conducting an infrastructure planning project on WaaS clouds. Each step of this methodology is described as follows.

Understanding the system and user requirements

In the first methodology phase, we gather knowledge about the functioning of the system, its sub-systems/components, and how they interoperate. We should have precise knowledge about the physical topology and the logical relationships of its modules. Relevant **performance and dependability metrics** related to Service Level Agreements should be determined in this phase. Datasheets provided by the manufacturers, experimental data, and results published in the literature are the primary sources of information concerning quantitative aspects of the system.

Proposal of performance and dependability models

By applying the knowledge gained in the initial stage, the modeler should then compose and parametrize the formal models for evaluating the performance and dependability metrics identified in the user SLAs. The modeler can employ **hierarchical modeling** to deal with the complexity of cloud environments (MATOS et al., 2015). Within this approach, the modeler divides the complete model into a **top-level model** for representing the system at a high-level view, and **submodels** for representing the behavior of individual modules.

The **distribution of the identified random variables** dictates the employed modeling technique. Fault trees, Reliability block diagrams, Continous Time Markov Chains, and Stochastic Petri nets (solved by CTMC generation) assume exponential times. Phase-type

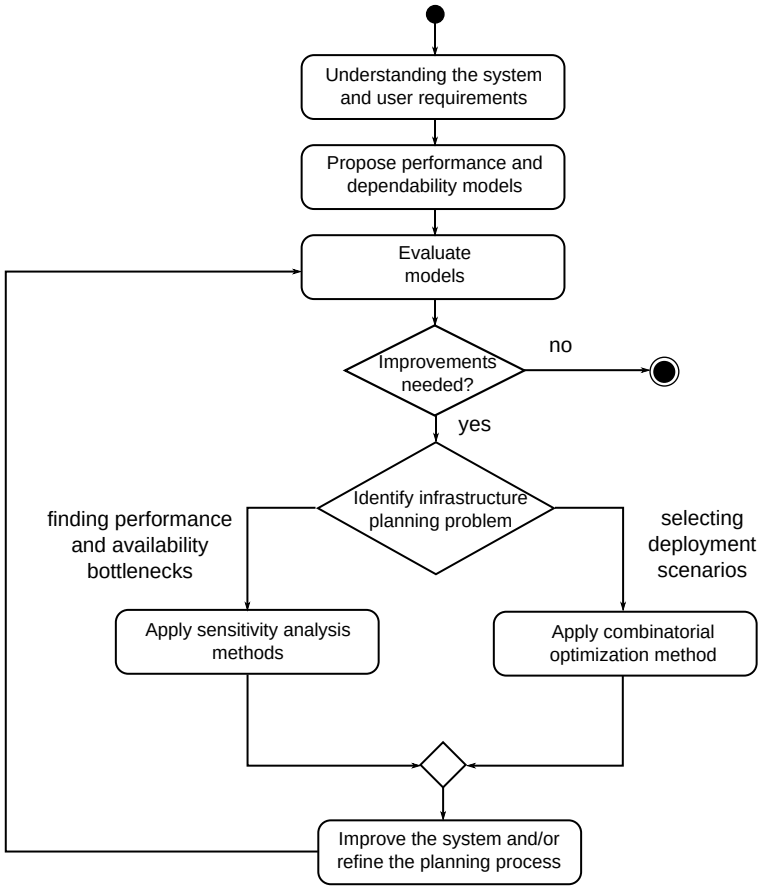


Figure 15 – Support methodology for optimization of cloud infrastructures and services

distributions can approximate general distributions. This technique, however, can increase the state-space of the underlying model considerably. If the phase-type approximation is not feasible, it may be necessary to employ discrete event simulation for obtaining the metrics of interest.

An important characteristic of model-based infrastructure planning projects is the need for exhaustive evaluation of the models in order to select the best scenarios among a large solution space or to discover the relative importance of various input parameters. Therefore, the ability to solve an individual model and combination of parameters in a timely manner is an issue to be addressed in this stage of the methodology.

Evaluate metrics

In the **evaluate metrics** stage, we compare the performance and reliability levels measured from the models developed in the previous step to the expected values documented in the SLAs. If the SLAs are met, we finish the project and wait for structural changes, disruptions in the system or increases in the workload. Otherwise, we proceed to the next step in the methodology.

Identify infrastructure planning problems

When building, maintaining, or deploying cloud applications and infrastructures, there are many non-trivial decisions to be made in order to satisfy Service Level Agreements and avoiding the waste of budget, energy, and computational resources. In this stage, we identify the infrastructure planning problems and the alternatives for improving non-functional aspects. Partitioning a cloud into clusters, setting up redundancy arrangements, and purchasing more reliable and powerful components are the main means for improving performance and availability aspects of a cloud infrastructure.

To make the best use of the organization budget, we must find the relative order of importance of the system components in the metric of interest. In this way, the administrator can prioritize the aspects of the system that are prone to improvement, and select the measures that will yield the best return, i.e, the **bottlenecks** regarding the metric of interest. Alternatively, the administrator may need to select a deployment configuration among a large number of possible scenarios. When the choice is not obvious, heuristic and meta-heuristic algorithms can help the administrator to find near-optimal solutions.

Apply sensitivity analysis methods to find performance and availability bottlenecks

Sensitivity analysis is the task of finding which components, when upgraded, promote the maximum improvement on the metric of interest. It consists of evaluating the model under many combinations of parameters and applying statistical methods to sort the model parameters by the level of impact to the selected metric. Partial derivatives, design of experiments (DoE) (JAIN, 1990), the percentage differences method (HOFFMAN; GARDNER, 1983), and variating one parameter at a time are examples of sensitivity analysis techniques.

Applying the design of experiments (DOE) technique consists of taking a parameter list, a list of values (called levels) for each parameter (called factors), and performing a series of experiments with all possible combinations of factors and values. There are various possible designs for an experiment. One possible alternative is to run the experiment for all combinations of levels using all factors, which is called a *full factorial design*. A drawback of this alternative is that, even for a small number of parameters, the amount of experiments to be performed could be very large. One solution is to use only two levels which is called a *two-level factorial design*. The percentage difference technique consists of changing one parameter over a list of values while holding the other parameters fixed, and calculating the percentage difference in the output metric considered. We perform this step for each parameter in our list and sort them from the highest difference to the lowest. The formula for obtaining the percentage difference is shown in the following (HOFFMAN; GARDNER, 1983):

$$SI = \frac{D_{max} - D_{min}}{D_{max}}$$

, where SI is the sensitivity index for the selected parameter, D_{max} is the maximum value of the output metric, and D_{min} is the minimum value.

The nature of the formal model and the number of parameters dictate the suitability of each technique. Partial derivatives can be employed if the model can be represented by a deterministic and differentiable function. DoE methods enable capturing the combined effect of parameters, but they suffer from combinatorial explosion when testing a significant number of parameters. Detailed hierarchical models are prone to this problem. One alternative is applying sensitivity analysis in each submodel individually and then creating a composite ranking (JúNIOR, 2016)

Apply optimization methods to select deployment scenarios

This activity consists of finding near-optimal deployment configurations that either minimize or maximize some metric of interest (e.g., monetary cost, annual throughput, steady-state availability, and so on). If the process of finding such configurations cannot be guided by established procedures, and if the solution space of configurations is vast, near-optimal configurations can be found by making use of meta-heuristic optimization methods.

An example of model-based optimization method is depicted in the flow diagram of Figure 16. This diagram represents the process of systematically exploring a solution space S according to a strategy dictated by the optimization algorithm (e.g.: hill climbing, genetic algorithm, ant colony, etc.). The objective function value for a solution $s \in S$ is obtained by the process described bellow. First, the solution s is converted to a performance or dependability model m, and a set of input parameters p. Then, the modeling framework runtime is invoked to extract the performance/dependability metric of interest. The obtained metric is the considered objective function value for the solution s. The algorithm will update its state with the newly discovered solution and apply an iterative process until some stopping criteria is reached. At the end of the process, a near-optimal deployment configuration is obtained according to the chosen objective function.

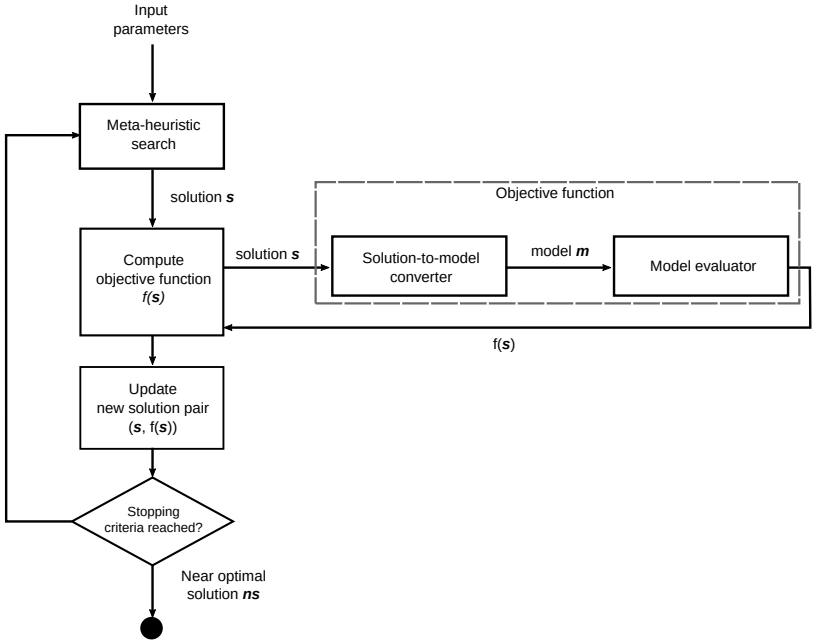


Figure 16 – Optimization method for selection of configuration/deployment scenarios in WaaS clouds

4.2 PERFORMANCE AND DEPENDABILITY MODELING FRAMEWORK

When applying the methodology described in the previous section, the modeler should rely on a capable and flexible modeling framework. In the introduction, we identified the main features that such framework should provide for assisting model-based infrastructure planning of WaaS clouds:

- Good support for hierarchical modeling and many different modeling paradigms;
- Good support for symbolic evaluation;
- Support for on-the-fly model generation and evaluation via third-party tools;
- Support for high-level modeling formalisms.

We developed our modeling framework as a new runtime and a set of extensions for the Mercury tool. By doing so, we were able to reuse the modeling capabilities of the tool and focus on the enhancements needed to meet the requirements listed above. The Mercury tool (MACIEL et al., 2017) is an integrated platform for performance and dependability modeling. This tool has been maintained by MoDCS research group [1] and supports the following formalisms: discrete/continuous Markov chains, fault trees, stochastic Petri nets, reliability block diagrams, and Energy Flow Models (EFMs). Mercury offers more than 25 probability distributions for discrete event simulation, SPN and CTMC transient and steady-state analysis, many methods for computing sensitivity indices for CTMC, RBD,

[1] <http://www.modcs.org/>

and SPN models, reliability-related importance indices, and moment matching of empirical data. Figure 17 presents the layered architecture of the Mercury tool. The modeling framework developed in this thesis is represented as the grey blocks in the diagram.

The bottom layer depicted in Figure 17 corresponds to the core of the tool, i.e., the modules invoked for solving the user models. The numeric evaluators and the discrete event simulator are implemented in this layer. Classes defined at this level provide the following functionalities: linear algebra operations, ordinary differential equations solvers, random number generators, and so on.

The middle layer corresponds to the runtime module for using the tool. The available runtimes are the scripting runtime and the graphical user interface (GUI) runtime. The GUI runtime offers drag-and-drop oriented builders for composing models. The scripting runtime offers a convenient way to describe an experimental design and perform a batch execution for each combination of parameters. In the uppermost layer, there are the graphical and scripting constructs for composing and evaluating models. We have implemented a translator module that converts graphical models to a script representation (Figure 18). The modeler can use the GUI runtime to create and validate a model, and then convert the graphical model to a script to perform a large set of experiments. The runtime engine can be invoked from the command line, allowing the usage of shell utilities and other scripting languages for processing the output metrics and producing charts and tables.

The upper layer exports the modeling capabilities of the tool to users through a runtime. The user can compose and evaluate models graphically or via the command line interface. In addition to the formalisms mentioned above, we implemented the object Petri net and DRBD formalisms to facilitate the modeling of WaaS clouds. The modeling Application Programming Interface (API) allows third-party software (written in the Java programming language) to connect to the scripting runtime, load scripts, and evaluate model metrics. The meta-modeling capabilities enable the creation of models with a variable structure. Finally, the plugin system enables the user to add new capabilities to the Mercury tool without having to recompile its source code.

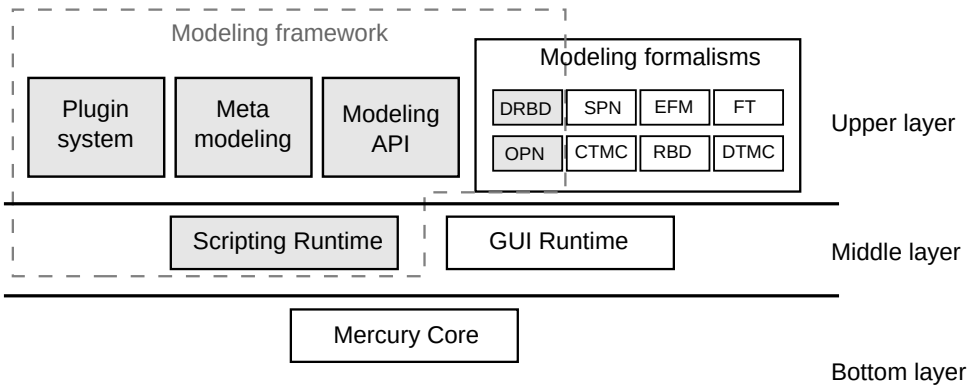


Figure 17 – Mercury tool layered architecture

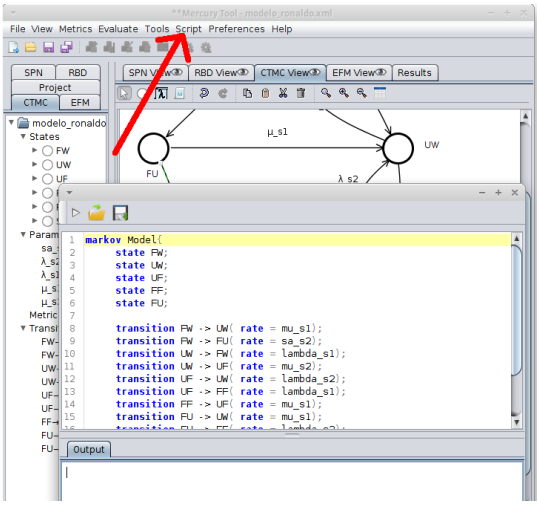


Figure 18 – Creating a script from the graphical user interface

4.2.1 Mercury Scripting Language

Graphical modeling tools allow users to compose formal models with ease, but they complicate the creation of models with a dynamic/complex structure, the hierarchical arrangement of different models, and the automatic evaluation of models with different parameter configurations. To overcome this problem, we created a scripting language for the Mercury tool that supports the combination of different modeling approaches into a single project. Two important features added to Mercury by this language are listed below:

- Better support for **hierarchical modeling**. The resulting metrics of a model can serve as an input parameter for top-level models;
- Better support for **symbolic evaluation**. The input parameters of models can contain expressions and variables that are assessed at evaluation time;

Those features are also found in the SHARPE tool (SAHNER; TRIVEDI, 1987), that features a scripting language as well. However, Mercury has some unique features as compared to the other tools:

1. **Hierarchical transitions on SPN models**. While CPN Tools (JENSEN; KRISTENSEN; WELLS, 2007) and TimeNet (GERMAN et al., 1995) offer support to hierarchical transitions, they only provide this feature for colored Petri net models;
2. **Phase-type distributions** (BREUER; BAUM, 2005). Phase-type distributions can be used to approximate general distributions in a Markovian model. The drawback of using it is that it complicates the structure of the model. The scripting language now allows for the representation of a transition delay regarding phase-type models and, therefore, simplifies the models;

3. **Event-based programming** for Petri net simulations. The scripting language allows the creation of user-defined functions, and those functions can be registered as events associated with transitions. This feature allows to create and modify simulation variables, the net marking, parameters, and even the Petri net structure itself;

4. **Object Petri nets**. Our definition of object Petri nets allows not only the tokens but all elements which form a Petri net to be treated as objects. This characteristic implies that we can override the behavior of certain elements such as places or transitions, and apply object-oriented principles when designing our models. This formalism is covered in Section 4.2.3;

5. Support to **meta-modeling** constructs. MSL offers constructs that allow specifying a class of Petri net models based on a model template. This feature is described in Section 4.2.4.

6. **Dynamic reliability block diagrams** for reliability and availability modeling. This formalism combines the simplicity of combinatorial models with the the ability of representing failure dependencies and redundancy arrangements between model blocks. DRBDs are presented in Section 4.2.2.

In (OLIVEIRA et al., 2017), we provide an introduction about the scripting language and the items from 1 to 3 in the previous list. The remaining of this chapter describes the items 3 to 6, as they are essential to the WaaS infrastructure planning case studies of Chapter 6.

4.2.2 Dynamic reliability block diagrams

We implemented the DRBD formalism as an alternative to RBDs and SPNs for creating availability and reliability models. It combines the simple graphical notation of RBDs with the ability to represent failure relationships and redundancy arrangements. To extract metrics from a DRBD model, however, it is necessary to convert it to state-based models that can be solved by numerical or simulation methods.

In our implementation, DRBDs models are a particular case of SPN models. We developed a method for converting the DRBD blocks and structure to SPN models. The choice of SPNs offers some advantages. By converting DRBDs to SPNs models, it is possible to employ numerical evaluation methods (based on CTMC generation) if the model has a manageable state-space and the failure and repair rates are exponential. Large or non-exponential DRBD models can be solved by discrete-event simulation. Additionally, it is possible to use SPN reward rates to obtain metrics such as the Capacity Oriented Availability (COA). This process is illustrated in Subsection 6.1.3.

The following subsections describe the rules for converting DRBD constructs to SPN structures.

4.2.2.1 Single Component

A DRBD block is converted to an SPN with two places and two transitions, as described in Figure 19 (a). A token inside the place with suffix *up* indicates that the block is functional; otherwise, the block is in a non-functional state. Transition c_fail causes the block to go from the functional to non-functional state, and the c_repair transition represents the restoration of the component.

4.2.2.2 Series-Parallel Arrangement Mapping

Similar to classical RBDs, DRBD blocks can be arranged into series and parallel structures. We use a pair of states for each series/parallel segment of the model, as shown in Figure 19 (b). In this example, we have three blocks mapped to SPN according to the rules for single component mapping. The operational state of the whole arrangement is represented by s_up and s_down states connected to two immediate transitions. Both transitions are controlled by guard expressions. In this example, transition s_fail has the following guard expression:

$$(\#c1\_up = 0)OR(\#c2\_up = 0)OR(\#c3\_up = 0)$$

The # operator followed by a place name indicates the number of tokens inside the place. Therefore, this expression evaluates to true if some *up* place loses its token, since a series arrangement requires all components be working in order to be considered operational. A parallel arrangement uses the same expression structure, but uses an AND logical operator, since this arrangement only fails if all components fail at the same time. Transition s_repair has as a guard expression the logical negation of the previous one:

$$NOT((\#c1\_up = 0)OR(\#c2\_up = 0)OR(\#c3\_up = 0))$$

4.2.2.3 SDEP block mapping

The SDEP construct is not present in the classical RBD formalism. A SDEP block is connected to a source block and one or more target blocks, via directed arcs. The failure of the source block provokes the immediate failure of all target blocks. This behavior is represented by the SPN elements shown in Figure 19 (c). It was added an immediate transition ($trigger\_f$) that moves a token from $c2\_up$ to the $c2\_down$ place. Due to the inhibitor arc connected to the $trigger\_f$ transition and the $c1\_up$ place, this transition only fires in case the source component fails, causing the failure of the target components.

4.2.2.4 K-out-of-N Mapping

A *k-out-of-n* structure is a block arrangement that requires at least k out of n components ($k <= n$) be working properly, in order to the k-out-of-n group be considered functional. Figure 19 (d) shows the mapping of a DRBD k-out-of-n mapping to an SPN. In this case, we have two possible outputs. The SPN of the left side shows the mapping of a k-out-of-n block with the traditional semantics of RBD: all blocks can fail or be repaired simultaneously. The place *up_components* contains the number of functional elements from the k-out-of-n structure. Each time the transition *c_fail* fires, it represents a failure of a block. In this case, we assume that all blocks have the same failure rate. Transitions *c_fail* and *c_repair* must be configured to have infinite server semantics, i.e., all tokens can be processed in parallel by the transition.

In a scenario with a limited number of *repair facilities* to serve all failed components, we use the SPN of the right-hand side. In this scenario, the number of components that can be repaired in parallel is determined by the number of repair stations available (*repair_stations* place). To facilitate the assessment of the availability of the whole structure, we added the *k_up* and *k_down* places, with a pair of transitions configured with guard expressions. The guard expression of *c_fail* transition is $\#up\_components < k$, whilst the expression of *c_repair* transition is $\#up\_components \geq k$.

4.2.2.5 SPARE Block Mapping (Cold-Standby)

SPARE blocks represent redundancy arrangements between two components and are useful for representing the cold-standby and warm-standby modes. Hot-standby redundancy can be represented by connecting blocks in parallel. In the cold- and warm-standby modes, a component assumes the role of the primary, and another one acts as a spare. Under normal conditions, the primary provides the specified service. When the primary fails, the spare is activated and starts to provide the service until the primary is restored. In cold-standby mode, the spare is maintained as deactivated, and it is assumed that it does not suffer the effects of failures in this state.

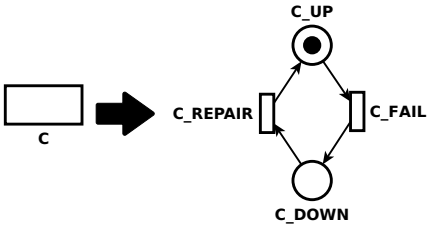
Figure 19 (e) shows the mapping of a SPARE block configured to work in cold-standby mode. Place $c2\_off$ represents the state when the spare component is deactivated. A token is set in $c2\_off$, and place *c2_up* has no tokens in the initial marking. Transition *activate_spare* is only enabled when the primary component fails, due to the inhibitor arc connected to the *c1_up* place. When the primary component is repaired, the transition *deact_spare* fires and moves the token from place *c2_up* to $c2\_off$ place.

Similarly, as for the series/parallel constructs, we added the *s_up* and *s_down* places to represent the operational state of the redundancy arrangement. We use the following guard expression in the s_fail transition: $\#c1\_up = 0$ AND $\#c2\_up =$. The logical negation of this expression is the guard expression of transition *s_repair* .

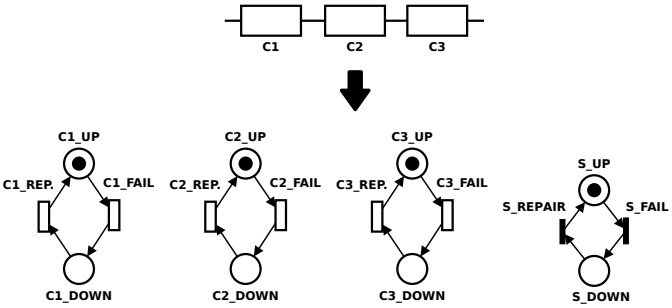
4.2.2.6 SPARE block mapping (Warm-standby)

In the cold-standby mode, the system is unavailable between the moment when the primary component fails and the activation of the spare. If the mean time to activate the spare component is too long, the system availability may become compromised. In the warm-standby mode, the spare component is kept activated; however, it does not receive any workload if the primary component is working. If the primary fails, the spare assumes the requests, with a very short switchover time when compared to the cold-standby mode. The spare may fail while not processing a workload, but generally it has a lower failure rate.

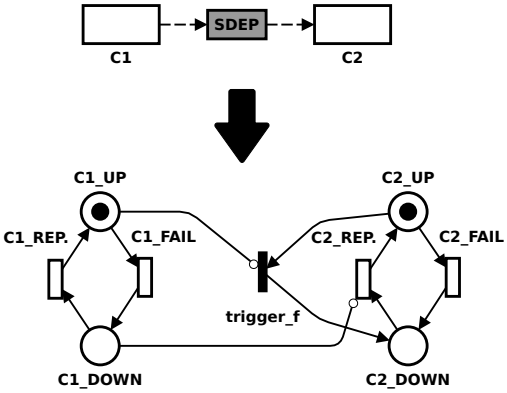
Figure 19 (f) shows the mapping from a SPARE block to a SPN using this redundancy scheme. We added places $c2\_off\_up$ and $c2\_off\_down$ to represent the state when the spare component is not configured to process a workload. The immediate transition $activate\_spare$ becomes enabled when the token goes from $c1\_up$ to $c1\_down$. Notice that the switchover is instantaneous since we are using an immediate transition. If we want to represent the activation time in the model, we could use a timed transition instead.



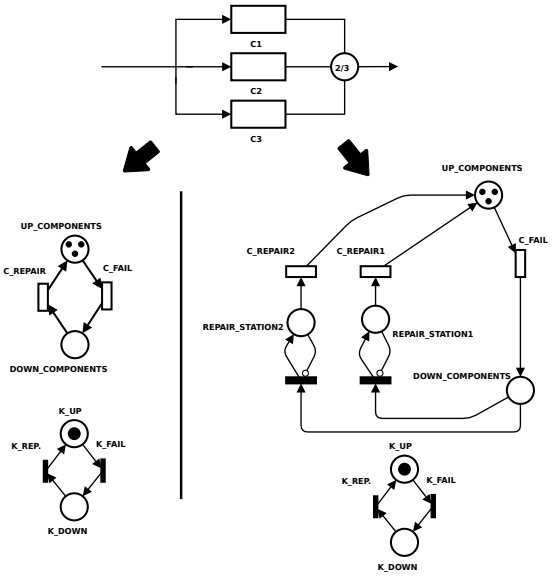
(a) Single component



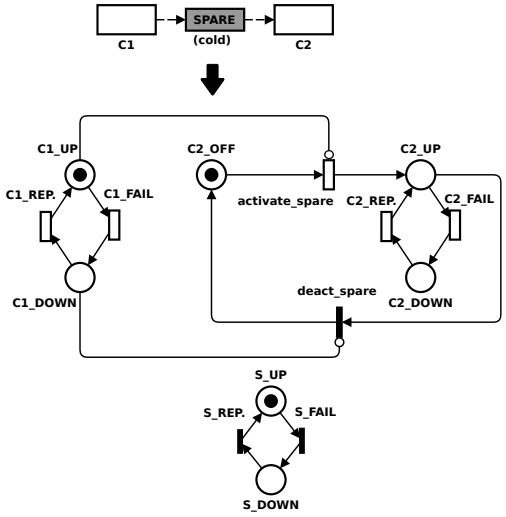
(b) Series/parallel arrangement



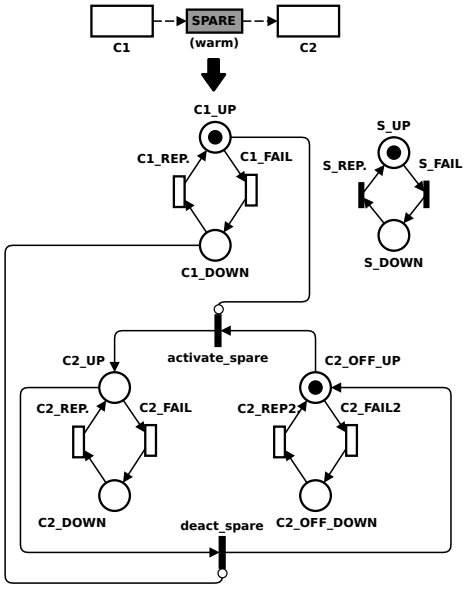
(c) SDEP block



(d) K-out-of-n arrangement



(e) Cold-standby SPARE



(f) Warm-standby SPARE

Figure 19 – Mapping of DRBD structures to Stochastic Petri Nets submodels

4.2.3 Object oriented Petri nets

Modelers can represent a potentially large Markov chain with an equivalent and concise stochastic Petri net. Sometimes, however, even Petri nets can become fairly intricate when trying to model complex real-world phenomena. High-level Petri nets such as colored and object-oriented Petri nets have equivalent descriptive power than Place/Transition nets, but they offer better **descriptive comfort** (LAKOS, 1993), i.e., they offer constructs for creating less intricate models their the P/T nets counterparts. In this work, we adopt the object Petri net formalism as the modeling language for creating performability models of WaaS clouds.

