2015 IEEE 7th International Conference on Cloud Computing Technology and Science

A Formal Approach for QoS Assurance in the Cloud

Parisa Heidari
Electrical and Computer Engineering Department,
Western University,
London, Ontario, Canada, N6A 3K7
pheidar2@uwo.ca

Hanifa Boucheneb
Computer Engineering Department,
École Polytechnique de Montréal,
P.O. Box 6079, Station Centre-ville,
Montréal, Québec, Canada, H3C 3A7
hanifa.boucheneb@polymtl.ca

Abdallah Shami
Electrical and Computer Engineering Department,
Western University,
London, Ontario, Canada, N6A 3K7
ashami2@uwo.ca

***Abstract*—Cloud computing is an attractive business model offering cost-efficiency and business agility. Recently, the trend is that small and large businesses are moving their services to cloud environments. The quality of service is always negotiated between the cloud users and the cloud providers and documented in the service level agreement (SLA). Yet assuring–or even measuring–the quality of the provided service can be challenging. This paper proposes a formal approach for quantifying the quality of service in the cloud systems as promised in the SLA. The proposed approach uses controller synthesis to find a system configuration that meets the SLA requirements. The formal approach suggested in this paper is based on, but not limited to, Time Petri Nets (TPN). As a case study, we focus on service availability as a key performance indicator in the SLA and for a sample set of resources providing a service, we determine the system configuration satisfying the SLA.**

***Index Terms*—Cloud Computing; Formal Verification; Controller Synthesis; Quality of Service (QoS); Service Level Agreement (SLA); Availability; Time Petri Nets (TPN)**

I. Introduction

Cloud computing [1] provides a suitable business model to integrate various on-demand and pay-per-use services. With the proliferation of cloud computing environments, more companies are moving their services to cloud platforms to benefit from the cost-efficiency advantages and the agility with which business can be conducted [2]. Cloud providers promise to deliver a specified level of quality to their customers; the expected level of quality of service is determined in the Service Level Agreement (SLA) and is supposed to be guaranteed by the cloud providers.

An SLA document usually has a proprietary structure, but generally it explains the type and quality of the service agreed upon between the cloud provider and the cloud consumers [3]. Moreover, SLA determines the penalties in case of deviation from the negotiated QoS. Quality of service is delineated through Key Performance Indicators (KPI) such as response time. Cloud consumers are responsible for investigating the best profitable choice, and the provider who is offering their desired level of quality with a competitive price and enhanced support wins the client. Cloud providers in turn need to find the optimal system configuration with the least cost while meeting the expected level of QoS.

Assuring the quality of service promised in the SLA is very critical for the cloud providers. If the promised QoS is not satisfied the provider is penalized. The cloud providers often need to over-provision their resources to guarantee that QoS agreed in the SLA is met. On the other hand, there is always a trade-off between the cost and quality. The better the quality being offered, the more cost is imposed to the providers–which of course leads to more energy consumption and an increased carbon footprint. For example, service availability is a KPI that is defined as the percentage of time the service is available for the end users in a given duration. A service is considered highly available if it is accessible 99.999% of the time [4] (a.k.a., five nines). The availability is obtained from:

$$Availabiliy = MTTF/(MTTF + MTTR)$$

where MTTF stands for Mean Time To Failure and MTTR stands for Mean Time To Recover. Online services are expected to be always available but not every online service has to be highly available. The question is how to find the optimal system design that satisfies the quality of service as promised in the SLA without incurring unnecessary costs to the providers and, consequently, to the environment.

