TU

Runtime Prediction of Service Level Agreement Violations for Composite Services

Extended Version of the
NFPSLAM-SOC 2009 Paper

Philipp Leitner, Branimir Wetzstein,
Florian Rosenberg, Anton Michlmayr,
Schahram Dustdar, Frank Leymann

TUV-1841-2010-02   March 22, 2010

SLAs are contractually binding agreements between service providers and consumers, mandating concrete numerical target values which the service needs to achieve. For service providers, it is essential to prevent SLA violations as much as possible to enhance customer satisfaction and avoid penalty payments. Therefore, it is desirable for providers to predict possible violations before they happen, while it is still possible to set counteractive measures. We propose an approach for predicting SLA violations at runtime, which uses measured and estimated facts (instance data of the composition or QoS of used services) as input for a prediction model. The prediction model is based on machine learning regression techniques, and trained using historical process instances. We present the architecture of our approach and a prototype implementation, and validate our ideas based on an illustrative example.

Keywords: Service-oriented Computing, Web Services, SLA, Prediction of SLA Violations

Runtime Prediction of Service Level Agreement Violations for Composite Services

Philipp Leitner[1], Branimir Wetzstein[2], Florian Rosenberg[3], Anton Michlmayr[1],
Schahram Dustdar[1], Frank Leymann[2]

[1] Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040, Vienna, Austria
lastname@infosys.tuwien.ac.at

[2] Institute of Architecture of Application Systems
University of Stuttgart
Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

[3] CSIRO ICT Centre
GPO Box 664
Canberra ACT 2601, Australia
florian.rosenberg@csiro.au

Abstract. SLAs are contractually binding agreements between service providers and consumers, mandating concrete numerical target values which the service needs to achieve. For service providers, it is essential to prevent SLA violations as much as possible to enhance customer satisfaction and avoid penalty payments. Therefore, it is desirable for providers to predict possible violations before they happen, while it is still possible to set counteractive measures. We propose an approach for predicting SLA violations at runtime, which uses measured and estimated facts (instance data of the composition or QoS of used services) as input for a prediction model. The prediction model is based on machine learning regression techniques, and trained using historical process instances. We present the architecture of our approach and a prototype implementation, and validate our ideas based on an illustrative example.

1 Introduction

In service-oriented computing [1], finer-grained basic functionality provided using Web services can be composed to more coarse-grained services. This model is often used by Software-as-as-Service providers to implement value-added applications, which are built upon existing internal and external Web services. Very important for providers and consumers of such services are Service Level Agreements (SLAs), which are legally binding agreements governing the quality that the composite service is expected to provide (Quality of Service, QoS) [2]). SLAs contain Service Level Objectives (SLOs), which are concrete numerical

target values (e.g., "maximum response time is 45 seconds"). For the provider it is essential to not violate these SLOs, since typically violations are coupled with penalty payments. Additionally, violations can negatively impact service consumer satisfaction. Therefore, it is vitally important for the service provider to be aware of SLA violations, in order to react to them accordingly.

Typically, SLA monitoring is done *ex post*, i.e., violated SLOs can only be identified after the violation happened. While this approach is useful in that it alerts the provider to potential quality problems, it clearly cannot directly help preventing them. In that regard an *ex ante* approach is preferable, which allows to predict possible SLA violations before they have actually occurred. The main contribution of this paper is the introduction of a general approach to prediction of SLA violations for composite services, taking into account both QoS and process instance data, and using estimates to approximate not yet available data. Additionally, we present a prototype implementation of the system and an evaluation based on an order processing example. The ideas presented here are most applicable for long-running processes, where human intervention into problematic instances is possible. Our system introduces the notions of checkpoints (points in the execution of the composition where prediction can be done), facts (data which is already known in a checkpoint, such as the response times of already used services) and estimates (data which is not yet available, but can be estimated). Facts and estimates can refer to both typical QoS data (e.g., response times, availability, system load) and process instance data (e.g., customer identifiers, ordered products). Our implementation uses regression classifiers, a technique from the area of machine learning [3], to predict concrete SLO values.

The rest of the paper is structured as follows. In Section 2 we briefly introduce an illustrative example which we will use in the remainder of the paper. In Section 3 we detail the general concepts of our prediction approach. In Section 4 we described the implementation of a prototype tool, which we use for evaluation in Section 5. Finally, we provide an overview of relevant related work in Section 6 and conclude the paper in Section 7.