In our Object Petri Net implementation, not only the tokens (as in (VALK, 2004)), but all the elements that compose a Petri net are treated as objects (even the net itself). Thus, we can pass a net-object reference to other objects and form composite structures. An object Petri net instance, as it is defined in this work, is formed by the following tuple.

$$OPNNet = (P, IT, TT, IA, OA, F)$$

, where:

- P is a set of **OPNPlace** instances, $P = (p_1, p_2, p_3, ...)$,
- IT is a set of **OPNImmediateTransition** instances, $IT = (it_1, it_2, it_3, ...)$,
- TT is a set of **OPNTimedTransition** instances, $TT = (tt_1, tt_2, tt_3, ...)$,
- IA is a set of **OPNInputArc** instances, $T = (ia_1, ia_2, ia_3, ...)$,
- OA is a set of **OPNOutputArc** instances, $T = (oa_1, oa_2, oa_3, ...)$,
- F is a set of **OPNFilter** instances, $F = (f_1, f_2, f_3)$.

Figure 20 shows the class diagram for the above-mentioned classes. All classes that define a specific Petri net element (place, transition, arcs, filters), and the **OPNNet** class itself, derive from an abstract class named **OPNObject**. The **OPNObject** class implements a parent-child relationship to express part-whole structures in the model. Certain classes define overridable methods that can be used to intercept certain events during the construction of a Petri net object or during the simulation/space-state generation. In the remaining of this section, we refer to these overridable methods as **hook methods**.

The **OPNNet** class defines the following methods:

- **getEnabledTransitions** - This method returns a list with all transitions that are enabled in the current marking. A transition is enabled if it is possible to form at least one binding with tokens from input places, satisfying the filters that are set for the transition, if any. If there are any immediate transition enabled, no single timed

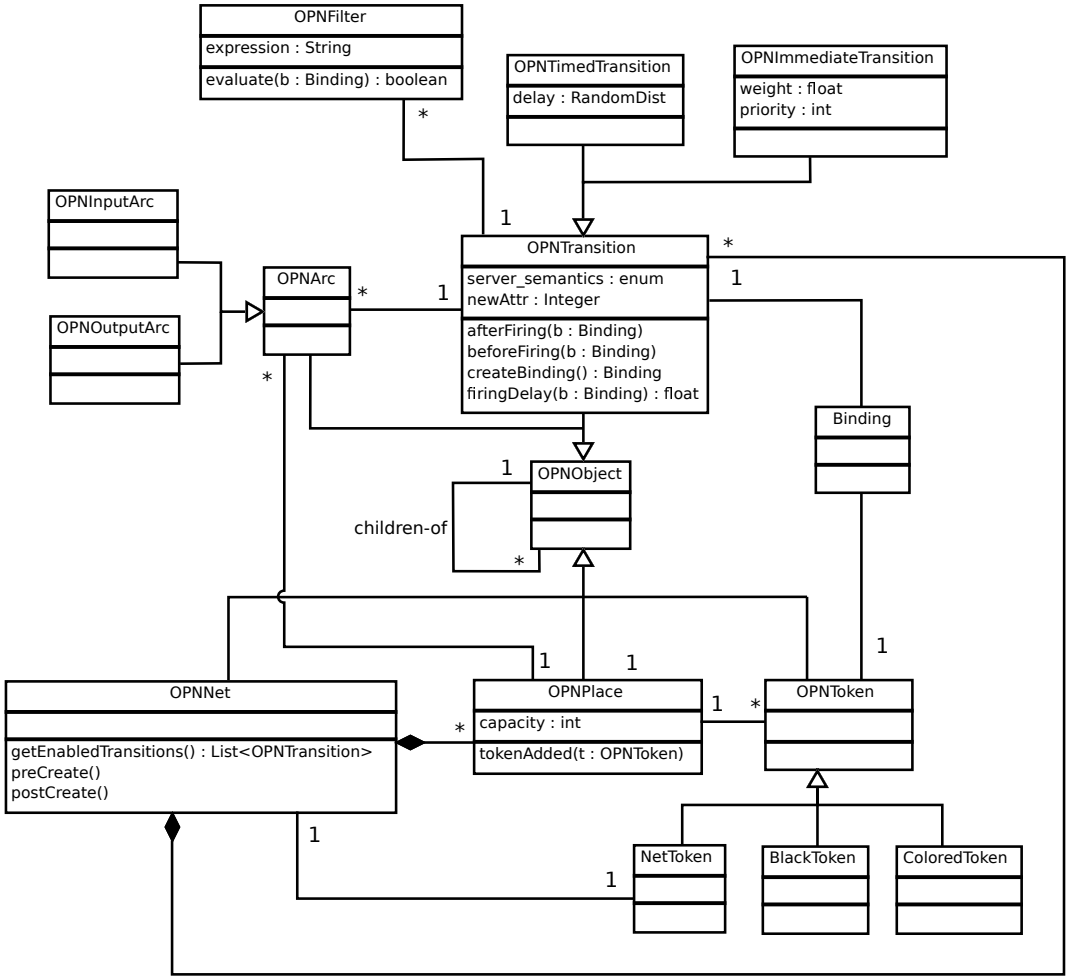


Figure 20 – Object-oriented Petri net modeling framework - class diagram

transition is added to the result list. Additionally, only the immediate transitions with priority equal to the higher priority among them are returned;

- **preCreate** and **postCreate** - The **preCreate** hook method is called after the **OPNNet** instance is created, and before any element (place, transition, etc.) is added to the net. The **postCreate** method is invoked after all elements are added to the net. These methods can be overridden in order to specify custom behavior to be triggered before a simulation or state-space generation process.

The places, transitions, arcs, and filters instances of an **OPNNet** are children elements of the net object. Additionally, an **OPNNet** can contain nested nets as children. This scheme allows for a modular representation of Object Petri nets.

The **OPNPlace** class defines common behavior for places. The tokens of a place are kept in a queue structure. The overridable **tokenAdded** method can be used to capture the event of adding a token to a place. By overriding this method it is possible to manipulate the token instances as they move from one place to another. This feature can be useful to synchronize nested nets with their enclosing nets.

An **OPNPlace** instance should specify a queueing policy for the tokens that are kept in its queue. The queueing policy defines how the tokens are removed from the place on the occurrence of a firing event. Consider, for instance, a place p with the following token queue: (tk_1, tk_2, tk_3, tk_4). By firing a transition t that takes one token from p, the resulting place queue can be:

- (tk_2, tk_3, tk_4), by adopting the FIFO queueing policy;
- (tk_1, tk_2, tk_3), by adopting the LIFO queueing policy;
- (tk_1, tk_2, tk_4), by adopting the random policy and choosing the third token randomly;

An OPN transition can be an instance of the **ONPTimedTransition** or the **OPNImmediateTransition** classes. The base abstract class **OPNTransition** provides the common behavior shared among them. Timed and immediate transitions have the same semantics they have in SPN models: timed transitions, when enabled, schedule a clock and fire at a later point in time. Immediate transitions fire promptly after being enabled, without advancing the clock. The **OPNTransition** class define the following hook methods:

- **beforeFiring**, called when the transition is about to fire;
- **afterFiring**, called after the transition fires;
- **generateDelay**, invoked to generate the firing delay;
- **getBinding**, invoked to select a binding with tokens available from input places;

The **OPNToken** class can be extended to allow the representation of tokens with specific attributes and methods. Object tokens can respond to method invocations as they move from place to place and when transitions process them. We provide three concrete classes that extend the **OPNToken** class for providing black, colored and net tokens. A net token keeps a reference to an **OPNNet** instance.

The **OPNInputArc** and **OPNOutputArc** instances represent the input and output arcs of a transition, as explained in the background section. They do not define any particular hook method. Since object token instances can contain meaningful data as colored tokens in CPNs, **OPNInputArc** and **OPNOutputArc** should declare the bindings that should be formed when a transition fires. The instances of the **OPNFilter** class represent boolean expressions that restrict certain bindings to be formed. In the following subsection, we discuss binding and timing semantics for OPN transitions.

4.2.3.1 Binding semantics for OPN transitions

In elementary Petri nets, the enabling of a transition is dictated by numeric weights of arcs connected to its pre-set, as described in Subsection 2.6. For colored and object Petri nets, numeric weights are no longer sufficient for expressing the firing behavior of a transition. Instead, we annotate input arcs with variable names. A **binding** is the association of tokens from the pre-set of the transition to the variable names annotated on input arcs. As in a colored Petri net, a transition in our object Petri net formalism is enabled if at least one binding can be formed.

To illustrate the concept of bindings, consider the object Petri net of Figure 21. The tokens in the places **P0** and **P1** are instances of the **Person** class, depicted in the class diagram in the right side of the figure. To form a valid binding for the transition T0, a token from **P0** should be associated with the variable name **x** and a token from **P1** should be associated with the variable name **y**. If **T0** follows a random queueing policy, the possible bindings for the initial marking are denoted by the set:

$$Bindings_{T0} = \{(x \rightarrow t_1, y \rightarrow t_2) \mid (t_1, t_2) \in P0.tokens \times P1.tokens\}$$

$$= \{(x \rightarrow t1, y \rightarrow t4), (x \rightarrow t1, y \rightarrow t5), (x \rightarrow t1, y \rightarrow t6), (x \rightarrow t2, y \rightarrow t4), \ldots\}$$

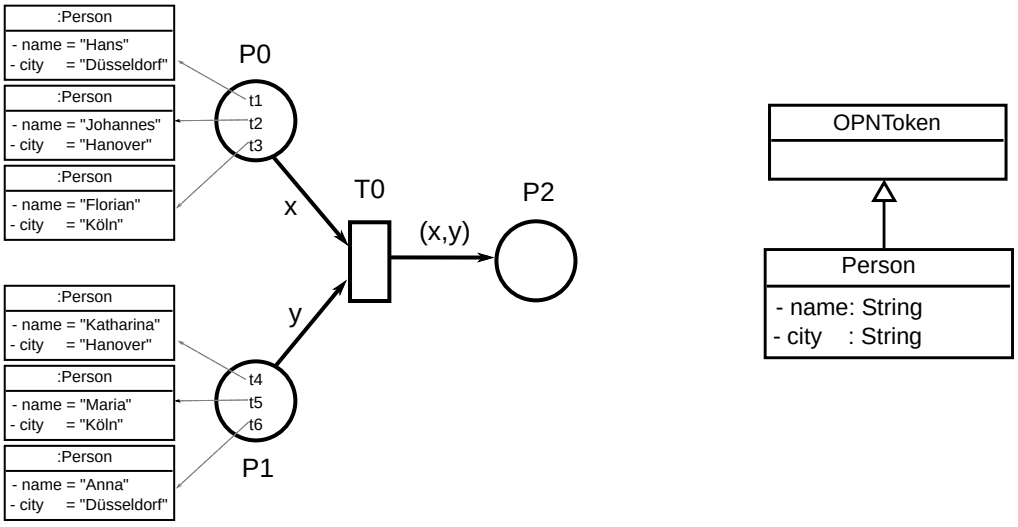


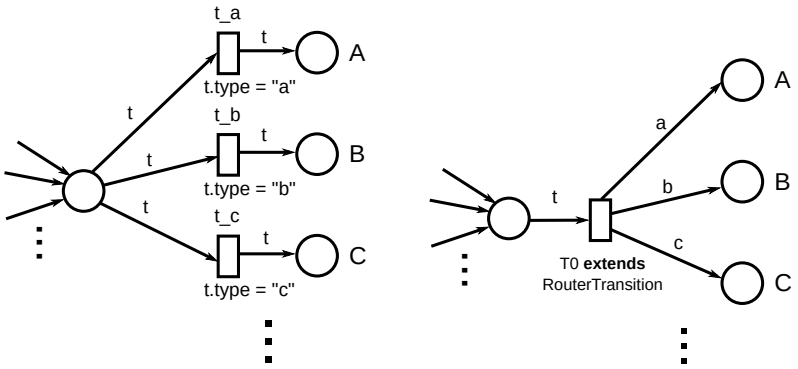
Figure 21 – Example of object Petri net

A transition can be associated with one or more filter elements in order to restrict the bindings that can be formed. A filter element is represented by a boolean expression composed of relational/logic operators and binding elements. For instance, if the transition **T0** is associated with the filter expression $x.city = y.city$, the list of possible bindings will be equal to:

$$\{(x \to t1, y \to t6), (x \to t2, y \to t4), (x \to t3, y \to t5)\}$$

Our modeling framework allows modifying the default firing behavior by overriding the **beforeFiring** hook method declared in the **OPNTransition** class. This mechanism can be used either to create custom behavior associated with a specific transition or to create reusable general purpose special transitions. To exemplify this feature, consider a model that should route tokens to output places according to a label field in the token object. This routing scheme is depicted in the object Petri net of Figure 22 a).

Suppose we want to implement the same behavior of all transitions and filters of the model of Figure 22 a) within a single transition. We can implement this feature by extending the **OPNTransition** class as in the Listing 4.1. The model depicted in Figure 22 b) employs the custom transition class. When using the **RoutingTransition** class, we adopt the following convention: the name of the variable binding corresponds to the type of the token to be routed through the corresponding arc. The other binding variables that are not assigned to any valid token point to a null token. By default, this null token is discarded by the output places.



(a) (b)

Figure 22 – Routing object tokens according to a **type** field

```
class RouterTransition extends OPNTimedTransition{
  @Override
  public void beforeFiring( Binding b ){
    String type = b.token("t").getType();

    b.setBinding( type, b.token("t") );
  }
}
```

Listing 4.1 – Extending the OPNTimedTransition class to implement routing behavior

4.2.4 Meta-modeling

In the literature, meta-modeling is referred to the process of using higher level languages to generate models for some fundamental modeling formalism (BRETON; BÉZIVIN, 2001). For example, since UML is way more widespread than Petri nets, many works in literature propose the conversion of UML diagrams to SPN models for facilitating the task of modeling a particular application domain (BERNARDI; DONATELLI; MERSEGUER, 2002; BENDER et al., 2008; ANDRADE et al., 2009). It is possible also to see Petri nets as meta-models for automata or Markovian models. A particular (non-marked) Stochastic Petri Net can define a class of related CTMC models. By changing the number of tokens of a place, we change the topology of the underlying embedded CTMC.

In this work, we adopt a meta-modeling scheme that consists of defining parameters that can change the structure of a model. In this work, we employ two meta-modeling techniques. Conditional rules define parts of a model that can be included or removed to the final model, based on the result of logical expressions. Repetition rules specify a repeating structure that occurs in the model. The meta-model should be expanded in order to obtain a concrete model which can be used for analysis.

To illustrate how our meta-modeling approach works, consider the classical problem of the dining philosophers. N philosophers are sitting around a circular table. The philosophers alternate between eating and thinking, and they should use two forks for eating. The forks are shared with neighbors, and a philosopher can eat only if the nearby philosophers are thinking (and therefore both forks are free). A Petri net representation of the dining philosophers problem is given in Figure 23. It can be observed that the model has a recurring structure (highlighted in the figure).

The meta-model for creating dining philosophers models with any number of philosophers is given in the script of Listing 4.2. We use two repeating rules, one for declaring the places, and the other for declaring the transitions. The hash (#) symbol is used to specify an identifier with a suffix that is given by a numeric expression, enabling us to create the places $fork_1$, $fork_2$, $fork_3$ dynamically, and so on. The dollar ($) operator is used to retrieving the contents of a variable, similar to the Perl, PHP or Bash languages. When we omit this operator, the value is not retrieved until the model is solved. In short: we use the $ operator in temporary variables (like the i variable in the script), and we do not use this operator when we want to declare a model parameter (the np variable, for instance). We use a conditional rule in order to make each i-th philosopher refer to the fork of the $i+1$-th philosopher, for $i = 1, 2, \ldots, np-1$. The last philosopher, however, refers to the first philosopher's fork, and closes the circle.

```
SPN Model{
  for i in range(1, np){
    place eating#($i);
    place fork#($i)( tokens = 1 );
```

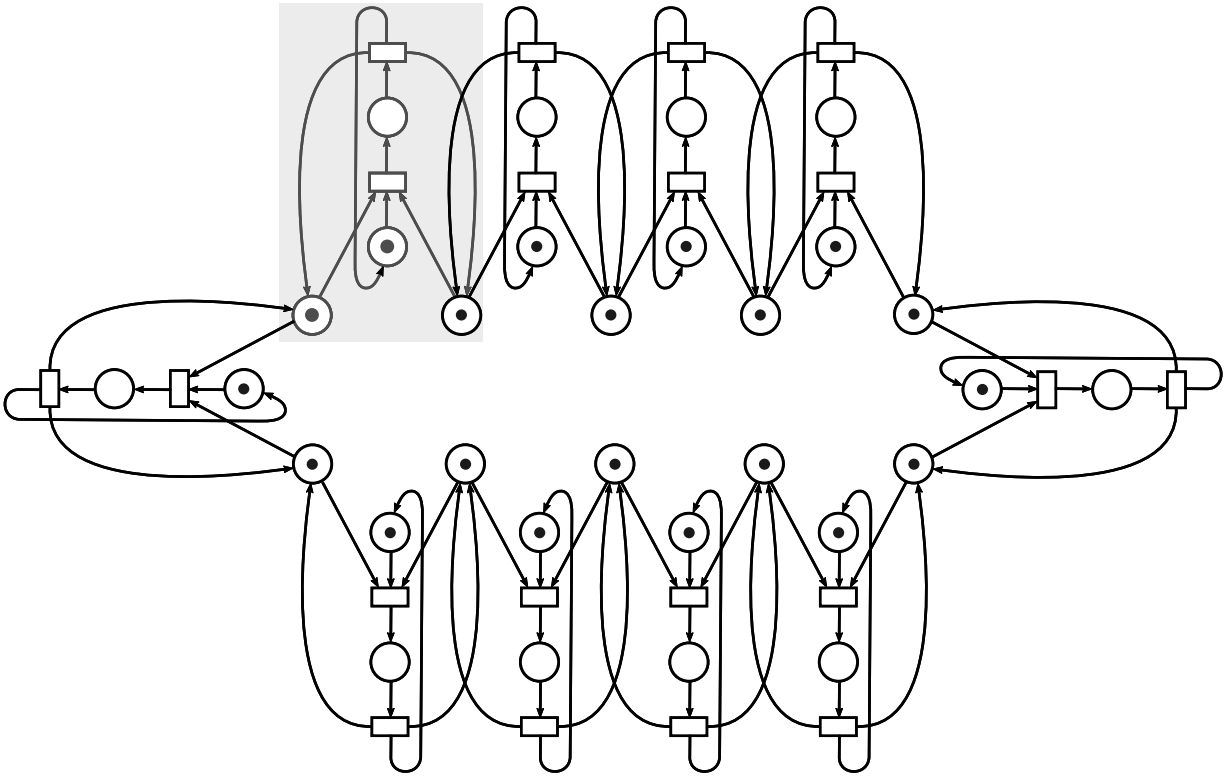


Figure 23 – Dining philosophers Petri net model with 10 philosophers

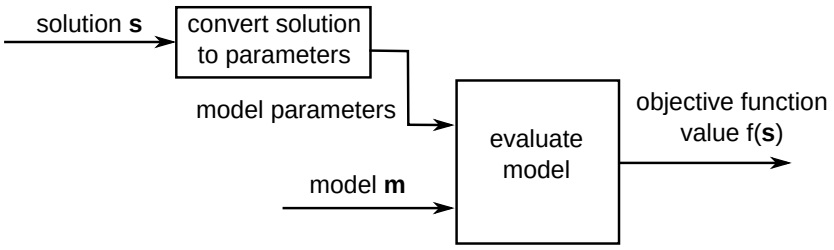
```
    place thinking#($i)( tokens = 1 );
  }

  for i in range(1, np){

    if($i == $np){
      $next = 0;
    }else{
      $next = $i + 1;
    }

    transition eat#($i)(
      inputs = [fork#($i), thinking#($i), fork#($next)],
      outputs = [eating#($i)],
    );
    transition think#($i)(
      inputs = [eating#($i)],
      outputs = [fork#($i), fork#($next), thinking#($i)],
    );
  }
}
```

Listing 4.2 – Mercury script for dining philosophers meta-model

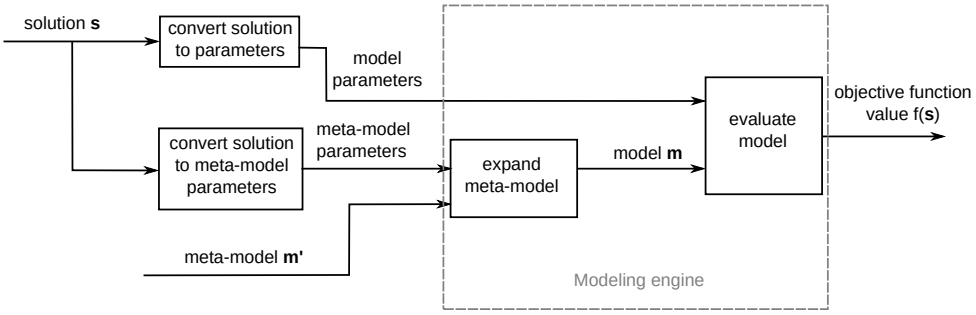
4.2.5 Model-generator algorithms

A crucial step of the method described in Figure 16 is connecting a heuristic or meta-heuristic algorithm to the modeling framework to obtain a objective function value via model evaluation. The algorithmic representation of a solution s should be convertible to a tuple $t = (i, m)$, where i is the a set of input parameters (e.g., transition/state rates, probabilities, etc.), and m is a performance or reliability model.

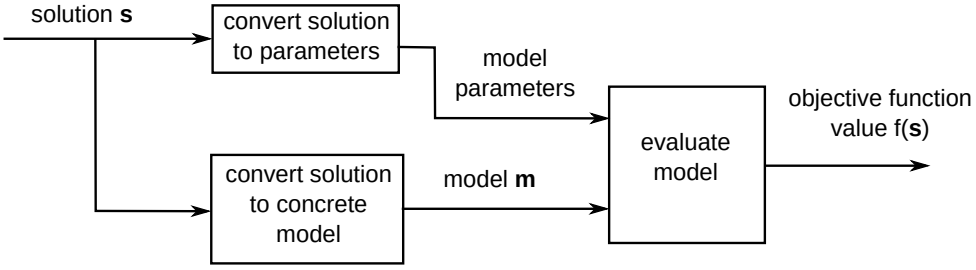
We identified three possible scenarios in which this conversion can happen. They are represented in the Figures 24. The first scenario (Figure 24 a)) represents the simplest scenario. Every tuple t corresponding to a solution s in the solutinon space S share a same fixed model m. Each particular solution represents a different combination of parameters. In the second and third scenarios (Figures 24 b) and c)), each solution s is associated with a different model m that should be generated at running time. The second scenario employs meta-models, i.e., configurable templates that can be expanded to concrete models, and the third scenario employs model generator algorithms.



(a) First scenario



(b) Second scenario (meta-modeling scheme)



(c) Third scenario (model-generator algorithm)

Figure 24 – Converting solution s to pair (p, m) - parameters and model

A model generator algorithm can use the modeling API for on-the-fly model generation, or it can generate a script representation of the model to be loaded at runtime. It is also

possible to combine model generators and meta-models. This approach is adopted in the case study of Section 6.3.

4.3 FINAL REMARKS

This chapter presented the modeling framework proposed for assisting model-based infrastructure planning of Workflow-as-a-Service clouds. The framework supports the requirements of infrastructure planning activities by providing methods for coupling the model evaluator engine to sensitivity analysis and combinatorial optimization tools. On the other hand, it also enables the creation of sophisticated performance and availability models of Workflow-as-a-Service clouds by providing good assistance for hierarchical modeling and high-level modeling formalisms such as DRBDs and OPNs.

5 MODELS

This chapter presents performance and availability models of Workflow-as-a-Service clouds. Section 5.1 introduces the considered WaaS system. First, it provides a general overview of a WaaS cloud, and then it describes the cloud underlying architecture in details. Section 5.2 presents the availability models for the first and second case studies presented in Chapter 6. Section 5.3 presents a combinatorial-optimization model for solving the cloud redundancy allocation problem studied in the second case study. In Section 5.4, we present the object-oriented Petri net model for performability of a cloud workflow application, used in the third case study of Chapter 6. Finally, in Section 5.5, we make some final remarks about this chapter.

5.1 WORKFLOW-AS-A-SERVICE CLOUD - AN OVERVIEW

Figure 25 provides an overview of the modeled system. The cloud infrastructure consists of a collection of hosts that acts either as workers or part of a cloud manager. A work node is a server that runs the virtual machines of the users. The cloud manager is a subsystem composed of the software components for handling the requests of the users and orchestrating the cloud infrastructure. Employing more than one physical machine to run the necessary services is recommended to avoid a single point of failure.

As in (CHRISTODOULOPOULOS et al., 2007), we assume that job requests arrive according to a Poisson process. Each job is composed of a series of subtasks with precedence constraints defined as a DAG. A predefined strategy establishes the scheduling of the subtasks. This strategy determines the number of virtual machines allocated from the cloud provider and the tasks executed on each of them. The scheduling can be generated by some heuristic (First Come-First Served, Shortest Job First, Heterogeneous Earliest Finish Time, etc.) or meta-heuristic algorithm (taboo search, genetic algorithm, ant colony, etc.).

In order to reduce the complexity of the model and, therefore, reducing the simulation time, we assume the instantaneous provisioning of machines to users. It is important to stress, however, that different provisioning strategies could have been implemented in the model in case of a need for evaluating the impact of such strategies on the output metric. In (QUIROZ et al., 2009), for instance, a pool of pre-allocate VMs is suggested as a means of reducing the responsiveness of a cloud platform.

In this work, we adopt the Eucalyptus platform (EUCALYPTUS, 2013) as the basis of our cloud architecture. We chose the Eucalyptus platform since it was the most mature open-source private cloud software at the time this project started. The focus of the Eucalyptus platform enabling the creation of private cloud infrastructures that are fully

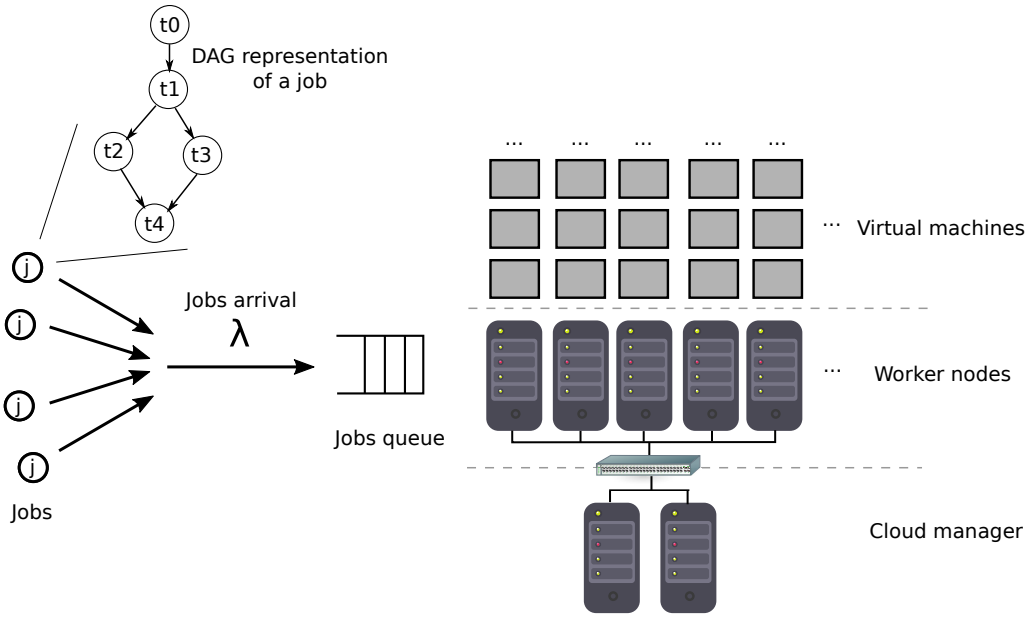


Figure 25 – Modeled system

compatible with the Amazon Elastic Compute Cloud (EC2) and Simple Storage (S3) services.