Choosing the suitable resource entities from the COTS[1] components, such that the user requirements are met and the provider's profit is maximized, is usually seen as an optimization problem that investigates available resources and their specifications and ends with recommending which resources to choose. However, quality of service does not depend solely on the type of the resource entities providing that service. Sometimes the system configuration plays a critical role in the quality of service delivered to the end-users [5]. In order to achieve long-term success, the quality of service should be monitored intently. A service is available if the resources providing that service are failing less frequently (higher MTTF), and the service is recovered shortly (lower MTTR) [6]. Yet it is important to note that the service is not recovered unless the failure is detected, and that a failure is detected faster if the service is monitored more frequently (which impacts the standard performance of the system). Based on the system

[1]Commercial off-the-shelf

978-1-4673-9560-1/15 $31.00 © 2015 IEEE
DOI 10.1109/CloudCom.2015.36

configuration, the service is recovered either by repairing the failed entities or by failing over the service to some other redundant entities that can handle the service. If the redundant resources are synchronized with the entities providing the service, the failover delay is decreased. Failure detection time and recovery time are configurable to some extent.

In this paper, the main objective is to identify the system configuration required to meet the expected level of QoS while considering available resource specifications and configuration. We consider not only the characteristics of the COTS components (such as their MTTF), but also the configurable specifications of the system design (e.g., detection time). A formal controller synthesis approach that finds the appropriate system configuration meeting the user requirements as per SLAs is the main contribution of this paper. We apply some controller synthesis techniques to create the most accommodating controller that meets the desired requirements, as established in the SLA. The proposed approach investigates whether the system design is sufficient to guarantee an SLA requirement; if not, then the proposed approach identifies how to correct the configuration design. The formal approach presented here is based on Time Petri Nets (TPN). However, application of controller synthesis for QoS assurance is not limited to Time Petri Nets and can be extended to other formal models (e.g., Timed Automata).

The rest of the paper is organized as follows: Section II provides some background knowledge about formal verification and controller synthesis, Time Petri Nets as a known formal model, and the forward on-the-fly controller synthesis algorithm of [7] for safety and reachability properties. Section III describes the suggested approach for locating the system configuration that meets the SLA promises. The formal controller synthesis approach discussed in this paper is based on the controller synthesis algorithm of [7] that finds a controller to enforce a system to meet a given behavior. Section IV is dedicated to a brief survey of some relevant fieldwork and literature. Finally, Section V concludes the paper.

II. Background

A. Formal Verification and Controller Synthesis

System designers manage the functional and non-functional requirements that should be guaranteed in the implementation. Considering the cost of a failure in critical systems, it is favorable to verify the system design before any implementation to make sure that all of the requirements will be met. Formal methods provide reliable means to verify whether the system will satisfy the properties of interest or find a counter example where the property of interest is violated. Note that verification is different from simulation. Simulation may show some errors in the behavior of the system but cannot guarantee that the system is error free or behaves as expected.

In order to verify the system formally, the behavior of the system is modeled according to the mathematical formulas; subsequently, some model-checker techniques are applied to the model to verify that it is satisfying the desired requirements (or to give a counter example). Some of the known models are algebraic modeling languages, Automata and Petri Nets. These models have a mathematical disposition, but the last two have user-friendly graphical interfaces that conceal the complicated analysis. Many extensions are already suggested in the literature to both Automata and Petri Nets making them more expressive.

Model-checking techniques are useful to locate the violations where the model is not behaving as expected. Once a model is not satisfying the expected requirement, controller synthesis methods are applied to determine how to correct the system to achieve the desired requirement.

Ramadge and Wonham introduced the theory of "control" based on the formal languages in [8]. The controller enforces a discrete event system (DES) to behave consistently. Subsequently, the theory of control was extended to the other models such as Timed Automata [9] and Time Petri Nets [10]. In these cases the control specification is articulated according to model states instead of model language. Typically, the system is modeled and the property of interest is determined. The desired controller should guarantee the satisfaction of those properties. The existence of such controller and its implementation is then investigated.

In controller synthesis, actions are categorized in two disjoint sets: controllable and uncontrollable. Controllable actions are those managed by the controller; the controller acts on the controllable actions to enforce the property of interest while it does not have any control of the uncontrollable actions.