2 Illustrative Example

To illustrate the ideas presented in this paper we will use a simple purchase order scenario (see Figure 2 below). In this example there are a number of roles to consider: a reseller, who is the owner of the composite service, a customer, who is using it, a banking service, a shipping service, and two external suppliers. The business logic of the reseller service is defined as follows. Whenever the reseller service receives an order from the customer, it first checks if all ordered items are available in the internal stock. If this is not the case, it checks if the missing item(s) can be ordered from Supplier 1, and, if this is not the case, from Supplier 2. If both cannot deliver the order has to be cancelled, otherwise the missing items are ordered from the respective supplier. When all ordered items are available she will (in parallel) proceed to charge the customer using the banking service and initialize shipment of the ordered goods (using the Shipping

Service). We have borrowed this example from [4], please refer to this work for more information.

In this case study, the reseller has an SLA with its customers, with an SLO specifying that the end-to-end response time of the composition cannot be more than a certain threshold of time units. For every time the SLO is violated the customer is contractually entitled a discount for the order. Note that even though our explanations in this paper will be based on just one single SLO, our approach can be generalized to multiple SLOs. Additionally, even though we present our approach based on a numerical SLO, our ideas can be also applied to estimation of nominal objectives. However, SLOs need to adhere to the following requirements: (1) they need to be non-deterministic (following the definition in [5]), and (2) they cannot be defined as aggregations over multiple executions. Requirement (1) is not so much functionally important, but our prediction approach is not very useful otherwise (e.g., for SLOs concerning security requirements). Requirement (2) is a limitation of our current approach, which we plan on working on as part of our future work.

3   Predicting SLA Violations

In this section we present the core ideas of our approach towards prediction of SLA violations. Generally, the approach is based on the idea of predicting concrete SLO values based on whatever information is already available at a concrete point in the execution of a composite service. We distinguish three different types of information. (1) **Facts** represent data which is already known at prediction time. Typical examples of facts are the QoS of already used services, such as the response time of a service which has already been invoked in this execution, or instance data which has either been passed as input or which has been generated earlier in the process execution. (2) **Unknowns** are the opposites of facts, in that they represent data which is entirely unknown at prediction time. Oftentimes, instance data which has not yet been produced falls into this category. If important factors are unknown at prediction time the prediction quality will be very bad, e.g., in our illustrative example a prediction cannot be accurate before it is known whether the order can be delivered from the reseller's internal stock. (3) **Estimates** are a kind of middle ground between facts and unknowns, in that they represent data which is not yet available, but can be estimated. This is often the case for QoS data, since techniques such as QoS monitoring [5] can be used to get an idea of e.g., the response time of a service before it is actually invoked. Estimating instance data is more difficult, and generally domain-specific.

The overall architecture of our system is depicted in Figure 1. The most important concept used is that the user defines **checkpoints** in the service composition, which indicate points in the execution where a prediction should be carried out. The exact point in the execution model which triggers the checkpoint is called the **hook**. Every checkpoint is associated with one **checkpoint predictor**. Essentially, the predictor uses a function taking as input all facts
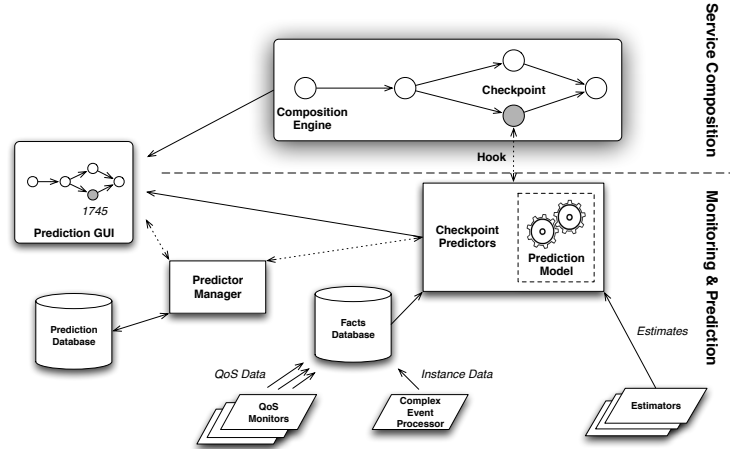
Fig. 1: Overall System Architecture

which are already available in the checkpoint, and, if applicable, a number of estimates of not yet known facts, and produces a numerical estimation of the SLO value(s). This function is generated using machine learning techniques We refer to this function as the **prediction model** of a checkpoint predictor. Facts are retrieved from a **facts database**, which is filled using a number of **QoS monitors** (which provide QoS data) and a **Complex Event Processing** (CEP) engine (which extracts and correlates the instance data, as emitted by the process engine). A detailed discussion of our event-based approach to monitoring is out of scope of this paper, but can be reviewed in related work [4, 6]. **Estimators** are a generic framework for components which deliver estimates. Finally, the prediction result is transferred to a graphical user interface (**prediction GUI**), which visualizes the predicted value(s) for the checkpoint. A **predictor manager** component is responsible for the lifecycle management of predictors, i.e., for initializing, destroying and retraining them. Additionally, predictions are stored in a **prediction database** to be available for future analysis.