A Eucalyptus cloud is composed of a set of servers, each of them assuming different roles. The *frontend* servers run the services that manage the entire cloud infrastructure, namely: the *Cloud Controller* (CLC), responsible for the cloud management; *Walrus*, that provides an object storage service compatible with Amazon S3; and a *PostgreSQL* database management system, for storing information about the entire cloud, e.g., users credentials, instances, volumes, and so on.

The *worker nodes* are the servers where the virtual machines will be deployed. They run an instance of the *Node Controller* (NC) software from the Eucalyptus framework. This service has the following functions: discovering the computational capabilities of the nodes (i.e., memory size, number of processors, and disk capacity), sending this information to the Cluster Controller (CC), and accepting requests from the CC (e.g., start a VM, kill a VM, receive a VM image).

The worker nodes can be grouped into distinct clusters named *Availability Zones*. Worker nodes and *cluster managers* form each cluster. A cluster manager runs the following services: the *Cluster Controller* (CC) and the *Storage Controller* (SC). The CC manages a set of worker nodes. It is responsible for balancing the requests sent by the CLC between the worker nodes and monitoring the status and resources of each node. The SC provides a service compatible with the Amazon EBS (Elastic Block Store) service, providing persistent storage volumes to be attached to VMs deployed on worker nodes.

All Eucalyptus services (except the NC) can be configured to work in warm-standby redundancy (BAUER; ADAMS; EUSTACE, 2011) with another instance of the same service on a different machine to avoid single-point-of-failure problems (WOLSKI,).

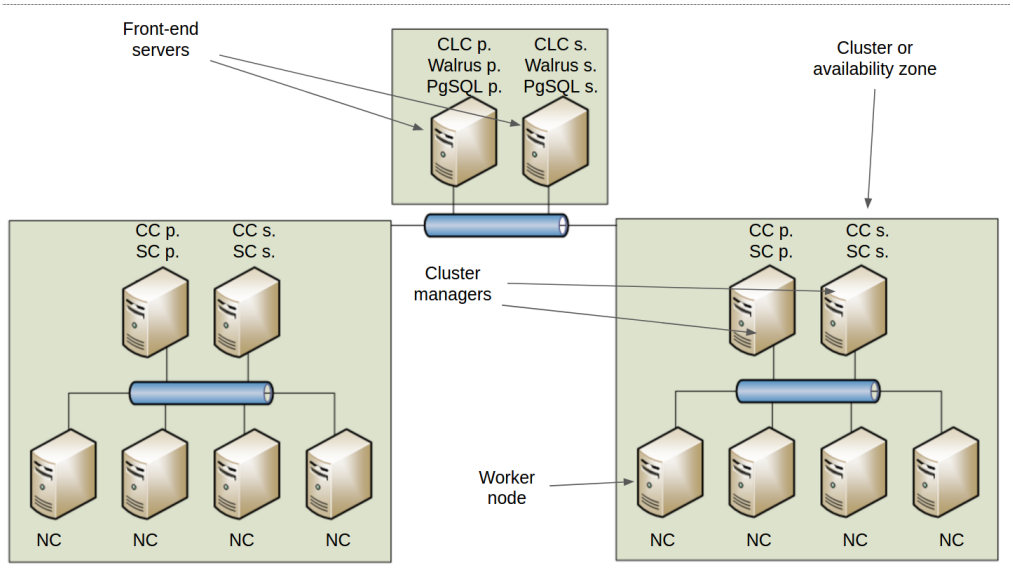


Figure 26 – Eucalyptus cloud architecture with two availability zones

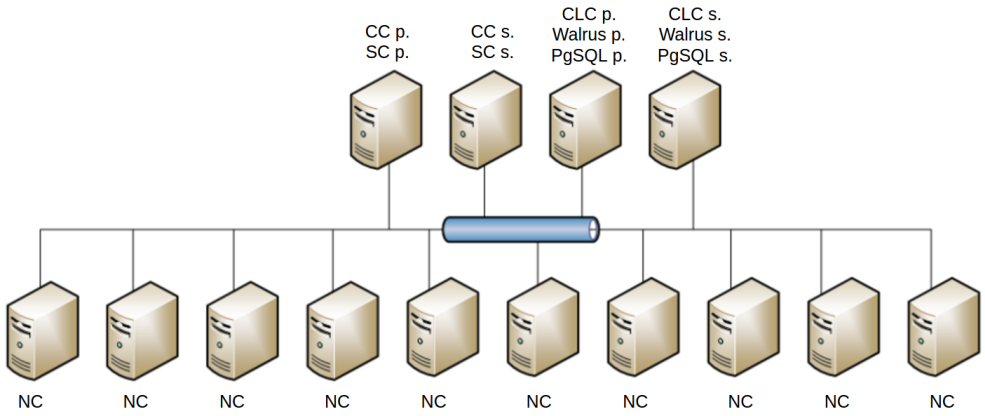


Figure 27 – Eucalyptus cloud architecture with one availability zone

Figure 26 shows an example of a Eucalyptus cloud with two frontend nodes, and two clusters (or availability zones). Each cluster contains two cluster managers and four worker nodes. Figure 27 presents a different cloud architecture with the same number of physical servers, which includes only one availability zone. The point to be noted here is that, for a certain number of physical servers, there are many different ways to create a Eucalyptus cloud. Each possible architecture might have distinct characteristics regarding availability, response time, and other performance and dependability metrics (MACIEL, 2016).

5.2 AVAILABILITY MODELS

In this section, we present the availability models for the cloud architecture described in the previous section. Considering the complexity of a Eucalyptus private cloud, we adopted hierarchical modeling to assess the availability of the entire system. Figure 28 shows the top level RBD model. This model describes the steady-state availability metric from the viewpoint of the cloud administrator. The system is considered to be available if

the infrastructure can support the deployment of a minimum number of VMs to the cloud users. According to our top-level model, the system is available if the cloud controller is functioning and at least one working availability zone is able to provide this minimum number of VMs. The number of blocks for the parallel structure is treated as an input parameter by employing the meta-modeling approach.

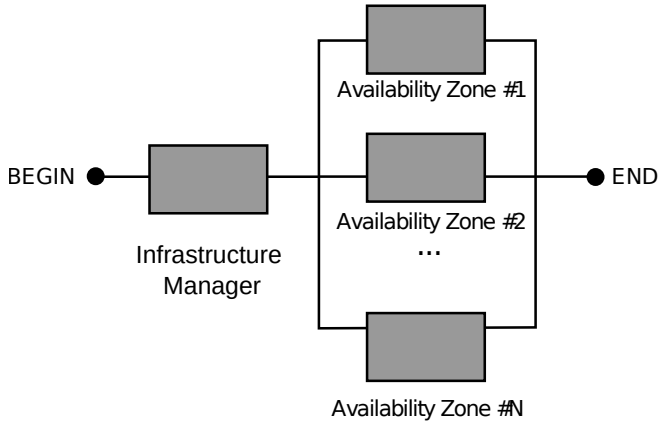


Figure 28 – Top-level model for the cloud infrastructure

In the graphical notation, a shaded (grey) RBD block represents a component evaluated by a sub-model within the hierarchical modeling approach. A white block is a single component described in terms of MTTF (mean time to failure) and MTTR (mean time to repair) values.

The *Infrastructure manager* block corresponds to the set of essential services for managing the cloud infrastructure: the Cloud Controller (CLC), the Walrus storage controller, and the PostgreSQL database. All these three services must be functioning for the users to access the cloud services properly. In order to achieve high availability in this subsystem, each service must be configured to run in conjunction with another standby instance of the same service running on a different machine. When the frontend services run inside the same machine, they also run inside the same JVM. A JVM failure causes the failure of all services that run on it.

The availability model of the infrastructure manager cannot use combinatorial formalisms (RBDs or fault trees) since such formalisms are not able to express failure relationships between the components and redundancy mechanisms. Therefore, we must use state-space based models for this purpose. We adopted the DRBD formalism, which offers the simplicity of traditional RBDs while providing the required semantics. The DRBD model for the cloud controller is presented in Figure 29. It comprises two subsystems in parallel, each of which corresponds to a physical server that runs the management services of Eucalyptus. Each series grouping is composed by the following blocks: Hardware (HW), Operating System (OS), Java Virtual Machine (JVM), Cloud Controller (CLC), Walrus (W), and the PostgreSQL database (PGSQL). We used SDEP blocks to indicate the fault relationships between blocks, and SPARE blocks to represent the warm-standby

redundancy arrangements between the cloud management services. In the model of Figure 29, we adopt SDEP blocks to express that a failure in the Java Virtual Machine provoke the immediate collapse of the CLC and Walrus components, as they are services written in the Java language running in the same JVM. The PostgreSQL database is not affected by JVM failures since it runs on its own process.

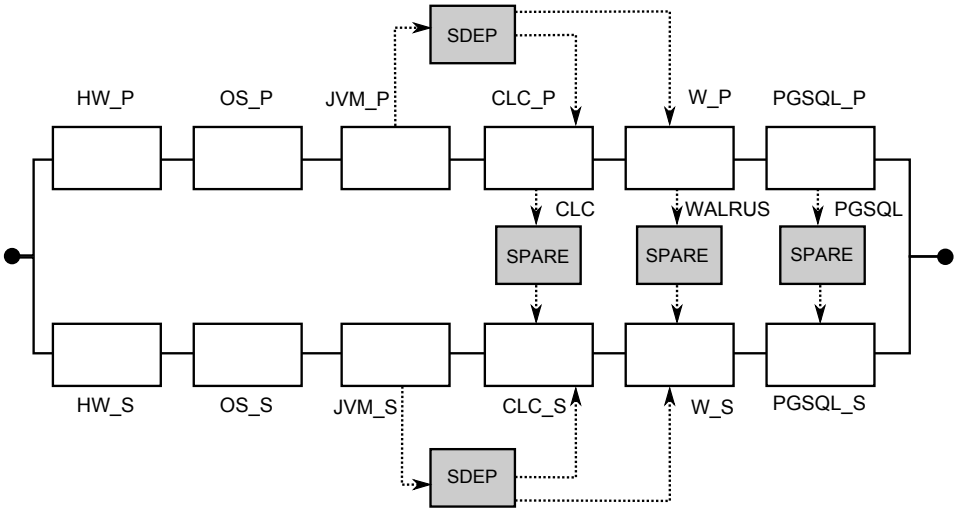


Figure 29 – Availability model for the Infrastructure Manager

The SPN model generated from the infrastructure manager DRBD has a total of 32 places and 52 (timed and immediate) transitions. Creating such a model manually is an error-prone process, and the graphical representation would be difficult to comprehend due to a large number of crossing arcs. This fact demonstrates the conciseness and expressiveness power of our approach.

An availability zone is composed of a cluster manager, a SAN (Storage Area Network) storage, and a set of worker nodes. We consider an availability zone to be working correctly if the three mentioned subsystems are functional. The RBD model for an availability zone is displayed in Figure 30. The cluster manager comprises the Cluster Controller and Storage Controller services. The DRBD model for the cluster manager is presented in Figure 31, and is similar to the infrastructure manager model. Notice that there is no SDEP block connecting the JVM with the CC, since the CC service does not run in the JVM. The SAN storage is depicted as a single block with associated MTTF and MTTR.

The worker nodes subsystem is depicted as a k-out-of-n structure in the model of Figure 30. Each block of this structure is similar to the others and is evaluated by another model. The model for a worker server is represented in Figure 32 by an RBD model composed of the following blocks: Hardware (HW), Operating System (OS), Virtual Machine Monitor (VMM), and Node Controller (NC).

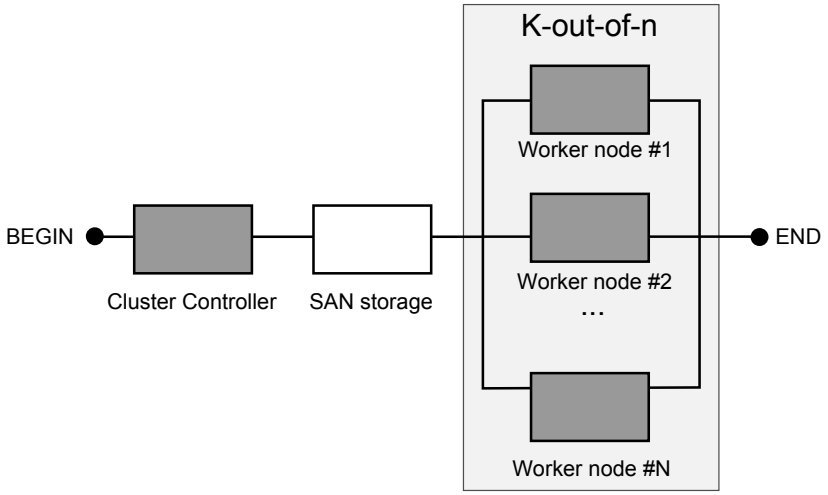


Figure 30 – Availability zone model

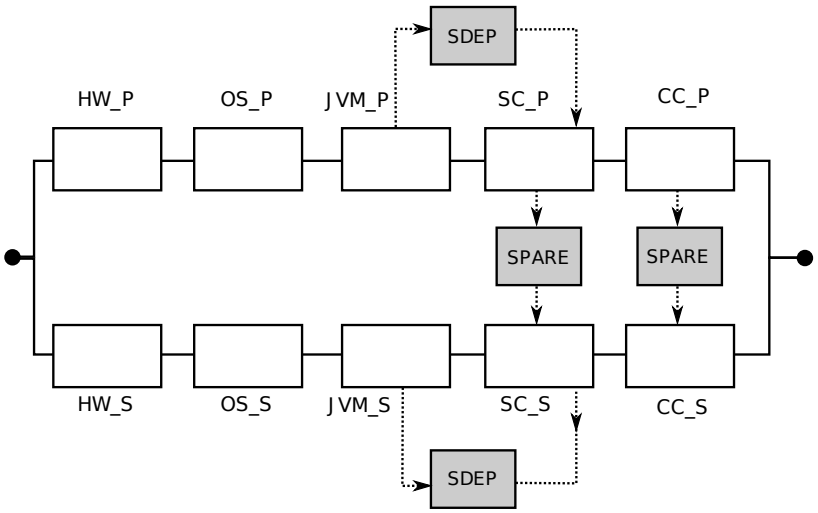


Figure 31 – Cluster Manager model

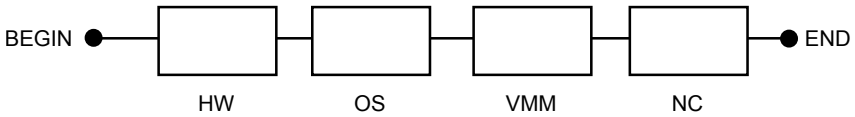


Figure 32 – Worker node model

5.3 CLOUD REDUNDANCY ALLOCATION PROBLEM OPTIMIZATION MODEL

Section 5.1 shown that having a limited number of resources available to create a private cloud, there will be many different forms to arrange those resources to compose the cloud. In this section, we propose a combinatorial optimization model for solving the following question:

Considering the available resources, and a set of constraints, such as minimum availability and a minimum number of workers that must be available, can we find near-optimal configurations regarding availability (steady state and capacity oriented) and acquisition cost?

The cloud infrastructure is composed of a group of n servers, whose possible roles are *manager* (of a cluster or the entire cloud) or *worker*. The role is defined by the software that we install on the server: if we install the *Node Controller* (NC) it will be a worker; otherwise, it will be a manager. The cloud could also be divided into m clusters/availability zones. The following definition describes a Eucalyptus configuration as a solution of the optimization model.

A Eucalyptus cloud configuration S is defined as:

$$S = (\boldsymbol{CLC}, \boldsymbol{W}, \boldsymbol{CC}, \boldsymbol{SC}, \boldsymbol{PGSQL}, \boldsymbol{NC}, \boldsymbol{AV\_ZONE}),$$

a seven-tuple of vectors, where:

1. $\boldsymbol{CLC}$, where $CLC_i \in \{0, 1\}$, and $i = \{1..n\}$
2. $\boldsymbol{W}$, where $W_i \in \{0, 1\}$, and $i = \{1..n\}$
3. $\boldsymbol{CC}$, where $CC_i \in \{0, 1\}$, and $i = \{1..n\}$
4. $\boldsymbol{SC}$, where $SC_i \in \{0, 1\}$, and $i = \{1..n\}$
5. $\boldsymbol{PGSQL}$, where $PGSQL_i \in \{0, 1\}$, and $i = \{1..n\}$
6. $\boldsymbol{NC}$, where $NC_i \in \{0, 1\}$, and $i = \{1..n\}$
7. $\boldsymbol{AV\_ZONE}$, where $AV\_ZONE_i \in \{0..m\}$, and $i = \{1..n\}$

The vectors: $\boldsymbol{CLC}$, $\boldsymbol{W}$, $\boldsymbol{CC}$, $\boldsymbol{SC}$, $\boldsymbol{PGSQL}$, and $\boldsymbol{NC}$, express if the corresponding service is installed on i-th node (1 if the service is installed, 0 otherwise). The $\boldsymbol{AV\_ZONE}$ vector defines the availability zone of i-th node. If AV_ZONE_i is equals to zero, the i-th node is a cloud manager, otherwise, the value indicates the zone that the node belongs (as a cluster manager or worker node).

As an illustration, Table 2 provides the representation of the cloud depicted in Figure 26 as a solution of our optimization model. The vectors are represented as the columns of the table. By inspecting the elements on each vector, we can determine the role and availability zone of each server. In our example, the fifth server is a cluster manager of the first availability zone, running the CC and SC services ($CC_5 = 1$, $SC_5 = 1$, and $AV\_ZONE_5 = 1$).

The optimization model for the cloud redundancy allocation problem can be described as:

$$\textit{maximize } A(\boldsymbol{S}),$$

subject to:

Table 2 – Matricial representation of the cloud from Figure 26

i	CLC[i]	WALRUS[i]	CC[i]	SC[i]	PGSQL[i]	NC[i]	AV_ZONE[i]
1	1	0	0	0	0	0	0
2	1	1	0	0	0	0	0
3	0	1	0	0	1	0	0
4	0	0	0	0	1	0	0
5	0	0	1	1	0	0	1
6	0	0	1	1	0	0	1
7	0	0	0	0	0	1	1
8	0	0	0	0	0	1	1
9	0	0	0	0	0	1	1
10	0	0	0	0	0	1	1
11	0	0	1	0	0	0	2
12	0	0	1	1	0	0	2
13	0	0	0	1	0	0	2
14	0	0	0	0	0	1	2
15	0	0	0	0	0	1	2
16	0	0	0	0	0	1	2
17	0	0	0	0	0	1	2

1. $$1 \leq |ccNodes(j)| \leq 2, \forall j \in \{1..m\}$$

2. $$1 \leq |scNodes(j)| \leq 2, \forall j \in \{1..m\}$$

3. $$1 \leq \sum_{i=1}^{n} CLC_i \leq 2$$

4. $$1 \leq \sum_{i=1}^{n} W_i \leq 2$$

5. $$1 \leq \sum_{i=1}^{n} PGSQL_i \leq 2$$

6. $$\sum_{j=1}^{m} ncNodes(j) \geq minWorkers,$$

where the objective function $A(S)$ is the steady-state availability of the solution S. This value is calculated by converting the vector representation to an RBD model. The functions $ccNodes(j)$, $scNodes(j)$, and $ncNodes(j)$ return a set containing the indexes of the servers

of an availability zone j that run the CC, SC, and NC services, respectively. They are defined as:

$$ccNodes(j) = \{i | AV\_ZONE_i = j, CC_i = 1\},\ i = \{1..n\} \tag{5.1}$$

$$scNodes(j) = \{i | AV\_ZONE_i = j, SC_i = 1\},\ i = \{1..n\} \tag{5.2}$$

$$ncNodes(j) = \{i | AV\_ZONE_i = j, NC_i = 1\},\ i = \{1..n\} \tag{5.3}$$

In our example, $ccNodes(1)$ is $\{5, 6\}$ and has cardinality equals to $|ccNodes(2)| = 2$.

Constraints 1-5 define the minimum and maximum of each management service instance. In a Eucalyptus cloud, we could have one or two instances of the CLC, WALRUS and PostgreSQL services, and one or two instances of the CC and SC services per cluster. Constraint 6 imposes that the number of worker nodes must be greater than the $minWorkers$ parameter.

5.3.1 Neighbor function

For the i-th server, the number of possible combinations is equal to $2^5 * (m + 1)$, since the CLC_i, W_i, CC_i, SC_i, $PGSQL_i$, and NC_i elements have two alternatives (0 or 1), and the AV_ZONE_i element has $(m + 1)$ alternatives (0 for cloud managers, or the number of the availability zone). Therefore, for a cloud formed by a set of n physical servers divided into frontend machines and m clusters, the number of possible solutions is $(64m)^n$. However, the vast majority of solutions will violate the optimization problem constraints, and therefore will be unacceptable solutions. Moreover, among the feasible solutions, there will be many solutions representing the same architecture.

Therefore, we must apply a strategy for helping to reduce the search space, and avoiding the generation of unacceptable or repeated solutions. A heuristic was implemented on the neighbor function of the optimization algorithm. The algorithm starts with a randomly generated solution and starts an iterative process to refine its quality. On the first iteration, it applies the neighbor function to the initial solution in order to generate a set of slightly modified solutions. Then, it compares the value of the objective function for the initial solution and neighbors and then selects the best solution found. In the next iterations, it will apply the neighbor function to the best solution of the previous step, until it satisfies some stopping criteria (e.g. performing a number of iterations).

The local search algorithm randomly chooses one of the following operations to produce a neighbor solution:

1. **Merge two availability zones** - Two availability zones i and j are selected randomly. All servers of zone j are converted into worker nodes of zone i;

2. **Split one availability zone** - One availability zone i is randomly selected, and half of the nodes of the zone will generate another availability zone. This operation could generate an unfeasible solution, since the modified zones could have an unacceptable number of worker nodes;

3. **Exchange nodes between zones** - Two availability zones i and j are selected randomly. One worker node from availability zone i is transferred to zone j;

4. **Cluster controller to worker** - One availability zone i is selected at random. If the cluster i has two or more manager nodes, a manager node is converted into a worker node, and the cluster management services are reallocated to the remaining managers;

5. **Worker to cluster controller** - One availability zone i is randomly selected. If the zone has less than four cluster managers, a worker node from i is converted into a manager;

6. **Cluster node to cloud manager** - One availability zone i is randomly selected. If the cloud has less than six cloud managers, a worker node is removed from the zone, and is converted into a cloud manager;

7. **Cloud manager to cluster node** - One availability zone i is selected at random. If the cloud has two or more cloud managers, one of them is removed from the cloud manager group and converted to a worker node of the cluster i.

5.3.2 Objective function

The objective function $A(S)$ of the optimization model described in this section converts a cloud architecture described in the matrix representation to an RBD availability model. The objective function value of a solution s is the steady-state availability metric obtained from this model. We employ a model generator algorithm that takes as the input a matrix representation of a cloud deployment and produces as the output an RBD availability model.

The DRBD model of Figure 29 suffers from state-space explosion if we represent three or more frontend nodes. Therefore, we had to adopt different submodels for the infrastructure and cloud managers blocks. Instead of using DRBD models, we employed RDB models with repeating blocks to represent the distribution of the Eucalyptus services among the frontend and cluster manager nodes. The pure RBD models are less accurate than the DRBD equivalent, but they are more suited to the objective function of the problem.

The RBD model for the Infrastructure Manager is represented in Figure 33. It is composed of three blocks in series: *CLC* (Cloud Controller), *Walrus*, and *PGSQL*. The submodels of each block (CLC, W and PG) are depicted in Figures 34 a), 34 b), and 34 c), respectively. The repeated blocks, namely, $HW1$, $OS1$, $JVM1$, $HW2$, $OS2$, and $JVM2$, represent the same component appearing in different places in the model. For solving an RBD model with repeated blocks, we use the pivotal decomposition technique presented in Section 2.4.

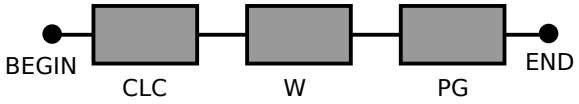
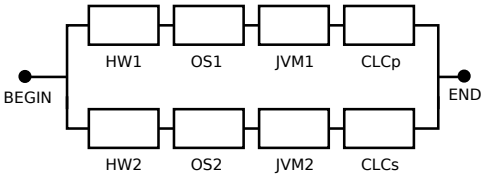
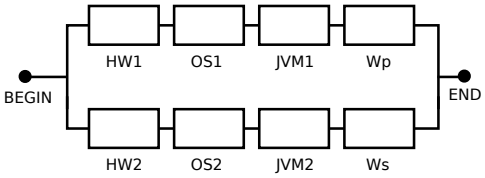


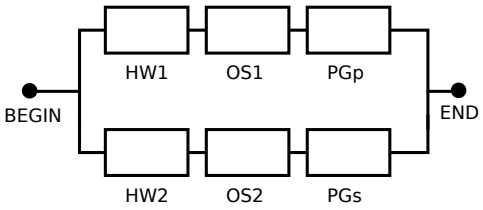
Figure 33 – Infrastructure manager model



(a) CLC refined model



(b) W refined model



(c) PG refined model

Figure 34 – Infrastructure manager RBD submodels

The Cluster Manager model is shown in Figure 35, and it is similar to the Infrastructure Manager model. Its submodels are described in Figures 36 a) and b).

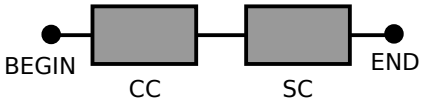
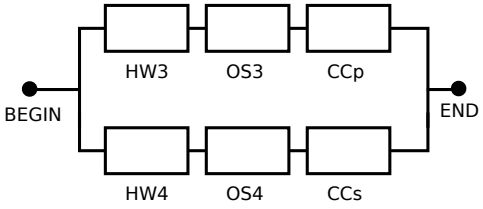


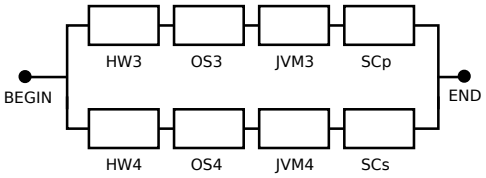
Figure 35 – Cluster Manager model

5.3.3 Capacity Oriented Availability

In the studied system, some failures do not provoke system unavailability but can degrade performance. A cloud will not be considered unavailable after some failures on workers nodes if it can still provide a minimum number of workers. However, having fewer worker



(a) Cluster Controller (CC)



(b) Storage Controller (SC)

Figure 36 – Sub models from the Cluster Manager model

nodes available means that the processing capability of the cloud was reduced. In the same way, a failure of a cluster manager will not cause unavailability if the cloud has another functional cluster, but half of the worker nodes of that cloud will be inaccessible to the users while the cluster manager is unavailable.

We can measure the impact of failures on the system performance with the Capacity Oriented Availability metric. This metric measures the proportion of the whole system that is being actually delivered to its clients. It is a value ranging from 0 (no service is delivered at any point in time) to 1 (the system is able to deliver its full processing power all the time).

The COA metric can be evaluated by using a Markov Reward Model (REIBMAN; SMITH; TRIVEDI, 1989). In this model, each state is associated with a reward rate that represents the processing power that is being delivered in that state. The metric is obtained by summing up the product between the reward and the stationary probability of each state:

$$COA = \sum_{i=0}^{n} \pi_i * r_i,$$

where n is the size of the state space of the Markov Reward Model, π_i is the stationary probability of being in state i, and r_i is the reward rate assigned to the state i.