B. Time Petri Nets (TPN)

Petri Nets [11] are directed graphs composed of two types of nodes: Places and Transitions. Time Petri Nets [12] are extensions of Petri Nets where each transition has a time interval. Formally, TPN is presented by a tuple $(P, T, Pre, Post, M_0, Is)$ where P stands for the set of Places and T stands for the set of transitions. P and T are non-empty disjoint sets. In a controllable Time Petri Net, transitions are partitioned in controllable and uncontrollable transitions (T_c, T_u respectively) where $T_c \cap T_u = \emptyset$ and $T = T_c \cup T_u$. Pre and $Post$ are the backward and the forward incidence functions ($Pre, Post : P \times T \to \mathbb{N}, \mathbb{N}$ is the set of non-negative integers), M_0 is the initial marking ($M_0 : P \to \mathbb{N}$), and Is is the static interval function ($Is : T \to \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \{\infty\})$), where $\mathbb{Q}^+$ is the set of non-negative rational numbers. Is associates with each transition t an interval called the static firing interval of t. Bounds $\downarrow Is(t)$ and $\uparrow Is(t)$ of the interval $Is(t)$ are respectively the minimum and maximum firing delays of t.

Let M be a marking and t a transition. Transition t is enabled for M if and only if all required tokens for firing t are present in M, i.e., $\forall p \in P, M(p) \geq Pre(p,t)$. In this case, the firing of t leads to the marking M' defined by: $\forall p \in P, M'(p) = M(p) - Pre(p,t) + Post(p,t)$. We denote $En(M)$ the set of transitions enabled for M, i.e., $En(M) = \{t \in T \mid \forall p \in P, Pre(p,t) \leq M(p)\}$.

The TPN state is calculated based on clock or interval characterizations [13]. Our focus here is on the second char-

acterization where the TPN state is defined as a pair (M, I), where M is a marking and I is a firing interval function $(I : En(M) \rightarrow \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \{\infty\}))$. In other words, the bounds of $I(t)$ are respectively the minimal and maximal remaining times before t can fire. Note that for any real value $v \geq \downarrow Is(t)$, if $\uparrow Is(t) = \infty$, then $\uparrow Is(t) - v = \infty$ and $max(0, \downarrow Is(t) - v) = 0$. This means that for a TPN with unbounded firing intervals, many clock states may infinitely map to the same interval state. In such a case, all these states will obviously exhibit the same future behavior. We consider here the interval characterization as it induces much more abstractions.

The TPN state space is the structure $(\mathcal{Q}, \rightarrow, q_0)$, where: $q_0 = (M_0, I_0)$ is the initial interval state of the TPN and $\mathcal{Q} = \{q | q_0 \stackrel{*}{\rightarrow} q\}$($\stackrel{*}{\rightarrow}$ being the reflexive and transitive closure of $\rightarrow$) is the set of reachable states of the model. The state space is usually abstracted to remove some irrelevant details while preserving the properties of interest. Many abstraction methods are proposed in the literature such as the *state class graph (SCG)* [14], the *zone based graph (ZBG)* [15], and so on. These methods mainly differ in the state characterization (interval or clock), the properties they preserve, etc. SCG is based on the interval states and preserves the linear properties. In this paper, we are interested in linear properties and SCG is a suitable method for our calculations.

C. A forward on-the-fly approach for controller synthesis

In [7], [16], the authors have proposed a forward on-the-fly method for controller synthesis of Time Petri Nets where the undesired behavior of the system is determined and defined as the set of bad markings. In essence, the state class graph [14] of the given TPN is explored on-the-fly and path by path to collect those sequences leading to bad states (states having bad markings). For a given class α and a sequence ω feasible from α and leading to a bad marking, the subclass of α leading to a bad marking is calculated. Ultimately, the controller should eliminate these subclasses. The algorithm starts from the last states leading to a bad marking and tries to eliminate the subclasses leading to the bad states. In the event that the algorithm cannot regulate the behavior of the system from the last states (for example, there is no controllable transition or the whole class is leading to a bad state), then the algorithm will act in some earlier states. In [16], it is proven that the controller calculated by this method is maximally permissive, meaning that it imposes the least restriction on the normal behavior of the system. The authors have also shown that this method will definitely give the controller if it exists. In other words, if the algorithm fails to calculate the controller, such controller does not exist; thus, there is no way to enforce the system to satisfy our property of interest.

