Formal verification of service level agreements through distributed monitoring*

Behrooz Nobakht[1,2], Stijn de Gouw[2,3], and Frank S. de Boer[1,3]

[1] Leiden Advanced Institute of Computer Science
Leiden University
bnobakht@liacs.nl
[2] SDL Fredhopper
{bnobakht,sgouw}@sdl.com
[3] Centrum Wiskunde en Informatica
{frb,cdegouw}@cwi.nl

Abstract. In this paper, we introduce a formal model of the availability, budget compliance and sustainability of distributed services, where service sustainability is a new concept which arises as the composition of service availability and budget compliance. The model formalizes a distributed platform for monitoring the above service characteristics in terms of a parallel composition of task automata, where dynamically generated tasks model asynchronous events with deadlines. The main result of this paper is a formal model to optimize and reason about service characteristics through monitoring. In particular, we use schedulability analysis of the underlying timed automata to optimize and guarantee service sustainability.

Keywords: runtime monitoring, service availability, budget compliance, service sustainability, distributed architecture, cloud computing, service level agreement

1 Introduction

Cloud computing provides the elastic technologies for virtualization. Through virtualization, software itself can be offered as a service (Software as a Service, SaaS). One of the aims of SaaS is to allow service providers to offer reliable software services while scaling up and down allocated resources based on their availability, budget, service throughput and the Service Level Agreements (SLA). Thus, it becomes essential that virtualization technologies facilitate elasticity in a way that enables business owners to *rapidly* evolve their systems to meet their customer requirements and expectations.

The fundamental technical challenge to a SaaS offering is maintaining the quality of service (QoS) promised by its SLA. In SaaS, providers must ensure a

* This paper is funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services, http://www.envisage-project.eu.

consistent QoS in a dynamic virtualized environment with variable usage patterns. Specifically, virtualized environments such as the cloud provide elasticity in resource allocation, but they often do not offer an SLA that can guarantee constant resource availability. As a result, SaaS providers are required to react to resource availability at runtime. Furthermore, by offering a 24/7 software service, SaaS providers must be able to react to certain service usage patterns, such as an increase in throughput to ensure the SLA is maintained.

Runtime monitoring [20,4] is a dynamic analysis approach based on extracting relevant information about the execution. Runtime monitoring may be employed to collect statistics about the service usage over time, and to detect and react to service behavior. This latter ability is fundamental in the SaaS approach to guarantee the SLA of a service and is the focus of this paper.

The monitoring model that is presented in this paper is designed to *observe* in real-time certain service characteristics and *react* to them to ensure the evolution of the system towards its SLA. Asynchronous communication is an essential feature of a monitoring model in a distributed context. Asynchronous communication accomplishes non-intrusive observations of the service runtime. Further, the monitoring model is expected to operate according to certain real-time constraints specified by the SLA of the service. Satisfying the real-time constraints is the main challenge in a distributed monitoring model.

In this paper, we formalize service availability and budget compliance in a distributed deployment environment. This formalization is based on high-level task automata models [1,9,13]. The automata capture the real-time evolution of the resources provided by a distributed deployment platform and the above two main service characteristics. These task automata represent the real-time generation of the asynchronous events extended with deadlines [3,22] by the monitoring platform for managing resources (i.e. allocation or deallocation). The main result of this paper is a formal model to optimize and reason about the above service characteristics through monitoring. In particular, the *schedulability* of the underlying timed automata implies service availability and budget compliance. Furthermore, we introduce a composition of service availability and budget compliance which captures service sustainability. We show that service sustainability presents a multi-objective optimization problem.

2 Related Work

Vast research work present different aspects of runtime monitoring. We focus on those that present a line of research for distributed deployment of services.

MONINA [12] is a DSL with a monitoring architecture which supports certain mathematical optimization techniques. A prototype implementation is available. Accurately capturing the behavior of an in-production legacy system coded in a conventional language seems challenging: it requires developing MONINA components, which generate events at a specified fixed rate, there are no control structures (if-else, loops), the data types that can be used in events are predefined, and there are no OO-features. We use ABS [15], an executable mod-

eling language that supports all of these features and offers a wide range of tool-supported analyses [5,25]. The mapping from ABS to timed automata [1] allows to exploit the state-of-the-art tools for timed automata, in particular for reasoning about real-time properties (and, as we show, SLAs using schedulability analysis [9]). MONINA offers *two pre-defined* parameters that can be used in monitoring to adapt the system: cost and capacity. Our service metric function generalizes this to *arbitrary user-defined* parameters, including cost and capacity.

Hogben and Pannetrat examine in [11] the challenges of defining and measuring availability to support real-world service comparison and dispute resolution through SLAs. They show how two examples of real-world SLAs would lead one service provider to report 0% availability while another would report 100% for the same system state history but using a different period of time. The transparency that the authors attempt to reach is addressed in our work by the concept of monitoring window and expectation tolerance in Section 4. Additionally, the authors take a continuous time approach contrasted with ours that uses discrete time advancements. Similarly, they model the property of availability using a two-state model.

