Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, eds.

SIMULATION BASED VALIDATION OF QUANTITATIVE REQUIREMENTS IN SERVICE ORIENTED ARCHITECTURES

Falko Bause
Peter Buchholz
Jan Kriege
Sebastian Vastag

Informatik IV,
TU Dortmund,
D-44221 Dortmund

ABSTRACT

Large Service Oriented Architectures (SOAs) have to fulfill qualitative and quantitative requirements. Usually Service Level Agreements (SLAs) are defined to fix the maximal load the system can accept and the minimal performance and dependability requirements the system has to provide. In a complex SOA where services use other services and thus performance and dependability of a service depend on the performance and dependability of lower level services, it is hard to give reasonable bounds for quantitative measures without performing experiments with the whole system. Since field experiments are too costly, model based analysis, often using simulation is a reliable alternative. The paper presents an approach to model complex SOAs and the corresponding SLAs hierarchically, map the model on a simulator and analyze the model to validate or disprove the different SLAs.

1 INTRODUCTION

Service Oriented Architectures (SOAs) are a common paradigm to realize large distributed software systems. SOAs are based on the loose coupling of different processes which interact via service calls. The combination of different services is denoted as orchestration that enables a user to build complex distributed systems in an efficient and reliable way (Peltz 2003). Apart from functional correctness an adequate Quality of Service (QoS) is the key aspect in any SOA. The system has to assure a predefined level of performance and availability to meet the requirements of the user. The quality of service requirements are usually written down in a contract denoted as *Service Level Agreement* (SLA) which, however, considers apart from the specification of quantitative aspects like performance and availability also many other aspects like financial aspects, user expectations or hardware and software requirements (Trienekens, Bouman, and van der Zwan 2004). The focus of this paper is on the parts of an SLA which specify the performance and availability of services. Our goal is to integrate these parts of the SLA in the model specification and validate them using simulation. If we talk of an SLA in the sequel, we mean the parts of the whole contract that consider quantitative aspects.

There are two challenges when dealing with SLAs, namely to specify them adequately and to validate them. Both are of practical and theoretical interest and are, of course, dependent since validation requires a formal or semi formal specification. We will use a restricted set of patterns to express SLAs. However, even if SLAs are formally specified, the validation can be cumbersome. Often validation is done during runtime by monitoring the actual service calls (Skene et al. 2007). This approach is necessary to assure that the contract which has been concluded is fulfilled from both sides. I.e., the provider supplies an adequate QoS and the user brings an adequate load into the system. However, for a provider it is also interesting to know which QoS he or she is able to guarantee for a given load. For a user it is interesting to know which QoS he or she is going to expect from a service that is composed of different services from different providers each with its own SLA. In general both problems can be analyzed using measurements of the running system with some benchmark workload. However, it is often better to use model based analysis to validate SLAs in these cases.

The simulative validation of SLAs in a SOA requires an adequate model of the architecture and the integration of the SLAs in the model. Modeling of SOAs has to consider both, the workflow of the services and also the communication network. The complexity of the systems usually requires a rather high level model. Only very few simulation approaches are currently available that allow a natural modeling of SOAs, (Sarjoughian et al. 2008) is one of the few examples. In this

978-1-4244-5771-7/09/$26.00 ©2009 IEEE

paper, we present an approach to integrate SLAs in a model of a SOA and validate them via simulation. The model class we use is a specific class of hierarchical process chains that are mapped on simulation models (Bause et al. 2008b) and which have already shown to be eligible for the modeling of SOAs (Bause et al. 2008a). In this paper we extend the approach of (Bause et al. 2008a) by integrating SLAs in the model description and by developing methods to statistically evaluate the SLAs using simulation experiments. This allows us to perform a statistically well founded validation of SLAs.

The paper is structured as follows. In the next section we briefly present the ProC/B approach to model SOAs. Then we show how to specify SLAs in a model of a SOA using predefined patterns. Afterwards, in Sect. 4, the validation of SLAs via simulation is introduced and an example is presented in Sect. 5. The paper ends with the conclusions.