3.1 Checkpoint Definition

At design-time, the main issue is the definition of checkpoints in the composition model. For every checkpoint, the following input needs to be provided: (1) The hook, which defines the concrete point in the execution that triggers the prediction, (2) a list of available facts, (3) a list of estimates, and the estimator component as well as the parameters used to retrieve or calculate them, (4) the retraining strategy, which governs at which times a rebuilding of the prediction model should happen, and (5) as a last optional step, a parameterization of the machine learning technique used to build the prediction model. After all these

inputs are defined the checkpoint is deployed using the predictor manager, and an initial model is built. For this a set of historical executions of the composite service need to be available, for which all facts (including those associated with estimates) have been monitored. If no or too little historical data is available the checkpoint is suspended by the predictor manager until enough training data has been collected. The amount of data necessary is case-specific, since it vastly depends on the complexity of the composition. We generally use the Training Data Correlation as a metric for evaluating the quality of a freshly trained model (see below for a definition), however, a detailed discussion of this is out of scope of this paper. After the initial model is built the continuous optimization of the predictor is governed by the predictor manager, according to the retraining strategy. Finally, the checkpoint can be terminated by the user via the prediction GUI. We will now discuss these concepts in more depth.
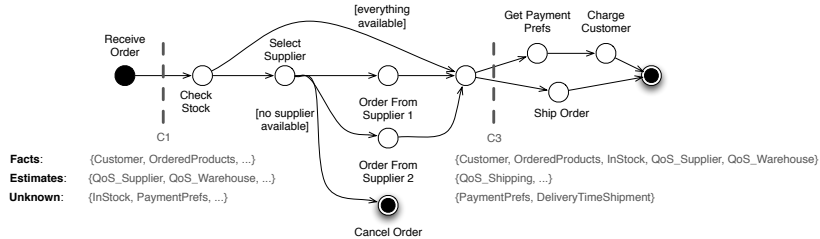

Fig. 2: Illustrative Example With Possible Checkpoints

Hooks Hooks can be inserted either before or after any WS-BPEL activity (for instance, an *Invoke* activity). Generally, there is a tradeoff to take into account here, since early predictions are usually more helpful (in that they rather allow for corrections if violations are predicted), but also less accurate since less facts are available and more estimates are necessary. Figure 2 depicts the (simplified) example from Section 2, and shows two possible checkpoints. In C_1 the only facts available are the ones given as input to the composition (such as a customer identifier, or the ordered products). Some other facts (mainly QoS metrics) can already be estimated, however, other important information, such as whether the order can be served directly from stock, is simply unavailable in C_1, not even as an estimate. Therefore, the prediction cannot be very accurate. In checkpoint C_3, on the other hand, most of the processes important raw data is already available as facts, allowing for good predictions. However, compared to C_1, the possibilities to react to problems are limited, since only the payment and shipping steps are left to adapt (e.g., a user may still decide to use express shipping instead of the regular one if a SLA violation is predicted in C3). Finding good checkpoints at which the prediction is reasonably accurate and still timely enough to react to problems demands for some domain knowledge about influential factors of composition performance. Dependency analysis as discussed in [6]

can help providing this crucial information. Dependency analysis is the process of using historical business process instance data to find out about the main factors which dictate the performance of a process. When defining checkpoints, a user can assume that the majority of important factors of influence need to be available as either facts or at least as good estimates in order to achieve accurate predictions.

Facts and Estimates: Facts represent all important information which can already be measured in this checkpoint. This includes both QoS and instance data. Note that the relationship between facts and the final SLO values does not need to be known (e.g., a user can include instance data such as user identifiers or ordered items, even if she is not sure if this has any relevance for the SLO). However, dependency analysis can again be used to identify the most important facts for a checkpoint. Additionally, the user can also define estimates. In the example above, in C_1 the response time of the warehouse service is not yet known, however, it can e.g., be estimated using a QoS monitor. Since estimating instance data is inherently domain-specific, our system is extensible in that more specific estimators (which are implemented as simple Java classes) can be integrated seamlessly. Estimates are linked to facts, in the sense that they have to represent an estimation of a fact which will be monitorable at a later point.