The following research works provide a language or a framework that allows to formalize service level agreements (SLA). However, they do not study how such SLAs can be used to monitor the service and evolve it as necessary. WSLA [18] introduces a framework to define and break down customer agreements into a technical description of SLAs and terms to be monitored. In [21], a method is proposed to translate the specification of SLA into a technical domain directed in SLA@SOI EU project. In the same project, [8] defines terms such as availability, accessibility and throughput as notions of SLA, however, the formal semantics and properties of the notions are not investigated. In [6], authors describe how they introduce a function how to decompose SLA terms into measurable factors and how to profile them. Timed automata is used in [24] to detect violations of SLA and formalize them.

Johnsen [16] introduce "deployment components" using Real-Time ABS [3]. A deployment component enables an application to acquire and release resources on-demand based on a QoS specification of the application. A deployment component is a high level abstraction of a resource that promotes an application to a resource-aware level of programming. Our work is distinguished by the fact that we separate the monitors from the application (service) themselves. We argue that we aim to design the monitoring model to be as *non-intrusive* as possible to the service runtime. Thus, we do not deploy the monitors inside the service runtime.

In Quanticol EU project[4], authors in [7] and [10] use statistical approaches to observe and guarantee service level agreements for public transportation. We also present that service characteristics can be composed together. This means that evolving a system based on SLAs turns into a multi-object optimization problem. In addition, in COMPASS EU project[5], CML [26] defines a formal language to

[4] Quanticol EU project with no. 600708: `http://quanticol.eu/`
[5] COMPASS EU project with no. 287829: `http://www.compass-research.eu/`

model systems of systems and the contracts between them. CML studies certain properties of the model and their applications. CML is used in the context of a Robotics technology to model and ensure how emergency sensors should react and behave according to the SLAs defined for them. Our approach is similar to provide a generic model for service characteristics definition, however, we utilize timed and task automata.

3 SDL Fredhopper Cloud Services

In this section, we introduce a running example in the context of SDL Fredhopper. We use the example in different parts of the paper and also in the experiments.

SDL Fredhopper develops the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). Fredhopper Cloud Services provides several SaaS offerings on the cloud. These services are exposed at endpoints. In practice these endpoints typically are implemented to accept connections over HTTP. For example, one of the services offered by these endpoints is the Fredhopper Query API, which allows users to query over their product catalog via full text search[6] and faceted navigation[7].

A customer of SDL Fredhopper using Query API owns a *single* HTTP endpoint to use for search and other operations. However, internally, a number of resources (virtual machines) are used to deliver Query API for the customer. The resources used for a customer are managed by a load balancer. In this model of deployment, each resource is launched to serve *one* instance of Query API; i.e. resources are *not* shared among customers.

When a customer signs a contract with SDL Fredhopper, there is a clause in the contract that describes the minimal QoS levels of the Query API. For example, we have a notion of query per second (QPS) that defines the number of completed queries per second for a customer. An agreement is a bound on the expected QPS and forms the basis of many decisions (technical or legal) thereafter. The agreement is used by the operations team to set up an environment for the customer which includes the necessary resources described above. The agreement is additionally used by the support team to manage communications with the customer during the lifetime of the service for the customer.

Maintaining the services for more than 250 customers on more than 1000 servers is not an easy operation task [8]. Thus, to ensure the agreements in a customer's contract:

- The operation team maintains a monitoring platform to get notifications on the current metrics.

[6] http://en.wikipedia.org/wiki/Full_text_search
[7] http://en.wikipedia.org/wiki/Faceted_navigation
[8] Figures are indication of complexity and scale. Detailed confidential information may be shared upon official request.

- The operation team performs *manual* intervention to ensure that sufficient resources are available for a customer (launching or terminating).
- The monitoring platform depends on *human* reaction.
- The cost that is spent for a customer on the basis of safety can be *optimized*.

In this paper, we use the notion of QPS as an example in the concepts that are presented in this research. We use the example here to demonstrate how the model that is proposed in this research can address the issues above and alleviate the *manual* work with *automation*. The manual life cycle depends on the domain-specific and contextual knowledge of the operations team for every customer service that is maintained in the deployment environment. This is labor-intensive as the operations team stands by 24×7. In such a manual approach, the business is forced to over-spend to ensure service level agreements for customers.

4 Distributed Monitoring Model

We introduce a distributed monitoring platform and its components and discuss some underlying assumptions and definitions. Further, we define the notion of service availability and service budget compliance. In the deployment environment (e.g., "the cloud"), every server from the IaaS provider is used for a *single* service of a customer, such as the Query Service API for a customer of SDL Fredhopper (c.f. Section 3). Typically, multiple servers are allocated to a single customer. The number of servers allocated for a customer is not visible to the customer. The customer uses a single endpoint - in the load balancer layer - to access all their services.

The ultimate goal is to maintain the environment in such a way that customers and their end users experience the delivered services up to their expectations while minimizing the cost of the system. The first objective can be addressed by adding resources; however, this conflicts with the second goal since it increases the cost of the environment for the customer. In this section, we formalize the above intuitive notions as *service availability* and *service budget compliance*.