2 MODELING OF SOA USING HIERARCHICAL PROCESS CHAINS

The approach presented in this paper uses hierarchical process chains, namely *ProC/B* models, to specify SOAs and the corresponding SLAs. *ProC/B* (Bause et al. 2002) is a process chain based modeling approach which formalizes the process chain paradigm introduced by (Kuhn 1995, Kuhn 1999). Originally *ProC/B* was developed and used for the description and analysis of logistics networks within the collaborative research center ”Modelling of Large Logistics Networks” 559 (Collaborative Research Center 559). Only recently *ProC/B* has been extended to meet the requirements that are necessary to describe components of SOAs (Bause et al. 2008a).

ProC/B models consist of function units (FUs) that capture the structure of a system and process chains (PCs) that model the behavior. PCs are composed of process chain elements (PCEs) which specify the load at one level. FUs might offer services that can be used by activities of PCs. Those services are again described by PCs. Furthermore, they can use other services, which are offered by internal or imported FUs, thus introducing a hierarchy. *ProC/B* offers two kinds of standard FUs that have predefined services and are used at the lowest level of the hierarchy: servers act as traditional queues and describe the consumption of time, while so-called counters support the manipulation of passive resources and describe the consumption of space. User defined FUs are used in the higher levels of the hierarchy. They offer services to their environment that are internally realized by process chains which may use some services of lower level FUs. The hierarchy of *ProC/B* has to form an acyclic graph and is based on a calling semantics of services. A service is called from a process chain element of an upper level and the upper level process is blocked until the service call returns.

Figures 1-3 present an example of a *ProC/B* model. The top level of the model is shown in Figure 1 where the behavior of a process type called `caller` is defined by a PC. Process chain `caller` contains two PCEs named `count_callers` and `call_service`. Those names are arbitrarily assigned by the user. PCE `count_callers` is a so-called CODE element which allows for the specification of code blocks in *ProC/B* syntax which are executed during simulation. More complex examples of code blocks are shown in Listings 1 and 2. PCE `call_service` calls a service of the FU `Service_Provider`. Each process of process chain `caller` contains a local variable `success` which is initially set to `FALSE` and might have changed after finishing the service call. Access notations to parameters and variables of processes are prefixed with keyword *data* for technical reasons in order to distinguish them from global variables. FU `Model` contains also a variable `no_callers` which is globally accessible for all processes within this FU. In this example the global variable is used to count the number of callers. Processes of type `caller` are incarnated with exponentially distributed interarrival times as specified by the source element of Figure 1. The service `service` of the `Service_Provider` is specified by a process chain as shown in Figure 2.

The first activity in FU `Service_Provider` is again a service call, now using service `start_trans` of FU `Transaction`. This FU manages the amount of parallel active service requests to FU `Processor` which executes the requests of the `caller` processes. FU `Processor` is a standard FU which acts as a traditional queue. Access to the `Processor` is granted by setting variable `data.success` to true and using this variable in a boolean OR-connector selecting the upper branch, in case `data.success` is true or the lower (empty) branch otherwise. The upper branch specifies the call of service `request` of FU `Processor` followed by a call to service `commit` of FU `Transaction`.

Service calls of the `caller` processes compete with processes of process chain `maintenance` which are incarnated in batches of size 2 with an exponentially distributed batch interarrival time and also terminate within FU `Service_Provider`. Their behavior is similar to the one of the processes of type `caller`.

The internals of FU `Transaction` are shown in Figure 3. FU `Pool` is a counter modeling the use of passive resources which is here a “token pool” used for managing the concurrent access to FU `Processor`. In this example the pool is represented by an integer value initially set to the default value 0 and is only allowed to take a value within the interval $[0,16]$. Service `change` requests for a change of the counter’s value. A change to the counter is immediately granted iff its result respects the specified upper and lower bounds; otherwise the requesting process gets blocked until the change becomes possible. The predefined service `alter_or_skip` has a more sophisticated semantics. In this example the requesting