III. SLA Formal Verification

In this section, we expound a solution based on controller synthesis to design a system configuration to satisfy SLA requirements with the least impact on the normal behavior of the system. Step one: the system behavior is modeled formally. In this approach, we model the system behavior using Time Petri Nets. Then, we apply the controller synthesis approach of [7] to the model and explore the entire state space of the system to verify whether the desired property is satisfied in all states and paths. As proven in [16], if there is any state where the property of interest is violated, the controller synthesis algorithm of [7] will locate a controller to force the property of interest in the whole state space. If the algorithm fails to locate said controller, then such a controller does not exist.

We focus on the service availability as a main KPI discussed in the SLA, and explain how the controller synthesis algorithm of [7], [16] is helping in QoS assurance. For better clarification, we apply the approach to a hypothetical example. Suppose a cloud provider is operating a content delivery network (CDN) as a service to the end-users, and service availability is the requirement discussed in the SLA. In order to guarantee the availability of the service, the provider has to consider some redundant resources to continue providing the service if the main resources fail. Different redundancy models are already introduced in the literature [4] (from the cloud provider's perspective). They mainly differ in the number of redundant standby units considered for each service, the number of active units that can provide the service (restricted by one or extended to more), and whether the resources providing a service can act as the standby resources of other services at the same time. From the service perspective, we can designate all of these redundancy models to two main groups: active/active and active/passive. In active/active–which is typically used for stateless services–multiple redundant resources provide the service and a load-balancer is employed to balance the workload among them. In active/passive configuration, one set of resources are providing the service and one or multiple sets of redundant resources are reserved to manage the service in case the active resources fail.

Service availability is usually accompanied with another KPI: service continuity. The function of "service continuity" is to maintain the service from where it was interrupted while the "availability" is to make the service available regardless of whether its state is preserved. In case of the stateful services, a synchronization solution is required to be integrated among the redundant resources. For example, some check-pointing solutions are required to capture and save the state periodically and then synchronize the resources. Once the active entities fail, the service is failed over to the passive redundant entities and continued from the last check-pointed state. In case of stateful services, the redundant resources can have different levels of synchronization compared to the active ones. Those levels are as follows: the redundant resources may not be instantiated (called spare); they can be instantiated but not synchronized with the active ones (cold standby); they can be instantiated, and maintain the state locally but will delay execution (warm standby); or the redundant resources can be instantiated and aware of the state of the active entities (hot standby). The redundant resources can only provide the service when they are instantiated, are aware of the state, and are in their execution mode. Thus, in case of stateful services, the

resources should be instantiated, the state should be fetched, parsed and executed. As soon as the redundant entity is in its execution mode, it can take the active role and provide the service from where it was interrupted while in the case of stateless services (or if service continuity of a stateful service is not considered), the service is provided once the resources are instantiated and are in their execution mode.

To configure the system, the CDN as a service provider needs to choose the resources providing the service, select the redundancy model and determine some configurable options such as the level of synchronization between the redundant resources and the monitoring interval of the service. The final configuration should satisfy the required QoS (service availability and continuity in this case). For the sake of simplicity, we assume the service is provided by a single component, hosted on a virtual machine (VM). The CDN provider chooses an active/passive configuration and communicates with two VMs in two different zones/regions from the underlying Infrastructure provider. Each VM can host a component. The component on the primary VM is in its execution mode and provides the service (active). The service is stateful and the state is captured and saved periodically. In case of a failure occurring on the active side, the service is failed over to the component hosted on the secondary VM (passive). The passive entity can be a hot, warm, or cold standby, or even a spare unit. As explained above, the time needed for the service recovery varies according to the type of redundant entity. Hot standby has the lowest recovery time while warm, cold and spare have higher recovery times, respectively. Moreover, the service is recovered only after being detected, and detection time depends on the monitoring frequency. The lower recovery time–which may cost more for the provider–is achieved by having a minimized detection time, a higher synchronization level, and enhanced COTS components that reach to their execution mode faster. In addition, a better synchronization level and lower detection time have more impact on the normal performance of the system. The provider needs to choose the most efficient options for the standby type and monitoring interval such that the service is recovered fast enough to meet the SLA availability requirement while imposing the least cost and performance overhead.