We then develop a distributed monitoring platform that aims to optimize these service characteristics in a deployment environment. The monitoring platform works in two cyclic phases: *observation* and *reaction*. The observation phase takes measurements on services in the deployment environment. Subsequently, the corresponding levels of the service characteristics are calculated. In the reaction phase, if needed, a platform API is utilized to make the necessary changes to the deployment environment (e.g. adjust the number of allocated resources) to optimize the service characteristics. The monitoring platform builds on top of a real-time extension of the actor-based language ABS [15]. To ensure non-intrusiveness of the monitor with the running service, each monitor is an active object (actor) running on a separate resource from that which runs the service itself, and the components of the monitoring platform communicate through *asynchronous messages* with deadlines [16].

Below, we discuss assumptions and basic oncepts that will be used in the analysis of the formal properties of the monitoring platform and corresponding theorems. We assume that the external infrastructure provider has an *unlimited* number of resources. Further, we assume that all resources are of the *same type*; i.e. they have the same computing power, memory, and IO capacity. Finally, we assume that every resource is initialized within at most t_i amount of time.

In our framework time T is a universally shared clock based on the NTP [9] that is used by all elements of the system in the same way. T is discrete. We fix that the unit of time is *milliseconds*. This level of granularity of time unit means that between two consecutive milliseconds, the system is not observable. For example, we use the UTC time standard for all services, monitors and platform API. We refer to the current time by t_c.

We denote by r a resource which provides computational power and storage and by s a general abstraction of a service in the deployment environment. A service exposes an API that is accessible through a delivery layer, such as HTTP. In our example, a service is the Query API (c.f. Section 3) that is accessible through a single HTTP endpoint.

In our framework, *monitoring platform* P is responsible for (de-)allocation of resources for computation or storage. We abstract from a specific implementation of the monitoring platform P through an API in Listing 1. There is only *one* instance of P available. In this paper, P internally uses an external infrastructure provisioning API to provide resources (e.g. AWS EC2). The term "platform" is interchangeably used for monitoring in this paper. The platform provides a method `getState(Service s)` which returns the number of resources allocated to the given service s at time t_c.

Listing 1: Platform API

```
1 interface Platform {
2   void    allocate(Service s);
3   void    deallocate(Service s);
4   Number  getState(Service s);
5   boolean verify_α(Service s);
6   boolean verify_β(Service s);
7 }
```

We use monitoring to observe the external behavior of a service. We formalize the external behavior of a service with its service-level agreement (SLA). An SLA is a contract between the customer (service consumer) and the service provider which defines (among other things) the minimal quality of the offered service, and the compensation if this minimal level is not reached. To formally analyze an SLA, we introduce the notion of a service metric function. We make basic measurements of the service externally in a given monitoring window (a duration). The service metric function aggregates the basic measurements into a single number that indicates the quality of a certain service characteristic (higher numbers are better).

Basic measurement $\mu(s, r, t)$ is a function that produces a real number of a *single* monitoring check on a resource r allocated to service s at some time t. For example, for SDL Fredhopper cloud services, a basic measurement is the number of completed queries at the current time.

[9] `https://tools.ietf.org/html/rfc1305`

Service Metric f_s is a function that aggregates a sequence of basic non-negative measurements to a single non-negative real value: $f_s : \bigcup_n \mathbb{R}^n \to \mathbb{R}$. For example, for SDL Fredhopper cloud services, the service metric function f_s calculates the average number of queries per second (QPS) given a list of basic measurements.

Monitoring Window is a duration of time τ throughout which basic measurements for a service are taken.

Monitoring Measurement is a function that aggregates the basic measurements for a service over its resources in the last monitoring window. The last monitoring window is defined as $[t_c - \tau, t_c]$. To produce the monitoring measurement, f_s is applied. Formally:

$$\mu(s, r, \tau) = f_s\big(\langle \mu_i(s, r, t) \rangle_{i=0}^{\infty}\big) \text{ where } t \in [t_c - \tau, t_c]$$

in which $\mu_i(s, r, t)$ is the i-th basic measurement of services s on resource r at time t where $t \in [t_c - \tau, t_c]$.

Definition 1 (Service Availability $\alpha(s, \tau, t_c)$)**.** First, we need a few auxiliary definitions before we can define service availability.

Service Capacity $\kappa_\sigma(s, \tau) = \sum_{r \in \sigma(s)} \mu(s, r, \tau)$ denotes the capability of service s that is the aggregated monitoring measurements of its resources over the monitoring window τ and $\sigma(s)$ is the number of allocated resources to service s.

Agreement Expectation $E(s, \tau, t_c)$ is the minimum number of requests that a customer expects to complete in a monitoring window τ. The agreement expectation depends on the current time t_c because the expectation may change over time. For example, SDL Fredhopper customers expect a different QPS during Christmas.

We define the availability of a service $\alpha(s, \tau, t_c)$ in every monitoring window τ as:

$$\alpha(s, \tau, t_c) = \frac{\kappa_\sigma(s, \tau)}{E(s, \tau, t_c)}$$

Capacity Tolerance $\varepsilon_\alpha(s, \tau)) \in [0, 1]$ defines how much $\kappa_\sigma(s, \tau)$ can deviate from $E(s, \tau, t_c)$ in every time span of duration τ.

