City Research Online

City, University of London Institutional Repository

Citation: Lorenzoli, D. & Spanoudakis, G. (2010). EVEREST+: Run-time SLA violations prediction. In: Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing. (pp. 13-18). New York: ACM. ISBN 978-1-4503-0452-8 doi: 10.1145/1890912.1890915

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/5167/

Link to published version: https://doi.org/10.1145/1890912.1890915

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

EVEREST+: Run-Time SLA Violations Prediction

Davide Lorenzoli
School if Informatics, City University
Northampton square, London,
EC1V OHB, UK
+44 20 7040 3722
Davide.Lorenzoli.1@soi.city.ac.uk

George Spanoudakis
School if Informatics, City University
Northampton square, London,
EC1V OHB, UK
+44 20 7040 8413
G.Spanoudakis@soi.city.ac.uk

ABSTRACT

Monitoring the preservation of QoS properties during the operation of service-based systems at run-time is an important verification measure for checking if the current service usage is compliant with agreed SLAs. Monitoring, however, does not always provide sufficient scope for taking control actions against violations as it only detects violations after they occur.

In this paper we describe a model-based prediction framework, EVEREST+, for both QoS predictors development and execution. EVEREST+ was designed to provide a framework for developing in an easy and fast way QoS predictors only focusing on their prediction algorithms implementation without the need for caring about how to collect or retrieve historical data or how to infer models out of collected data. It also provides a run-time environment for executing QoS predictors and storing their predictions.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications, C.4 [**Performance of Systems**]: Measurement techniques

General Terms

QoS Measurement, Algorithms

Keywords

Run-time QoS Prediction, Prediction Framework

1. INTRODUCTION

Monitoring the preservation of quality of service (QoS) properties during the operation of service-based systems at run-time is an important verification measure for checking if the current usage and behaviour of the services deployed by the system is compliant to the Service Level Agreements (SLAs) set for these services. The monitoring of QoS properties specified in an SLA has received significant attention in the literature and several approaches and monitoring systems have been developed to support it [12][11][1][7]. Most of these approaches and systems, however, can only support the detection of a QoS property violation once it has occurred. Thus, they do not provide sufficient support for taking control actions that could prevent violations or warning the relevant parties that violations are likely to occur in the future.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MW4SOC'10, November, 2010, Bangalore, India.
Copyright 2010 ACM 978-1-4503-0452-8/10/11…$10.00.

The prediction of violations of QoS properties of software systems has been the subject of research outside the area of SLA monitoring. This work has focused on prediction related to different types of properties including, for example, software systems failures [13], system dependability [4], security [15], and parameters of system infrastructures such as server workloads, CPU loads, and network throughput [3][6]. Related techniques have been based on wide spectrum of prediction algorithms ranging from time series analysis [2] to mean-value prediction techniques [3] or belief-based reasoning [8].

Three limitations of existing techniques that make them falling short of providing adequate support for run-time prediction of SLA violations are:

- They tend to focus on system infrastructure properties (e.g., network and server properties) rather than service level application based properties (e.g., service throughput, mean time to failure).
- They tend to focus on the prediction of specific types of properties without providing a more generic framework for building predictors that can cover a wide or even the whole spectrum of service properties that can be part of an SLA
- They are not integrated with environments for monitoring SLAs for service-based systems

The latter limitation is important as the lack of relevant integration prevents the development of support for proactive management of service-based systems and SLAs including, for example, proactive service discovery by service clients in cases where the QoS properties in SLAs of the services used by a system are forecasted to fail, proactive negotiation of new SLAs with existing customers in cases where providers detect that their SLAs are due to be violated, or proactive provision of further service capacity in the same case.