Retraining Strategy: Generally, the prediction model needs to be rebuilt whenever enough new information is available to significantly improve the model. The retraining strategy is used to define when the system should check whether rebuilding the prediction model is necessary. Table 1 summarizes all retraining strategies available, and gives examples. The custom strategy is defined using Java code, all other strategies are implemented in our prototype and can be used and configured without any additional code.

Strategy	Retrains ...	Example
periodic	... in fixed intervals	*every 24 hours*
instance-based	... whenever a fixed number of new instances have been received since the last training	*every 250 instances*
on demand	... on user demand	–
on error	... if the mean prediction error exceeds a given threshold	*if $\bar{e} > T$*
custom	... if a user-defined condition applies	*whenever more than 10 orders from customer 12345 have been received*

Table 1: Predictor Retraining Strategies

Prediction Model Parameterization: A user can also define the machine learning technique that should be used to build the prediction model. This is done

by specifying an algorithm and the respective parameterization for the WEKA toolkit[4], an open source machine learning toolkit which we internally use in our prototype implementation. In this way the prediction quality can be tuned by a machine learning savvy user, however, we also provide a default configuration which can be used out of the box.

3.2 Run-Time Prediction

At runtime, the prediction process is triggered by lifecycle events from the WS-BPEL engine. These are events emitted by some engines (such as Apache ODE[5]), which contain lifecycle information about the service composition (e.g., `ActivityExecStartEvent`, `VariableModificationEvent`, `ProcessCompletion-Event`). Our approach is based on these events, therefore, a WS-BPEL engine which is capable of emitting these events is a preliminary of our approach. When checkpoints are deployed we use the hook information to register respective event listeners. For instance, for a checkpoint with the hook "After invoke CheckStock" we generate a listener for `ActivityExecEndEvent`s which consider the invoke activity "CheckStock". We show the sequence of actions which is triggered as soon as such an event is received in Figure 3.
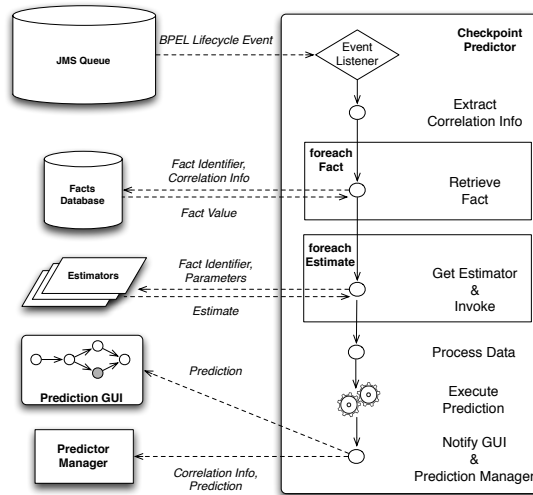

Fig. 3: Runtime View On Checkpoint Predictors

After being triggered by a lifecycle event the checkpoint predictor first extracts some necessary correlation information from the event received. This in-
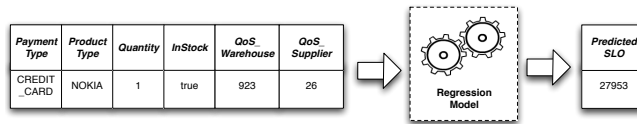

Fig. 4: Black-Box Prediction Model

cludes the process instance ID as assigned by the composition engine, the instance start time (i.e., the time when the instance was created) and the timestamp of the event. This information is necessary to be able to retrieve the correct facts from the facts database, which is done for every fact in the next step (e.g., in order to find the correct fact "CustomerNumber" for the current execution the process instance ID needs to be known). When all facts have been gathered, the predictor also collects the still missing estimates. For this, for every estimate the predictor instantiates the respective estimator component (if no instance of this estimator was available before), and invokes it (passing all necessary parameters as specified in the checkpoint definition). The gathered facts and estimates are then converted into the format expected by the prediction model (in the case of our prototype, this is the WEKA Attribute-Relation File Format ARFF[6]), and, if necessary, some data cleaning is done. Afterwards, the actual prediction is carried out by passing the gathered input to the prediction model producing a numerical estimation of the SLO value. This prediction is then passed to the prediction GUI (for visualization) and the prediction manager.