Assume the provider chooses a component that, on average, fails once a month and a VM that fails every 4 months. As per the SLA agreement, the required service availability is 99.99% which means the total service outage of 52.56 minutes is acceptable on a yearly basis. We expect 15 failures per year and therefore, every failure needs to be recovered within 210.24 seconds. The component is monitored periodically and the frequency is configurable with the default value of 120 seconds, meaning that the failure will be detected within 2 minutes. Once the failure is detected, the faulty resource should be cleaned up, then the redundant component hosted on the secondary VM needs some time to reach to its execution mode and provide the service. The time required to clean up the faulty entity, in addition to the time the redundant entity needs to reach to its execution mode, is between $[50, 60]$ sec

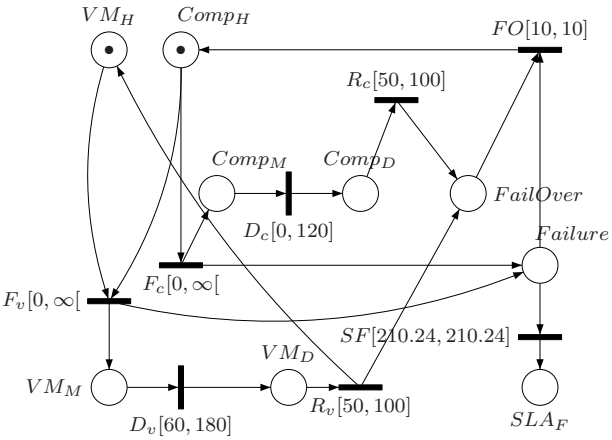

Fig. 1. TPN model of a simple CDN as a service. R_c, R_v, D_c and D_v are the controllable transitions

for hot standby, $]60, 70]$ sec for warm standby, $]70, 80]$ sec for cold standby and finally $]80, 100]$ sec for a spare, respectively. VM failure is detected within $[60, 180]$ seconds. Once a component failure due to VM failure is detected again, the service is failed over to the redundant resources. Component and VM monitoring intervals and recovery time are configurable (controllable actions). Failover takes 10 seconds.

In order to find the convenient detection time and synchronization level, first we model the behavior of the system with TPN. Fig. 1 illustrates the corresponding TPN. At the beginning, both the component providing the service and its hosting VM are healthy. $Comp_H$ and VM_H places model the healthy component and VM. At some point, the component fails and transition F_c is fired leaving one token in the $Failure$ place and one token in the $Comp_M$ place (monitoring component). Once the component failure is detected, the transition D_c is fired and the token reaches to $Comp_D$ (failure detected). After the redundant component reaches to its execution mode the transition R_c is fired and a token is generated in the $FailOver$ place. After the failover delay, the transition FO is fired, which consumes the tokens from $Failure$ and $FailOver$ places. The service is recovered and failed over to the passive entity thus, one token is generated in the $Comp_H$ place.

The component can also fail due to VM failure. Transition F_v models the VM failure and consumes both the tokens of $Comp_H$ and VM_H and subsequently generates two tokens one in the $Failure$ place and one in the VM_M place. Transition D_v models the detection time required for detecting the VM failure and once fired leaves a token in the VM_D place. Again the redundant component reaches to its execution mode after a delay (the transition R_v). Once R_v is fired, one token is generated in each of VM_H and $FailOver$ places. In this model, the transitions R_c, R_v, D_c and D_v are controllable. However, timing interval associated to R_v and R_c are the same, as they are both modeling the time required for the component to reach to its execution mode. The transition SF models the outage per failure and is fired if the service is not

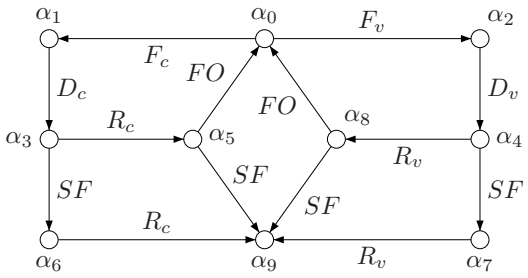

Fig. 2. The State Class Graph of the TPN given at Fig.1

recovered within the outage interval meeting the SLA. Then, if SLA_F is marked, we have a forbidden marking which should be avoided.