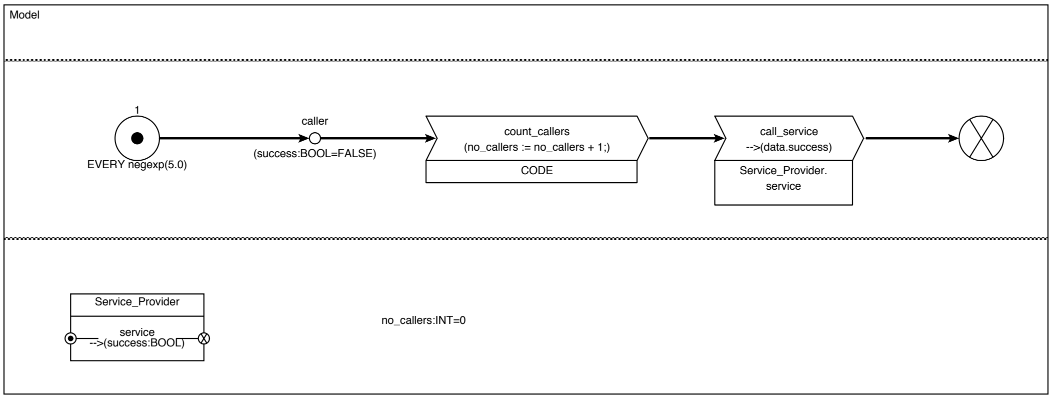

Figure 1: Example of a *ProC/B* model

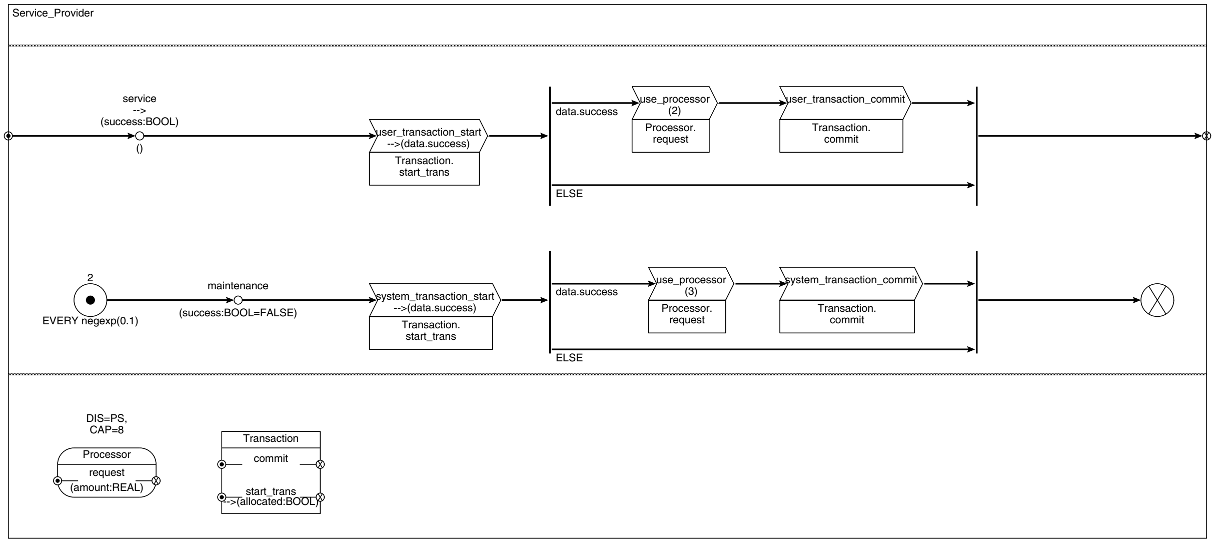

Figure 2: FU `Service_Provider`

process tries to increase the counter value by 1 or, if not possible, the counter value is not increased (increase by 0). The actual increase value is returned and stored in variable `data.my_amount`, such that `data.allocated` is set to true if the integer value of the pool has been increased by 1. In the model of Figures 1-3 no process will be blocked when calling service `commit`, since a decrease of the pool value will only happen after a corresponding increase.

ProC/B allows for the specification of measures in an elaborate way. Standard measures like for example throughputs or response times can be measured at any FU. Additionally, user-defined measures (called rewards in *ProC/B*) can be used for serially collecting values, for estimating rates and for the description of trajectories. While the user-defined rewards have to be updated manually using a model element designed for those updates, the standard measures are updated automatically whenever a process enters or leaves a FU. Moreover *ProC/B* supports the itemization of the measurement streams, i.e. when measuring at an FU it is possible to consider only processes that moved along a specific path through the model hierarchy before using a service of that FU and to ignore all other processes for the measure.