Note that the "intelligence" that actually implements the prediction of the SLO values is encapsulated in the prediction model. Since we (usually) want to predict numerical SLO values the prediction model needs to be a regression model [3]. We consider the regression model to be a black-box function which takes a list of numeric and nominal values as input, and produces a numeric output (Figure 4). Generally, our approach is agnostic of how this is actually implemented. In our prototype we use multilayer perceptrons (a powerful variant of neural networks) to implement the regression model. Multilayer perceptrons are trained iteratively using a back-propagation technique (maximization of the correlation between the actual outcome of training instances and the outcome that the network would predict on those instances), and can (approximately) represent any relationship between input data and outcome (unlike simpler neural network techniques such as the perceptron, which cannot distinguish data which is not separable by a hyperplane [7]). If a non-numerical SLO should be predicted, a different technique suitable for classification (as opposed to regression) needs to be used to implement the prediction model, e.g., decision trees such as C4.5 [8].

---

[6] http://www.cs.waikato.ac.nz/~ml/weka/arff.html

3.3 Evaluation of Predictors

Another important task of the prediction manager is quality management of predictors, i.e., continually supervising how predictions compare to the actual SLO values once the instance is finished. Generally, we use three different quality metrics to measure the quality of predictions in checkpoints, which are summarized in Table 2. The first metric, *Training Data Correlation*, is a standard machine learning approach to evaluating regression models. We use it mainly to evaluate freshly generated models, when no actual predictions have yet been carried out. This metric is defined as the statistical correlation between all training instance outcomes and the predictions that the model would deliver for these training instances. The definition given in the table is the standard statistical definition of the correlation coefficient between a set of predicted values P and a set of measured values M. However, note that this metric is inherently overconfident in our case, since during training all estimates are replaced for the facts that they estimate (i.e., the training is done as if all estimates were perfect). Therefore, we generally measure the prediction error later on, when actual estimates are being used. However, a low training data correlation is an indication that important facts are still unknown in the checkpoint, i.e., that the checkpoint may be too early.

Name	Definition
Training Data Correlation	$corr = \frac{cov(P,M)}{\sigma_p \sigma_m}$
Mean Prediction Error	$\bar{e} = \frac{\sum_{i=0}^{n} |m_i - p_i|}{n}$
Prediction Error Standard Deviation	$\sigma = \sqrt{\frac{\sum_{i=0}^{n} (e_i - \bar{e})^2)}{n}}$

Table 2: Predictor Quality Metrics

This can be done using the *Mean Prediction Error* $\bar{e}$, which is the average (Manhatten) difference between predicted and monitored values. In the definition in Table 2, n is the total number of predictions, p_i is a predicted value, and m_i is the measured value to prediction p_i. Finally, we use the *Prediction Error Standard Deviation* (denoted here simply as σ) to describe the variability of the prediction error (i.e., high σ essentially means that the actual error for an instance can be much lower or higher than $\bar{e}$). In the definition, e_i is the actual prediction error for a process instance ($m_i - p_i$). These metrics are mainly used to give the user an estimation of how trustworthy a given prediction is. Additionally, the `on error` retraining strategy triggers on $\bar{e}$ exceeding a certain threshold.

4   Tool Implementation

In order to verify our approach we built a prototype prediction tool in the Java programming language. Our core implementation is based on our earlier work on event-based monitoring and analysis (as presented in [6] and [4]). Data persistence is provided using a simple MySQL[7] database and Hibernate[8]. We have integrated two different approaches to QoS monitoring: firstly, QoS data as provided by the event-based QoS monitoring approach discussed in [6], and secondly, the QoS data provided by server- and client-side VRESCo [9] QoS monitors [5]. In order to enable event-based monitoring we have used Apache ActiveMQ[9] as JMS middleware. Finally, as has already been discussed, we use the open-source machine learning toolkit WEKA to build prediction models. WEKA is integrated in our system using the WEKA Java API. In addition to the actual prediction tool we have also prototypically implemented the illustrative example as presented in Section 2, as a testbed to verify our ideas (this will be discussed in more detail in Section 5). We have used the WS-BPEL engine Apache ODE, mainly because of ODE's strong support for BPEL lifecycle events. We have also set up the necessary base services which are used in the example (e.g., supplier services, banking service, stock service) using Apache CXF[10].

```
1   <cpdl:checkpoints
2     xmlns:cpdl="http://www.infosys.tuwien.ac.at/2009/cpdl">
3
4     <checkpoint
5       name="beforeGetPaymentPrefs"
6       activityName="getPaymentPrefs" breakBefore="true"
7       predictor="weka.classifiers.functions.MultilayerPerceptron">
8
9       <update type="periodically" value="5"/>
10      <class ppmRef="ORDER_FULFILLMENT_LEAD_TIME" />
11      <fact ppmRef="RESPONSE_TIME_WAREHOUSE" />
12      <fact ppmRef="ORDER_INSTOCK" />
13      <!-- more facts -->
14      <estimate name="getPaymentPrefsResponseTime" type="integer">
15        <estimatorClass
16          class="at.ac.tuwien.infosys.branimon.VrescoQoSEstimator"/>
17        <argument value="ResponseTime"/>
18        <argument value="CustomerService"/>
19        <estimatedField ppmRef="RESPONSE_TIME_GETPAYMENTPREFS" />
20      </estimate>
21
22      <!-- more estimates -->
23    </checkpoint>
24
25  <checkpoints>
```