Service Guarantee Time t_G is the duration within which a customer expects service availability reaches an acceptable value after a violation. Typically, t_G is an input parameter from the customer's contract.

Example 1. Intuitively, $\alpha(s, \tau, t_c)$ presents the actual capability of a service s over a time period τ compared to the expectation on the service $E(s, \tau)$. For values $\alpha(s, \tau, t_c) \ll 1 - \varepsilon_\alpha(s, \tau))$, the resource for service s are at "under-capacity" while for values $\alpha(s, \tau, t_c) \gg 1 + \varepsilon_\alpha(s, \tau))$, there is "over-capacity". The goal is optimize $\alpha(s, \tau, t_c)$ towards a value of 1.

For example, we expect a query service to be able to complete 10 queries per second. We define the monitoring window $\tau = 5$ minutes; thus, $E(s, \tau, t_c) = 10 \times 60 \times 5 = 3000$. Suppose we allocate only one resource to the service, measure the service during a single monitoring window τ and find $\mu(s, r, \tau) = 2900$. Then $\alpha(s, \tau, t_c) = \frac{2900}{3000} = 0.966$. If we have $\varepsilon_\alpha(s, \tau)) = 0.03$, this means that service s is under-capacity because $\alpha(s, \tau, t_c) < 1 - \varepsilon_\alpha$.

Definition 2 (Budget Compliance $\beta(s,\tau)$)**.** We first provide a few auxiliary definitions.

Resource Cost $€(r,\tau) \in \mathbb{R}^+$ is the cost of resource r in a monitoring window τ which is determined by a fixed resource cost per time unit.

Service Cost $€_\sigma(s,\tau) \in \mathbb{R}^+$ is the cost of a service s in a monitoring window τ and defined as $€_\sigma(s,\tau) = \sum_{r\in\sigma(s)} €(r,\tau)$.

Service Budget $B(s,\tau)$ specifies an upper bound of the expected cost of a service in the time span τ. Intuitively $B(s,\tau)$ is the allowed budget that can be spent for service s over the time span τ. The service budget is typically chosen to be fixed over any time span τ.

We are now ready to define service budget compliance $\beta(s,\tau)$ that, intuitively, represents how a service complies with its allocated budget:

$$\beta(s,\tau) = \frac{€_\sigma(s,\tau)}{B(s,\tau)}$$

Budget Tolerance $\varepsilon_\beta(s,\tau) \in [0,1]$ specifies how much the service cost $€(s,\tau)$ can deviate from $B(s,\tau)$ in every time span of duration τ.

Service Guarantee Time t_G is similar to that defined for service availability.

Example 2. Assume every resource on the environment costs 1 (e.g. €) per hour. Suppose we set a budget of 1.5 per hour for every service, allocate *one* resource to the service and define a monitoring window of $\tau = 5$ minutes. Every hour has 12 monitoring windows. This means that each resource costs $€(r,\tau) = \frac{1}{12} \approx 0.08$ per monitoring window. Since there is only one resource, the service cost is $€(s,\tau) = \sum_{r\in\sigma(s)} €(s,\tau) \approx 0.08$ per monitoring window. On the other hand, if we calculate the budget for one monitoring window, we have $B(s,\tau) = \frac{1.5}{12} = 0.125$ per monitoring window. This yields budget compliance as $\beta(s,\tau) = \frac{0.08}{0.125} = 0.64$.

The formal definitions of service availability and budget compliance provide a rigorous basis for automatic deployment of resource-aware services with an appropriate quality of service, taking costs into account. This in particular includes automated scaling up or down of the service with the help of monitoring checks that are installed for the service. The fundamental challenge in ensuring service availability and budget compliance is that they have *conflicting* objectives:

$$\alpha(s,\tau,t_c)\uparrow \iff \beta(s,\tau)\downarrow$$

Intuitively, if more resources are used to ensure the availability of a service; then $\alpha(s,\tau,t_c)$ increases. However, at the same time, the service costs more; i.e. budget compliance $\beta(s,\tau)$ decreases.

5 Service Characteristics Verification

In this section, we use timed automata and task automata to model the behavior of a monitoring platform P, the deployment environment E, and the monitoring

components for service availability $\alpha(s, \tau, t_c)$ and budget compliance $\beta(s, \tau)$. [13] defines a task automata as an extension of timed automata in which each task is a piece of executable program with (b, w, d): best/worst time and deadline of the task. A task automata uses a scheduler for the tasks to schedule each task with a location on a queue.

Modeling the elements of the monitoring platform is necessary to be able to study certain properties of the system. The most important goal of a monitoring platform is to enable the autonomous operation of a set of services according to their SLA. Thus, it is essential how to analyze that the monitoring platform can provide certain guarantees about the service and its SLA. In addition, it is important be able to verify the monitoring platform through model checking and schedulability analysis. Using timed automata and task automata facilitates model checking and verification through formal method tools such as UPPAAL [2] supporting advanced methods such as state-space reduction [19].