In this paper, we introduce a new framework, which supports the prediction of potential violations of QoS properties in SLAs. This framework has been developed as part of a generic monitoring framework for checking SLAs at run-time, called EVEREST [15]. Our prediction framework provides an integrated architecture for SLA monitoring and prediction that supports the latter activity through the deployment of a built-in set of model-based predictors (including for example a generic predictor for constraints regarding mean values of QoS properties). Our framework receives specifications of the QoS properties, which need to be monitored and predicted, expressed

in a general high level SLA specification language developed as part of FP7 EU project SLA@SOI, and a specification of the prediction parameters and model that should be used for generating predictions of violations of these properties. Based on the input QoS and prediction model specifications, the framework generates automatically operational monitoring specifications expressed in Event Calculus [5] to enable not only the classic (i.e., non predictive) monitoring of the required QoS properties but also the acquisition and recording of run-time information that will be required for predicting potential violations of the given QoS properties. Following the generation of operational monitoring specifications, the framework activates these specifications to perform run-time monitoring, and uses the data gathered during monitoring to compute automatically the prediction model identified for the relevant QoS property and generate the required forecasts.

The architectural design of the framework enables its extension by new QoS predictors that may be required for specific types of properties, and provides a language that enables the users of the framework to declare how new predictors can fit in the generic prediction approach advocated by it and the type of predictor(s) that should be used for a given property if more than one predictors are available for the property by specifying appropriate configuration specifications.

Our integrated monitoring-prediction framework addresses the lack of integration of monitoring and prediction systems and the tendency to provide specific prediction algorithms instead of a generic framework for building predictors. Our framework provides a coherent approach to data collection and analysis both for monitoring and prediction purpose. Moreover, it supports the developing of monitoring and prediction uses-designed algorithm that can be used for extending the framework abilities. Also, it defines a single point of access for configuring the integrated framework.

In the rest of this paper, we compare our approach with existing work (Section 2), present the key concepts underpinning our framework (Section 3), present our framework key points and its architecture (Section 4 and 5), present the specifications used by our framework (Section 6). We also present an initial experimental evaluation of the framework (Section 7) and outline directions for future work (Section 8).

2. RELATED WORK

Several different approaches to QoS monitoring have been proposed in literature (e.g. [12][11][1]) and recommendations about QoS metrics measurement for web services have been described in [14].

Michlmayr et al. [11] present an event-based QoS monitoring and SLA violation detection framework. They developed client and service side monitoring and integrated them in the VRESCo [10], a run-time environment for service-oriented computing. At the moment VRESCo supports a limited list of QoS properties. Our approach, like [11], can monitor both client and server side, but it doesn't have a fixed list of supported QoS properties. Indeed, users can specify new properties to be monitored as EC-Assertions.

Sahai et al. [12] present an automated and distributed SLA monitoring engine. They use both client and service side collected information. There is not a fixed set of monitorable properties, but to add a new property a new SLA evaluator component must be developed and deployed into the framework. Our approach does not require any new components to be developed and deployed to monitor a new QoS property. It is only required to write a new AC-Assertion specification.

De Luc et al. [1] present a middleware component for monitoring services and delivery timely and coherent monitoring data to business processing using them in run-time decision making settings. This work focuses on data collection and how to efficiently deliver it to other components. Our approach also detects when monitored data violates QoS requirements.

Leitner et al. [7] present an approach for predicting SLA violations at run-time. The Prediction approach requires the definition at design-time of checkpoints for each BPEL subjected to prediction. Moreover, it does not support the prediction of aggregate properties. Our approach does not require defining any checkpoints; in fact a prediction can be requested at any time. It can also predict aggregate properties.

All the described approaches focus on monitoring or prediction only. On contrary, our approach integrates monitoring and prediction with in a same coherent framework. Moreover, we provide a more generic framework for building predictors that can cover a wide or even the whole spectrum of service properties that can be part of an SLA.

3. BACKGROUND: The EVEREST Monitoring Framework

EVEREST is a generic monitoring engine for checking violations of software system properties expressed in an Event Calculus (EC) based language called *EC-Assertion* at run-time. EVEREST has been used for monitoring different types of properties of software systems including functional security and dependability properties [15]. It has also been applied for monitoring SLA guarantee terms for service-based systems [9]. Whilst a full description of EVEREST is beyond the scope of this paper, in this section we provide an overview of the language that it uses to express monitorable SLA guarantee terms to enable the reader understand how the prediction specifications used by the prediction framework relate to specifications of these terms.