The COA model for the studied cloud architecture can be obtained by converting the RBD availability model to a SRN, and then converting the SRN to a Markov Reward Model. Figure 37 displays the corresponding COA model of the cloud architecture shown in 26. The pair of places CLM_UP/CLM_DOWN corresponds to the operational status of the cloud manager. The $CM1\_UP/CM1\_DOWN$ and $CM2\_UP/CM2\_DOWN$ pairs correspond to the cluster managers, and the $W1\_UP/W1\_DOWN$ and $W2\_UP/W2\_DOWN$ pairs represent the worker nodes in both availability zones. The firing time of the failure and repair transitions are obtained by computing the MTTF and MTTR of the RBD

models presented in this section. The transitions associated with the failures and repairs of workers nodes have infinite server semantics.

The COA metric for the cloud of Figure 26 is given by the reward rate:

$$r_i = \begin{cases} \text{available\_nodes}/(w1 + w2), & \text{if } \#CLM\_UP = 1 \\ 0, & \text{otherwise} \end{cases}$$

$$\text{available\_nodes} = (\#CM1\_UP \times \#W1\_UP + \#CM2\_UP \times \#W2\_UP)$$

The notation $\#p$ represents the number of tokens in p. The variables $w1$ and $w2$ represent the number of worker nodes on each availability zone. The expression $\#CM\_i\_UP \times \#W\_i\_UP$ denotes the number of available worker nodes on the availability zone i. If the cluster manager (CM) of the i-th availability zone is up, $\#CM\_i\_UP$ is equal to one and W_i is equal to the number of available workers in the i zone. If $\#CM\_i\_UP$ is equal to zero, there is no available worker, since the cluster manager must be operational in order to the workers be accessible.

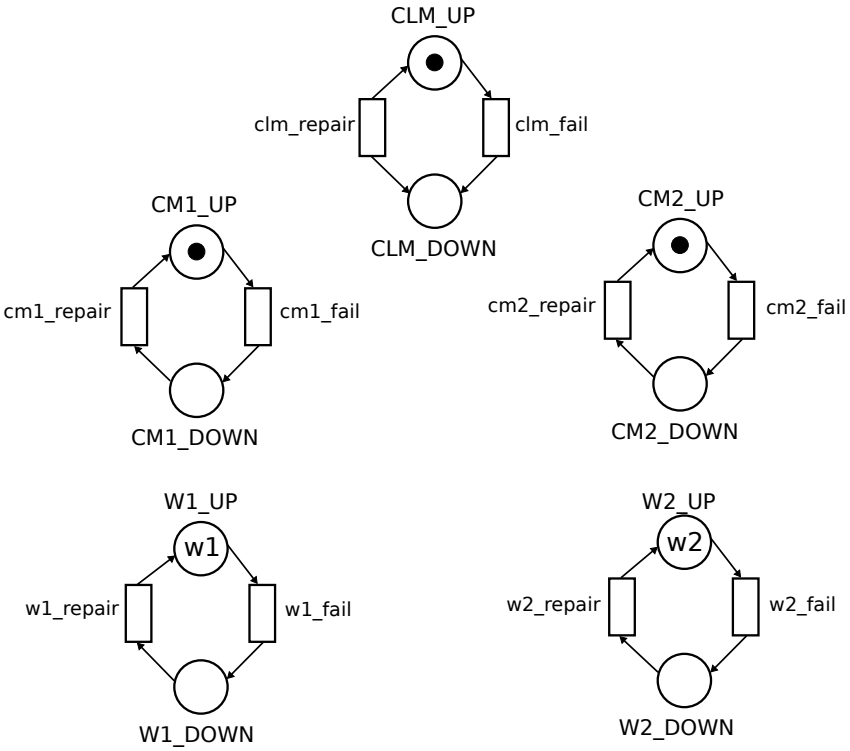


Figure 37 – Capacity oriented availability model

5.4 PERFORMABILITY MODEL

The proposed performability model is an Object Petri net that emulates the cloud infrastructure depicted in Figure 25 subjected to an influx of job submissions. It can be used for obtaining performance metrics at cloud level, such as job throughput, waiting time, and response time. It also allows measuring the impact of failure events on those metrics. Figure 38 shows the conceptual view of the performability model as a UML object diagram. It is composed of two Petri nets, namely, the performance and the availability model. The performance model represents the workload (job arrivals), the scheduling of resources (virtual machines to the jobs), job executions, and the releasing of virtual machines. The UML association denotes that the performance model knows (keeps a reference to) the availability model. The availability model represents the failure and repair events which affect the performance model.

In the following subsections, we describe the performance, availability, and job execution submodels.

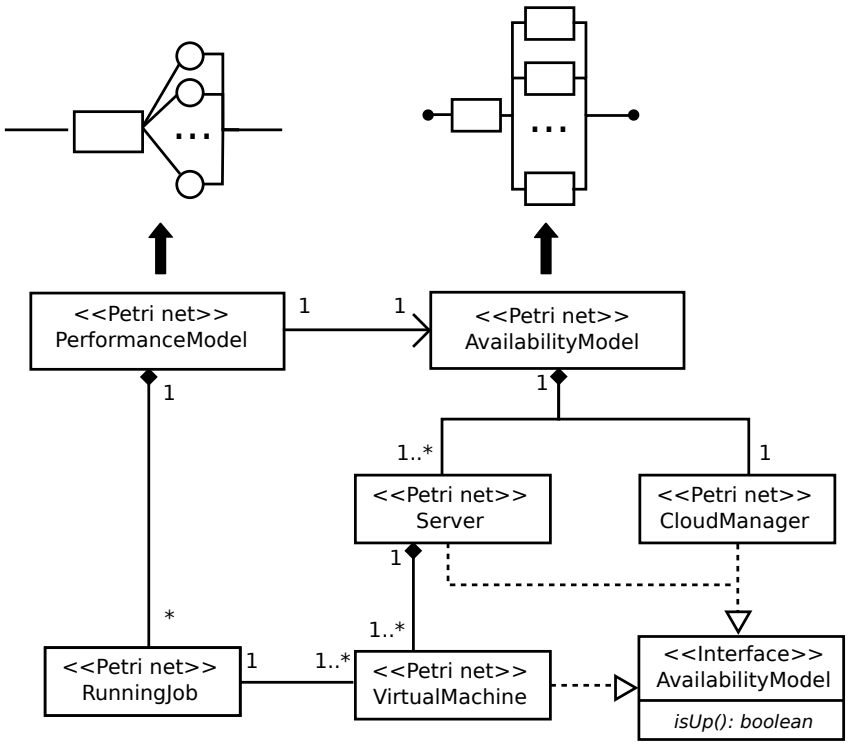


Figure 38 – Class diagram performability model

5.4.1 Performance model

The performance model represents the following events:

- arrival of jobs;
- queuing of jobs, in case of unavailability of resources;

- discarding of jobs, in case of a full queue,
- allocation of resources when a job is scheduled to execution
- the release of resource when a job is done (or when it failed during execution).

Figure 39 depicts the performance model as an open queue accepting a job submission influx with rate equals to λ. Each job will be promptly executed in case there are available resources in the cloud. Otherwise, it will be enqueued or discarded (if the queue is full). A job will also be discarded if the cloud manager is unavailable since in this case, it is not possible to allocate the cloud resources for the job execution. Another possibility for the failure of a job submission is when some virtual machine failure prevents the execution of one or more subtasks. A virtual machine can fail due to software (operating system or hypervisor) or hardware faults.

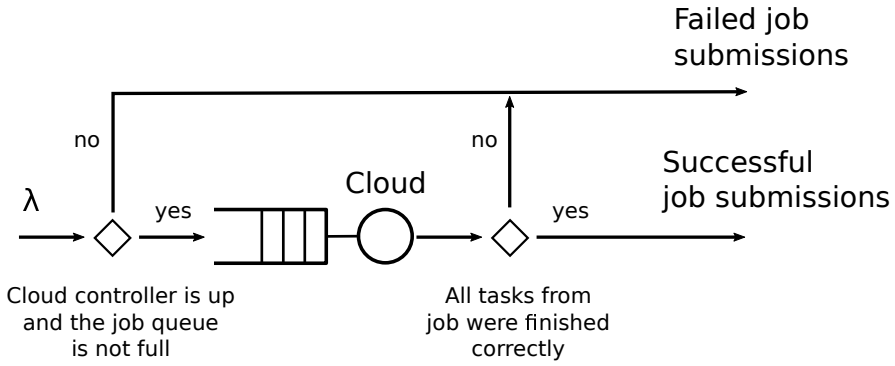


Figure 39 – Performance model (high level description)

Figure 40 illustrates the life cycle of a job. The **ready** state refers to a newly created job about to be submitted to the cloud. After being submitted to the cloud for execution, it enters the queue of jobs, moving to the **queued** state. If there are sufficient resources for running it (i.e., available VMs), it allocates the necessary resources and begins to execute (entering the **running** state). If a VM failure occurs during the execution of a computing/communication task, the whole job is aborted, and the job reaches the **failed** final state. If all tasks are successfully completed, it moves to the final **completed** state.

In the Figure 41, we provide a detailed description of the performability model as a Stochastic Object Net. The transition **arrivals** represents the workload generation. Each time this transition fires, a new job request token is sent to the **job_queue** place. The **job_request** object token contains information about the DAG to be executed (its runtime characteristics) and the scheduling to be applied (number of VMs to be provisioned and the distribution/ordering of tasks on them). The **discard** transition is fired when the queue capacity is full. The **schedule** transition fires when the number of tokens in the **available_VMs** place is greater than the **vms** attribute of the request token in the **queue** place. The **available_VMs** place contains net tokens corresponding to idle and

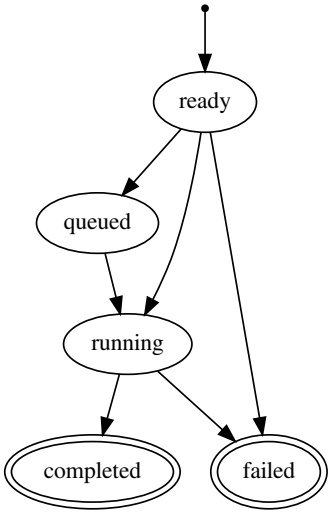


Figure 40 – Life cycle of a job submission

functioning VMs, i.e., VMs that are operational and that were not allocated by a job request.

A token in the **running** place corresponds to an active job execution (i.e.: in the state **running** according to Figure 40). A **JobToken** instance models the execution of the workflow application. The associated nested net will execute and reach an absorbing state. If no failure occurs during the execution of computing/communication tasks, the final state will be marked as **completed**. Otherwise, the final status will be marked as failed. When a **JobToken** instance reaches an absorbing state, it will be moved to the **completed_jobs** or **failed_jobs** places according to its status. The provisioned VMs are released to the pool of idle VMs.

5.4.2 Availability model

Failure of virtual machines and physical servers have a substantial influence on the system performance. Job requests are enqueued if the cloud cannot provide the specified number of virtual machines. The decrease in the number of virtual machines, due to the effect of failures, causes an increase in the average waiting time and a reduction on the system throughput. When a job request completes, the allocated virtual machines are released and become available for processing further requests. If a failure occurs during the processing of a job, the failed virtual machines/worker nodes are subject to repair, and the processing is canceled. In case of a failure on the cloud manager, arriving job requests are discarded, since this is the component that acts as an interface between the users and the cloud infrastructure.

The availability sub-model describes the operational state (up/active or down/failed) of the cloud manager and the worker nodes and running virtual machines. It is represented as an SPN model integrated into the performability model. The cloud manager is defined as a fixed structure and the worker nodes and virtual machines are dynamically generated

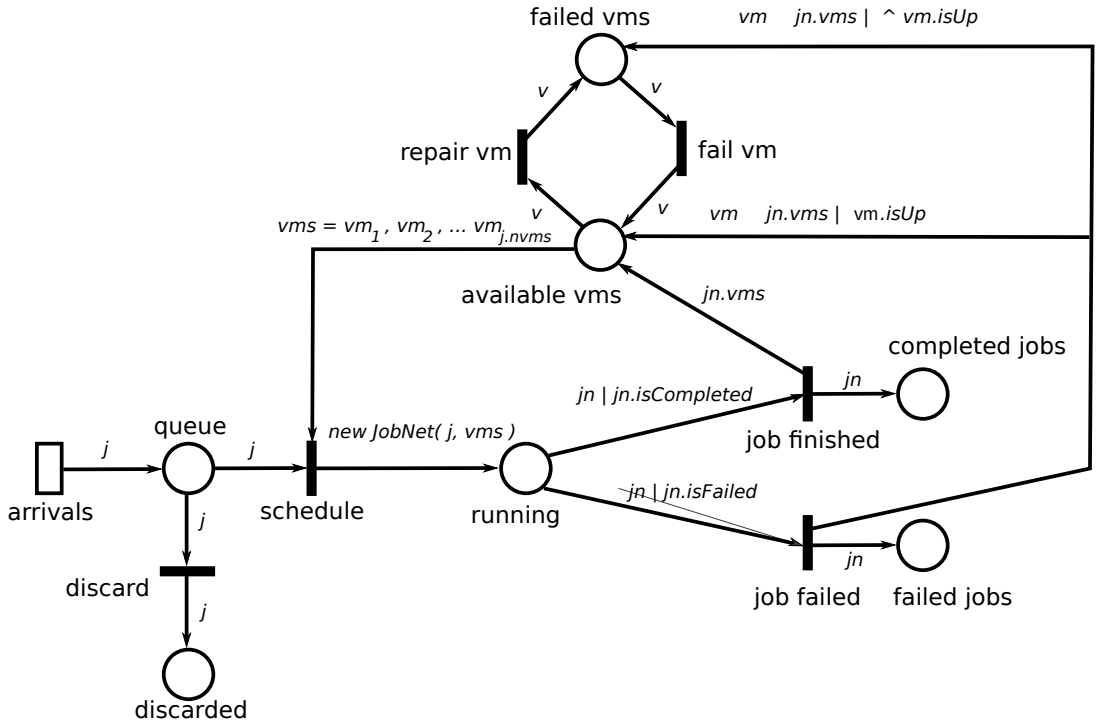


Figure 41 – Performance model (detailed Stochastic Object Net description)

according to two parameters: the number of worker nodes and the maximum number of running virtual machines per server. Figure 42 illustrates the model generated for two worker nodes with two virtual machines per node. Each component is defined as a pair of places for defining the operational state of the component, and a pair of transitions for causing failure/repair events. An immediate transition controlled by an inhibitor arc is used for updating the operational state of every virtual machine associated to a physical server.

Since DRBD models are convertible to SPNs, we developed a mechanism that converts SPNs to OPNs in order to reuse the infrastructure manager model depicted in Figure 29 into the performability model. If the infrastructure manager is in a failed state, the **schedule** transition from the performance model disables. This situation may incur in jobs being discarded, even if there are available and idle VMs.

5.4.3 Job model

The job model simulates the execution of tasks and their communication events. A job model is instantiated in the performance (top-level) model, when a job token is scheduled, and moves to the **running** place. The job model does not have a fixed structure, being algorithmically generated instead. The Algorithm 1 is a pseudo-code representation of the job model generator algorithm. This algorithm takes a DAG and scheduling as input, and produces an SPN performance model as output. A different SPN model will be generated for each particular scheduling. In the resulting model, each task is represented

by a place-transition pair. The place represents the inputs for the task and the transition represents the task processing event. A processing transition for a task t must forward a token for each input place of dependent tasks. For a pair of tasks related by a precedence relationship, there are two possibilities:

- **The tasks are scheduled to the same VM.** In this case, the processing transition from the source task has an output arc connected to the inputs place from the destination task.
- **The tasks are scheduled to different VMs.** In this case, it is necessary to include an additional place-transition pair for representing the communication. This transition is connected to the inputs place from the destination task via an output arc.

Besides the representation of data relationships from the DAG structure, the SPN model should express the temporal constraints imposed by the scheduling. If a set of tasks t_1, t_2, ..., t_n are scheduled to the same VM (in this order), the processing of a task t_i should be enabled only after the execution of the previous task t_{i-1}, in addition to the data constraints of the DAG. In our algorithm, this is ensured by connecting control places to the processing transitions via input arcs.

To illustrate our method, we used the DAG from Figure 1 and the scheduling displayed in Figure 2 a) as the input for our algorithm. The resulting SPN model is displayed in Figure 43. The control places and connected arcs are depicted in gray.

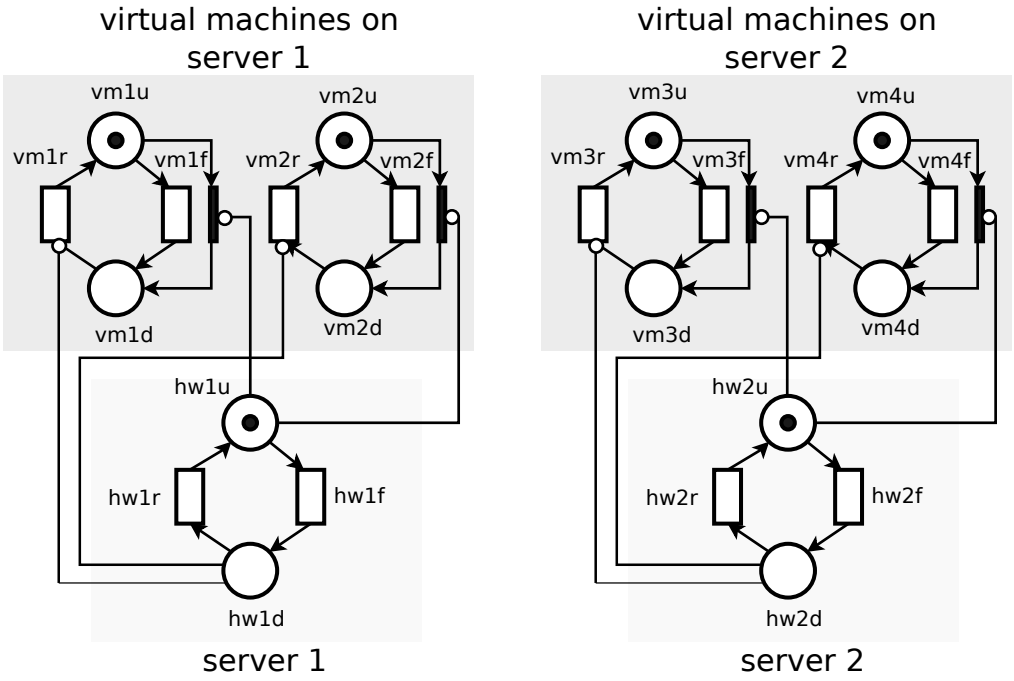


Figure 42 – Model generated with structural parameters set to $number\_of\_servers = 2$, $vms\_per\_server = 2$

Algorithm 1: DAG+scheduling to SPN conversion algorithm

Data: DAG and scheduling specifications
Result: A Stochastic Petri net

```
for Each task t in the DAG do
    create a place pl for the dependencies of task t;
    create a transition trans for the processing of task t;
    connect pl and trans with an output arc;
    if t has no parent tasks then
        add one token to pl;
for Each task t in the DAG do
    let pt be the processing transition for task t ;
    for Each task ct connected to t by an output arc do
        if t and ct are scheduled to the same VM then
            let ip be the inputs place for ct;
            let pct be the processing transition for task ct;
            connect ip and pct by an output arc;
        else
            create a new place outp for the output from t to ct;
            let pct be the processing transition for task ct;
            create a new transition sendt for representing the communication
              between t and ct;
            connect t and outp by an output arc;
            connect outp and sendt by an input arc;
            connect sendt and pct by an outpyt arc;
for Each processing transition pt in the SPN do
    set the weight of the output arc of pt to be the number of input arcs;
for Each VM v do
    for Each task t scheduled to v do
        if t has predecessor then
            let t_pred be the predecessor task to t on the scheduling;
            connect the processing transition of t_pred and the input place of t by an
              input arc;
```

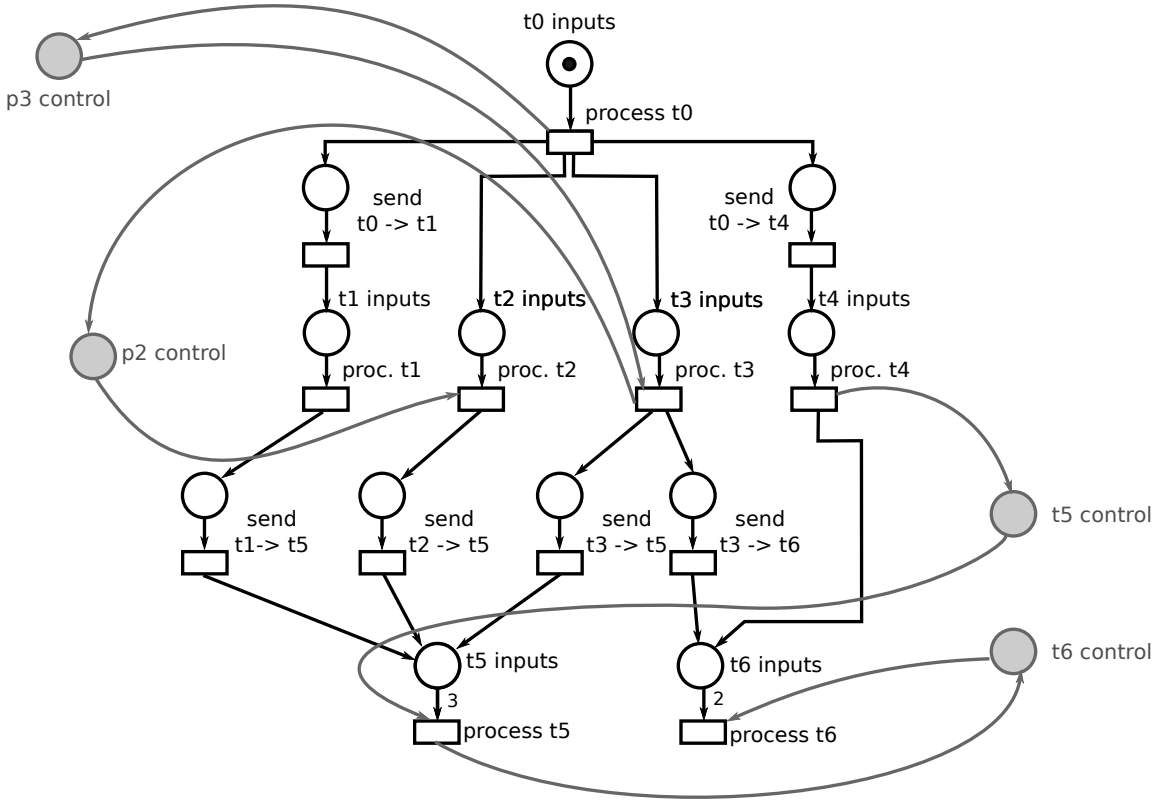


Figure 43 – DAG and scheduling converted to an SPN model

5.5 FINAL REMARKS

This chapter presented a series of models, meta-models, and model-generator algorithms suited for infrastructure planning of WaaS clouds. These models employ hierarchical modeling techniques and use the RBD, DRBD, SPN, and OPN formalisms. They enable the evaluation of the steady-state availability, capacity-oriented availability, and performance metrics such as throughput and job makespan. In the next chapter, we will present case studies that demonstrate the capabilities of the proposed framework, in conjunction with the models featured in this chapter, concerning infrastructure planning activities.

6 CASE STUDIES

This chapter presents three case studies, selected to demonstrate the suitability of the proposed framework and models in assisting the infrastructure planning of Workflow-as-a-Service clouds. The first case study applies sensitivity analysis for finding the most influential parameters on the steady-state and capacity oriented availability. In the second case study, we solve the cloud redundancy allocation problem described in Section 5.3. Finally, in the third case study, we investigate the scheduling problem on cloud workflow applications, considering a performability metric (the system throughput under the effect of hardware and VM failures) as the objective function.

6.1 CLOUD INFRASTRUCTURE AVAILABILITY ENHANCEMENT VIA STOCHASTIC MODELING AND SENSITIVITY ANALYSIS

Availability is one of most fundamental attributes of cloud systems since low availability can incur in revenue losses and affect customer confidence (PATEL; RANABAHU; SHETH, 2009). We can improve this attribute by acquiring more reliable software and hardware or by employing redundancy. Such measures must be carefully chosen, since they can incur in extra costs. A method to find a better way to increase the availability of a cloud is creating availability models and applying sensitivity analysis techniques (HAMBY, 1994) in order to find the most influential parameters on the metric of interest.

In this case study, we apply sensitivity analysis techniques in conjunction with the availability models shown in the previous chapter. We combine two sensitivity analysis techniques: the percentage difference and the DoE method. To deal with a large parameter list, we first apply the percentage difference method (that is faster to conduct) to refine the parameter list for the DoE method (that demands more time to finish and can get the most information about the combined effect of different parameters). Then, we compare the list of parameter impacts on the metric of interest produced by the two techniques and draw our conclusions.

6.1.1 Availability Evaluation

Table 12 shows the complete list of parameters used to evaluate the proposed model. Each parameter has a short description, a baseline value, and minimum and maximum values. The baseline value is used to evaluate the model in our baseline scenario, i.e., a configuration of parameters that is common in real-world deployments according to our literature review and experiments in testbed setups. The maximum and minimum values are used to perform the sensitivity analysis. The parameter "*extra workers*" denotes the number of workers that exceed the minimum required. In the k-out-of-n block for the

Table 3 – Input parameters for the models

Parameter name	Description	Baseline value (h)	Minimum	Maximum
av_zones	Number of availability zones	1	1	5
extra_workers	Number of extra worker nodes in a cluster	1	1	5
repair_teams	Number of repair teams	1	1	5
mttf_hw	Mean time to failure of a physical server	8760	2000	12000
mttr_hw	Mean time to repair of a physical server	1.66667	0.5	4
mttf_os	Mean time to failure of the operating system	2893	2000	4000
mttr_os	Mean time to repair of the operating system	0.25	0.25	2
mttf_jvm	Mean time to failure of the JVM	2893	2000	4000
mttr_jvm	Mean time to repair of the JVM	0.25	0.25	2
mttf_service	Mean time to failure of Eucalyptus components	788	500	2000
mttr_service	Mean time to repair of Eucalyptus components	1	0.25	3
mttf_vmm	Mean time to failure of the hypervisor	2990	2000	4000
mttr_vmm	Mean time to repair of the hypervisor	1	0.25	3
act_service	Mean time to activate a standby component	0.005	0.001	0.1
mttf_san	Mean time to failure of Storage Area Network	9223372	100000	1000000
mttf_san	Mean time to repair of Storage Area Network	1	0.25	3

availability zone model, this value is equal to $n - k$. The MTTF and MTTR of hardware, operating system, hypervisor, JVM, and Eucalyptus services are obtained from (DANTAS et al., 2012). Since this work does not consider the PostgreSQL component, we used the same MTTR and MTTF values of the other software components for parametrizing this component. The MTTF and MTTR for the SAN storage were obtained from the model presented in (KIM; MACHIDA; TRIVEDI, 2009).

We consider software components in a spare mode to be less likely to fail due to software aging issues (the accumulation of errors during the software runtime that leads to a failure), memory leaks, and running erroneous instructions. Therefore, we adopted the strategy used by (GUIMARÃES; MACIEL; MATIAS, 2013) of setting the failure rate of standby components to 50% of the failure rate for active components.

The results for the baseline scenario are summarized in Table 4. Even when using redundancy in the essential components of Eucalyptus, this baseline scenario could not reach the mark of four nines. For sake of comparison, this is the availability level supported by the Amazon EC2 service as defined on its SLA (AMAZON, 2019). The steady-state availability of the private cloud system considered in this study was 99.98286%. This result indicates that the accumulated system outages in a year (i.e., the annual downtime) may add up to about 1.5 hours of system unavailability. Such results indicate that there is some room for improvement in this cloud. Some possible actions that could improve the system availability include: (i) acquire more reliable hardware and operating systems in order to have longer time spans between failures; (ii) hire more staff for the repair teams; (iii) purchase equipment to increase the number of extra worker servers in a cluster; and (iv) arrange machines to add more clusters (availability zones) to the cloud, among other options.