We use task automata as defined in [9,14,13]. Task automata are an extension of timed automata [1]. In addition, we design the automata for the monitoring platform using the real-time extension of task automata presented in [13] p. 92 in which the author presents a mapping from Real-Time ABS [16] to the equivalent task automata.

A task type is a piece of executable program/code represented by a tuple (b, w, d), where b and w respectively are the best-case and worst-case execution times and d is the deadline. In a task automata, there are two types of transitions: *delay* and *discrete*. A delay transition models the execution of a running task by idling for other tasks. A discrete transition corresponds to the arrival of a new task. When a new task is triggered, it is placed into a certain position in the queue based on a scheduling policy [23,22]. Examples of a scheduling policy are FIFO or EDF (earliest deadline first). The scheduling policy is modeled as a timed automaton Sch. Every task has its own stop watch. The scheduler also maintains a separate stop watch for each task to determine if a task misses its deadline. All stop watches work at the same clock speed specified by T.

We design separate automata for each service s characteristic: service availability $\alpha(s, \tau, t_c)$ by an automata M_{α_s} and service budget compliance $\beta(s, \tau)$, by an automata M_{β_s}. Each automaton is responsible for one goal: to optimize the service characteristic. M_{α_s} aims to improve $\alpha(s, \tau, t_c)$ whereas M_{β_s} aims to improve $\beta(s, \tau)$. M_{α_s} uses `allocate` to launch a new resource in the environment and improve the service s. In contrast, M_{β_s} uses `deallocate` to terminate a resource to decrease the cost of the service.

We use task automata to design M_{α_s}. Periodically, M_{α_s} checks whether the service availability is within the thresholds, taking tolerance into account (Definition 1). If the condition fails, M_{α_s} generates a task for monitoring platform P to allocate a new resource to service s with a deadline of τ. We define the period to be τ. We use the semantics of a task automata in [13] p. 92 in the transitions of the task automata. Figure Fig. 1a and Fig. 1b present M_{α_s} and M_{β_s}. Both M_{α_s} and M_{β_s} share state with the monitoring platform P. The state keeps the current number of resources for a service s that is denoted by $\sigma(s)$. All timed

automata and task automata in the monitoring platform have shared access to $\sigma(s)$. In the automata, we use a conditional statement to check the service characteristics $\alpha(s,\tau,t_c)$ or $\beta(s,\tau)$. If the condition fails, M_{α_s} requests P to `allocate` a new resource to s and M_{β_s} requests P to `deallocate` a resource. In addition, M_{α_s} triggers a new task $\texttt{verify}_\alpha$ with deadline t_G. Intuitively, this means the service characteristic $\alpha(s,\tau,t_c)$ is verified to be within the expected thresholds after at most t_G time.

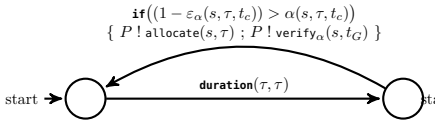

Fig. 1a: M_{α_s} task automata for $\alpha(s,\tau,t_c)$

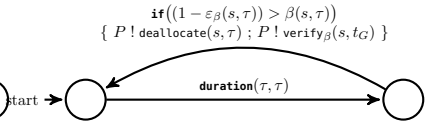

Fig. 1b: M_{β_s} task automata for $\beta(s,\tau)$

We use a separate task automaton for each service characteristic to verify the SLA of the service after t_G time. Respectively, M_{V}^{α} and M_{V}^{β} execute tasks $\texttt{verify}_\alpha$ and $\texttt{verify}_\beta$ (Figures Fig. 2a and Fig. 2b). M_{V}^{α} uses **`await`** to ensure the condition of the SLA. In addition, the task is controlled by the scheduler using a deadline that is specified as t_G in the generated task $\texttt{verify}_\alpha(s,t_G)$ in M_{α_s}. If t_G passes before the guard statement in **`await`** statement holds, it leads to a *missed deadline*.

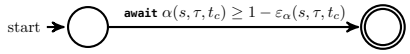

Fig. 2a: M_{V}^{α} to execute $\texttt{verify}_\alpha$

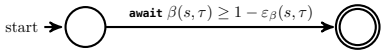

Fig. 2b: M_{V}^{β} to execute $\texttt{verify}_\beta$

Both M_{α_s} and M_{β_s} are specific to one particular service s. A generalized automaton for all services is obtained as their parallel composition:$M_\alpha = (\|_s\ M_{\alpha_s})$ and $M_\beta = (\|_s\ M_{\beta_s})$. The tasks generated by M_α and M_β (triggered by the calls to `allocate` and `deallocate`) are executed by the task automata for platform M_P.