More specifically, the SLA terms that can be checked by EVEREST are expressed as EC-Assertion monitoring rules and/or assumptions of the form: `body ⇒ head`. The semantics of a monitoring rule of this form is that when the `body` of the rule evaluates to `True`, its `head` must also evaluate to `True`. The semantics of assumption of this form is that when the `body` of the rule evaluates to `True`, its `head` is deduced by EVEREST. The body and head of EC-Assertion rules and assumptions are defined in terms of standard EC predicates:

(a) *Happens(e,t,R(lb,ub))* – This predicate denotes that an instantaneous event *e* occurs at some time *t* with in the time range *R(lb,ub)*, where *lb*≤*ub* are *R* lower and upper bounds.
(b) *HoldsAt(f,t)* – This predicate denotes that a state (a.k.a. fluent) f holds at time *t*
(c) *Initiates(e,f,t)* and *Terminates(e,f,t)* – These predicates denote the initiation and the termination of a fluent *f* by an event e at a time *t* respectively, and
(d) *Initially(f)* which denotes that a fluent holds at the start of the operation of a system.

An example of an SLA term specified in EC-Assertion is shown in Table 1. The formulas in the table check whether the mean time to repair of a service (MTTR) *_Srv*, i.e., is the mean length of the periods of time over which a service does not respond to operation calls and is therefore unavailable, is always below a given threshold *K*, i.e., *MTTR≤K*.

Rule R1:

Happens(e(_id1, _Snd, _Srv, Call(_O), _Srv), t_1, [t_1,t_1]) ∧
Happens(e(_id2, _Srv, _Snd, Response(_O), _Srv), t_2, [t_1,t_1+d]) ∧
∃ _PN, _STime, _MTTR: ***HoldsAt***(Unavailable(_PN, _Srv, _STime), t_1)) ∧
HoldsAt(MTTR(_Srv, _PN, _MTTR), t_1)) ⇒
_MTTR < K

Assumption R1.A1:

Happens(e(_id1, _Snd, _Srv, Call(_O), _Srv), t_1, [t_1,t_1]) ∧
¬***Happens***(e(_id2, _Srv, _Snd, Response(_O), _Srv), t_2, [t_1,t_1+d]) ∧
¬∃ _PeriodNum, _STime,: ***HoldsAt***(Unavailable(_PeriodNum, _Srv, _STime), t_1)) ∧
∃ _PN, _MTTR: ***HoldsAt***(MTTR(_Srv, _PN, _MTTR), t_1)) ⇒
Initiates(e(_id1, _Snd, _Srv, Call(_O), _Srv), Unavailable(_PN+1, _Srv, t_1), t_1) ∧
Terminates(e(_id1, _Snd, _Srv, Call(_O), _Srv), MTTR(_Srv, _PN, _MTTR), t_1) ∧
Initiates(e(_id1, _Snd, _Srv, Call(_O), _Srv), MTTR(_Srv, _PN+1, _MTTR), t_1

Assumption R1.A2:

Happens(e(_id1, _Snd, _Srv, Call(_O), _Srv), t_1, [t_1,t_1]) ∧
Happens(e(_id2, _Srv, _Snd, Response(_O), _Srv), t_2, [t_1,t_1+d]) ∧
∃ _PeriodNum, _STime: ***HoldsAt***(Unavailable(_PeriodNum, _Srv, _STime), t_1) ⇒
Terminates(e(_id1, _Snd, _Srv, Call(_O), _Srv), Unavailable(_PeriodNum, _Srv, _STime), t_1+1)

Assumption R1.A3:

Happens(e(_id1, _Snd, _Srv, Call(_O), _Srv), t_1, [t_1,t_1]) ∧
Happens(e(_id2, _Srv, _Snd, Response(_O), _Srv), t2, [t_1,t_1+d]) ∧
∃ _PeriodNum, _STime,: ***HoldsAt***(Unavailable(_PeriodNum, _Srv, _STime), t_1)) ∧
∃ _PN, _MTTR: ***HoldsAt***(MTTR(_Srv, _PN, _MTTR), t_2)) ⇒
Terminates(e(_id1, _Snd, _Srv, Call(_O), _Srv), MTTR(_Srv, _PN, _MTTR), t_2) ∧
Initiates(e(_id1, _Snd, _Srv, Call(_O), _Srv), MTTR(_Srv, _PN, (_MTTR*(_PN−1) +(t1 − STime))/_PN), t_2)