Each alternative has a significant cost involved and, considering that the organization has a limited budget, the decision makers must carefully analyze and discover which

Table 4 – Results for the baseline scenario

Metric	**Value**
Availability (%)	99.98286
Availability (nines)	3.76597
Annual downtime (hours)	1.502
Capacity oriented availability (%)	99.77022

Table 5 – Sensitivity ranking from percentage difference - availability

Parameter	**Effect**
extra_workers	0.018774496
mttr_service	0.002150095
act_service	$8.013075495 \times 10^{-}4$
mttf_service	$6.730110415 \times 10^{-}4$
mttr_vmm	$4.210018701 \times 10^{-}4$
av_zones	$4.152073960 \times 10^{-}4$
mttr_os	$3.072028529 \times 10^{-}4$
repair_teams	$1.960734570 \times 10^{-}4$
mttr_hw	$1.790365769 \times 10^{-}4$
mttf_vmm	$1.052793474 \times 10^{-}4$
mttf_hw	$6.483898160 \times 10^{-}5$
mttf_os	$3.843260727 \times 10^{-}5$
mttr_jvm	$2.193959603 \times 10^{-}5$
mttf_jvm	$9.543993829 \times 10^{-}6$
mttf_san	$8.999990999 \times 10^{-}7$
mttr_san	$2.981555017 \times 10^{-}7$

solution will improve the availability with minimum investment. Our parametrized model allows sensitivity analysis to be applied so we can list the parameters in order of influence to the selected metric. The sensitivity analysis is shown in the next subsections.

6.1.2 Sensitivity analysis - Availability

Table 5 shows the parameters ranked by the sensitivity index of percentage difference. The parameters with the largest indexes are *extra_workers*, *mttr_service*, and *act_service*. Therefore, we can expect them to have large influences on system availability and to deserve attention from cloud administrators. On the other hand, parameters related to the JVM and SAN components (*mttr_jvm*, *mttf_jvm*, *mttf_san*, *mttr_san*) are on the bottom of the ranking, with lower sensitivity indexes than most of the other parameters. In particular, the SAN is already highly reliable and small variations in its mean time to failure and repair have little significance on the overall system.

From the graphs of Figure 43, we can conclude that an increase of the number of repair

teams, availability zones, or extra workers can bring benefits as a first step, but subsequent increases produce low gains for availability compared to changes in other parameters. The reader should note that some parameters are inversely proportional to the steady-state availability. For instance, an ineffective repair team would increase the MTTR of hardware and software components and, consequently, decreasing the steady state availability of the system.

Table 6 shows the ranking of the parameters and respective effects on the metric of interest (availability), as computed from the DOE analysis with the first ten parameters from the percentage difference index. The list is ordered by the absolute value of the effect, since this method may produce negative values. The order of the parameters is different from the previous method, as the DOE technique computes the metric for each possible combination of parameters (e.g., 1,024 combinations when considering ten parameters). Thus, this method can obtain more information about the combined effect of different parameters than the percentage difference formula.

The one-parameter-at-time sensitivity analysis also reveals useful information. It can be noticed in Figures 43 a) and f) that, in spite of the high sensitivity indexes for the "*av_zones*" and "*extra_workers*" parameters as found by the DOE method, adding a second extra node or employing more than two availability zones do not provoke a significant improvement in availability; hence, such investment be considered not worthwhile.

Table 6 – Sensitivity ranking from DOE - availability

Parameter	**Effect**
av_zones	0.016357221
extra_workers	0.016167285
mttr_service	−0.008554587
mttf_service	0.006234579
mttr_vmm	−0.002528422
mttr_os	−0.001503708
mttf_vmm	$9.936709812 \times 10^{-}4$
mttr_hw	$-9.900866044 \times 10^{-}4$
act_service	$-5.863656932 \times 10^{-}4$
repair_teams	$1.812525481 \times 10^{-}4$

6.1.3 Sensitivity analysis - Capacity Oriented Availability

A cloud system comprises a set of worker nodes, and each of them represents a fraction of the total computing power of the cloud. A failure on a worker node may not bring the whole system down, but reduces the processing capability of the cloud. The *capacity oriented availability* (COA) (HEIMANN; MITTAL; TRIVEDI, 1990) is the metric that measures the proportion of the whole system that is actually being delivered to its clients.

We evaluate this metric by converting the SPN model to a Markov Reward Model (REIBMAN; SMITH; TRIVEDI, 1989). The metric is obtained by summing up the product between the reward and the stationary probability of each state:

$$COA = \sum_{i=0}^{n} \pi_i * r_i,$$

where n is the size of the state space of the Markov Reward Model, π_i is the stationary probability of being in state i, and r_i is the reward rate assigned to the state i. The reward rate can be defined using the following function:

$$r_i = \begin{cases} \#workers\_ups/N, & \text{if } \#s\_up = 1 \\ 0, & \text{otherwise} \end{cases}$$

In this expression, $\#workers\_ups$ represents the number of tokens in the $workers\_ups$ place. Each token in this place represents a functional worker node. N represents the total of worker nodes available in the cloud. $\#s\_up$ represents the number of tokens in the s_up place, which can be zero, if the system is unavailable, or one, if the system is functional.

Table 4 shows the capacity oriented availability for the baseline scenario. This result shows that 0.23% of the cloud processing power is wasted due to the effect of failures. Tables 7 and 8 show the sensitivity ranking for the capacity oriented availability, obtained from the percentage difference and DOE techniques, respectively. We notice that, while the effect of the parameters is higher the capacity oriented availability metric, the order of the parameters is the same. Therefore, we conclude that any action taken to improve the steady-state availability of the cloud will be appropriate to improve the capacity oriented availability. Having different order of parameters in the previous tables would mean a compromise between prioritizing the improvement of steady-state or the capacity-oriented availability.

6.1.4 First case study - final remarks

Providing a cost effective cloud service regarding availability is not an easy task. To enhance availability, one must employ redundancy schemes (by acquiring extra hardware), buy more expensive and reliable equipment, or to hire more repair technicians. In this case study, we applied a sensitivity analysis for finding the most influential parameters on the system steady-state availability. The experimental results have shown differences between the parameter ranking produced by the DoE and the percentage difference technique. Despite being important parameters according to the DoE ranking, adding a third extra worker node or availability zone does not provide a substantial increase in the steady-state availability. Additionally, we have shown that the steady-state and the capacity oriented

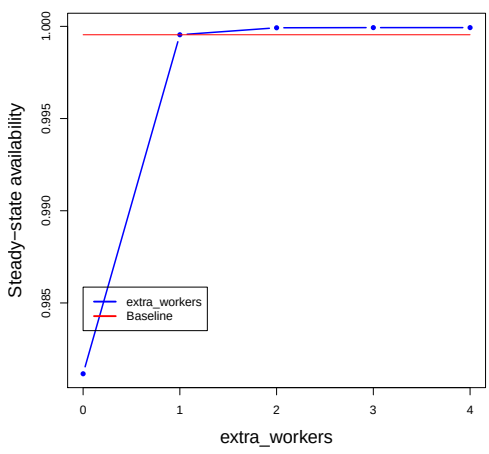
Table 7 – Sensitivity ranking from percentage difference - capacity oriented availability

Parameter	**Effect**
extra_workers	0.016886150
mttr_service	0.005442452
mttf_service	0.002114982
mttr_vmm	0.001302266
mttr_os	8.859865768 $\times 10^{-}4$
act_service	8.013005235 $\times 10^{-}4$
mttr_hw	5.702922094 $\times 10^{-}4$
mttf_vmm	3.458005798 $\times 10^{-}4$
mttf_hw	1.985703476 $\times 10^{-}4$
repair_teams	1.960597226 $\times 10^{-}4$
mttf_os	9.863198674 $\times 10^{-}5$
mttr_jvm	2.193741554 $\times 10^{-}5$
mttf_jvm	9.539648186 $\times 10^{-}6$
mttf_san	8.987305360 $\times 10^{-}7$
mttr_san	2.987713762 $\times 10^{-}7$

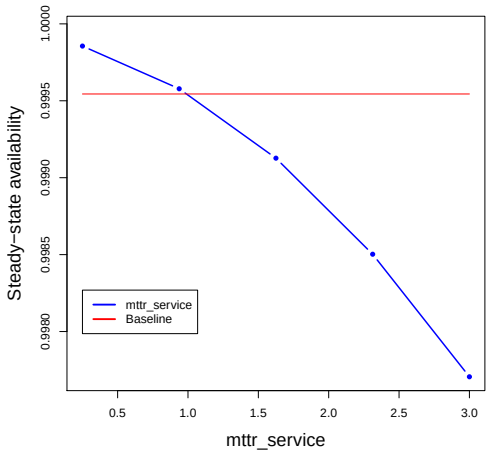
Table 8 – Sensitivity ranking from DOE - capacity oriented availability

Parameter	**Effect**
extra_workers	0.029160852
mttr_service	−0.018776881
mttf_service	0.013512220
mttr_vmm	−0.005593606
mttr_os	−0.003308422
mttr_hw	−0.002387835
mttf_vmm	0.002199690
mttf_hw	0.001044982
act_service	$-7.337189558 \times 10^{-}4$
repair_teams	$4.832393291 \times 10^{-}4$

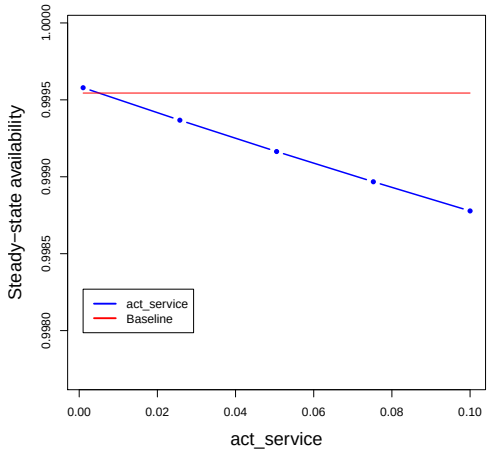
availabilities have the same parameter ranking for both the DoE and the percentage difference techniques.


Steady−state availability
1.000
0.995
0.990
0.985
extra_workers
Baseline
0
1
2
3
4
extra_workers

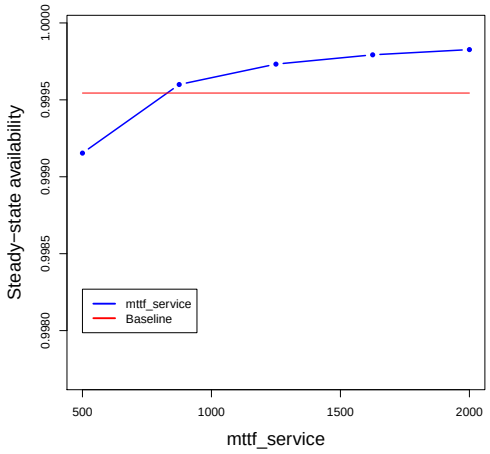
(a)


Steady−state availability
1.0000
0.9995
0.9990
0.9985
0.9980
mttr_service
Baseline
0.5
1.0
1.5
2.0
2.5
3.0
mttr_service

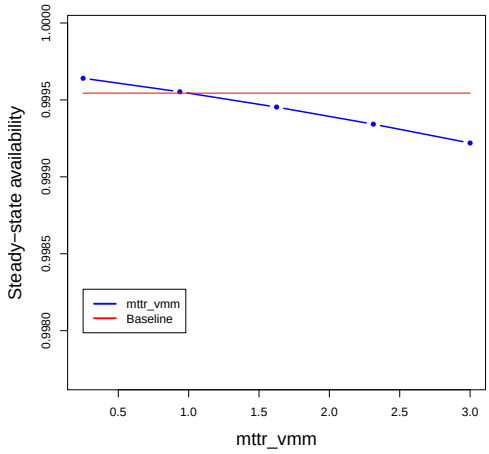
(b)


Steady−state availability
1.0000
0.9995
0.9990
0.9985
0.9980
act_service
Baseline
0.00
0.02
0.04
0.06
0.08
0.10
act_service

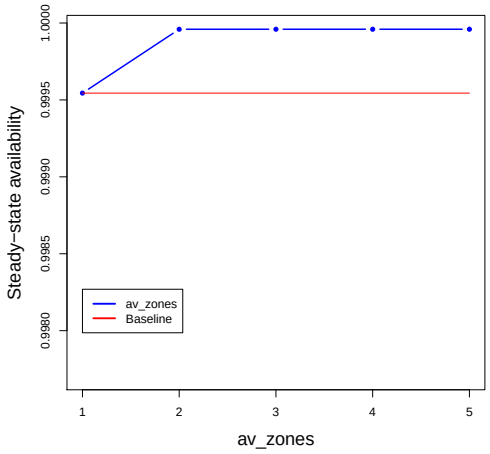
(c)


Steady−state availability
1.0000
0.9995
0.9990
0.9985
0.9980
mttf_service
Baseline
500
1000
1500
2000
mttf_service

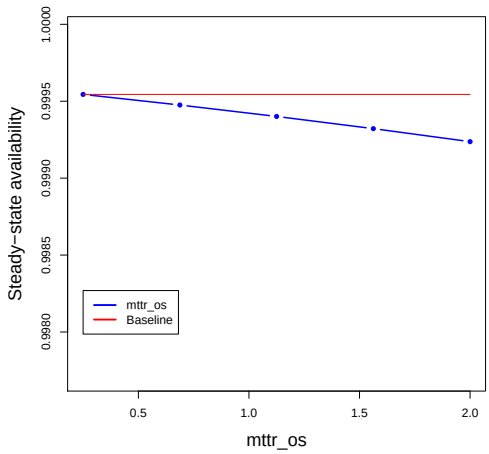
(d)


Steady−state availability
1.0000
0.9995
0.9990
0.9985
0.9980
mttr_vmm
Baseline
0.5
1.0
1.5
2.0
2.5
3.0
mttr_vmm

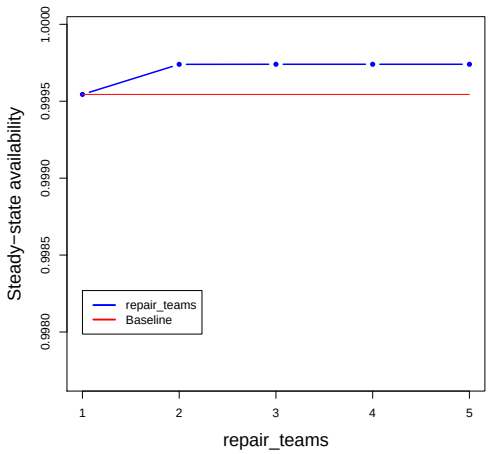
(e)


Steady−state availability
1.0000
0.9995
0.9990
0.9985
0.9980
av_zones
Baseline
1
2
3
4
5
av_zones

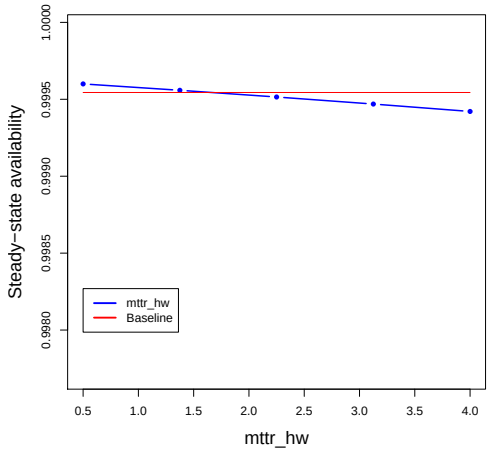
(f)



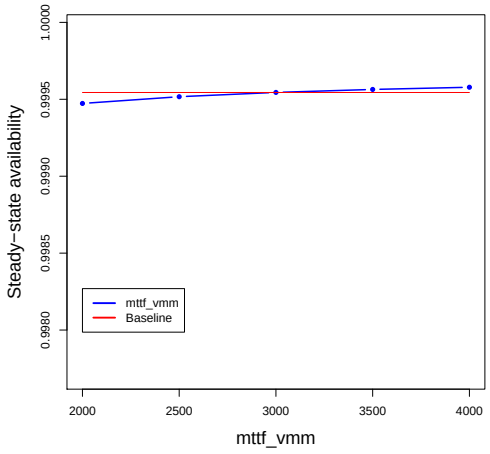
(g)



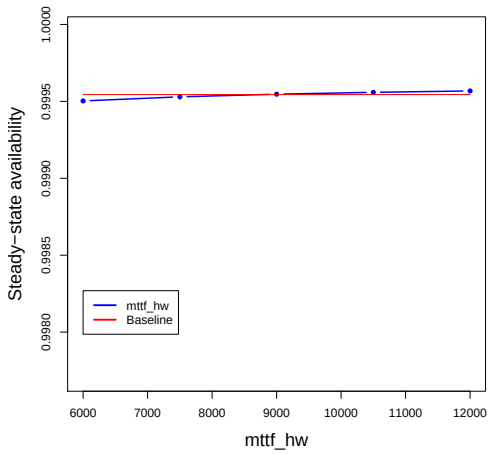
(h)



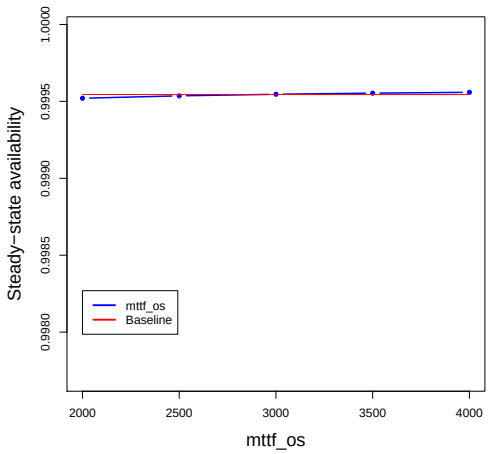
(i)



(j)



(k)



(l)

Figure 43 – One at a time sensitivity analysis (availability) - Top 12 parameters

6.2 DEPENDABILITY AND COST OPTIMIZATION OF PRIVATE CLOUD INFRASTRUCTURES

In this case study, we employed the optimization model described in Section 5.3 to find near-optimal deployment configurations of a cloud infrastructure concerning steady-state availability and acquisition cost. In this study case, we employ the local search algorithm described in Section 6.2.1 to select the best deployment configuration for a number n of servers in the cloud infrastructure. We then apply a modified version of the bisection algorithm (described in the Section 6.2.2) to find a near-optimal number of servers when considering a specified availability level. The experimental results shown in Section 6.2.3 demonstrate the effectiveness of the combined usage of the local search and the bisection algorithms in maximizing the steady-state/capacity oriented availability of the cloud while reducing the acquisition cost.

6.2.1 Multi-start hill climbing algorithm

Running a traditional local optimization algorithm using only one machine and a single processor has some tradeoffs that are stated below. If we want to achieve a good solution, we must increase the number of iterations and, as a consequence, we increase the running time. On the other hand, if we want to reduce the running time of the algorithm, we reduce the number of iterations and may not achieve a good quality solution. One way to mitigate this tradeoff using local optimization is to perform a multi-start algorithm (BOESE; KAHNG; MUDDU, 1994): the algorithm generates a certain number of initial solutions, and each solution can run on a dedicated processor. Using this parallel approach, we can increase the quality of the generated solution, and reduce the running time, by using more processors.

We implemented the multi-start hill climbing algorithm using the Akka concurrency framework (AKKA...,). Figure 44 shows the architecture of the optimization algorithm. The big rectangles with thicker lines represent physical nodes containing an Akka environment. Inside those rectangles, the actors (lightweight processes) that run in each node are displayed. An arrow connecting two actors indicates that the source maintains a reference to the target. Using a reference, an actor can send a message to another actor, and the actor that receives the message acts according to the content of the message. The dotted arrows represent references that point to an actor located in another machine.

The *Global Optimizer* actor is the starting point of our system. The user sends a message to this actor containing the cloud requirements, and the parameters of the optimization algorithm. Next, the Global Optimizer generates some initial solutions, according to the number of the *global iterations* parameter. The initial solutions will be distributed to the *Local Optimizers* registered in the environment. On the local optimizer, a hill climbing traversal will be done, configured to perform some iterations according to the *local iterations* parameter.

Each local iteration of the algorithm consists of generating and solving an availability model. Each hill climbing traversal is performed on a *Worker* actor in a dedicated thread. The number of worker actors that will be created is the number of processors of the node. Each local optimum found by a Local Optimizer will be sent back to the Global Optimizer. At the end of the process, the best solution found will be presented to the user.

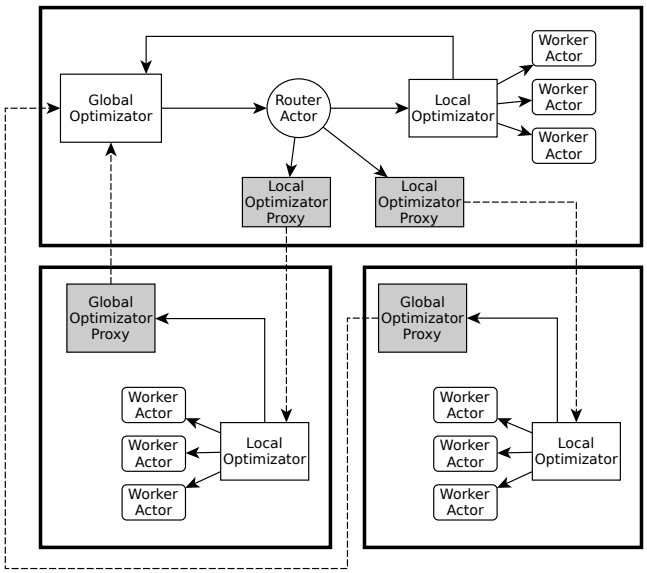


Figure 44 – Architecture of the distributed optimization algorithm

6.2.2 Bisection Algorithm for Optimizing Acquisition Cost

In this section, we discuss the proposed algorithm for optimizing the acquisition cost. The acquisition cost is defined as follows:

$$C_{acq}(S) = \sum_{i=1}^{n} P_i \tag{6.1}$$

In this work, we consider that all servers have the same configuration and price ($P_i = P$, for every i from 1 to n). Therefore, increasing the number of servers will also increase the acquisition cost of the private cloud infrastructure. The main question is: how to decrease the acquisition cost while optimizing the availability metric? At first, it seems contradictory to increase the availability while reducing the number of machines (considering the operational mode specified by the user), since we need to employ extra machines to be used as spares in redundancy schemes to achieve high availability. However, as we will show, employing more nodes than required may not necessarily increase the availability. Thus, we can reduce the number of additional machines considering the availability level that we are interested in maintaining.

To understand this fact, consider one particular execution of our optimization algorithm. We set the following parameters:

- 128 global iterations
- 30 local iterations
- Minimum of 40 worker servers that must be available
- Total of 100 servers comprising the cloud infrastructure

For this example, we configured our hill-climbing algorithm implementation to store the cost of each randomly generated solution.

The plot of the availability for the generated solutions is shown in Figure 13 (the X axis in the plot is an identification number for the generated solutions). Each black dot is a different solution found by the algorithm. Notice that the majority of the solutions present a high availability. Therefore, in this case, there are more resources than necessary to build a fault-tolerant cloud infrastructure. The outliers at the bottom of the chart show some poor random solutions generated by the algorithm. Not employing redundancy at the cloud manager or cluster managers, or creating a cluster with exactly the required number of workers (a k-out-of-n block with $k = n$, i.e., a series grouping) will produce a significantly inferior availability level compared to the solutions that avoid those choices.

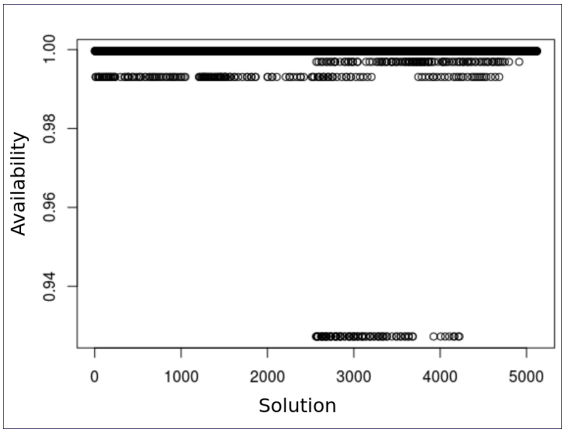


Figure 45 – Stationary availability of generated solutions for a particular run

The proposed hill-climbing optimization algorithm deals with a fixed number of nodes and attempts to find combinations that lead to high availability. Since the total number of nodes is not modified during the iterative process, it does not optimize the acquisition cost. To optimize the acquisition cost it is necessary to reduce the number of nodes until the point the minimal availability is reached. To find this number of nodes, we adopted the bisection algorithm. This algorithm finds the first value greater than a specified value, considering an ordered list of elements. The Listing 2 shows our modified version of the bisection algorithm to optimize the acquisition cost.

The presented bisection algorithm has the following input parameters:

Algorithm 2: Bisection algorithm to optimize acquisition cost of a private cloud

Data: $low, high, MinAv$
Result: Near optimal cloud configuration

```
if MinAv < A_optm(low) then
    return low;
if MinAv > A_optm(high − 1) then
    return high;
while True do
    if lo + 1 == high then
        return low;
    else
        return low + 1;
    mi = (high + low/2;
    if MinAv ≥ A_otpm(mi) then
        high = mi;
    else
        low = mi;
```

- low - the minimal number of nodes that can be used to build our cloud, satisfying the user requirements, but without employing any redundancy (therefore, producing a low availability);
- $high$ - Any number that generates a high-available cloud configuration when applying our hill-climbing algorithm;
- $MinAv$ - the expected availability level that the cloud should provide. If we adopt 0.9999 as $MinAv$, for instance, we are interested in finding the minimal number of servers that can be used to build a private cloud infrastructure with four nines of availability.

The value computed by the algorithm is the integer number i, located in the interval $[low, high]$, that satisfies the following expression:

$$(MinAv > A_{optm}(i-1)) \; and \; (MinAv < A_{optm}(i))$$

The $A_{otpm}(i)$ function consists of running the proposed multi-start hill-climbing algorithm, changing the number of nodes to i, and keeping the other parameters, e.g., the minimum number of worker nodes, global iterations, local iterations, minimum availability. In other words, it should produce a number i of nodes of a private cloud infrastructure, such as, if we remove one node, the availability will decrease to a value less than the $MinAv$ parameter.

Since the hill-climbing algorithm is subject to randomness, it is possible that a particular run of the bisection algorithm generates i, such as $A_{otpm}(i) < A_{otpm}(i-1)$. We can partially reduce the occurrence of this effect by adapting the algorithm in the following way: in the beginning, we can adopt a small number of global iterations, since a quick result should be sufficient in the early stages of the algorithm. As it runs, we can increase the number of global iterations to perform a more exhaustive search and avoid the problem mentioned earlier.

6.2.3 Experimental results

In the experiments for this case study, we adopted the same parameters from the first case study, displayed in Table 12. Our initial test consisted in solving a small-sized problem with the hill-climbing algorithm and also applying a brute-force search to examine how close to the global maximum is the solution provided by the optimization algorithm. This experiment was performed on a Dell Optiplex 980 machine with an Intel i5 processor (2.6 GHz). The total number of the nodes of the cloud architecture was set to 25, and we variate the required minimum number of worker nodes from five to eight. The total execution time of the multi-start hill climbing algorithm was set to five minutes. Table 9 shows the obtained results. The obtained values suggest that we can obtain values close to the global minimum without having to explore all the solution space exhaustively.