As already mentioned the *ProC/B* formalism was extended in (Bause et al. 2008a) for the modeling of SOAs. These extensions include a timeout mechanism that can be used to specify an upper limit for the amount of time a process will wait for the return from a call to a service. In SOAs such a behavior is important since in a complex system it cannot be assured that a service call always returns in time or even returns at all. If the service call is not finished in time, the calling process has to proceed, of course, the subsequent activities may depend on whether the call timed out or not. For the

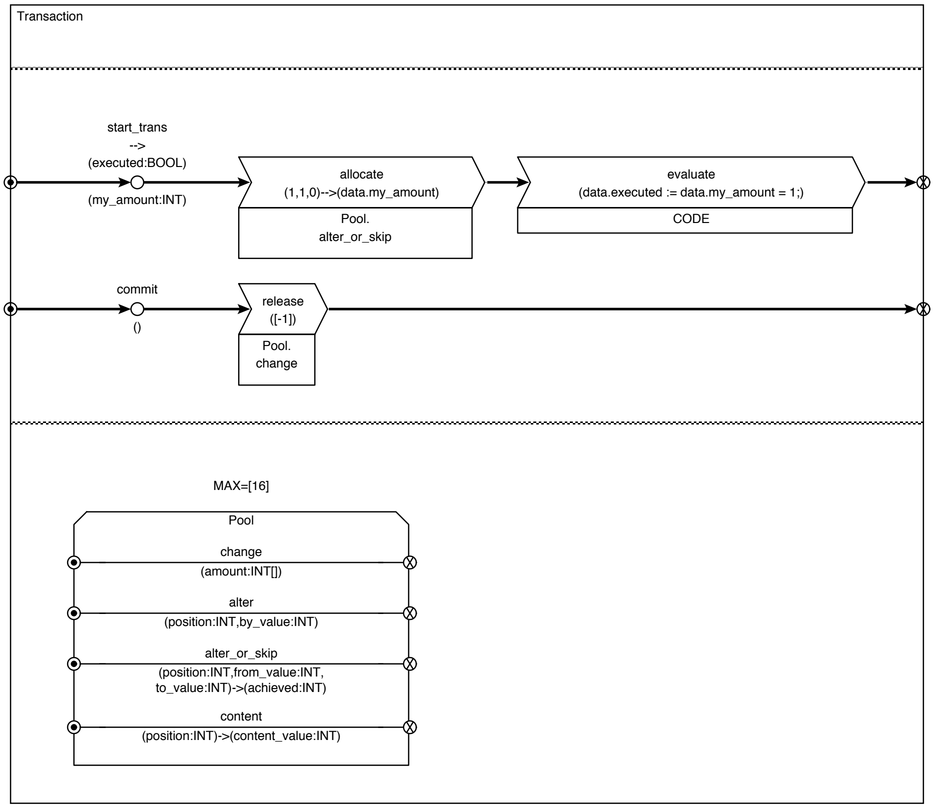
Transaction
start_trans
-->
(executed:BOOL)
(my_amount:INT)
allocate
(1,1,0)-->(data.my_amount)
Pool.
alter_or_skip
evaluate
(data.executed := data.my_amount = 1;)
CODE
commit
()
release
([-1])
Pool.
change
MAX=[16]
Pool
change
(amount:INT[])
alter
(position:INT,by_value:INT)
alter_or_skip
(position:INT,from_value:INT,
to_value:INT)->(achieved:INT)
content
(position:INT)->(content_value:INT)

Figure 3: FU `Transaction`

modeling and analysis of *ProC/B* models a toolset exists which provides a graphical user interface for model specification and tools that transform a *ProC/B* model into the input language of different analysis tools, allowing for an automated analysis of the models. Currently available are transformers for two simulators, a queueing network solver and a tool for the analysis of stochastic Petri nets. The transformers for the simulators are able to handle any *ProC/B* model whereas the latter two transformers assume some restrictions. For more details on *ProC/B* and the corresponding toolset we refer the reader to (Bause et al. 2002, Bause et al. 2008b, Bause et al. 2008a). One analysis option of the toolset is the simulation of *ProC/B* models by using *OMNeT++*. *OMNeT++* is a public-source simulation environment that has been developed for the modeling of communication protocols and several model frameworks are available for *OMNeT++* with an emphasis on network models. Using this analysis option of the toolset *ProC/B* models may contain FUs that are realized by *OMNeT++* models, such that the process flow of the Web services is described in *ProC/B* at a higher level and the underlying network structure is specified by an *OMNeT++* model. The amount of data for communication over the network can be stated in the *ProC/B* model and is passed to the *OMNeT++* model where the transfer of the data over the network is simulated. Since also the process chain elements can be translated to *OMNeT++*, the whole model accounts for both, a high-level SOA description and the description of the underlying network structure at a lower level. The complete model can be simulated with the *OMNeT++* simulation engine.