Table 1. EC formula for monitoring MTTR

More specifically, rule R1 in Table 1 checks for MTTR violations when a call of an operation of the service *_Srv* is served after a period of unavailability. The first two conditions in the rule check whether a served operation call has occurred. The latter two conditions check whether this happens at a time when the service has been unavailable.

The first assumption in Table 1 (R1.A1) initiates the fluent *Unavailable(_PN+1, _Srv, t_1)* to represent a period of service unavailability. This fluent is initiated when service call occurs (i.e., the call represented by the event *_id1*), no response to this call is produced within *d* time units, and at the time of the occurrence of the call the service is not already unavailable (i.e., no fluent of the form *Unavailable(_PeriodNum, _Srv, _STime)* already holds). The number of the new unavailability period is determined by increasing the variable *_PN* whose current value is extracted from the fluent *MTTR(_Srv, _PN, _MTTR)* which keeps a record of the number of the past periods of unavailability of the service (i.e., *_PN*) and the mean length of time during which the service remained unavailable in each of these periods (i.e., the value of the variable *_MTTR*). As a new period of unavailability is initiated for the service, the assumption also re-initiates the fluent *MTTR(_Srv, _PN, _MTTR)* in order to increase the number of unavailable periods *_PN*.

The second assumption (R1.A2) terminates the fluent that represents a currently active period of service unavailability (i.e., the fluent *Unavailable(_PeriodNum, _Srv, _STime)*) when a served service call occurs (i.e., the call represented by the event *_id1*) and at the time of the occurrence of this call the service is not unavailable (i.e., a fluent of the form *Unavailable(_PeriodNum, _Srv, _STime)* holds).

The third assumption (R1.A3) updates the fluent that represents the mean length of consecutive periods of service unavailability (i.e., the value stored in the variable *_MTTR* of the fluent *MTTR(_Srv, _PN, _MTTR)*) when a served service call occurs (i.e., the call represented by the event *_id1*) and at the time of the occurrence of this call the service is not unavailable (i.e., a fluent of the form *Unavailable(_PeriodNum, _Srv, _STime)* holds). The new mean value is computed as the mean of the mean of the previous *_PN-1* observations that is stored as the current value of *_MTTR* and the new period of unavailability (t_1−*STime*).

4. OVERVIEW OF OUR PREDICTION APPROACH

At a high level, our framework assumes that a prediction problem can be formulated as follows: Given a request for predicting whether a QoS property will be satisfied at some future time point t_e that is received at a time point t_c, prediction is the computation of the probability that the QoS property will be satisfied at t_e. The computation of this probability will, in general, be based on estimating the probability of different values for specific variables that underpin the QoS property and/or the QoS property itself. These probabilities can be computed by fitting probability distribution functions to historical data of these variables.

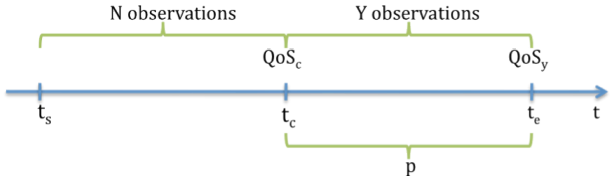

Figure 1. Prediction framework common definitions

Figure 1 illustrates this general formulation. More specifically, t_c in the figure is the time point at which a prediction is requested, t_e is the time point in the future that the prediction is required for, p is the prediction window (i.e., $p=t_e-t_c$), N is the number of QoS observations between t_s and t_c, Y is the number of future QoS observations between t_c and t_s, QoS_c is the value of the observed QoS at the time point t_c, and QoS_y is the value of the predicted QoS at the time point t_e.