Table 9 – Hill-climbing X brute-force solution

Minimum number of workers	Brute force	Hill climbing
5	0.9999930063	0.9999928918
6	0.9999929222	0.9999928917
7	0.9999928916	0.9999927377
8	0.9999928488	0.9999918988

On the study of larger cloud architectures, we configured the distributed optimization algorithm in the following way. We set up a cluster composed by eight VMs on Amazon EC2 platform. Each VM is an instance of c4.2xlarge, possessing eight cores and 15 GB of primary memory (AMAZON, 2016). We adopted 128 global iterations, so each processor performs two executions of the hill-climbing algorithm, and each execution performed 20 local steps. The minimal number of nodes was set to 40, e.g., a cluster will be considered unavailable if less than 40 worker nodes are active.

In order to evaluate how the availability of the computed near-optimal solution increases as we add more nodes in the cloud, we performed the experiment summarized in Figure 46. Each point represents the availability of the near-optimal solution for a given number of nodes. We started with 45 nodes and increased this number by five units until 125. Notice that there are two almost plain segments and a significant difference between 85 and 90

nodes. The reason is that beyond a certain point inside this interval is possible to build a cloud with two availability zones and employ redundancy on the cloud/cluster managers. In the almost plain segments, the difference between the points is slight at the beginning of the segment, and nonexistent from a certain point. It means that from a certain point, adding new machines to the cloud produces no increase in the availability. Our version of the bisection algorithm will help to reduce the number of machines that add nothing to the availability and increase the acquisition cost.

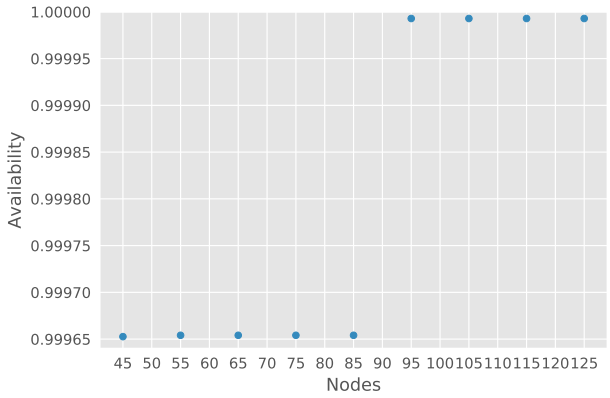


Figure 46 – Near-optimal availability X number of nodes - up to two availability zones)

Figure 47 shows a slightly different experiment. In this experiment, we are interested in seeing whether adopting more than two availability zones will produce an increase in the availability or not, as shown in Figure 46, from one to two availability zones. The obtained results shows no significant increase in the availability by adopting more than two availability zones.

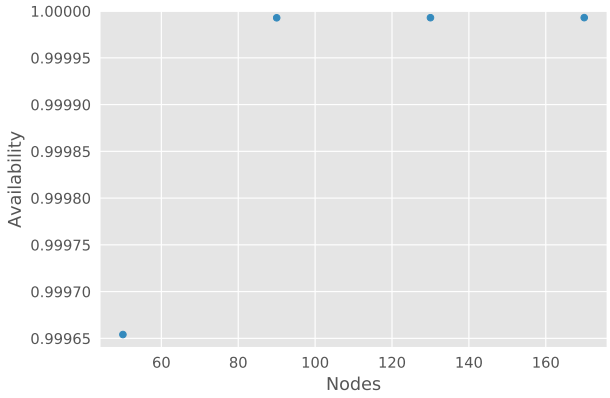


Figure 47 – Near-optimal availability X number of nodes - one to four availability zones

Figure 48 shows the results obtained by the bisection algorithm. We used the interval $[80, 100]$ as the input parameters since we guess that the best point concerning the acquisition cost will be in this range, as we can observe from Figure 46. The minimum availability that we used was 0.99999, corresponding to five nines of availability, and

only 5.26 minutes of downtime per year. Our method produced a solution with 88 nodes. It means that 88 is our best guess of the smallest number of nodes that can be used to build a private Eucalyptus cloud with this availability level. Using the value 0.9999 as the minimum availability parameter produces a solution with 87 nodes and steady state availability equals to 0.99997481 (four nines of availability). Adding one more node produces a solution with five nines of availability.

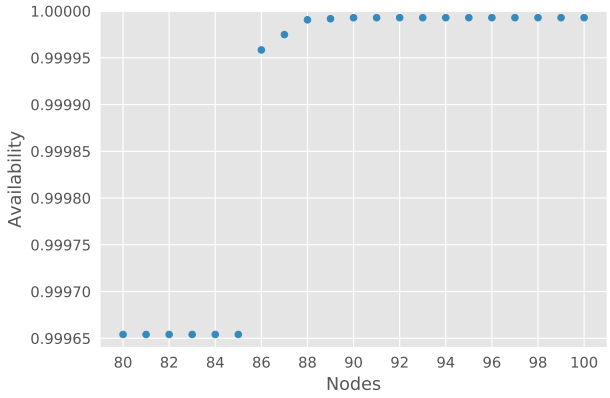


Figure 48 – Near-optimal availability X number of nodes - bisection algorithm

Table 10 shows the results obtained by performing the same experiment depicted in Figure 46, but using the Capacity Oriented Availability as the metric to be optimized. These results indicate that the COA metric is not as impacted by the employed redundancy arrangement as the steady-state availability. Moreover, due the stochastic nature of the multi-start hill-climbing algorithm, we observe a not monotonic behavior, i.e., we can observe some decreases in the COA metric when increasing the number of redundant nodes in the performed experiment.

Table 10 – Near-optimal capacity oriented availability X number of nodes

Total number of nodes	Capacity Oriented Availability
45	0.9977772035
55	0.9977772023
65	0.9977771947
75	0.9977771977
85	0.9977772078
95	0.9977771752
105	0.9977771780
115	0.9977771822
125	0.9977772006

6.2.4 Second case study - final remarks

Creating a highly available and cost-effective private cloud infrastructure is not a simple task, considering a large number of possible configurations. Analytical models can be used to predict the availability of particular configurations, but manually creating models for many different solutions is not a good method to find an optimal configuration. In this case study, we applied the proposed optimization method for increasing the availability of private cloud infrastructures. We employed an adapted version of the hill climbing algorithm that translates each random solution to an analytical model and can use the steady-state/capacity oriented availability as the objective function. To minimize the acquisition cost (i.e., reducing the number of servers of the infrastructure) while achieving high availability, we combine the bisection method with the hill climbing algorithm. The experiments done using the developed framework allowed us to gain some useful insights about availability aspects of the considered system

6.3 PERFORMABILITY EVALUATION AND OPTIMIZATION OF WORKFLOW APPLICATIONS

This case study aims to solve the multiprocessor scheduling problem of scientific workflows running in a cloud environment, using a performability metric as the objective function. Given a workflow described by a DAG $G = \{T, E\}$, our objective is to find a scheduling S of tasks in T on m virtual machines that maximizes the throughput of jobs. The number m is not fixed and must be determined by the optimization method.

The flow diagram of Figure 49 presents a high-level overview of the adopted optimization method. The workflow DAG has a list of tasks and their dependencies, processing time of each task and communication time between dependent tasks running on different processors. The computing and communication times of a DAG can either be deterministic or follow a specific random distribution (normal, exponential, Erlang, and so on). The cloud infrastructure parameters define the number of physical servers, the maximum number of virtual machines that each host can provide, and the failure/repair/switchover rates of the physical/virtual machines. The simulation/optimization parameters configure the simulation engine (e.g., number of replications for an individual simulation) and the optimization algorithm (e.g., population size, number of elite chromosomes, number of generations and so on).

The optimization algorithm explores the solution space for the input DAG until the stopping condition is reached, which is defined by the control parameters. Then, a near-optimal scheduling solution is provided by the algorithm. The user can perform further analysis of the obtained solution and evaluate additional performance/reliability metrics, such as average waiting/response time, discard rate of tasks/jobs, the probability of completing a job. The method can be used interactively, i.e., the user can modify

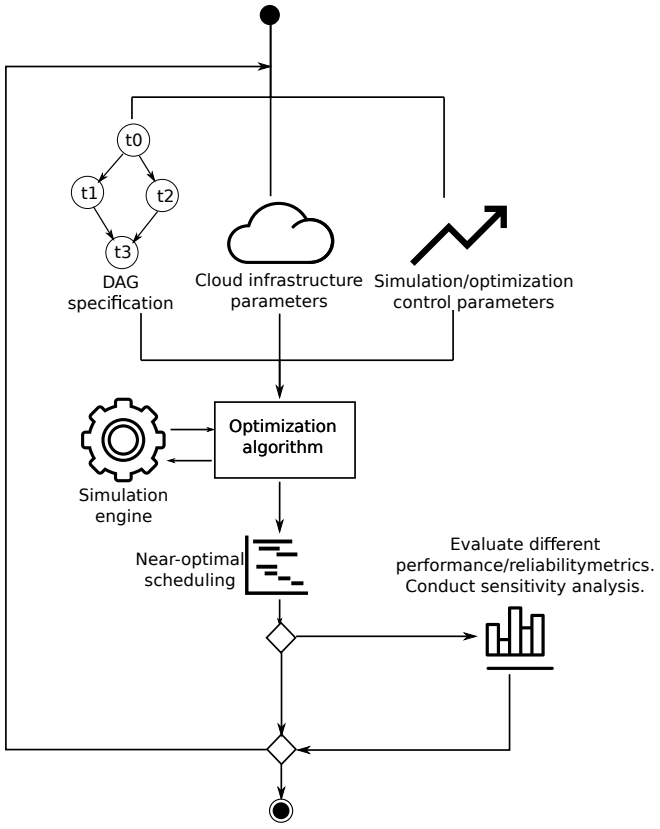


Figure 49 – Overview of the proposed method

the cloud/control parameters and repeat the process, obtaining new scheduling and performability metrics.

The next subsections explain further each part that composes the optimization method.

6.3.1 Genetic Algorithm with Stochastic Fitness Function

The activity diagram of Figure 50 describes the optimization algorithm adopted in this paper. It is a genetic optimization procedure that uses the performability model of Section 5.4 for computing the fitness value of explored chromosomes. The chromosome representation consists of a pair of vectors representing the ordering of tasks and the mapping of tasks to the processors, as illustrated in Figure 51. The partially-mapped crossover (PMX) operator (GOLDBERG; LINGLE et al., 1985) was adopted to generate the offspring of a population. Two random chromosomes are randomly selected for creating a pair of children (parents with higher fitness value are more likely to be selected). The process to generate the children is defined as follow. First, it creates a copy of the parents. A paired subinterval is randomly selected and switched among the children. Then, a mapping function is applied to convert the repeated alleles (i.e.: the units of information that compose the chromosome) outside the random subinterval. Figure 52 shows an example of crossover operator. The mutation operator modifies a chromosome with a random operation by swapping the order of two tasks or changing a task to a different processor.

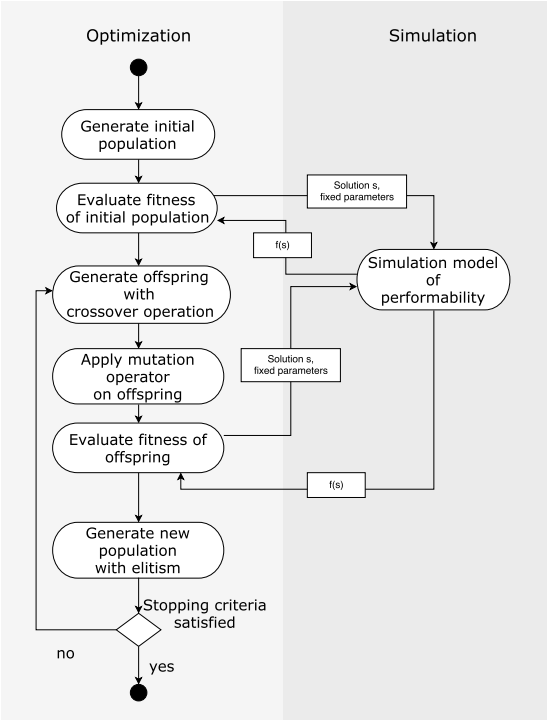


Figure 50 – Genetic algorithm with a stochastic fitness function

The mutation operator is illustrated in Figure 53.

Processors:	0	0	0	1	1	1	1	1
Tasks:	0	3	1	4	2	5	6	7

Figure 51 – Chromosome representation

Figure 54 shows the activity diagram of the method to obtain a fitness value of a solution s. The method takes as input a solution s from the search space of available schedulings, a set of fixed parameters, a template model, and a set of auxiliary models. The solution s is parsed to obtain a set of solution parameters. The fixed and solution parameters are combined into a single set. The joint solution-fixed parameters set is divided into two categories: structural and non-structural. The structural parameters define the fixed structures of the model and the arcs/edges that connect them. Non-structural parameters define the delay/rate parameters and information such as probabilities, and buffer capacity. The combined set of structural/non-structural parameters and the base models are used as input for generating the final simulation model for a solution.

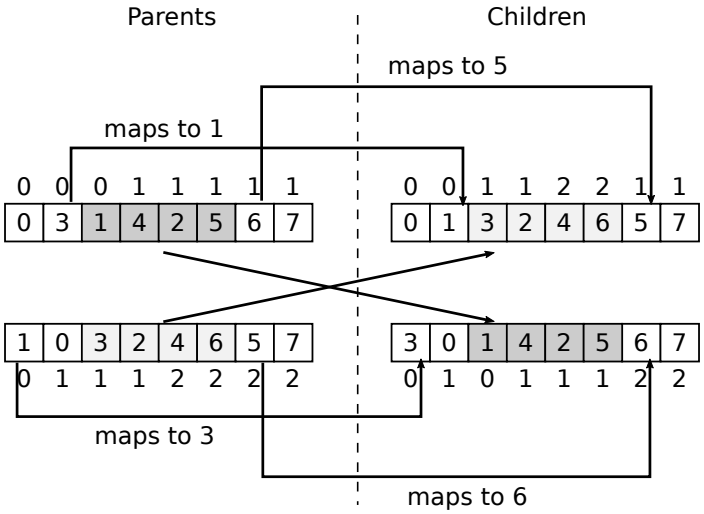


Figure 52 – PMX crossover operator

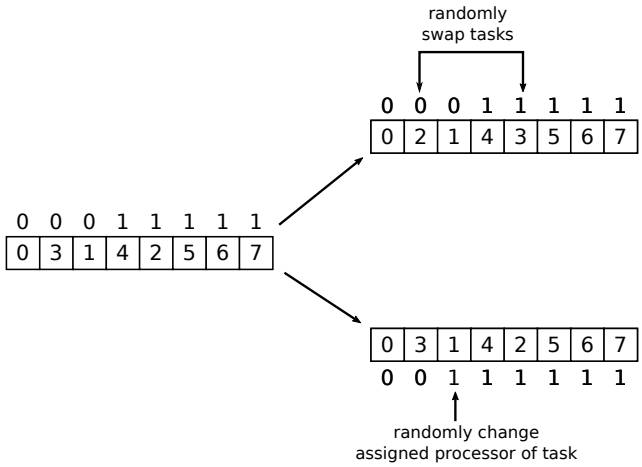


Figure 53 – Mutation operator

6.3.2 Optimization and Performability Analysis of a Small Sized DAG Application

In this case study, we present two scenarios to evaluate the performability optimization method. The first scenario shows the analysis of a small-sized workflow application/cloud infrastructure. In the second scenario, we examine a real workflow application and a greater cloud infrastructure. In both scenarios, the metric used for the fitness function was the average number of completed jobs in one year. We use a Fujitsu Primergy RX200 S7 server to conduct the experiments. The server has the configuration shown in Table 11.

In the first scenario, we choose a small-sized DAG application for being able to apply a brute force analysis and compare the results found by the meta-heuristic algorithm with the global maximum. Table 12 shows the input parameters for the performability model. We adopted the mean time to failure, repair, and activation times for physical and virtual servers as defined in (KIM; MACHIDA; TRIVEDI, 2009). The DAG from Figure 1 was adopted in this study. We consider the computing and communication times to follow a normal distribution, as in (CAI et al., 2017). The mean values are obtained from the graph

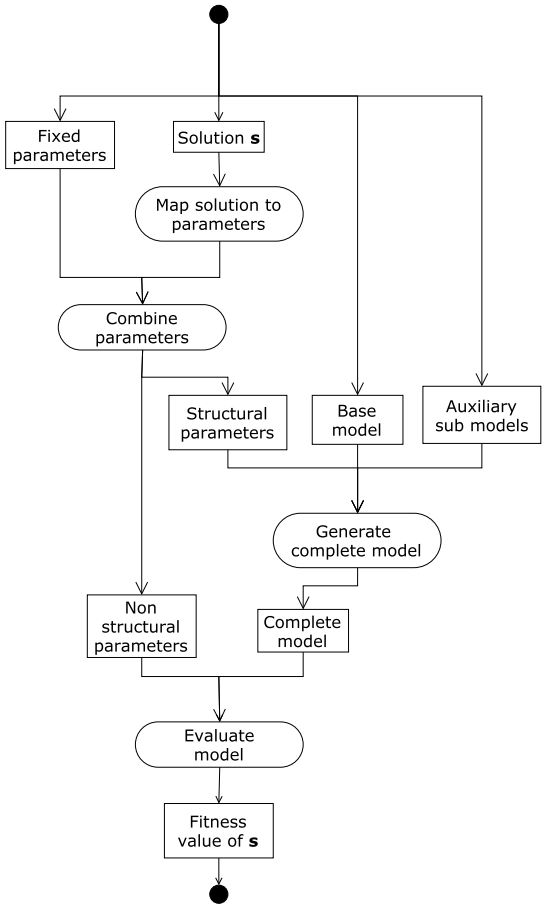


Figure 54 – Method to obtain the fitness value for a solution s

nodes and edges and the standard deviation is assumed to be 10% of the mean value.

The scatter plot from Figure 55 shows the fitness values (the number of completed jobs per year) for all possible schedulings of the DAG from Figure 1, evaluated by brute force. The solution space has 7840 different schedulings. Each solution is assigned to a unique integer identifier which is displayed on the plot x-axis. It can be observed that the scheduling and the number of provisioned virtual machines play a critical role in the throughput of the system. The difference between the worst and the best solutions is approximately 2430 jobs per year.

Due to the solution space not being too big, we adopted a small population of ten

Table 11 – Server used for experiments - specifications

CPU 1, CPU 2	Intel Xeon E5-2650 (8 cores and 16 threads)
Memory	(10 banks of) DIMM DDR3 1600 MHz, 4GB
Operating system	Debian 7, Linux kernel version 4.9
JVM	Oracle JDK version 1.7

chromosomes for testing the genetic algorithm in this scenario. We configured the algorithm to keep two elite chromosomes from the previous population in each iteration and employed a mutation probability of 0.05. Figure 57 shows the population average and highest fitness values for each generation. A horizontal line represents the fitness value for the global optimum at the top of the plot. It is possible to observe that the increase in the average fitness value after each generation leads to the discovery of a new elite chromosome after the fifth generation. In the tenth generation, the average fitness gets closer to the best fitness.

To analyze the impact of failures on the number of completed jobs per year, we made a sensitivity analysis on the mean time to failure for physical and virtual machines. Each parameter ranges from 20% to 200% of the base value shown in Table 12. Figure 56 presents the results of the sensitivity analysis, considering the best scheduling found by brute force. For the collected metrics, we indicate the 95% confidence intervals alongside the average values. For each point on the plots, we obtained 500 samples from the simulation model. Figures 56 a) and b) show the impact of the hardware and virtual machine MTTFs on the number of processed jobs per year and figures 56 c) and d) show the sensitivity analysis of the failure ratio of jobs. The analysis reveals that the system throughput and job failure ratio are sensitive to VM failures. It also can be noticed that as the hardware/virtual machine MTTFs increase, the differences between adjacent points in the plots become less pronounced and the confidence intervals overlap.

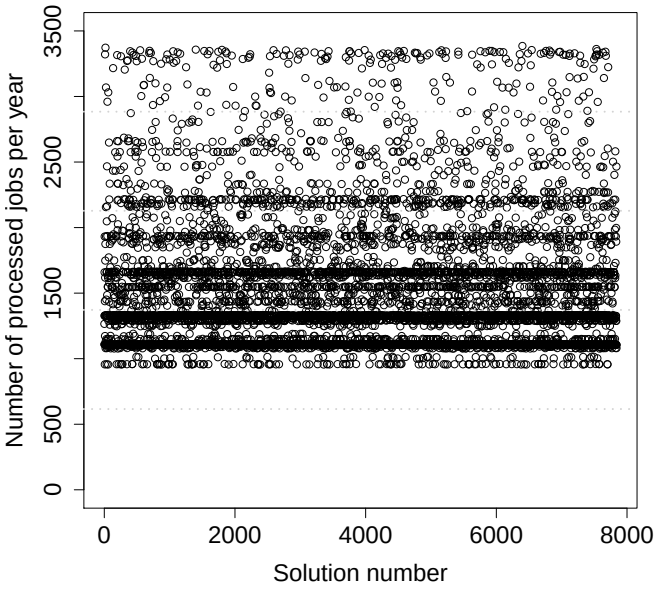
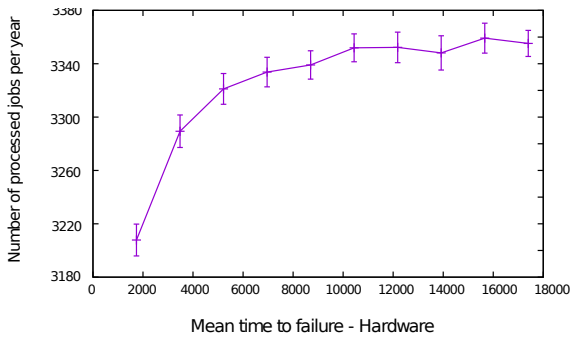
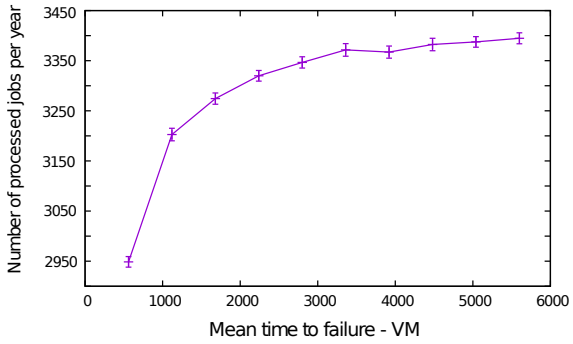


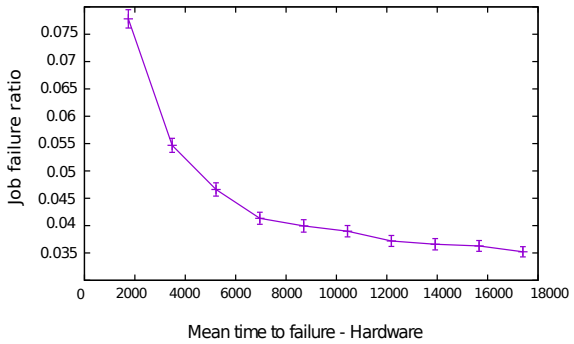
Figure 55 – Scatter plot of all solutions evaluated by brute force



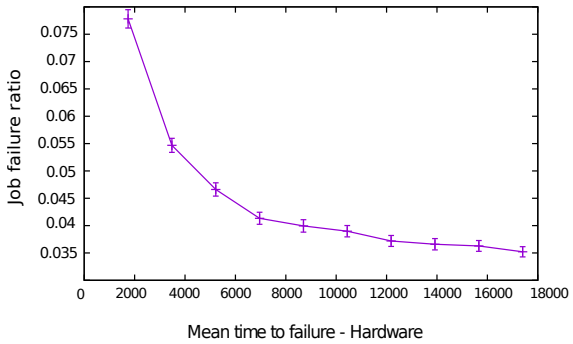
(a) MTTF Hardware × Number of processed jobs



(b) MTTF VM × Number of processed jobs



(c) MTTF Hardware × Job failure ratio



(d) MTTF VM × Job failure ratio

Figure 56 – Sensitivity analysis - first scenario (95% confidence interval)

6.3.3 Optimization of a LIGO Workflow Application

In the second scenario, we used two scientific workflows: the LIGO Inspiral Analysis workflow (BROWN et al., 2007) (Figure 58 a)) and a randomly generated DAG (Figure 58 b)). The LIGO workflow was created with the Pegasus Workflow Generator available in (JUVE; BHARATHI, 2014). This workflow generator creates synthetic workflows based on traces collected from real-world scientific workflows. An ad-hoc algorithm created the random DAG. Computing and communication times for the random DAG were generated

Table 12 – Model parameters

Parameter	Value
Mean time to failure - physical machine	8760 h
Mean time to failure - idle physical machine	13140 h
Mean time to failure - virtual machine	2880 h
Mean time to repair - physical machine	1 h
Mean time to repair - virtual machine	1 h
Arrival rate of jobs	1/2.5 (1/h)
Number of workers	20
VMs per worker	3
Activation time of standby server	0.004 (h)
Number of replications for the simulation	30

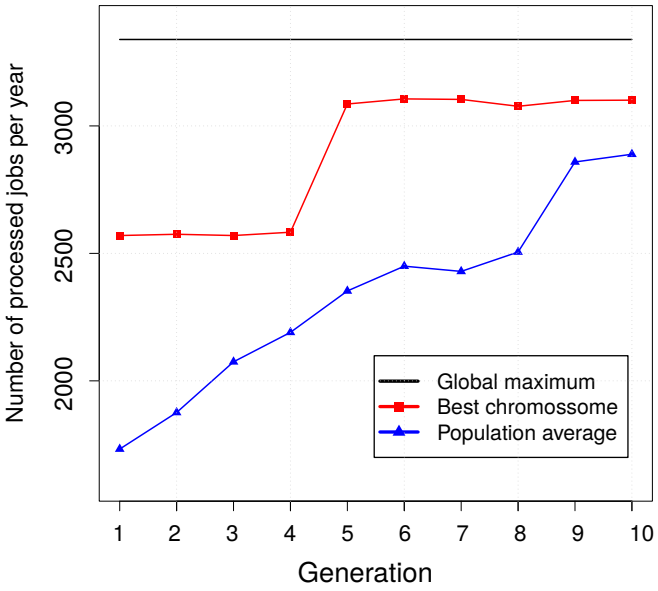


Figure 57 – Average and max fitness value (number of processed jobs per year) of each generation - first scenario

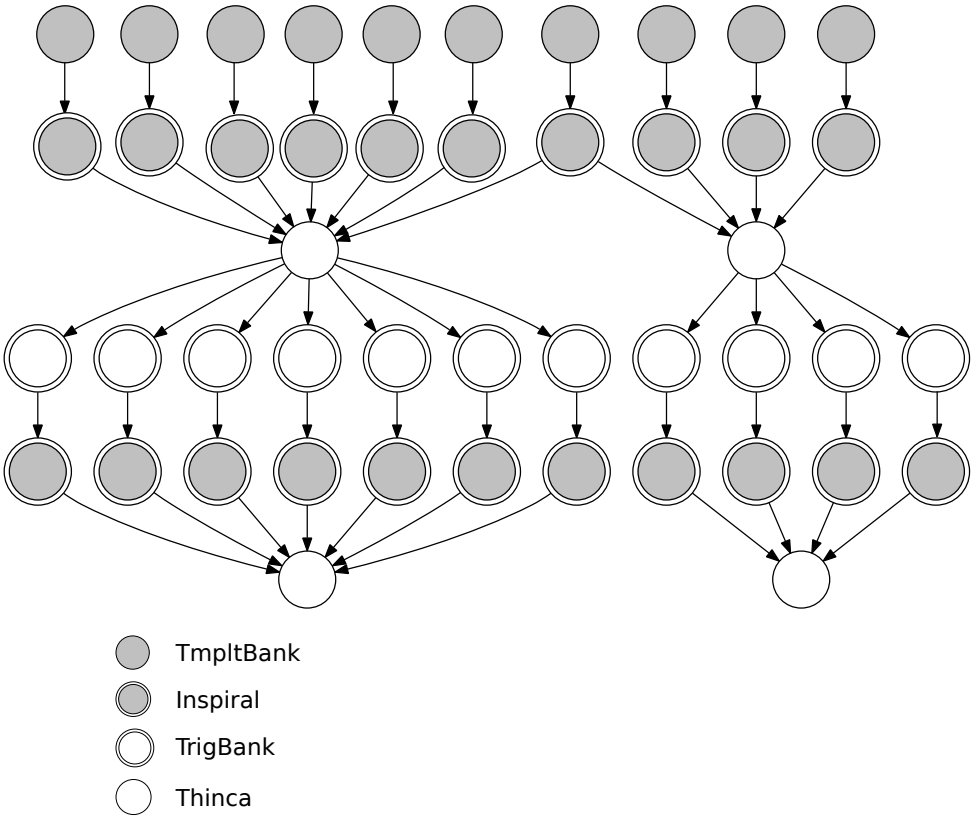
using a Uniform distribution with the interval from [1 min, 20 min] and [1min, 10 min], respectively.