According to the controller synthesis approach of [7] and [16], we calculate the state space of the TPN model of Fig. 1, which is presented in Fig. 2. TABLE I depicts the information concerning the state classes of Fig. 2. In order to meet the SLA availability requirement, α_6 , α_7 and α_9 should be avoided. To avoid α_9 from the state class α_5 the transition FO should fire before the transition SF. But FO is not a controllable transition, so the algorithm goes back one state and tries to control the system from α_3 by limiting the time interval of the controllable transition, R_c. In addition, at α_3, the transition R_c should fire before SF to prevent α_6 (i.e., $R_c < SF$). The condition is $50 \leq R_c \leq 90 \wedge 50 \leq R_c \leq 80$, and therefore $50 \leq R_c \leq 80$. In the same vein, to avoid α_9 from α_8, the transition FO should fire before the transition SF. But FO is not controllable, so the algorithm goes back one state and tries to control the system from α_4 by limiting time interval of R_v. In addition, at α_4 the transition FO should fire before SF to prevent α_7 (i.e., $R_v < SF$), which is not possible. At this point, the algorithm returns to the state α_2 and tries to control the system by limiting the transition D_v such that ultimately α_7 and α_9 are avoided; the result is that we end up with $60 \leq D_v \leq 120$. Then, given these resource types, the component monitoring interval can be up to 2 minutes, VM monitoring can be between 1 and 2 minutes and hot, warm and cold standby are at acceptable limits. Ultimately, the appropriate system configuration, as well as the synchronization level of the standby entities that satisfy the QoS, are calculated appropriately.

After illustrating the idea through a simple example, we can consider more complicated cases. Failure has different sources (e.g., failure of a component, VM, server, rack, datacenter and so on). Different failures, which nonetheless may be recovered after different delays, can be detected through different mechanisms and within different detection intervals. Owing to unspecified limitations or constraints, sometimes a recovery or detection interval is not configurable and is uncontrollable. The challenge is to control the system through the limited controllable actions. Moreover, if the service is provided through multiple components, the components can have different dependencies. The TPN model should correspond to

TABLE I
THE STATE CLASSES OF THE TPN IN FIG.1

$\alpha_0 : Comp_H + VM_H$	$0 \leq F_c < \infty \wedge 0 \leq F_v < \infty$
$\alpha_1 : Comp_M + VM_H$	$0 \leq D_c \leq 120 \wedge$ $210.24 \leq SF \leq 210.24$
$\alpha_2 : Failure + VM_M$	$60 \leq D_v \leq 180 \wedge$ $210.24 \leq SF \leq 210.24$
$\alpha_3 : Comp_D + Failure$ $+VM_H$	$90.24 \leq SF \leq 210.24 \wedge$ $50 \leq R_c \leq 100$
$\alpha_4 : Failure + VM_D$	$50 \leq R_v \leq 100 \wedge 30.24 \leq SF \leq 150.24$
$\alpha_5 : FailOver + VM_H$	$10 \leq FO \leq 10 \wedge 0 \leq SF \leq 160.24$
$\alpha_6 : Comp_D + VM_H$ $+SLA_F$	$0 \leq R_c \leq 9.76$
$\alpha_7 : SLA_F + VM_D$	$0 \leq R_v \leq 69.76$
$\alpha_8 : FailOver + VM_H$ $+Failure$	$10 \leq FO \leq 10 \wedge 0 \leq SF \leq 100.24$
$\alpha_9 : FailOver + VM_H$ $+SLA_F$	–

the exact behavior of the system. For each failure type, the controller synthesis approach will determine the appropriate time intervals (monitoring and recovery) that help to satisfy the SLA availability requirements. If the SLA considers multiple KPIs at the same time, the controller synthesis approach may calculate different intervals for the same timing and then the intersection of all of the calculated intervals corresponding to different KPIs should be considered.