3 INTEGRATION OF SLAs

Before we present the integration of SLAs in the process oriented model view, some approaches for the specification of SLAs are reviewed. Specification of a quantitative requirement has to be done in such a way that it can be validated which means that it can be measured at the running system (Raimondi, Skene, and Emmerich 2008, Skene et al. 2007) or it can be analyzed with some model (Menascé, Ruan, and Gomaa 2007). This implies that the specification has to include a precise description of the service and of the requirements (Trienekens, Bouman, and van der Zwan 2004). In general an SLA has to include two components, the specification of the maximal load and the minimal service level. Without a bound on the load, every service can be overloaded such that the required service level cannot be reached.

Quantitative aspects of SLAs have to be specified in a formal or at least semi formal way. Various approaches exist to specify SLAs like the XML based specification language WSLA (Ludwig et al. 2003) or extensions to UML (Skene, Lamanna, and Emmerich 2004, Teyssié 2006) to mention only two examples. To validate SLAs a precise semantics is necessary. Thus, SLAs may be described using logical formulas (Teyssié 2006) or by transferring them into a timed automaton (Raimondi, Skene, and Emmerich 2008) which may be used to verify that a trace resulting from a measurement or simulation of the system observes the requirements given in the SLA. However, usually the behavior of a complex system like a SOA is stochastic such that some violations of the SLA will occur and it has to be analyzed how often such violations occur. This requires some statistical evaluation which is, to the best of our knowledge, not considered in the literature on SLA validation yet. We will come back to this point in the following section after introducing our approach to specify SLAs in process models.

For a user it is usually very hard to specify an SLA formally from scratch using some logic or a timed automaton or even using some XML or UML based specification language. One approach to bypass the low level definition of properties is to use patterns that allow one to describe common properties at a higher level. This high level description is then transformed into a low level specification amenable for the used validation or verification technique. Patterns for functional properties in finite state systems can be found in (Dwyer, Avrunin, and Corbett 1999). These patterns are mapped onto logical formulas in some temporal logics like CTL or LTL which can be used as input for some model checker. However, this approach cannot be applied to express quantitative aspects since the logics have no notion of time or probability. Consequently, extended patterns for real time verification have been defined in (Konrad and Cheng 2005) and are based on the real time logics MTL or TCTL. This set of patterns includes patterns to express the maximal or minimal duration or the maximal and minimal response time and may be used to express some requirements included in SLAs. However, the patterns are developed for hard real time systems such that they can only be used to prove or disprove that a property always holds. This is, as already argued above, not adequate for SLAs of SOAs where we usually will have some, hopefully small, probability to miss a requirement. We extend the idea of patterns to express SLAs. In contrast to the mentioned approach our patterns are not translated into logical formulas, instead they define measurement streams in a simulation which are validated and statistically evaluated by simulation experiments.

As shown in section 2 services are specified by PCs and are offered by FUs. Thus, access points of services are represented by the service entry points at the FUs. SLAs are assigned to the entry points and allow one to express properties that are measurable at the entry and exit level of a service. This includes the times between initiation and termination of the service call and the status of the returning service call. We assume that the status of a service call is a Boolean variable `success` which is `TRUE` for a successful call and `FALSE` otherwise. Of course, `success` can be defined by some Boolean expression over the return parameters of the service call. If a service call has no parameters, then we assume that `success` is always `TRUE`. For some service call `scall`, `scall.success` is the value of the *success variable*, `scall.at` is the arrival time and `scall.st` is the sojourn time which is the time between start and termination of the service call.

Let sla be an SLA specification for some service, then sla.load describes the condition on the load (i.e., an upper bound for the calling instance), sla.perf describes the requirement on the duration of the service (i.e., a lower bound for the executing instance) and sla.avail describes the requirement on the availability of the service (i.e., a lower bound on the percentage of successful calls for the executing instance). We assume that each SLA has a *load* part and it has a *perf* or an *avail* part but not necessarily both.