The design of EVEREST+ enables the realization of different prediction models for QoS factors which are based on a common underlying principle: the estimation of probabilities of specific values (or ranges of values) for different variables that underpin the violation or otherwise of a QoS term, and the use of these probabilities in deriving the probability of the violation of the term in a given period.

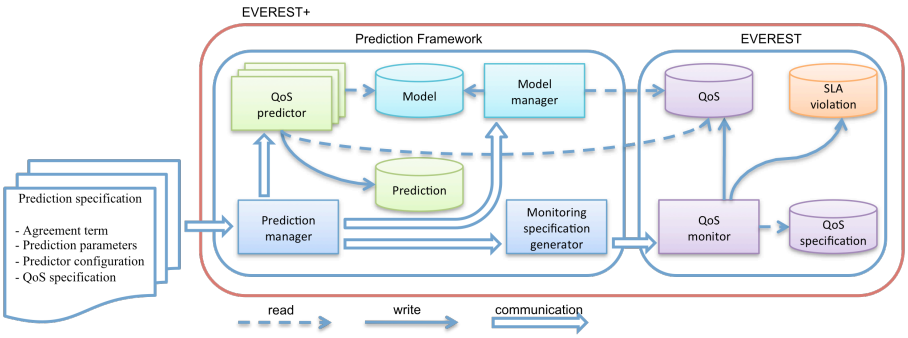

Figure 2. EVEREST+ components

In the case of the MTTR QoS term for a service, for instance, a prediction model that we have developed for checking whether MTTR $\leq K$ at a future time point t_e is based on the estimating the probability distribution functions of two variables: (a) the MTTR of a service itself and (b) the time between two successive non served calls of service operations, referred to as *time-to-failure* or TTF in the following.

More specifically, assuming that N is the number of TTR values recorded until t_c and y is the (yet unknown) number of TTR values that will be recorded during the period p (or, equivalently, the number of cases where a service became available again following a period of unavailability), to violate MTTR at time t_e the following inequality must be false.

$$(N*MTTR_c + y*MTTR_y)/(N+y) \leq K \qquad (1)$$

From (1), we can deduce that for the MTTR term to be violated it must be true that:

$$MTTRy \geq [K*(N+y) - N*MTTR_c]/y = MTTR_{crit} \qquad (2)$$

Given (2), there are two factors to take into account to predict MTTR:

- $Pr(y)$, that is the probability to observe y failures in the prediction time period p.
- $Pr(MTTR_y > MTTR_{crit})$, that is the probability of having $MTTR_y > MTTR_{crit}$.

The probability to violate the QoS term constraint $MTTR \leq K$ at the end of p time units is approximated by formula (3).

$$Pr(\bigwedge_{y=1}^{M} E_y) = \begin{cases} 1 - \sum_{y=1}^{M}[Pr(y) * Pr(MTTR_{crit} > MTTR_y)], & MTTR_c > K \\ \sum_{y=1}^{M}[Pr(y) * Pr(MTTR_{crit} \leq MTTR_y)], & MTTR_c \leq K \end{cases} \qquad (3)$$

$Pr(y)$ and $Pr(MTTR_y > MTTR_{crit})$ are computed by using density and cumulative probability functions. These probability functions are inferred by analyzing MTTR and TTF historical values collected by EVEREST, fitting different known probability functions to them, and selecting the function that has the best fit with the data. More specifically, during monitoring, EVEREST stores all the fluents defined in the EC formulas of Table 1 that required for monitoring the MTTR term including, for instance, run-time values of the variable _MTTR of the fluent *MTTR(_Srv, _PN, _MTTR)* from which the density and probability function of MTTR can be inferred. The prediction components of EVEREST+ use subsequently these historical values to identify the probability density and probability functions that have the best fit with the stored MTTR values and use these functions to estimate $Pr(MTTR_y > MTTR_{crit})$.

5. ARCHITECTURE OF EVEREST+

EVEREST+ has been designed with the general goal of providing a framework for developing QoS predictors in an easy and fast way by focusing only on prediction algorithm implementations without the need for caring about how to collect or retrieve historical data or how to infer statistical models out of the collected data. The architecture of EVEREST+, shown in Figure 2, includes two main components: (1) the EVEREST monitoring framework, and (2) the new prediction framework.