Fig. 5: Checkpoint Definition in XML Notion

---

[7] http://www.mysql.com/

[8] https://www.hibernate.org/

[9] http://activemq.apache.org/

[10] http://cxf.apache.org/

As discussed in Section 3, the main input for our approach is a list of checkpoint definitions. In our current prototype, these definitions are given in a proprietary XML-based language, which we refer to as CPDL (Checkpoint Definition Language). An exemplary excerpt can be seen in Figure 5. In the figure, a checkpoint, which is hooked before the execution of the invoke activity "getPaymentPrefs", is defined. A multilayer perceptron is used as prediction model. The checkpoint will be retrained periodically every 5 hours, and will predict the SLO ORDER_FULFILLMENT_LEAD_TIME. Then a number of available facts and estimates are specified. For estimates, an estimator class is given as a full qualified Java class name, which implements the actual prediction. Additionally, a number of arguments can be given to the estimator class. Finally, for every estimate a link to the estimated fact needs to be specified. Note that we do not define facts directly in CPDL. Instead, we reuse the model presented in [6], where we discussed a language for definition of facts using calculation formulae and XLink[11] pointers to WS-BPEL processes (so-called PPMs, process performance metrics). In CPDL, ppmRefs are identifiers which point to PPMs in such a model. The complete XML Schema definition of CPDL is available online[12].

5 Experimentation

In order to provide a first validation of the ideas presented we have implemented the illustrative example as discussed in Section 2, and run some experiments using our prototype tool. All experiments have been conducted on a single test machine with 3.0 GHz and 32 GByte RAM, running under Windows Server 2007 SP1. We have repeated every experiment 25 times and averaged the results, to reduce the influence of various random factors such as current CPU load or workload of the process engine.

Instances	Training [ms]	Instances	Prediction [ms]
100	3545	100	615
250	8916	250	630
500	17283	500	631
1000	31806	1000	647
(a) Training Overhead		(b) Prediction Overhead	

Table 3: Overhead for Training and Prediction

In Table 3 we have sketched the measured times for two essential operations of our system. Table 3(a) depicts the amount of time in milliseconds necessary to build or refresh a prediction model in a checkpoint. The most important factor here is clearly the time necessary to train the machine learning model,

---

[11] http://www.w3.org/TR/xlink/

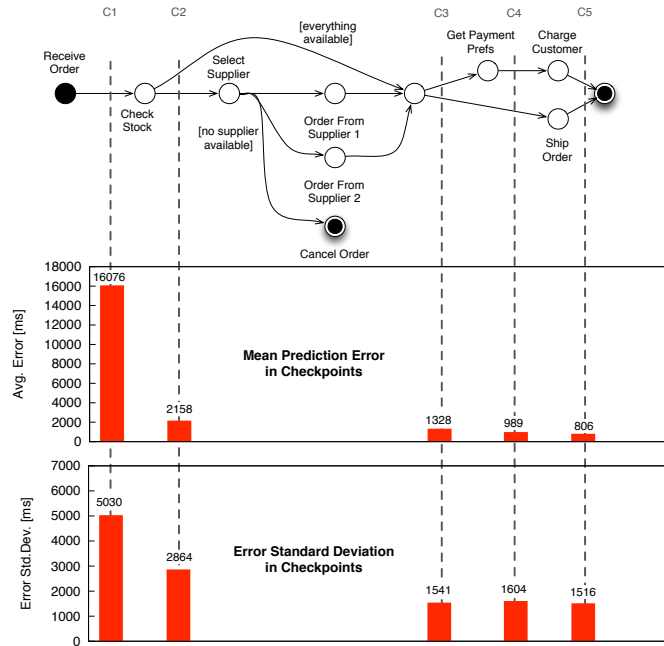[12] http://www.infosys.tuwien.ac.at/staff/leitner/cpdl/cpdl_model.xsd