We model monitoring platform P by a task automata M_P. The task types are $\{\texttt{A}(\texttt{allocate}), \texttt{D}(\texttt{deallocate})\}$. For task type `A` in M_P, we use $(b,w,d) = (t_i,\tau,\tau)$; i.e. the best-case execution time of a task is the resource initialization time, the worst-case is the length of the monitoring window, and the deadline is the length of the monitoring window. For task type `D` in M_P, we use $(b,w,d) = (0,\tau,\tau)$. We do not fix the scheduling policy Sch. The error state q_{err} in M_P is defined when either a deadline is missed or when the platform fails to provision a resource. Thus the monitoring platform P contains the following ingredients:

$$M_P = \langle M_{\mathsf{A}} \parallel M_{\mathsf{D}} \parallel M_{\mathsf{V}}^{\alpha} \parallel M_{\mathsf{V}}^{\beta}, \mathsf{Sch}, \tau \rangle$$

We define M_{A_s} as the timed automata to execute the tasks of type `allocate` in M_P. We use the model semantics presented in [13] p. 92 to design M_{A_s}. The resulting automata is presented in Figure 3.

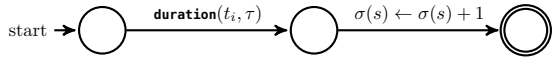

Fig. 3: M_{A_s}: Timed Automaton to execute task type `allocate` in M_P

Then, we define M_{A} in M_P as: $M_{\mathsf{A}} = \|_s\ M_{\mathsf{A}_s}$; i.e. the composition of all timed automata to execute a task `allocate` for some service s. Similarly, we design M_{D_s}

to execute task type `deallocate` in Figure 4. Therefore, we also have $M_{\mathtt{D}}$ in M_P as: $M_{\mathtt{D}} = \|_s M_{\mathtt{D}_s}$.

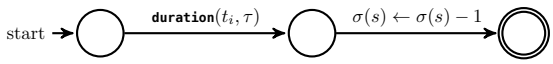

Fig. 4: $M_{\mathtt{D}_s}$: Timed Automaton to execute task type `deallocate` in M_P

For a particular service s, its automaton M_{α_s} regularly measures the service characteristics and calculates $\alpha(s, \tau, t_c)$. When s is under-capacity, M_{α_s} requests to `allocate` a new resource for s through monitoring platform P. This generates a new task in M_P that is executed by $M_{\mathtt{A}_s}$. When the task completes, the state of the service $\sigma(s)$ is updated; strictly increased. Thus, in isolation, the combination of M_{α_s} and $M_{\mathtt{A}_s}$ increase the value of service availability $\alpha(s, \tau, t_c)$ for service s over time. Similarly, in isolation, the combination of M_{β_s} and $M_{\mathtt{D}_s}$ increase the value of service budget compliance $\beta(s, \tau)$ for service s over time. Because in the latter, `deallocate` is used to decrease the cost of the service and as such increases $\beta(s, \tau)$.

In reality, resources might fail in the environment. The failure of a resource is not and cannot be controlled by the monitoring platform P. However, the failure of a resource affects the state of a service and its characteristics. Thus, we model the environment, including failures, as an additional timed automata, M_E. In M_E, in every monitoring window, there is a probability that some resources fail. For example, we present a particular instance of M_E in Figure 5. In this environment, in every monitoring, an unspecified constant (c) number of resources fail.

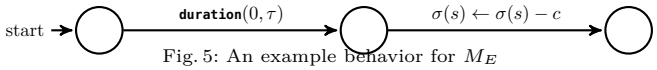

Fig. 5: An example behavior for M_E

We define system automata [13] (p. 33, Definition 3.2.7) for each service characteristic; $\mathcal{S}_\alpha$ for $\alpha(s, \tau, t_c)$ and $\mathcal{S}_\beta$ for $\beta(s, \tau)$:

$$\mathcal{S}_\alpha = M_\alpha \parallel M_E \parallel M_P \qquad \textit{and} \qquad \mathcal{S}_\beta = M_\beta \parallel M_E \parallel M_P$$

With the above automata that we designed for $\alpha(s, \tau, t_c)$ and $\beta(s, \tau)$, we are now ready to present the main results.

Theorem 1. If the SLA for service s on $\alpha(s, \tau, t_c)$ is violated, either:

- $\mathcal{S}_\alpha$ re-establishes the condition $\alpha(s, \tau, t_c) \geq 1 - \varepsilon_\alpha(s, \tau)$ (thereby satisfying the SLA) within t_G time, or,
- there exists at least one task $\mathtt{verify}_\alpha$ in $M_{\mathtt{V}}^\alpha$ with a missed deadline.

Proof. At any given time in T:

- If $\alpha(s, \tau, t_c) \geq 1 - \varepsilon_\alpha(s, \tau)$, then the SLA for service availability α is satisfied.
- If the above condition does not hold, on every monitoring window τ, M_α generates a new task `allocate` in $M_{\mathtt{A}}$. In addition, a new task $\mathtt{verify}_\alpha$ is generated with a deadline t_G. After a duration of t_G, the **`await`** statement allows $M_{\mathtt{V}}^\alpha$ to complete the task $\mathtt{verify}_\alpha$ only if the condition $\alpha(s, \tau, t_c) \geq 1 - \varepsilon_\alpha(s, \tau)$ holds. If this is not the case, since t_G has passed, the scheduler generates a missed deadline (moving to its error state).

□

Theorem 2. If the SLA for service s on $\beta(s,\tau)$ is violated, either:

- $\mathcal{S}_\beta$ re-establishes the condition $\beta(s,\tau) \geq 1 - \varepsilon_\beta(s,\tau)$ (thereby satisfying the SLA) within t_G time, or,
- there exists at least one task $\texttt{verify}_\beta$ in M_{V}^{β} with a missed deadline.