Table 13 shows the updated parameters for the second scenario. Unfortunately, increasing too much the cloud scale leads to a substantial computational effort to solve the simulation model. The low frequency of failure-related events demands the adoption of a simulation time large enough to capture those events. Having highly-frequent performance events in such simulation model means that a large number of events will be generated and processed by the simulation algorithm. The problem of having both low and high frequencies for events in the model is described in the literature as **stiffness** (GOŠEVA-POPSTOJANOVA; TRIVEDI, 2000).

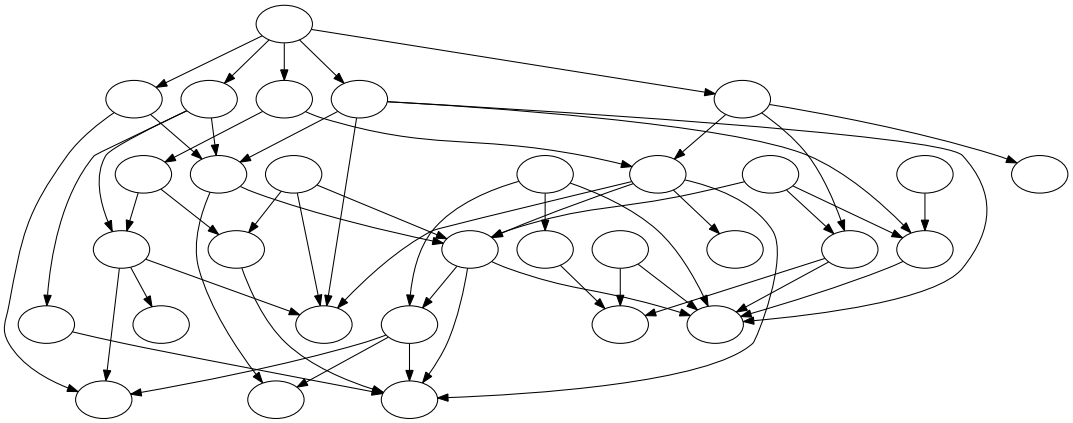
Table 13 – Model/configuration parameters - second scenario

Parameter	Value
Number of workers	50
VMs per worker	4
Number of replications for the simulation	10
Generations	25
Population size	40
Number of elite chromosomes	3

Figures 59 a) and 59 b) summarize the results of the genetic algorithm. They are displayed as boxplots for each generation produced by the optimization algorithm. Since we are using elitism (i.e., keeping the best chromossomes for each generation), the best



(a) LIGO workflow



(b) Randomly generated workflow

Figure 58 – DAGs used for the second scenario

solution is maintained until better solutions are found. In contrast to the first scenario, we noticed a more accentuated non-monotonic growth in the average fitness value (i.e., the average fitness value for the ith generation being smaller than the value for the $(i-1)$th generation). However, the algorithm can increase both the average and maximum fitness value in the long term.

The presented case study confirms the ability of the proposed simulation-based optimization method in solving the workflow scheduling problem from a performability viewpoint. Our method enables the optimization process to treat aspects that would be impossible to capture with a deterministic function, namely:

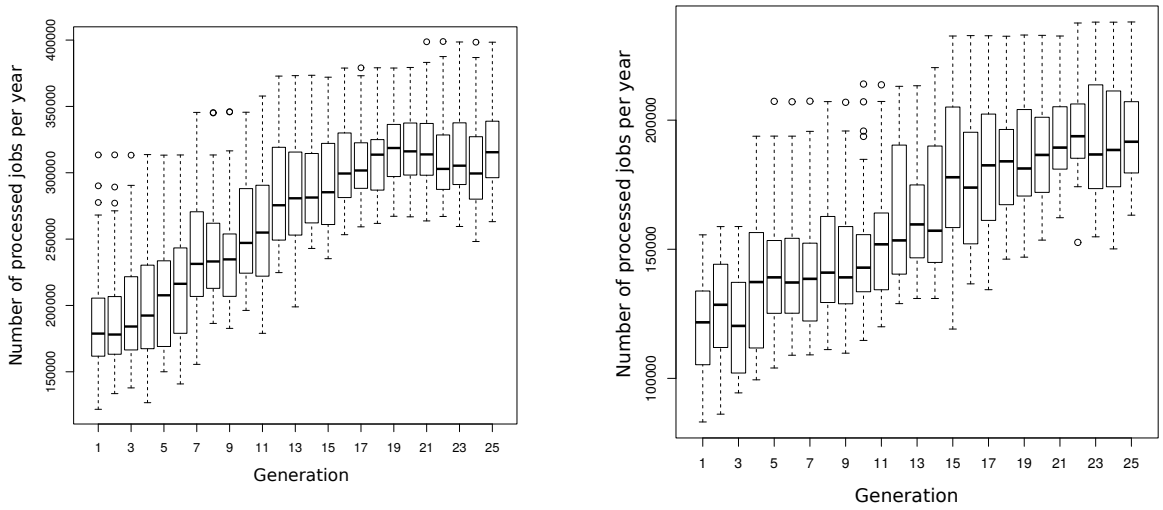
- Modeling non-deterministic and non-exponential computation/communication times;
- Capturing the failure relationships between servers and virtual machines;
- Modeling the provisioning of cloud resources to multiple users concurrently;
- Representing the influence of the cloud controller on the overall system performance.

Using such complex non-deterministic objective function (a discrete simulation model) did not cause the optimization algorithm to misbehave. The results for the LIGO and random workflows show the effectiveness of the generic operators (mutation and crossover) in avoiding getting stuck at a local maximum. New elite chromosomes were found multiple times in both scenarios.

6.3.3.1 Performance and reliability analysis

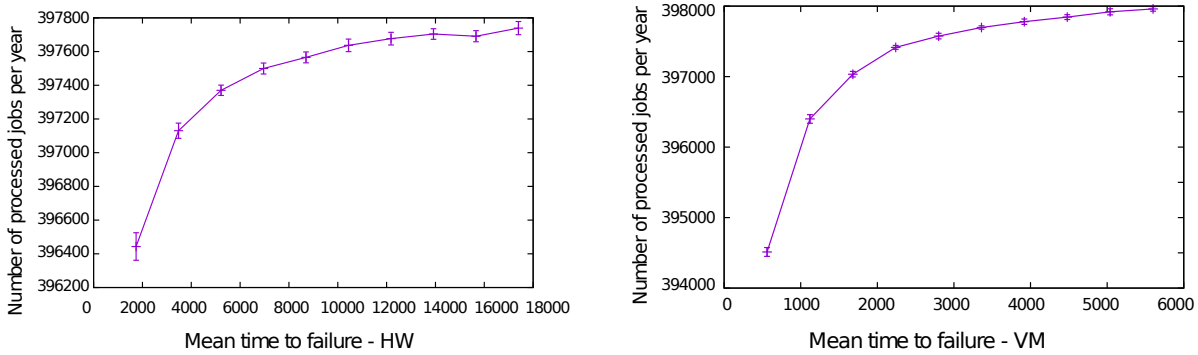
For evaluating the impact of failures on the second scenario, we performed a sensitivity analysis on the effect of hardware and VM failures in the adopted workflows. In this study, we consider the best scheduling obtained with the optimization method. The results are shown in Figure 60. We confirm the same pattern visualized in the previous section for the small DAG: the impact of hardware and VM failures diminishes as the reliability of these components reach a certain level.

Figure 61 shows the impact of failures on the cloud manager in the system throughput. It also allows us to evaluate the effectiveness of the warm-standby redundancy mechanism when contrasted with a single node cloud manager (without redundancy). Figure 61 indicates that for a small MTTF for a server node, there is a substantial increase in the number of processed jobs per year when using a redundant cloud manager. For a large MTTF, however, the difference between the mean number of processed jobs is minimal, and the confidence intervals overlap. These results indicate a negligible impact of the cloud-manager on system throughput when this subsystem is highly-available.



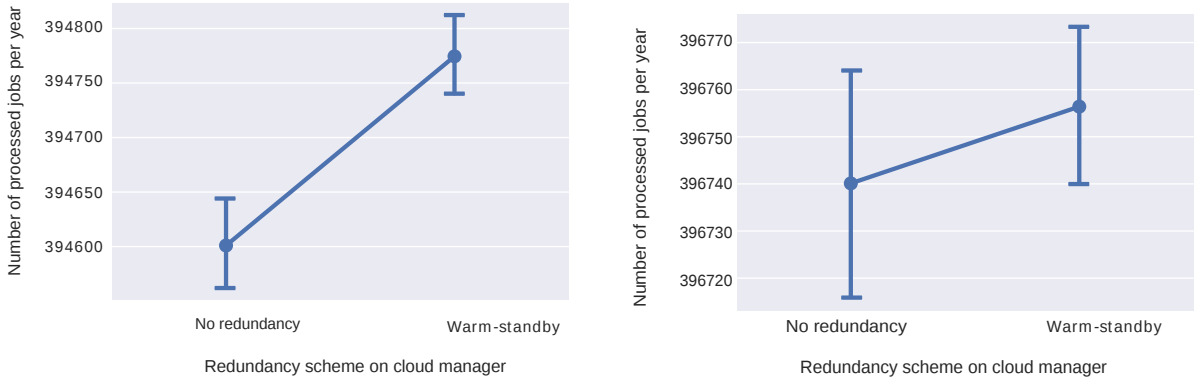
(a) LIGO workflow (b) Randomly generated DAG workflow

Figure 59 – Fitness values for each generation - second scenario



(a) MTTF Hardware × Number of processed jobs - LIGO workflow (b) MTTF VM × Number of processed jobs - LIGO workflow

Figure 60 – Sensitivity analysis - second scenario (95% confidence interval)



(a) MTTF Hardware = 1740h (b) MTTF Hardware = 8700h

Figure 61 – Influence of cloud manager failures on system throughput (95% confidence interval)

6.3.3.2 Random makespan kernel density estimation (LIGO workflow)

Considering non-deterministic communication and computation times for a workflow scheduling algorithm means that the makespan will be defined by a random variable instead of being a fixed value. We performed a kernel density estimation for the random makespan of the best scheduling found for the LIGO workflow. Figure 62 shows kernel density plot for the following scenarios: i) normally distributed times with standard deviation being equals to 10, 20, and 50% of the mean, respectively; ii) exponentially distributed times; and iii) deterministic times. The expected value for all distributions is equal to the value considered in the deterministic scenario.

It can be observed that increasing the variance of individual communication/computation times in the DAG increases the expected makespan. The exponential distribution presents a high dispersion and an expected value distant from the deterministic makespan. We, therefore, conclude that using Markov-chain based methods for analyzing workflow applications may not be a good choice since they assume exponentially distributed times. Phase-type distributions can be used to approximate non-exponential times, but using them can substantially increase the number of modeled states and cause the space-station explosion problem.

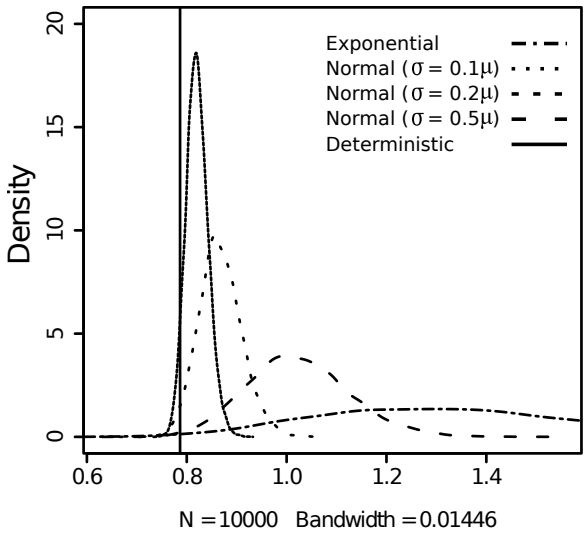


Figure 62 – Kernel density plot for Makespan of LIGO workflow (hours)

6.3.4 Third case study - final remarks

This case study studied the scheduling problem in cloud workflow applications. We adopted a performability metric, namely, the system throughput under the effect of failures, as the objective function of the problem. We employed automatic generation of performability models for a cloud application that takes as input the workflow description (as a DAG) and the infrastructure configuration. Our evaluation shows that the genetic algorithm is

efficient in optimizing both the number of virtual machines and the scheduling of the tasks regarding the system throughput.

7 CONCLUSIONS AND FUTURE WORK

The cloud computing paradigm possesses the potential to leverage advances in various areas of human knowledge by acting as infrastructure for running scientific applications. However, this potential is threatened by the intrinsic complexity of deploying distributed applications on clouds and maintaining cloud infrastructures. Cloud users and providers may face many challenges concerning the efficient usage of the available computing resources. From the cloud provider perspective, those challenges manifest as the tasks of redundancy planning, applying server/VM consolidation, performing load balancing, employing voltage scaling to reduce the energy footprint, and many others. From the user perspective, the proper provisioning of virtualized resources and the scheduling of computational tasks to these resources are the most critical issues.

Formal models enable the creation of mathematical representations of the system under study that are useful for answering "what-if" questions such as: *"what would be the expected waiting time if the number of incoming requests increases by a factor of ten?"*, or *"what would be the increase in the system availability if our company invests in an additional availability zone for our cloud infrastructure? Will we be able to meet the SLA expectations by making this investment?"*. Model-based infrastructure planning relies on the systematic evaluation of formal models in order to test a large number of parameter configurations or to search a near-optimal configuration scenarios among a large solution space. In this doctoral thesis, we have identified the main challenges in conducting model-based infrastructure planning of WaaS clouds, and proposed methods to overcome these challenges.

The remaining of this chapter summarizes the contributions achieved by this work, limitations of the proposed models, and suggest topics for further extending this research.

7.1 CONTRIBUTIONS

The main contribution of this work lies in the proposed modeling framework and the availability and performance models for conducting infrastructure planning activities in Workflow-as-a-Service cloud environments. The challenges and issues of this application area had driven the development of the proposed framework. Nevertheless, it is a general-purpose modeling framework, and it was used in different areas, such as[1]:

- Oliveira (OLIVEIRA, 2017) employed SPN modeling scripts in conjunction with the event-based simulation approach in order to build a cloud fault-injection tool based on SPN availability models;

[1] At the time of the writing of this thesis, other ongoing works are relying on the developed framework.

- Dantas (DANTAS, 2018) employed the SPN meta-modeling capabilities for creating a model for computing the capacity oriented availability of a virtualized infrastructure;
- Austregesilo et al. (AUSTREGESILO; CALLOU; MENEZES, 2018) used the modeling framework API for infrastructure planning of data-centers, employing genetic algorithms and Energy Flow models (CALLOU et al., 2013);
- Da Silva et al. (PINHEIRO et al., 2018) used a static code analyzer for converting mobile application code into SPN models. These automatically generated models were used to search the best deployment scenarios when considering the possibility of offloading some methods for remote cloud execution;
- Melo et al. (MELO et al., 2019) used DRBD models and the meta-modeling capabilities for steady-state and capacity oriented availability evaluation of hyper-converged clouds based on the OpenStack framework.

Our framework brought enhancements to the Mercury tool for performing model-based infrastructure planning of cloud workflow applications. Such enhancements include better support for hierarchical and symbolic evaluation, a comprehensive modeling API, advanced methods for SPN modeling (OLIVEIRA et al., 2017), meta-modeling capabilities, and high-level formalisms such as DRBDs and OPNs. The proposed method for DRBD evaluation include some novelties compared to the existing approaches, such as the K-out-of-N block and providing means to obtain the capacity oriented availability metric.

The uniqueness of our OPN formalism is to consider every Petri net element (e.g., places, transitions, arcs, and even the net itself) as objects, not only the tokens. We allow the modeler to extend the base classes and override the original behavior of such elements by means of implementing event methods. The advantages of this approach are twofold. We reap the benefits of object orientation when modeling with Petri nets. We can extend the base classes and declare special arcs, transitions, places, and those components can be easily reutilized. On the other hand, by extending the base classes and intercepting the Petri net events, we can specify how multiple nets in the same object space interact, and create elaborate models.

The capabilities of the modeling framework enabled us to devise a method for planning cloud workflow applications, targeting the optimization of a composite performance and availability metric. This method comprises the use of a meta-heuristic algorithm coupled with a performability model that provides the fitnesses of explored solutions. For being able to represent the combined effect of scheduling and component failures, we adopted discrete event simulation for the performability model. The obtained experimental results have shown the effectiveness of the hybrid simulation-optimization approach for optimizing the number of allocated virtual machines and the scheduling of tasks by considering the performability objective function.

Another contribution made by this work was the combination of the local search and the bisection methods for solving the redundancy allocation problem in private cloud infrastructures. By applying the proposed method, a cloud provider is able to determine a near-optimal number of server nodes for sustaining an expected availability level, and a near-optimal deployment configuration in terms of availability zones and redundancy arrangements of cloud services.

7.2 LIMITATIONS OF THE PROPOSED WORK

Combinatorial optimization and performance/dependability modeling are two areas in computer science with distinct challenges and objectives. Combining combinatorial optimization models with intricate performance and reliability models may introduce additional issues and impediments, such as high computation times for fitness values in meta-heuristic algorithms.

The main limitation of our work is the scalability of the performability models. Scaling-up towards hundreds or thousands of worker nodes would make a significant increase in the simulation time for a single scheduling, and also demand parallel simulation techniques. Some techniques proposed in Simulation/Optimization related works may not work properly with dependability or performability models. As an example, consider the technique proposed by (JUAN et al., 2015) of using an approximate deterministic model for speeding up the computation of the objective function. Applying this technique to a performability or reliability model would lead to distortions due to the large variability of the distributions for failure rates. Therefore, it is necessary to investigate novel strategies to alleviate the runtime of performability simulation models employed in conjunction with our approach. Surrogate models (GORISSEN et al., 2010) is one of such strategies. This technique consists of training a machine learning model for being used as a substitute for the simulation model, as machine learning models, once trained, can produce results faster than a computationally intensive simulation model.

7.3 FUTURE WORK

Besides the scheduling of workflow applications and the redundancy allocation problem in cloud environments, we intend to address other cloud-related infrastructure planning problems in future works. The following list covers some of the main problems explored in the cloud computing literature (the list is not exhaustive):

- Virtual machine placement (GAO et al., 2013) (XU; FORTES, 2010);
- Load balancing (MISHRA; JAISWAL, 2012) (RAMEZANI; LU; HUSSAIN, 2014);
- Web services composition (WANG; YANG; MI, 2015) (JULA; SUNDARARAJAN; OTHMAN, 2014);

- Power consumption management (GHAFARI et al., 2013) (DENG et al., 2016).

Some issues concerning combinatorial optimization aspects of model-based infrastructure planning will also be addressed in future works. We intend to study techniques for speeding up the computation of objective functions with stochastic components. Moreover, it would be worthy to explore more optimization algorithms such as bee/ant colony optimization, particle swarm, simulated annealing, and others. Future works in this direction should evaluate the suitability of employing meta-heuristic algorithms in conjunction with our approach and suggest enhancements and adaptations for these algorithms.

We also intend to extend our performability models for dealing with heterogeneity by considering virtual machines with different computational capabilities. Furthermore, we are interested in implementing fault tolerance schemes such as checkpointing and task-replication in our model.

REFERENCES

AKKA Framework. <http://akka.io/>. Last access: 2017-05-15.

ALWABEL, A.; WALTERS, R.; WILLS, G. Desktopcloudsim: Simulation of node failures in the cloud. *International Conference on Cloud Computing, GRIDs, and Virtualization*, p. 29, 2015.

AMAZON, I. Ec2 instance types – amazon web services (aws). *URL: https://aws.amazon.com/ec2/instance-types/*, 2016.

AMAZON, I. *Amazon Compute Service Level Agreement*. 2019. Available at: <https://aws.amazon.com/compute/sla/>.

ANDO, E.; NAKATA, T.; YAMASHITA, M. Approximating the longest path length of a stochastic dag by a normal distribution in linear time. *Journal of Discrete Algorithms*, Elsevier, v. 7, n. 4, p. 420–438, 2009.

ANDRADE, E.; MACIEL, P.; CALLOU, G.; NOGUEIRA, B. A methodology for mapping sysml activity diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. In: IEEE. *Digital Society, 2009. ICDS'09. Third International Conference on*. [S.l.], 2009. p. 266–271.

ARABNEJAD, H.; BARBOSA, J. G. A budget constrained scheduling algorithm for workflow applications. *Journal of grid computing*, Springer, v. 12, n. 4, p. 665–679, 2014.

ARMBRUST, M.; FOX, A.; GRIFFITH, R.; JOSEPH, A. D.; KATZ, R.; KONWINSKI, A.; LEE, G.; PATTERSON, D.; RABKIN, A.; STOICA, I. et al. A view of cloud computing. *Communications of the ACM*, ACM, v. 53, n. 4, p. 50–58, 2010.

AUSTREGESILO, M.; CALLOU, G.; MENEZES, J. A strategy based on genetic algorithms to optimize data center architectures. *IEEE Latin America Transactions*, Inderscience, 2018.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B. et al. *Fundamental concepts of dependability*. [S.l.]: University of Newcastle upon Tyne, Computing Science, 2001.

BAUER, E.; ADAMS, R.; EUSTACE, D. *Beyond redundancy: how geographic redundancy can improve service availability and reliability of computer-based systems*. [S.l.]: John Wiley & Sons, 2011.

BENDER, D. F.; COMBEMALE, B.; CRÉGUT, X.; FARINES, J. M.; BERTHOMIEU, B.; VERNADAT, F. Ladder metamodeling and plc program validation through time petri nets. In: SPRINGER. *European Conference on Model Driven Architecture-Foundations and Applications*. [S.l.], 2008. p. 121–136.

BERNARDI, S.; DONATELLI, S.; MERSEGUER, J. From uml sequence diagrams and statecharts to analysable petri net models. In: ACM. *Proceedings of the 3rd international workshop on Software and performance*. [S.l.], 2002. p. 35–45.

BITAM, S. Bees life algorithm for job scheduling in cloud computing. In: *Proceedings of The Third International Conference on Communications and Information Technology*. [S.l.: s.n.], 2012. p. 186–191.

BOESE, K. D.; KAHNG, A. B.; MUDDU, S. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Research Letters*, Elsevier, v. 16, n. 2, p. 101–113, 1994.

BOLCH, G.; GREINER, S.; MEER, H. de; TRIVEDI, K. S. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. [S.l.]: John Wiley & Sons, 2006.

BOLCH, G.; GREINER, S.; MEER, H. de; TRIVEDI, K. S. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. [S.l.]: John Wiley Sons, 2006.

BONET, P.; LLADÓ, C. M.; PUIJANER, R.; KNOTTENBELT, W. J. Pipe v2. 5: A petri net tool for performance modelling. In: *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*. [S.l.: s.n.], 2007.

BOOK, R. V. et al. Michael r. garey and david s. johnson, computers and intractability: A guide to the theory of *np*-completeness. *Bulletin (New Series) of the American Mathematical Society*, American Mathematical Society, v. 3, n. 2, p. 898–904, 1980.

BOSSE, S.; SPLIETH, M.; TUROWSKI, K. Multi-objective optimization of it service availability and costs. *Reliability Engineering & System Safety*, Elsevier, v. 147, p. 142–155, 2016.

BOYD, M. A. Dynamic fault tree models: techniques for analysis of advanced fault tolerant computer systems. Duke University, 1992.

BRETON, E.; BÉZIVIN, J. Towards an understanding of model executability. In: ACM. *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001*. [S.l.], 2001. p. 70–80.

BREUER, L.; BAUM, D. *An introduction to queueing theory: and matrix-analytic methods*. [S.l.]: Springer Science & Business Media, 2005.

BROWN, D. A.; BRADY, P. R.; DIETZ, A.; CAO, J.; JOHNSON, B.; MCNABB, J. A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis. In: *Workflows for e-Science*. [S.l.]: Springer, 2007. p. 39–59.

BUX, M.; LESER, U. Dynamiccloudsim: Simulating heterogeneity in computational clouds. *Future Generation Computer Systems*, Elsevier, v. 46, p. 85–99, 2015.

CAI, Z.; LI, Q.; LI, X. Elasticsim: A toolkit for simulating workflows with cloud resource runtime auto-scaling and stochastic task execution times. *Journal of Grid Computing*, Springer, v. 15, n. 2, p. 257–272, 2017.

CAI, Z.; LI, X.; RUIZ, R.; LI, Q. A delay-based dynamic scheduling algorithm for bag-of-task workflows with stochastic task execution times in clouds. *Future Generation Computer Systems*, Elsevier, v. 71, p. 57–72, 2017.

CALHEIROS, R. N.; RANJAN, R.; BELOGLAZOV, A.; ROSE, C. A. D.; BUYYA, R. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, Wiley Online Library, v. 41, n. 1, p. 23–50, 2011.

CALLOU, G.; MACIEL, P.; TUTSCH, D.; ARAÚJO, J. Models for dependability and sustainability analysis of data center cooling architectures. In: IEEE. *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*. [S.l.], 2012. p. 1–6.

CALLOU, G.; MACIEL, P.; TUTSCH, D.; FERREIRA, J.; ARAÚJO, J.; SOUZA, R. Estimating sustainability impact of high dependable data centers: a comparative study between brazilian and us energy mixes. *Computing*, Springer, v. 95, n. 12, p. 1137–1170, 2013.

CAROLAN, J.; GAEDE, S.; BATY, J.; BRUNETTE, G.; LICHT, A.; REMMELL, J.; TUCKER, L.; WEISE, J. Introduction to cloud computing architecture. *White Paper, 1st edn. Sun Micro Systems Inc*, 2009.

CHEN, W.; DEELMAN, E. Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In: IEEE. *E-Science (e-Science), 2012 IEEE 8th International Conference on*. [S.l.], 2012. p. 1–8.

CHEN, W.-N.; ZHANG, J. Ant colony optimization for software project scheduling and staffing with an event-based scheduler. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 1, p. 1–17, 2013.

CHIOLA, G.; FRANCESCHINIS, G.; GAETA, R.; RIBAUDO, M. Greatspn 1.7: graphical editor and analyzer for timed and stochastic petri nets. *Performance evaluation*, Elsevier, v. 24, n. 1-2, p. 47–68, 1995.

CHIOLA, G.; MARSAN, M. A.; BALBO, G.; CONTE, G. Generalized stochastic petri nets: A definition at the net level and its implications. *IEEE Transactions on software engineering*, IEEE, v. 19, n. 2, p. 89–107, 1993.

CHRISTODOULOPOULOS, K.; VARVARIGOS, E.; DEVELDER, C.; LEENHEER, M. D.; DHOEDT, B. Job demand models for optical grid research. In: SPRINGER. *International IFIP Conference on Optical Network Design and Modeling*. [S.l.], 2007. p. 127–136.

CHUOB, S.; POKHAREL, M.; PARK, J. S. Modeling and analysis of cloud computing availability based on eucalyptus platform for e-government data center. In: IEEE. *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*. [S.l.], 2011. p. 289–296.

COELHO, L. dos S. An efficient particle swarm approach for mixed-integer programming in reliability–redundancy optimization applications. *Reliability Engineering & System Safety*, Elsevier, v. 94, n. 4, p. 830–837, 2009.

COIT, D. W.; SMITH, A. E. Reliability optimization of series-parallel systems using a genetic algorithm. *IEEE Transactions on reliability*, IEEE, v. 45, n. 2, p. 254–260, 1996.