Fig. 6: Prediction Error in Checkpoints

e.g., to train the neural network in our illustrative example. This factor mainly depends on the number of training instances available. In Table 3(a) it can be seen that the time necessary for building the model depends linearly on the number of historical instances available. However, even for e.g., 1000 instances the absolute rebuilding time is below 32 seconds, which seems acceptable for practice, considering that model rebuilding can be done sporadically and offline. Additionally, when rebuilding the model, there is no time where no prediction model is available at all. Instead, the new model is trained offline, and exchanged for the last model as soon as training is finished. A more detailed discussion of these factors is out of scope of this paper for reasons of brevity. In Table 3(b) we have sketched the time necessary for actual prediction, i.e., the online part of the system. As can be seen this overhead is constant and rather small (well below 1 second), which seems very acceptable for prediction at run-time.

Even more important than the necessary time is the accuracy of predictions. To measure prediction accuracy, we have realized five checkpoints in the illustrative example (see top of Figure 6): C1 is located directly after the order is received, C2 after the internal warehouse is checked, C3 after eventual orders from external suppliers have been carried out, C4 during the payment and shipment process, and finally C5 when the execution is already finished. In each of those checkpoints we have trained a prediction model using 1000 historical process instances, and have specified all available data as facts. For not yet available

QoS metrics we have used the average of all previous invocations as estimate. Missing instance data has been treated as unknown. We have used each of those checkpoints to predict the outcome of 100 random executions, and calculated the Mean Prediction Error $\bar{e}$ and the Error Standard Deviation σ (both as defined in Section 3). As expected, $\bar{e}$ is decreasing with the amount of factual data available. In C1, the prediction is mostly useless, since no real data except the user input is available. However, in C2 the prediction is already rather good. This is mostly due to the fact that in C2 the information whether the order can be delivered directly from stock is already available. In C3, C4 and C5 the prediction is continually improving, since more actual QoS facts are available, and less estimates are necessary. Speaking in absolute values, $\bar{e}$ in e.g., C3 is 1328 ms. Since the average SLO value in our illustrative example was about 16000 ms, the error represents only about 8% of the actual SLO value, which seems satisfactory. Similar to $\bar{e}$, σ is also decreasing, however, we can see that the variance is still rather high even in C3, C4 and C5. This is mostly due to our experimentation setup, which included the (realistic) simulation of occasional outliers, which are generally unpredictable.

6 Related Work

The work presented in this paper is complementary to the more established concept of SLA management [10]. SLA management incorporates the definition and monitoring of SLAs, as well as the matching of consumer and provider templates. [10] introduces SLA management based on the WSLA language. However, other possibilities exist, e.g., in [11] the Web Service Offerings Language (WSOL) has been introduced. WSOL considers so-called Web service offerings, which are related to SLAs. Runtime management for WSOL, including monitoring of offerings, has been described in [12], via the WSOI management infrastructure. In our work we add another facet to this, namely the prediction of SLA violations before they have actually occurred. Inherently, this prediction demands for some insight into the internal factors impacting composite service performance. In [13], the MoDe4SLA approach has been introduced to model dependencies of composite services on the used base services, and to analyze the impact that these dependencies have. Similarly, the work we have presented in [4] allows for an analysis of the impact that certain factors have on the performance of service compositions. SLA prediction as discussed in this paper has first been discussed in [14], which is based on some early work of HP Laboratories on SLA monitoring for Web services [15]. In [14], the authors introduced some concepts which are also present in our solution, such as the basic idea of using prediction models based on machine learning techniques, or the trade-off between early prediction and prediction accuracy. However, the authors do not discuss important issues such as the integration of instance and QoS data, or strategies for updating prediction models. Additionally, this work does not take estimates into account, and relatively little technical information about their implementation is publicly available. A second related approach to QoS prediction has been pre-

sented recently in [16]. In this paper the focus is on KPI prediction using analysis of event data. Generally, this work exhibits similar limitations as the work described in [14], however, the authors discuss the influence of seasonal cycles on KPIs. This facet has not been examined in our work, even though seasons can arguably be integrated easily in our approach as additional facts.