We now introduce the syntax and semantics of the different parts of the SLA specification.

$$\textsf{sla.load} := \lambda_{avg} \mid t_{min} \mid (\Delta t, m) \mid (t_{min} \wedge (\Delta t, m))$$

where $\lambda_{avg}, t_{min}, \Delta t \in \mathbb{R}_{>0}$ and $m \in \mathbb{N}$. The terms have the following semantics:

λ_{avg} The average number of service calls per time unit is at most λ_{avg} which implies that the mean interarrival time of service calls is at least $(\lambda_{avg})^{-1}$. This measure is defined without a time interval for which the average values are computed such that additional assumptions are necessary to validate the SLA. These assumptions will be introduced in the following section.

t_{min} The time between two successive service calls is at least t_{min}.

$(\Delta t, m)$ In an interval of length Δt at most m calls arrive.

$(t_{min} \wedge (\Delta t, m))$ This requirement combines the two previous requirements. It is fulfilled by a call that arrives at time t, if for the arrival time of the previous call $\leq t - t_{min}$ holds and in $[t - \Delta t, t] \leq m$ calls arrived.

Similarly, the performance part can be specified as

$$\textsf{sla.perf} \;:=\; t_{avg} \mid t_{max} \mid (t_{max}, n, m)$$

where $t_{avg}, t_{max} \in \mathbb{R}_{>0}$, $n, m \in \mathbb{N}$, $n \leq m$ with the following semantics:

t_{avg} The average duration of a service call is at most t_{avg}. Like λ_{avg} this is an average value defined without any interval such that additional requirements are necessary for its validation.

t_{max} The maximal duration of a service call is t_{max}.

(t_{max}, n, m) n out of the last m service calls have a duration of at most t_{max}.

The performance part only considers service calls that terminated successfully.
Finally, we consider the availability requirements.

$$\textsf{sla.avail} \;:=\; p_{avg} \mid (n, m)$$

where $0 < p_{avg} \leq 1$, $n, m \in \mathbb{N}$, $n \leq m$ with the following semantics:

p_{avg} The average success probability of a call is at least p_{avg}. Again this an average measure without time interval.

(n, m) At least n out of the last m service calls had been successful.

The different requirements can be verified per service call or for all calls of a simulation run. The corresponding methods will be proposed in the following section.

4 SIMULATIVE VALIDATION OF SLAs

Three different parts of an SLA consider different measures of a service call. sla.load is based on `scall.at`, whereas sla.perf considers `scall.st` of all terminating service calls where `scall.success` is `TRUE`. Finally, sla.avail is based on the evaluation of `scall.success`. We have to distinguish between the average measures λ_{avg}, t_{avg}, p_{avg} and the remaining measures.

We start with the average measures which consider the long run behavior of the system. Without additional assumptions these measures cannot be validated or disproved on a finite run since the system can always run for a longer time and may then possibly reach the required service level. Therefore, we assume for the analysis of average values that the system has reached its steady state which is a common assumption for simulative analysis of stochastic systems. The steady state assumption has to be validated using available methods from stochastic simulation (Law and Kelton 2000, chap. 9). Let X be the random variable describing the required quantity (i.e., the interarrival or sojourn time or the success probability). Then it has to be validated that $E(X) \leq x_{avg}$ holds. For availability analysis usually a lower bound for the success probability is defined but $1 - p_{avg}$ can be used as upper bound for the miss probability. For statistical analysis it has to be taken into account that the observation of consecutive service calls results in correlated samples such that analysis methods for dependent sequences have to be applied (see (Law and Kelton 2000)). We use a batch means method with b batches of size k each (i.e., $h = bk$ equals the number of observed service calls) and assume that the batch means are independent and identically distributed. In steady state the required measures from the service calls (i.e., the interarrival times, sojourn times or success probabilities) are collected and mean values can be computed. Let x_i be the corresponding value of the ith service call, then

$$\bar{x} = \frac{1}{h}\sum_{i=1}^{h} x_i = \frac{1}{b}\sum_{i=0}^{b-1}\left(\frac{1}{k}\sum_{j=1}^{k} x_{ik+j}\right) \tag{1}$$

is the estimated mean for n observed service calls and

$$\hat{S}^2(x) = \frac{1}{b-1}\sum_{i=0}^{b-1}\left(\left(\frac{1}{k}\sum_{j=1}^{k} x_{ik+j}\right) - \bar{x}\right)^2 \tag{2}$$