COIT, D. W.; SMITH, A. E. Solving the redundancy allocation problem using a combined neural network/genetic algorithm approach. *Computers & operations research*, Elsevier, v. 23, n. 6, p. 515–526, 1996.

DALY, D.; DEAVOURS, D. D.; DOYLE, J. M.; WEBSTER, P. G.; SANDERS, W. H. Möbius: An extensible tool for performance and dependability modeling. In: SPRINGER. *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. [S.l.], 2000. p. 332–336.

DANTAS, J.; MATOS, R.; ARAUJO, J.; MACIEL, P. An availability model for eucalyptus platform: An analysis of warm-standy replication mechanism. In: IEEE. *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*. [S.l.], 2012. p. 1664–1669.

DANTAS, J.; MATOS, R.; ARAUJO, J.; MACIEL, P. Eucalyptus-based private clouds: availability modeling and comparison to the cost of a public cloud. *Computing*, Springer, v. 97, n. 11, p. 1121–1140, 2015.

DANTAS, J. R. *Planejamento de infraestrutura de nuvens computacionais para serviço de VoD streaming considerando desempenho, disponibilidade e custo*. Phd Thesis (thesis) — Universidade Federal de Pernambuco, 2018.

DAVIS, N. A.; REZGUI, A.; SOLIMAN, H.; MANZANARES, S.; COATES, M. Failuresim: A system for predicting hardware failures in cloud data centers using neural networks. In: IEEE. *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*. [S.l.], 2017. p. 544–551.

DENG, R.; LU, R.; LAI, C.; LUAN, T. H.; LIANG, H. Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption. *IEEE Internet of Things Journal*, IEEE, v. 3, n. 6, p. 1171–1181, 2016.

DINH, H. T.; LEE, C.; NIYATO, D.; WANG, P. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, Wiley Online Library, v. 13, n. 18, p. 1587–1611, 2013.

DISTEFANO, S.; PULIAFITO, A. Dependability modeling and analysis in dynamic systems. In: IEEE. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. [S.l.], 2007. p. 1–8.

DISTEFANO, S.; XING, L. A new approach to modeling the system reliability: dynamic reliability block diagrams. In: IEEE. *Reliability and Maintainability Symposium, 2006. RAMS'06. Annual*. [S.l.], 2006. p. 189–195.

DUGAN, J. B.; DOYLE, S. A. New results in fault-tree analysis. In: *Reliability and maintainability symposium*. [S.l.: s.n.], 1996. p. 568–573.

ENTEZARI-MALEKI, R.; TRIVEDI, K. S.; SOUSA, L.; MOVAGHAR, A. Performability-based workflow scheduling in grids. *The Computer Journal*, 2018.

EUCALYPTUS. *Eucalyptus - The Open Source Cloud Platform*. 2013. Eucalyptus Systems. Disponível em: http://open.eucalyptus.com/.

EVER, E. Performability analysis of cloud computing centers with large numbers of servers. *The Journal of Supercomputing*, Springer, v. 73, n. 5, p. 2130–2156, 2017.

GAO, Y.; GUAN, H.; QI, Z.; HOU, Y.; LIU, L. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences*, Elsevier, v. 79, n. 8, p. 1230–1242, 2013.

GARG, H.; RANI, M.; SHARMA, S. An efficient two phase approach for solving reliability–redundancy allocation problem using artificial bee colony technique. *Computers & Operations Research*, Elsevier, v. 40, n. 12, p. 2961–2969, 2013.

GARG, H.; SHARMA, S. Multi-objective reliability-redundancy allocation problem using particle swarm optimization. *Computers & Industrial Engineering*, Elsevier, v. 64, n. 1, p. 247–255, 2013.

GERMAN, R. A concept for the modular description of stochastic petri nets. In: *Proc. 3rd Int. Workshop on Performability Modeling of Computer and Communication Systems*. [S.l.: s.n.], 1996. p. 20–24.

GERMAN, R. *Performance Analysis of Communication Systems with Non-Markovian Stochastic Petri Nets*. New York, NY, USA: John Wiley & Sons, Inc., 2000. ISBN 0471492582.

GERMAN, R.; KELLING, C.; ZIMMERMANN, A.; HOMMEL, G. Timenet: a toolkit for evaluating non-markovian stochastic petri nets. *Performance Evaluation*, Elsevier, v. 24, n. 1-2, p. 69–87, 1995.

GHAFARI, S. M.; FAZELI, M.; PATOOGHY, A.; RIKHTECHI, L. Bee-mmt: A load balancing method for power consumption management in cloud computing. In: IEEE. *Contemporary Computing (IC3), 2013 Sixth International Conference on*. [S.l.], 2013. p. 76–80.

GHOSH, R.; TRIVEDI, K. S.; NAIK, V. K.; KIM, D. S. End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach. In: IEEE. *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*. [S.l.], 2010. p. 125–132.

GOLDBERG, D. E.; LINGLE, R. et al. Alleles, loci, and the traveling salesman problem. In: LAWRENCE ERLBAUM, HILLSDALE, NJ. *Proceedings of an international conference on genetic algorithms and their applications*. [S.l.], 1985. v. 154, p. 154–159.

GORISSEN, D.; COUCKUYT, I.; DEMEESTER, P.; DHAENE, T.; CROMBECQ, K. A surrogate modeling and adaptive sampling toolbox for computer based design. *Journal of Machine Learning Research*, v. 11, n. Jul, p. 2051–2055, 2010.

GOŠEVA-POPSTOJANOVA, K.; TRIVEDI, K. Stochastic modeling formalisms for dependability, performance and performability. In: *Performance Evaluation: Origins and Directions*. [S.l.]: Springer, 2000. p. 403–422.

GU, J.; HU, J.; ZHAO, T.; SUN, G. A new resource scheduling strategy based on genetic algorithm in cloud computing environment. *Journal of Computers*, v. 7, n. 1, p. 42–52, 2012.

GUIMARÃES, A. P.; MACIEL, P. R.; MATIAS, R. An analytical modeling framework to evaluate converged networks through business-oriented metrics. *Reliability Engineering & System Safety*, Elsevier, v. 118, p. 81–92, 2013.

GUPTA, R.; BHUNIA, A.; ROY, D. A ga based penalty function technique for solving constrained redundancy allocation problem of series system with interval valued reliability of components. *Journal of Computational and Applied Mathematics*, Elsevier, v. 232, n. 2, p. 275–284, 2009.

HAMBY, D. A review of techniques for parameter sensitivity analysis of environmental models. *Environmental monitoring and assessment*, Springer, v. 32, n. 2, p. 135–154, 1994.

HE, Q.; HU, X.; REN, H.; ZHANG, H. A novel artificial fish swarm algorithm for solving large-scale reliability–redundancy application problem. *ISA transactions*, Elsevier, v. 59, p. 105–113, 2015.

HEIMANN, D. I.; MITTAL, N.; TRIVEDI, K. S. Availability and reliability modeling for computer systems. *Advances in Computers*, v. 31, n. C, p. 175–233, 1990.

HOFFA, C.; MEHTA, G.; FREEMAN, T.; DEELMAN, E.; KEAHEY, K.; BERRIMAN, B.; GOOD, J. On the use of cloud computing for scientific workflows. In: IEEE. *eScience, 2008. eScience'08. IEEE Fourth International Conference on*. [S.l.], 2008. p. 640–645.

HOFFMAN, F.; GARDNER, R. Evaluation of uncertainties in environmental radiological assessment models. In: TILL, J.; MEYER, H. (Ed.). *Radiological Assessments: a Textbook on Environmental Dose Assessment*. Washington, DC: U.S. Nuclear Regulatory Commission, 1983. Report No. NUREG/CR-3332.

JAIN, R. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: John Wiley & Sons, 1990.

JANOUSEK, V. Pntalk: Object orientation in petri nets. Citeseer, 1995.

JEFF Bezos' Risky Bet. <https://www.bloomberg.com/news/articles/2006-11-12/jeff-bezos-risky-bet>. Last accessed: 2018-03-12.

JENSEN, K.; KRISTENSEN, L. M.; WELLS, L. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, Springer, v. 9, n. 3-4, p. 213–254, 2007.

JUAN, A. A.; FAULIN, J.; GRASMAN, S. E.; RABE, M.; FIGUEIRA, G. A review of simheuristics: Extending metaheuristics to deal with stochastic combinatorial optimization problems. *Operations Research Perspectives*, Elsevier, v. 2, p. 62–72, 2015.

JULA, A.; SUNDARARAJAN, E.; OTHMAN, Z. Cloud computing service composition: A systematic literature review. *Expert systems with applications*, Elsevier, v. 41, n. 8, p. 3809–3824, 2014.

JUVE, G.; BHARATHI, S. *Pegasus Synthetic Workflow Generator*. 2014. Available at: <https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator>.

JUVE, G.; DEELMAN, E.; VAHI, K.; MEHTA, G.; BERRIMAN, B.; BERMAN, B. P.; MAECHLING, P. Scientific workflow applications on amazon ec2. In: IEEE. *E-Science Workshops, 2009 5th IEEE International Conference on*. [S.l.], 2009. p. 59–66.

JúNIOR, R. de S. M. *Identification of availability and performance bottlenecks in cloud computing systems: Methodology based on hierarchical models and sensitivity analysis*. Phd Thesis (thesis) — Universidade Federal de Pernambuco, 2016.

KIM, D. S.; GHOSH, R.; TRIVEDI, K. S. A hierarchical model for reliability analysis of sensor networks. In: IEEE. *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*. [S.l.], 2010. p. 247–248.

KIM, D. S.; MACHIDA, F.; TRIVEDI, K. S. Availability modeling and analysis of a virtualized system. In: IEEE. *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*. [S.l.], 2009. p. 365–371.

KLEINROCK, L. *Queueing systems, volume 2: Computer applications*. [S.l.]: wiley New York, 1976.

KLIAZOVICH, D.; PECERO, J. E.; TCHERNYKH, A.; BOUVRY, P.; KHAN, S. U.; ZOMAYA, A. Y. Ca-dag: Modeling communication-aware applications for scheduling in cloud computing. *Journal of Grid Computing*, Springer, v. 14, n. 1, p. 23–39, 2016.

KOHNE, A.; SPOHR, M.; NAGEL, L.; SPINCZYK, O. Federatedcloudsim: a sla-aware federated cloud simulation framework. In: ACM. *Proceedings of the 2nd International Workshop on CrossCloud Systems*. [S.l.], 2014. p. 3.

KUMMER, O.; WIENBERG, F.; DUVIGNEAU, M.; KOHLER, M.; MOLDT, D.; ROLKE, H. Renew–the reference net workshop. *Petri Net Newsletter*, v. 56, p. 12–16, 1999.

KUO, W.; ZUO, M. J. *Optimal reliability modeling: principles and applications*. [S.l.]: John Wiley & Sons, 2003.

KWIATKOWSKA, M.; NORMAN, G.; PARKER, D. Prism: Probabilistic symbolic model checker. In: SPRINGER. *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. [S.l.], 2002. p. 200–204.

LAKOS, C. A. *The Role of Substitution Places in Hierarchiacal Coloured Petri Nets*. [S.l.]: Department of Computer Science, University of Tasmania, 1993.

LAKOS, C. A.; KEEN, C. D. *LOOPN-Language for object-oriented Petri nets*. [S.l.]: Department of Computer Science, University of Tasmania, 1991.

LAKOS, C. A.; KEEN, C. D. *LOOPN++: A new language for object-oriented Petri nets*. [S.l.]: Department of Computer Science, University of Tasmania, 1994.

LAPRIE, J.-C. *Dependability: Basic concepts and terminology*. [S.l.]: Springer, 1992.

LAVENDER, R. G.; SCHMIDT, D. C. Active object–an object behavioral pattern for concurrent programming. Citeseer, 1995.

LD, D. B.; KRISHNA, P. V. Honey bee behavior inspired load balancing of tasks in cloud computing environments. *Applied Soft Computing*, Elsevier, v. 13, n. 5, p. 2292–2303, 2013.

LIN, W.; WU, W.; WANG, J. Z. A heuristic task scheduling algorithm for heterogeneous virtual clusters. *Scientific Programming*, Hindawi Publishing Corporation, v. 2016, 2016.

LOGOTHETIS, D.; TRIVEDI, K. Time-dependent behavior of redundant systems with deterministic repair. In: *Computations with Markov Chains*. [S.l.]: Springer, 1995. p. 135–150.

MACIEL, P.; MATOS, R.; SILVA, B.; FIGUEIREDO, J.; OLIVEIRA, D.; FÉ, I.; MACIEL, R.; DANTAS, J. Mercury: Performance and dependability evaluation of systems with exponential, expolynomial, and general distributions. In: IEEE. *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*. [S.l.], 2017. p. 50–57.

MACIEL, P. R. M. Modeling availability impact in cloud computing. In: _____. *Principles of Performance and Reliability Modeling and Evaluation: Essays in Honor of Kishor Trivedi on his 70th Birthday*. Cham: Springer International Publishing, 2016. p. 287–320. ISBN 978-3-319-30599-8. Available at: <http://dx.doi.org/10.1007/978-3-319-30599-8_11>.

MACIEL, P. R. M.; TRIVEDI, K. S.; MATIAS, R.; KIM, D. S. *Dependability Modeling*. [S.l.]: IGI Global: Hershey, 2012. 53 -97 p.

MACIEL, P. R. M.; TRIVEDI, K. S.; MATIAS, R.; KIM, D. S. *Dependability Modeling*. [S.l.]: IGI Global: Hershey, 2012. 53 -97 p.

MAINKAR, V.; TRIVEDI, K. S. Sufficient conditions for existence of a fixed point in stochastic reward net-based iterative models. *IEEE Transactions on Software Engineering*, IEEE, v. 22, n. 9, p. 640–653, 1996.

MALAWSKI, M.; JUVE, G.; DEELMAN, E.; NABRZYSKI, J. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Computer Systems*, Elsevier, v. 48, p. 1–18, 2015.

MALHOTRA, M.; TRIVEDI, K. S. A methodology for formal expression of hierarchy in model solution. In: IEEE. *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*. [S.l.], 1993. p. 258–267.

MALHOTRA, M.; TRIVEDI, K. S. Power-hierarchy of dependability-model types. *IEEE Transactions on Reliability*, IEEE, v. 43, n. 3, p. 493–502, 1994.

MARKOV, A. A. Extension of the law of large numbers to dependent quantities. *Izv. Fiz.-Matem. Obsch. Kazan Univ.(2nd Ser)*, v. 15, p. 135–156, 1906.

MARSAN, M. A.; BALBO, G.; CONTE, G.; DONATELLI, S.; FRANCESCHINIS, G. *Modelling with generalized stochastic Petri nets*. [S.l.]: John Wiley & Sons, Inc., 1994.

MARSAN, M. A.; CONTE, G.; BALBO, G. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 2, n. 2, p. 93–122, 1984.

MATOS, R.; ARAUJO, J.; OLIVEIRA, D.; MACIEL, P.; TRIVEDI, K. Sensitivity analysis of a hierarchical model of mobile cloud computing. *Simulation Modelling Practice and Theory*, Elsevier, v. 50, p. 151–164, 2015.

MELL, P.; GRANCE, T. et al. *The NIST definition of cloud computing*. [S.l.], 2011.

MELO, C.; DANTAS, J.; ARAUJO, J.; MACIEL, P.; MATOS, R.; OLIVEIRA, D.; FE, I. Models for hyper-converged cloud computing infrastructures planning. *International Journal of Grid and Utility Computing*, Inderscience, 2019.

MENASCE, D. A.; ALMEIDA, V. A.; DOWDY, L. W.; DOWDY, L. *Performance by design: computer capacity planning by example*. [S.l.]: Prentice Hall Professional, 2004.

MEYER, J. F. On evaluating the performability of degradable computing systems. *IEEE Transactions on computers*, IEEE, n. 8, p. 720–731, 1980.

MEZMAZ, M.; MELAB, N.; KESSACI, Y.; LEE, Y. C.; TALBI, E.-G.; ZOMAYA, A. Y.; TUYTTENS, D. A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. *Journal of Parallel and Distributed Computing*, Elsevier, v. 71, n. 11, p. 1497–1508, 2011.

MISHRA, R.; JAISWAL, A. Ant colony optimization: A solution of load balancing in cloud. *International Journal of Web & Semantic Technology*, Academy & Industry Research Collaboration Center (AIRCC), v. 3, n. 2, p. 33, 2012.

NIEHOERSTER, O.; BRINKMANN, A. Autonomic resource management handling delayed configuration effects. In: IEEE. *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. [S.l.], 2011. p. 138–145.

NIEHORSTER, O.; BRINKMANN, A.; FELS, G.; KRUGER, J.; SIMON, J. Enforcing slas in scientific clouds. In: IEEE. *2010 IEEE International Conference on Cluster Computing*. [S.l.], 2010. p. 178–187.

OLIVEIRA, A. S. *SIMF: Um framework de injecao e monitoramento de falhas de nuvens computacionais utilizando SPN*. Phd Thesis (thesis) — Universidade Federal de Pernambuco, 2017.

OLIVEIRA, D.; MATOS, R.; DANTAS, J.; FERREIRA, J.; SILVA, B.; CALLOU, G.; MACIEL, P.; BRINKMANN, A. Advanced stochastic petri net modeling with the mercury scripting language. In: ACM. *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*. [S.l.], 2017. p. 192–197.

PANDA, S. K.; JANA, P. K. Efficient task scheduling algorithms for heterogeneous multi-cloud environment. *The Journal of Supercomputing*, Springer, v. 71, n. 4, p. 1505–1533, 2015.

PATEL, P.; RANABAHU, A. H.; SHETH, A. P. Service level agreement in cloud computing. In: *Cloud Workshops at OOPSLA09, Orlando, Florida, USA*. [S.l.: s.n.], 2009.

PINHEIRO, T. F. da S.; SILVA, F. A.; FÉ, I.; KOSTA, S.; MACIEL, P. Performance prediction for supporting mobile applications' offloading. *The Journal of Supercomputing*, Springer, p. 1–44, 2018.

QIU, X.; SUN, P.; GUO, X.; XIANG, Y. Performability analysis of a cloud system. In: IEEE. *Computing and Communications Conference (IPCCC), 2015 IEEE 34th International Performance*. [S.l.], 2015. p. 1–6.

QUIROZ, A.; KIM, H.; PARASHAR, M.; GNANASAMBANDAM, N.; SHARMA, N. Towards autonomic workload provisioning for enterprise grids and clouds. In: IEEE. *2009 10th IEEE/ACM International Conference on Grid Computing*. [S.l.], 2009. p. 50–57.

RAEI, H.; YAZDANI, N. Performability analysis of cloudlet in mobile cloud computing. *Information Sciences*, Elsevier, v. 388, p. 99–117, 2017.

RAMAKRISHNAN, L.; REED, D. A. Performability modeling for scheduling and fault tolerance strategies for scientific workflows. In: ACM. *Proceedings of the 17th international symposium on High performance distributed computing*. [S.l.], 2008. p. 23–34.

RAMEZANI, F.; LU, J.; HUSSAIN, F. K. Task-based system load balancing in cloud computing using particle swarm optimization. *International journal of parallel programming*, Springer, v. 42, n. 5, p. 739–754, 2014.

RANDELL, B. Dependability-a unifying concept. In: IEEE. *Computer Security, Dependability and Assurance: From Needs to Solutions, 1998. Proceedings*. [S.l.], 1998. p. 16–25.

REIBMAN, A.; SMITH, R.; TRIVEDI, K. Markov and markov reward model transient analysis: An overview of numerical approaches. *European Journal of Operational Research*, Elsevier, v. 40, n. 2, p. 257–267, 1989.

REISIG, W.; ROZENBERG, G. *Lectures on petri nets i: basic models: advances in petri nets*. [S.l.]: Springer Science & Business Media, 1998.

RIMAL, B. P.; MAIER, M. Workflow scheduling in multi-tenant cloud computing environments. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 28, n. 1, p. 290–304, 2017.

ROBIDOUX, R.; XU, H.; XING, L.; ZHOU, M. Automated modeling of dynamic reliability block diagrams using colored petri nets. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, IEEE, v. 40, n. 2, p. 337–351, 2010.

RODRIGUEZ, M. A.; BUYYA, R. A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 29, n. 8, 2017.

SAHNER, R. A.; TRIVEDI, K. S. Reliability modeling using sharpe. *IEEE Transactions on Reliability*, IEEE, v. 36, n. 2, p. 186–193, 1987.

SIGNORET, J.-P.; DUTUIT, Y.; CACHEUX, P.-J.; FOLLEAU, C.; COLLAS, S.; THOMAS, P. Make your petri nets understandable: Reliability block diagrams driven petri nets. *Reliability Engineering & System Safety*, Elsevier, v. 113, p. 61–75, 2013.

SOUSA, E.; MACIEL, P.; ARAÚJO, C. Performability evaluation of eft systems using expolinomial stochastic models. In: IEEE. *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*. [S.l.], 2009. p. 3328–3333.

STORM, C. *Specification and Analytical Evaluation of Heterogeneous Dynamic Quorum-Based Data Replication Schemes*. [S.l.]: Springer Science & Business Media, 2012.

TAWFEEK, M. A.; EL-SISI, A.; KESHK, A. E.; TORKEY, F. A. Cloud task scheduling based on ant colony optimization. In: IEEE. *Computer Engineering & Systems (ICCES), 2013 8th International Conference on*. [S.l.], 2013. p. 64–69.

TRIVEDI, K. S.; VASIREDDY, R.; TRINDALE, D.; NATHAN, S.; CASTRO, R. Modeling high availability. In: *PRDC*. [S.l.: s.n.], 2006. p. 154–164.

TRUONG, H.-L.; DUSTDAR, S. On analyzing and specifying concerns for data as a service. In: IEEE. *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*. [S.l.], 2009. p. 87–94.

VALK, R. Reference and value semantics for object petri nets. In: *Proceedings of Colloquium on Petri Net Technologies for Modelling Communication Based Systems*. [S.l.: s.n.], 1999. p. 169–187.

VALK, R. *Object Petri nets: Using the nets-within-nets paradigm, Advanced Course on Petri Nets 2003 (J. Desel, W. Reisig, G. Rozenberg, Eds.), 3098*. [S.l.]: Springer-Verlag, 2003.

VALK, R. Object petri nets. *Lecture notes in computer science*, Springer, v. 3098, p. 819–848, 2004.

VAQUERO, L. M.; RODERO-MERINO, L.; CACERES, J.; LINDNER, M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, ACM, v. 39, n. 1, p. 50–55, 2008.

VESELY, W. E.; GOLDBERG, F. F.; ROBERTS, N. H.; HAASL, D. F. *Fault tree handbook*. [S.l.], 1981.

VINAY, K.; KUMAR, S. D. Fault-tolerant scheduling for scientific workflows in cloud environments. In: IEEE. *Advance Computing Conference (IACC), 2017 IEEE 7th International*. [S.l.], 2017. p. 150–155.

VÖCKLER, J.-S.; JUVE, G.; DEELMAN, E.; RYNGE, M.; BERRIMAN, B. Experiences using cloud computing for a scientific workflow application. In: ACM. *Proceedings of the 2nd international workshop on Scientific cloud computing*. [S.l.], 2011. p. 15–24.

WANG, D.; YANG, Y.; MI, Z. A genetic-based approach to web service composition in geo-distributed cloud environment. *Computers & Electrical Engineering*, Elsevier, v. 43, p. 129–141, 2015.

WANG, J.; BAO, W.; ZHU, X.; YANG, L. T.; XIANG, Y. Festal: fault-tolerant elastic scheduling algorithm for real-time tasks in virtualized clouds. *IEEE Transactions On Computers*, IEEE, v. 64, n. 9, p. 2545–2558, 2015.

WANG, J.; KORAMBATH, P.; ALTINTAS, I.; DAVIS, J.; CRAWL, D. Workflow as a service in the cloud: architecture and scheduling algorithms. *Procedia computer science*, Elsevier, v. 29, p. 546–556, 2014.

WANG, L.; TAO, J.; KUNZE, M.; CASTELLANOS, A. C.; KRAMER, D.; KARL, W. Scientific cloud computing: Early definition and experience. In: IEEE. *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*. [S.l.], 2008. p. 825–830.

WANG, T.; CHANG, X.; LIU, B. Performability analysis for iaas cloud data center. In: IEEE. *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2016 17th International Conference on*. [S.l.], 2016. p. 91–94.

WEI, B.; LIN, C.; KONG, X. Dependability modeling and analysis for the virtual data center of cloud computing. In: IEEE. *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. [S.l.], 2011. p. 784–789.

WOLSKI, R. *The High Availability in the DNA of Eucalyptus*. <http://www.eucalyptus.com/blog/2012/06/04/high-availability-in-the-dna-of-eucalyptus>. Last access: 2015-02-05.

XIA, Y.; ZHOU, M.; LUO, X.; ZHU, Q.; LI, J.; HUANG, Y. Stochastic modeling and quality evaluation of infrastructure-as-a-service clouds. *IEEE Transactions on Automation Science and Engineering*, IEEE, v. 12, n. 1, p. 162–170, 2015.

XU, H.; XING, L. Formal semantics and verification of dynamic reliability block diagrams for system reliability modeling. In: *Proc. of the 11th International Conference on Software Engineering and Applications (SEA 2007)*. [S.l.: s.n.], 2007. p. 19–21.

XU, J.; FORTES, J. A. Multi-objective virtual machine placement in virtualized data center environments. In: IEEE. *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*. [S.l.], 2010. p. 179–188.

XU, Y.; LI, K.; HE, L.; ZHANG, L.; LI, K. A hybrid chemical reaction optimization scheme for task scheduling on heterogeneous computing systems. *IEEE Transactions on parallel and distributed systems*, IEEE, v. 26, n. 12, p. 3208–3222, 2015.

YEH, W.-C.; HSIEH, T.-J. Solving reliability redundancy allocation problems using an artificial bee colony algorithm. *Computers & Operations Research*, Elsevier, v. 38, n. 11, p. 1465–1473, 2011.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, Springer, v. 1, n. 1, p. 7–18, 2010.

ZHAO, C.; ZHANG, S.; LIU, Q.; XIE, J.; HU, J. Independent tasks scheduling based on genetic algorithm in cloud computing. In: IEEE. *Wireless Communications, Networking and Mobile Computing, 2009. WiCom'09. 5th International Conference on*. [S.l.], 2009. p. 1–4.

ZHAO, H. W.; TIAN, L. W. Resource schedule algorithm based on artificial fish swarm in cloud computing environment. In: TRANS TECH PUBL. *Applied Mechanics and Materials*. [S.l.], 2014. v. 635, p. 1614–1617.

ZHAO, M.; FIGUEIREDO, R. J. Experimental study of virtual machine migration in support of reservation of cluster resources. In: ACM. *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*. [S.l.], 2007. p. 5.

ZHENG, W.; SAKELLARIOU, R. Stochastic dag scheduling using a monte carlo approach. *Journal of Parallel and Distributed Computing*, Elsevier, v. 73, n. 12, p. 1673–1689, 2013.

ZHENG, W.; WANG, C.; ZHANG, D. A randomization approach for stochastic workflow scheduling in clouds. *Scientific Programming*, Hindawi, v. 2016, 2016.

ZHENG, Z.; WANG, R.; ZHONG, H.; ZHANG, X. An approach for cloud resource scheduling based on parallel genetic algorithm. In: IEEE. *Computer Research and Development (ICCRD), 2011 3rd International Conference on*. [S.l.], 2011. v. 2, p. 444–447.

ZHOU, A.; WANG, S.; SUN, Q.; ZOU, H.; YANG, F. Ftcloudsim: A simulation tool for cloud service reliability enhancement mechanisms. In: ACM. *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*. [S.l.], 2013. p. 2.

ZHU, X.; WANG, J.; GUO, H.; ZHU, D.; YANG, L. T.; LIU, L. Fault-tolerant scheduling for real-time scientific workflows with elastic resource provisioning in virtualized clouds. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 27, n. 12, p. 3501–3517, 2016.