Proof. Similar to the proof of Theorem 1. □

In practice, the guarantee of $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ in isolation to eventually evolve the system to satisfy the SLA is not enough. In reality, a service provider tries ensure both simultaneously to reduce their cost of service delivery while ensuring the delivered service is of the expectations agreed upon with the customer. However, these goals conflict. When $\alpha(s,\tau,t_c)$ increases because of adding a new resource, it means that service s costs more, hence $\beta(s,\tau)$ decreases. The same applies in the other direction: increasing $\beta(s,\tau)$ negatively affects $\alpha(s,\tau,t_c)$.

To capture the combined behavior of service availability and budget compliance, we compose them. We define *service sustainability* $\gamma(s,\tau)$ as the composition of $\alpha(s,\tau,t_c)$ and $\beta(s,\tau)$. We present the composition by system automata $\mathcal{S}_\gamma$ as:

$$\mathcal{S}_\gamma = \mathcal{S}_\alpha \parallel \mathcal{S}_\beta$$

Authors in [9] define that a task automata is *schedulable* if there exists no task on the queue that misses its deadline. The next theorem presents the relationship between schedulability analysis of service sustainability and satisfying its SLA.

Theorem 3. If $\mathcal{S}_\gamma$ is *schedulable* given input parameters (τ, t_i, t_G), then the SLA for both service characteristics $\alpha(s,\tau,t_c)$ and $\beta(s,\tau)$ is satisfied within t_G time after a violation.

Proof. When a violation of the SLA occurs in $\mathcal{S}_\gamma$, either $\mathcal{S}_\alpha$ or $\mathcal{S}_\beta$ (or both) start to evolve the service based on Theorems 1 and 2. Therefore, there exists at least one task of $\texttt{verify}_\alpha$ or $\texttt{verify}_\beta$ with a deadline t_G. Hence, if $\mathcal{S}_\gamma$ is schedulable, then neither $\texttt{verify}_\alpha$ nor $\texttt{verify}_\beta$ miss their deadline. Thus, both $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ are schedulable. This means that both $\texttt{verify}_\alpha$ and $\texttt{verify}_\beta$ complete successfully. Therefore, the SLA of the service is guaranteed within t_G after a violation in $\mathcal{S}_\gamma$. □

Using the algorithm presented in Chapter 6 [13], we translate the above task automata into traditional timed automata. This allows to leverage well-established model checking techniques such as UPPAAL [2] to determine the schedulability of $\mathcal{S}_\gamma$. Moreover, the results of the schedulability analysis serves as a method to optimize the input parameters of the monitoring model including τ and t_G.

6 Evaluation of the monitoring model

In this section, we evaluate the implementation of the monitoring model.

We set up an environment to evaluate how the monitoring evolves a service according to its SLA. In the environment, a single instance of monitoring platform is present to provide new resources as necessary. Every resource hosts only one service. We define two customers in the environment. For both customers, we deploy the same service, Fredhopper Query API. For every resource that hosts a service, we set up a monitor that measures QPS and reports it to the platform. Both customers run with the same SLA: the QPS expectation is $E(s, \tau, t_c) = 10$ and $\varepsilon_\alpha(s, \tau, t_c) = 0.1$. We launch every customer service with only one resource. Monitors observe the customer service and calculate the service availability of every customer service $\alpha(s, \tau, t_c)$.

We run the environment setup for different monitoring windows $\tau \in \{1, 5, 10\}$ (seconds). We fix the initialization time of a resource to $t_i = 2.5$ seconds. We set $t_G = 300$ seconds; i.e. we verify the service after this time and evaluate if the service is guaranteed based on its SLA.

Figure 6 plots the service availability $\alpha(s, \tau, t_c)$ over time with the different monitoring windows. The following summarizes the behavior:

- As the monitoring window τ increases, the system converges with a slower pace towards the expected $\alpha(s, \tau, t_c)$.
- When the monitoring window is chosen such that $\tau < t_i$, the evolution of the system becomes *non-deterministic*.
- The setting $\tau < t_i$ causes a missed deadline in $\texttt{verify}_\alpha$ because after a duration of t_G the service availability has not yet reached the expected value.

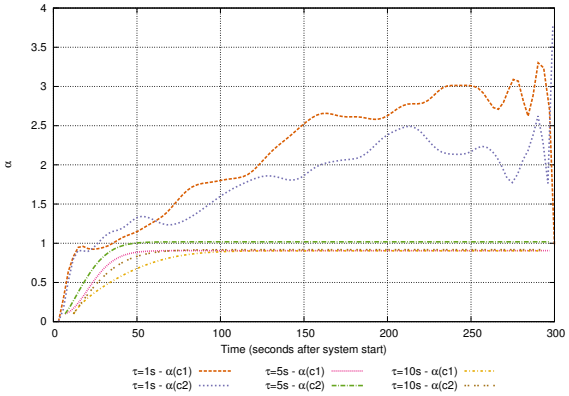

Fig. 6: Evolving $\alpha(s, \tau, t_c)$ with different τ

Every monitoring measurement is performed in a monitoring window τ. Monitoring measurements are aggregated and calculated in every window and form the basis of reactions necessary to evolve the service to meet their SLA. Thus, selection of an appropriate monitoring window length τ is crucial, as we also discussed how schedulability analysis can be used to optimize it. The authors in [11] present that for the same setup and deployment of services, measurements using different monitoring windows yield to very different understanding of service properties such as service availability. Therefore, it is essential to choose the value of τ such that monitoring measurements do not lead to *unrealistic* understanding and inappropriate reactions.