7   Conclusions

In this paper we have presented an approach to runtime prediction of SLA violations. Central to our approach are checkpoints, which define concrete points in the execution of a composite service at which prediction has to be carried out, facts, which define the input of the prediction, and estimates, which represent predictions about data which is not yet available in the checkpoint. We use techniques from the area of machine learning to construct regression models from recorded historical data to implement predictions in checkpoints. Retraining strategies govern at which times these regression models should be refreshed. Our Java-based implementation uses the WEKA Machine Learning framework to build regression models. Using an illustrative example we have shown that our approach is able to predict SLO values accurately, and does so in near-realtime (with an delay of well below 1 second).

As part of our future work we plan to extend the work presented here in three directions. Firstly, we want to improve the usability of our prototype by improving the GUI, especially with regard to the definition of checkpoints. Currently, this is mostly done on XML code level, which is clearly unsuitable for the targeted business users. Instead, we plan to incorporate a template-based approach, where facts and estimates are as far as possible generated automatically. Secondly, we want to generalize the ideas presented in this paper so that they are also applicable to aggregated SLOs, such as "Average Response Time Per Day". Thirdly, we plan to extend our prototype to not only report possible SLA violations to a human user, but to actively try to prevent them. This can be done by triggering adaptations in the service compositions, for instance using BPEL'n'Aspects [17]. However, more research needs to be conducted in order to define models of how possible SLA violations can best be linked to adaptation actions, i.e., how to best define which adaptations are best suited to prevent which violations.

Acknowledgements

References

1. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer **11** (2007)

2. Menascé, D.A.: Qos issues in web services. IEEE Internet Computing **6**(6) (2002) 72–75
3. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. 2 edn. Morgan Kaufmann (2005)
4. Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Leymann, F., Dustdar, S.: Monitoring and Analyzing Influential Factors of Business Process Performance. In: EDOC'09: Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference. (2009)
5. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: Comprehensive QoS Monitoring of Web Services and Event-Based SLA Violation Detection . In: MW4SOC 2009: Proceedings of the 4rd International Workshop on Middleware for Service Oriented Computing. (2009)
6. Wetzstein, B., Strauch, S., Leymann, F.: Measuring Performance Metrics of WS-BPEL Service Compositions. In: ICNS'09: Proceedings of the Fifth International Conference on Networking and Services, IEEE Computer Society (2009)
7. Haykin, S.: Neural Networks and Learning Machines: A Comprehensive Foundation. 3 edn. Prentice Hall (2008)
8. Quinlan, J.R.: Improved Use of Continuous Attributes in C4.5. Journal of Artificial Intelligence Research **4** (1996) 77–90
9. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-End Support for QoS-Aware Service Selection, Invocation and Mediation in VRESCo. Technical report, TUV-1841-2009-03, Vienna University of Technology (2009)
10. Dan, A., Davis, D., Kearney, R., Keller, A., King, R., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., Youssef, A.: Web Services on Demand: WSLA-Driven Automated Management. IBM Systems Journal **43**(1) (2004) 136–158
11. Tosic, V., Pagurek, B., Patel, K., Esfandiari, B., Ma, W.: Management applications of the web service offerings language (wsol). Information Systems **30**(7) (2005) 564–586
12. Tosic, V., Ma, W., Pagurek, B., Esfandiari, B.: Web Service Offerings Infrastructure (WSOI) – A Management Infrastructure for XML Web Services. In: NOMS'04: Proceedings of the IEEE/IFIP Network Operations and Management Symposium. (2004) 817–830
13. Bodenstaff, L., Wombacher, A., Reichert, M., Jaeger, M.C.: Monitoring Dependencies for SLAs: The MoDe4SLA Approach. In: SCC '08: Proceedings of the 2008 IEEE International Conference on Services Computing, Washington, DC, USA, IEEE Computer Society (2008) 21–29
14. Castellanos, M., Casati, F., Dayal, U., Shan, M.C.: Intelligent Management of SLAs for Composite Web Services. In: DNIS 2003: Proceedings of the 3rd International Workshop on Databases in Networked Information Systems. (2003) 28–41
15. Sahai, A., Machiraju, V., Sayal, M., Moorsel, A.P.A.v., Casati, F.: Automated SLA Monitoring for Web Services. In: DSOM '02: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, London, UK, Springer-Verlag (2002) 28–41
16. Zeng, L., Lingenfelder, C., Lei, H., Chang, H.: Event-Driven Quality of Service Prediction. In: ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing, Berlin, Heidelberg, Springer-Verlag (2008) 147–161
17. Karastoyanova, D., Leymann, F.: BPEL'n'Aspects: Adapting Service Orchestration Logic. In: ICWS 2009: Proceedings of 7th International Conference on Web Services, Los Angeles, CA, USA, IEEE (2009)