As discussed earlier, the EVEREST monitoring framework checks services at run-time to determine whether they behave according to SLAs QoS terms set for them. EVEREST checks QoS terms based on events intercepted from services by internal or external event captors. Whilst monitoring QoS terms, EVEREST stores QoS related information, including the computed QoS term values, the instances of QoS term violations and satisfactions, and the values of any other state variables (aka fluents) that have been taken into account in checking QoS terms (see Section 3). This information is available through an API that allows its retrieval from the internal EVEREST monitoring database (see *QoS* data store in Figure 2).

The *prediction framework* (PF) fits statistical distribution functions to different types of historical QoS data generated by EVEREST, selects the distribution functions that have the best fit with the data, and makes these functions and the "raw" QoS data available to different QoS predictors that are deployed in EVEREST+ as plug-ins. The prediction framework has three main components, namely the *model manager*, *QoS predictor*, and *prediction manager*. These components are described in the following.

5.1 Prediction Manager

The Prediction Manager component coordinates and supervises prediction tasks by managing prediction specifications, triggering components, and reporting prediction results. The operation of Prediction Manager is driven by prediction specifications. As shown in Figure 3, these specifications determine the QoS term that is to be monitored and forecasted *(qos_specification)* and the *prediction_parameters* whose statistical models will need to be determined in order to make predictions for this QoS term. They also specify *predictor configuration* parameters (e.g. type of predictor, prediction period). The prediction manager extracts and sends the QoS specification to EVEREST; selects and deploys the appropriate QoS predictor and sends the necessary configuration parameters to it, and sends the prediction configuration to the model

manager. Once all the above components are configured, QoS predictors begin to produce predictions and store them into the *prediction* database. It is prediction manager responsibility to fetch and report them when needed.

5.2 Model Manager

The Model Manager is in charge of coordinating a pool of *Model Calculator* components used for inferring models out of historical QoS data collected by EVEREST. The model manager also makes inferred models available to QoS predictors.

A Model Calculator component is a component implementing model-specific algorithm. EVEREST+ has a set of already implemented model calculators that can infer statistical distribution functions from historical QoS data, e.g., probability distribution (aka density) functions (PDF) and cumulative probability distribution functions (CDF). Statistical models are computed (and updated) at run-time, and stored in the *model* database. EVEREST+ also provides mechanisms for extending its default set model calculators.

The model manager also implements model-updating policies that can be specified in EVEREST+ configuration files. They can be time, data, or time/data driven. Time driven policies trigger model updating after a certain time period has passed from the last computed version of the model. Data driven policies trigger model updating after a certain amount of data has been computed after the last updating of the model. Time/data driven policies trigger model updating by considering both time elapsed and the data received after the last model updating. The triggering policy should be chosen with respect to the specific domain EVEREST+ is operating in. Variable data might suggest a data driven policy, whilst for homogeneous data a time driven policy would suit better.

5.3 QoS Predictors

QoS Predictor components are the components in EVEREST+ that implement specific QoS prediction algorithms. All QoS predictors extend a basic predictor component. The base predictor component provides the common functionalities required for accessing EVEREST historical QoS data and the statistical models inferred by the model calculator components. It also provides functionality for storing QoS prediction results. In this way, developers of specific QoS predictors can focus only on prediction algorithm implementation without implementing the above common core functionalities.

To render it deployable in the EVEREST+ framework, a QoS predictor must also provide a *prediction feature* list and *dependencies*. The list of prediction features indicates the QoS terms that the particular predictor can generate forecasts for. The dependency list indicates which data are required by a QoS predictor to make predictions. For instance, a predictor for MTTR based on the approach outlined in Section 4 (*MTTR_PRE* predictor) has one prediction feature only, i.e., the QoS term MTTR, and requires MTTR and TTF historical data to make its predictions.

5.4 Monitoring Specification Generator

The Monitoring Specification Generator component receives a prediction specification, translates it into a monitoring specification expressed in EC-Assertion, and forwards it to EVEREST. It also checks whether all needed resources required by QoS predictors are available.