If $\tau < t_i$, Theorem 1 does not hold because every task `allocate` in M_{A} misses its deadline. Thus, it is essential that $\tau \geq t_i$. Analogously, choosing monitoring window as $\tau \gg 2 \times t_i$ also has a counter-productive effect on the service deployments. In a real setting, different services may use different types of resources. In such a setting, the monitoring window should be chosen as the largest t_i of any resource type that is available in the platform: $\tau \geq \mathsf{max}(t_i)\ \forall r \in P$.

7 Future work

We continue to generalize the notion of the distributed service characteristics and investigate how the composition of an arbitrary number of such properties can be formalized and reasoned about. In the context of the ENVISAGE project, industry partners define their service characteristics in this framework and monitor the service evolution. Moreover, the work will be extended to generate parts of the monitoring platform based on an input of different SLA formalizations such as SLA$\star$ [17]. Currently, we are integrating our automated monitoring infrastructure into the in-production SDL Fredhopper cloud services (cf. Section 3).

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
2. G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.
3. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
4. K. Bratanis, D. Dranidis, and A. J. H. Simons. Towards Run-Time Monitoring of Web Services Conformance to Business-Level Agreements. volume 6303, pages 203–206. Springer, 2010.
5. R. Bubel, A. Flores-Montoya, and R. Hähnle. Analysis of executable software models. In *SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, pages 1–25, 2014.
6. Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. SLA decomposition: Translating service level objectives to system level thresholds. In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, pages 3–3. IEEE, 2007.
7. A. Coles, A. J. Coles, A. Clark, and S. Gilmore. Cost-sensitive concurrent planning under duration uncertainty for service-level agreements. In *ICAPS*, 2011.
8. M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour. Establishing and monitoring SLAs in complex service based systems. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 783–790. IEEE, 2009.
9. E. Fersman, P. Krcal, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
10. S. Gilmore, L. Gönczy, N. Koch, P. Mayer, M. Tribastone, and D. Varró. Non-functional properties in the model-driven development of service-oriented systems. *Software & Systems Modeling*, 10(3):287–311, 2011.

11. G. Hogben and A. Pannetrat. Mutant Apples: A Critical Examination of Cloud SLA Availability Definitions. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 379–386. IEEE, 2013.
12. Inzinger, Christian and Hummer, Waldemar and Satzger, Benjamin and Leitner, Philipp and Dustdar, Shahram. Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems. *Software – Practice and Experience*, 2014.
13. M. M. Jaghoori. *Time at your service: schedulability analysis of real-time and distributed services*. PhD thesis, Leiden University, 2010.
14. M. M. Jaghoori. Composing real-time concurrent objects refinement, compatibility and schedulability. In *Fundamentals of Software Engineering*, pages 96–111. Springer Berlin Heidelberg, 2012.
15. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, pages 142–164. Springer, 2012.
16. E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *Formal Methods and Software Engineering*, pages 71–86. Springer, 2012.
17. K. T. Kearney, F. Torelli, and C. Kotsokalis. SLA⋆: An abstract syntax for Service Level Agreements. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 217–224. IEEE, 2010.
18. A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.
19. K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 14–24. IEEE, 1997.
20. X. Logean, F. Dietrich, H. Karamyan, and S. Koppenhöfer. Run-time monitoring of distributed applications. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 459–474, 1998.
21. K. Mahbub, G. Spanoudakis, and T. Tsigkritis. Translation of SLAs into monitoring specifications. In *Service Level Agreements for Cloud Computing*, pages 79–101. Springer, 2011.
22. B. Nobakht, F. S. de Boer, and M. M. Jaghoori. The future of a missed deadline. In *Coordination Models and Languages*, pages 181–195. Springer, 2013.
23. B. Nobakht, F. S. de Boer, M. M. Jaghoori, and R. Schlatte. Programming and deployment of active objects with application-level scheduling. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1883–1888. ACM, 2012.
24. F. Raimondi, J. Skene, and W. Emmerich. Efficient online monitoring of web-service SLAs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 170–180. ACM, 2008.
25. P. Y. H. Wong, R. Bubel, F. S. de Boer, M. Gómez-Zamalloa, S. de Gouw, R. Hähnle, K. Meinke, and M. A. Sindhu. Testing abstract behavioral specifications. *STTT*, 17(1):107–119, 2015.
26. J. Woodcock, A. Cavalcanti, J. Fitzgerald, S. Foster, and P. G. Larsen. Contracts in CML. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 54–73. Springer, 2014.