For a given set of entities, appropriate monitoring interval, and the suitable synchronization level for the standby entities can be extracted using this approach. The advantage of this solution is that it mitigates (and even conceals) the complex mathematical calculations; the only thing a system integrator has to do is create the Petri Net model, which is similar to determining the behavioral state machine of the system.

IV. RELATED WORK

With the proliferation of Cloud Computing frameworks, assuring the quality of the services provided by the cloud remains challenging. Research that addresses this issue has been undertaken, and is currently underway. In the following paragraphs, we briefly review the most relevant contributions.

For the cloud consumers, selection of the most convenient cloud provider needs to consider various correlated indicators owing to its multi-choice selection capacity. In [17], the authors have introduced a trust model that helps cloud customers–especially those with mission critical applications– to choose the most reliable cloud provider offering them the most reliable resources. In [18], the authors have proposed a platform that benchmarks different cloud providers, considers different user expectations and finds the one that fits the user requirements. In fact, in [17] and [18] the principal question is which provider to choose.

In [19], the authors have introduced a new cloud model called SLA-aware service and a language CSLA to describe the quality of service. Then, they have used a feedback control loop to keep the performance as stipulated in the SLA. In other words, their control approach consists in finding an elasticity engine that increases/decreases the resources in

service, according to the increase/decrease of the workload. In this way, the performance is kept according to the SLA while the resource utilization is minimized.

In [20], the resource allocation as an optimization problem (e.g., in different datacenters) is discussed in relation to cost. The authors have proposed an algorithm to calculate the cost of each resource allocation that meets the SLA requirement; they then analyze and recommend the most profitable options. In [19] and [20], the objective is to provide the expected QoS while minimizing the costs. Both of them are considering the performance KPI (e.g., the response time).

From a high availability perspective, extrapolating a configuration for availability management that meets the user requirements is a challenging, time-consuming and error-prone task for system integrators. In [21], the authors have proposed to predict which components can potentially meet the required level of service availability. Then, they eliminate the components that cannot meet the availability level of interest and apply the configuration generation method proposed and implemented in [22], [23] for the components that passed their prediction test successfully. In [21], the availability management configuration is designed based on predicting the level of service availability that the component entities can provide. The final configuration is evaluated through simulation. In their approach, the failure rates and the recovery duration are all presented as the specification of the components; in our approach, we take into account not only the specification of the resource entities, but also some configurable characteristics of the system (such as detection time and the level of synchronization). We can use the prediction method proposed in [21] simply to reduce the number of available COTS components (inputs), and then apply our formal approach to find the convenient recovery and monitoring time.

V. Conclusion

In this paper, we presented a formal controller synthesis approach for QoS assurance in cloud environments. This approach locates the system configurations meeting the QoS level stipulated in the SLA. In this paper, our focus was on the service availability as a main KPI outlined in the SLA. Through our methodology, we modeled the system using TPN. However, applying controller synthesis for QoS assurance is not limited to TPN and can be extended to the other formal models like Timed Automata.

The advantage of this solution for the system integrators is to hide the complex mathematical calculations. A system integrator knows the behavioral state machine of the system and can easily create the corresponding TPN model. The proposed approach in this paper performs all of the calculations to find a configuration that meets the SLA objectives.

VI. Acknowledgements

This work has been partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC) and MITACS. We also thank Ali Kanso for the valuable comments and discussions.

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," 2011, accessed: 2015-02-09. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[2] "Business Agility in the Cloud," accessed: 2015-08-05. [Online]. Available: https://hbr.org/resources/pdfs/tools/Verizon_Report_June2014.pdf

[3] S. Frey, C. Luthje, and C. Reich, "Key Performance Indicators for Cloud Computing SLAs," in *EMERGING 2013 : The Fifth International Conference on Emerging Network Intelligence*, 2013, pp. 60–64.