6. PREDICTION SPECIFICATIONS

A prediction specification is a user-defined document that tells the prediction framework what to predict. To express prediction specifications we use the high level SLA specification language developed by the FP7 EU project SLA@SOI and extended it to support the specification of prediction requirements.

Besides prediction targets, a prediction specification carries information about how to configure QoS predictors and which information is needed by QoS predictors to operate. Moreover, it can contain EC rules (QoS specifications) to be used by monitoring framework to monitor new QoS terms.

An example of a prediction specification is given in Figure 3. As shown in the figure, a prediction specification specifies an *agreement_term* element for a service identified by a *service_id*. An *agreement_term*, identified by its unique id, can have one or more *guaranteed_state* sub-elements specifying the QoS term that the prediction is required for (i.e., MTTR in the example). Each guaranteed state has its own unique id too. The triplet (service id, agreement term id, guaranteed state id) is used to retrieve stored prediction results.

```
prediction_specification {
   serice_id = _Srv
   prediction.window = 10min
   Agreement_term {
      id = AG-1
      guaranteed_state {
         id= GS-1
         MTTR < K
      }
   prediction_parameters {
      qos { id = MTTR }
      qos { id = TTF }

   }
   prediction_configuration {
      predictor_id = MTTR_PRED
      configuration_property { history.window = 400 }

   }
   qos_specification {
      ec_formula_id = MTTR
      ec_formula = "The EC-Assertion formula" }
}
```

Figure 3. Example of prediction specification

The prediction specification tells the constraint that holds for the QoS specified in the guaranteed state, whose violation will be the subject of prediction (i.e., *MTTR<K* seconds in our example), and the window of the prediction (i.e., the time period in the future that the prediction should be concerned with). This window is set to 10 minutes in our example, meaning that the prediction required in this instance should be whether the MTTR of *_Srv* will be greater than or equal to *K* seconds within 10 minutes following the prediction request

Note that a prediction specification uses a QoS name that is also used in a QoS specification, and therefore, enables the QoS predictor to retrieve historical QoS data for the term in order to compute the statistical prediction model for it.

6.1 QoS Specification

QoS specifications are EC formulas given to EVEREST to instruct it for monitoring QoS terms, e.g., MTTR, TTF, and availability. An example of QoS specification for monitoring MTTR has been discussed in Section 2. Via QoS specifications it is possible to extend the set of monitorable QoS terms own by EVEREST. If a prediction about a new QoS term is required, and EVEREST doesn't have its QoS specification, it is sufficient to attach to a prediction specification, a QoS specification that describes how to monitor the new QoS term.

6.2 QoS Predictor Configuration

Since EVEREST+ supports user-defined QoS predictors, each predictor might require different kind of configurations. QoS predictor configuration provides a key-value pair based configuration policy.

For instance, in Figure 3, *MTTR_PRED* predictor receives a QoS predictor configuration for configuring the data history size it must use during its computation. QoS predictor configuration sets the available data history size to 400 (*history.window=400*).

7. EXPERIMENTAL RESULTS

The current implementation EVEREST+ is based on Java Platform Standard Edition 5.0 (J2SE 5.0) and uses MySQL 5.1 as DBMS. J2SE 5.0.

Monitoring and prediction can be time critical tasks. In real-time or high performance environments decision must taken within a few seconds or even a few milliseconds time. Therefore, the ability to provide results fast is crucial. EVEREST+ current implementation most consuming activity is model inferring from historical data. The inferring process fits up 43 statistical distributions given a set of data points.

We evaluated EVEREST+ performance with historical data of different sizes, from 50 to 20000 data points. Table 2 shows the input size, the number of inferred models, and the time consumed by the inferring process in the first, second, and third column respectively. As expected, the bigger is the history size, the longer the inferring process takes. However, it exceeds one-second time only for history sizes greater equal to 10000. Our experiments also highlighted how data point set sizes of 1000 and 5000 are sufficient to produce quality statistical models.

Data points	Inferred models	Inferring time (ms)
50	20	~322
100	20	~217
500	20	~217
1000	20	~222
5000	20	~656
10000	18	~1196
20000	18	~1701

Table 2 Model inferring performance

Moreover, the prediction algorithm execution time is of a few milliseconds only. Therefore, latencies of one or two seconds (in the worst case) are still acceptable in not time critical systems.

8. CONCLUSION

This paper presents a model-based prediction framework for detecting potential violations of QoS properties. The proposed approach key properties are generality and extensibility. It is general because it doesn't support a limited set of QoS only and it is extensible because the definition of which data to collect and how to analyse them can be specified using models (QoS specifications) and pluggable components (QoS predictors).

To proofing the validity of our approach we defined a QoS specification for monitoring MTTR QoS term values and we implemented a QoS predictor to predict about MTTR violations. Our experiments show that good performance results about the automatically inferred statistical models.

For future work, we plan to extend the set of available QoS predictors and automatically inferred models to create a robust and flexible support for QoS monitoring and prediction.

9. ACKNOWLEDGEMENT

This research has been supported by the EU Commission under the Framework 7 project SLA@SOI (grant n. 216556).

10. REFERENCES

[1] Duc B. L., P., Châtel Rivierre N., Malenfant J., Collet P., Truck I.: Non-functional data collection for adaptive business processes and decision-making. In Proceedings of the 4th MWSOC. ACM, 2009.

[2] Garg S. Garg, A. V. Moorsel A. V., K. Vaidyanathan K., and Trivedi K.: A methodology for detection and estimation of software aging. ISSRE, 0:283, 1998.

[3] Iyer R. K. Iyer and Rossetti D. J.: Effect of system workload on operating system reliability: A study on ibm 3081. IEEE Trans. Softw. Eng., 1985.

[4] Iyer R. K., Young L. T., and Iyer P. K.: Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data. *IEEE Trans. Comput.* 39, 4, 1990.

[5] Kowalski R. and Sergot M.: A logic-based calculus of events. New Gen. Comput., 4(1):67–95, 1986.

[6] Lee B-D, Schopf J M.: Run-Time Prediction of Parallel Applications on Shared Environments, Cluster Computing, IEEE International Conference on, p. 487, Fifth IEEE International Conference CLUSTER, 2003.

[7] Leitner P., Wetzstein B., Rosenberg F., Michlmayr A., Dustdar S. and Leymann F.: Runtime Prediction of Service Level Agreement Violations for Composite Services. The 3rd Workshop NFPSLAM-SOC, 2009.

[8] Lorenzoli D. and Spanoudakis G.: Detection of security and dependability threats: A belief based reasoning approach. SECURWARE, 0:312–320, 2009.

[9] Mahbub K. Spanoudakis G.: Monitoring WS Agreements: An Event Calculus Based Approach, Test and Analysis of Service Oriented Systems, (eds) L. Baresi, E. diNitto, Springer- Verlang, 2007

[10] Michlmayr A., Rosemberg F., Leitner P., and Dustdar S.: End-to-end support for QoS-aware service selection, invocation and mediation in VRESCo. Technical report, Vienna University of Technology, 2009.

[11] Michlmayr A., Rosenberg F., Leitner P., and Dustdar A.: Comprehensive QoS monitoring of Web services and event-based SLA violation detection. In Proceedings of the 4th MWSOC. ACM, 2009.

[12] Sahai A., Machiraju V., Sayal M., Moorsel A. P., and Casati F.: Automated SLA Monitoring for Web Services. In Proceedings of the 13th IFIP/IEEE international Workshop on Distributed Systems, 2002.

[13] Salfner F., Schieschke M., and Malek M.: Predicting failures of computer systems: a case study for a telecommunication system. IPDPS, 0:415, 2006.

[14] Thio N., Karunasekera S.. Automatic measurement of a Qos metric for Web Services. In proc. of ASWEC, 2005.

[15] Tsigkritis T., Spanoudakis G., and Lorenzoli D.: Diagnosis and Threat Detection Capabilities of the SERENITY Monitoring Framework,. cChapter 14, pages 239–271. Advances in Information Security. Springer US, 2009.