[4] M. Toeroe and F. Tam, *Service Availability: Principles and Practice*. Wiley, 2012.

[5] Y. Song, W. Tobagus, J. Raymakers, and A. Fox, "Is MTTR More Important Than MTTF for Improving User-Perceived Availability?" [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.88.182

[6] A. Kanso, M. Toeroe, and F. Khendek, "Comparing Redundancy Models for High Availability Middleware," *Computing*, vol. 96, no. 10, pp. 975–993, 2013.

[7] P. Heidari and H. Boucheneb, "Efficient method for checking the existence of a safety/ reachability controller for time Petri nets," in *10th International Conference on Application of Concurrency to System Design(ACSD)*, 2010, pp. 201–210.

[8] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[9] H. Wong-Toi and G. Hoffmann, "The control of dense real-time discrete event systems," *Technical report STAN-CS-92-1411, Stanford University*, 1992.

[10] A. S. Sathaye and B. H. Krogh, "Synthesis of real-time supervisors for controlled time Petri nets," *32nd Conference on Decision and Control*, vol. 1, pp. 235–236, 1993.

[11] C. Petri, "Kommunication mit Automaten." PhD Dissertation, University of Bonn, 1962.

[12] P. M. Merlin, "A study of the recoverability of computing systems," PhD dissertation, University of California, Irvine, United States, 1974.

[13] W. Penczek and A. Polrola, "Specification and model checking of temporal properties in time Petri nets and timed automata," *25th International conference on application and theory of Petri nets*, vol. 3099 of LNCS, pp. 37–76, 2004.

[14] B. Berthomieu and M. Diaz, "Modeling and verification of time dependent systems using time Petri nets," *IEEE Transactions on Software Engineering*, vol. 17, no. 3, 1991.

[15] H. Boucheneb, G. Gardey, and O. H. Roux, "TCTL model checking of time Petri nets," *Journal of Logic and Computation*, vol. 6, no. 19, pp. 1509–1540, 2009.

[16] P. Heidari and H. Boucheneb, "Maximally permissive controller synthesis for time Petri nets," *International Journal of Control*, 2012.

[17] M. Alhamad, T. Dillon, and E. Chang, "SLA-Based Trust Model for Cloud Computing," in *13th International Conference on Network-Based Information Systems*, 2013, pp. 321–324.

[18] M. Souidi, S. Souihi, S. Hoceini, and A. Mellouk, "An Adaptive Real Time Mechanism For IaaS Cloud Provider Selection Based on QOS Aspects," in *International Conference on Communications (ICC)*, 2015.

[19] D. Serrano, S. Bouchenak, , Y. Kouki, T. Ledoux, J. Lejeune, and J. Sopena, "Towards QoS-Oriented SLA Guarantees for Online Cloud Services," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2013, pp. 50–57.

[20] H. Goudarzi and M. Pedram, "Multi-dimensional SLA-based Resource Allocation for Multi-tier Cloud Computing Systems," in *4th International Conference on Cloud Computing*, 2011, pp. 324–331.

[21] P. Pourali, M. Toeroe, and F. Khendek, "Enhanced Configuration Generation Approach for Highly Available COTS Based Systems," in *9th International Conference on Availability, Reliability and Security (ARES)*, 2014, pp. 104–113.

[22] A. Kanso, M. Toeroe, A. Hamou-lhadj, and F. Khendek, "Generating AMF Configurations from Software Vendor Constraints and User Requirements," in *International Conference on Availability, Reliability and Security (ARES)*, 2009, pp. 454–461.

[23] A. Kanso, "Automated Configuration Design and Analysis for Service High-Availability," PhD Dissertation, Concordia University, Montreal, Canada, 2012. [Online]. Available: http://spectrum.library.concordia.ca/974790/9/Kanso_PhD_F2012.pdf