is an estimate for the variance of the x_i. With known estimators for mean and variance, confidence intervals can be computed using standard means. However, for the validation of SLAs, the estimated mean has to be compared with a standard defined by x_{avg}. This implies that one-sided confidence intervals are required. For some significance probability α, the requirement $E(X) \leq x_{avg}$ is validated if

$$\bar{x} + t_{b-1,1-\alpha}\sqrt{\frac{\hat{S}^2(x)}{b}} \leq x_{avg} \tag{3}$$

where $t_{b-1,1-\alpha}$ is the $1-\alpha$ quantile of the t-distribution with $b-1$ degrees of freedom. If (3) holds, then the requirement is met with a probability of at least $1-\alpha$. In summary, validation of average values uses standard measures of steady state simulation which use one sided confidence intervals.

The situation is different for the remaining measures. In these cases it has to be decided at a per service call level whether the call meets the SLA or not. As already argued, it is unrealistic that in a complex SOA all requirements will always be met. Thus, the question is how many calls violate an SLA. Under stochastic assumption the probability of violating an SLA is a random variable and for the expectation of this random variable (i.e., the probability of SLA violation) an estimate and confidence intervals can be computed if we assume again that the model is in steady state.

We denote by $scall_i$ the ith service call and begin with the requirements $(scall_i.at - scall_{i-1}.at) \geq t_{min}$ and $scall_i.st \leq t_{max}$. In both cases, we can define a variable x_i which becomes 1 if the requirement is met and 0 otherwise. The estimator for mean (1) and variance (2) can then be used and

$$\bar{x} \pm t_{b-1,1-\alpha/2}\sqrt{\frac{\hat{S}^2(x)}{b}} \tag{4}$$

is the two sided confidence interval for the probability that a call fulfills the SLA. Again the result is computed using standard means for simulation output analysis and the variables x_i.

The remaining measures require access to the last m service calls for their evaluation. $(\Delta t, m)$ is observed if $scall_i.at - scall_{i-m-1}.at > \Delta t$, for $i < m+1$ the SLA is always true. $(t_{min} \wedge (\Delta t, m))$ holds if both conditions hold. Similarly, (t_{max}, n, m) holds if

$$\sum_{j=0}^{m-1} \delta(scall_{i-j}.st \leq t_{max}) \geq n$$

where $\delta(x)$ is 1 for $x = true$ and 0 otherwise. Again $scall_i.st \leq t_{max}$ is true for $i \leq 0$. Finally, (n,m) holds for the ith service call if

$$\sum_{j=0}^{m-1} \delta(scall_{i-j}.success) \geq n \ .$$

where $scall_i.success$ is true for $i \leq 0$.

For each measure and service call a variable x_i is defined which becomes 1 if the SLA holds and 0 otherwise. With (1), (2) and (4) an estimate and confidence intervals for the probability that the SLA is fulfilled can be computed. If the user defines lower bounds for the probability that an SLA is fulfilled, then one sided confidence intervals (3) can be used to validate these bounds similar to the steady state case. Let $\bar{x}$ and $\hat{S}^2(n)$ be the estimated mean and variance of measure X. Let x^{up} be the upper bound of the measure that can be tolerated according to the SLA. Assume that $\bar{x} < x^{up}$ since otherwise the SLA holds with a probability of less than 50%. For $\bar{x} < x^{up}$ the SLA holds for the system with probability $1-\alpha$ where α is given by

$$\alpha = \arg\min_{0<\beta<0.5}\left(\bar{x} + t_{b-1,1-\beta}\sqrt{\frac{\hat{S}^2(n)}{n}} \leq x^{up}\right)$$

In a large SOA usually different SLAs have to be validated and for each SLA the load and performance/reliability parts have to be evaluated separately. If K is the number of measures that are evaluated and α_k is the significance probability for the kth measure, then the probability that all confidence intervals contain the *correct* results can be computed from the

Bonferroni inequality (Law and Kelton 2000) and is given by

$$1 - \sum_{k=1}^{K} \alpha_k \ .$$

This shows that for a large number of measures α_k has to be small to give meaningful results.

5 MODELING OF SLAs USING HIERARCHICAL PROCESS CHAINS

The general approach for the integration of SLAs into an existing *ProC/B* model is shown in Figure 4. Parts of the original model are marked in black in Figure 4 and the new parts that are necessary for analyzing the SLAs are marked in red.

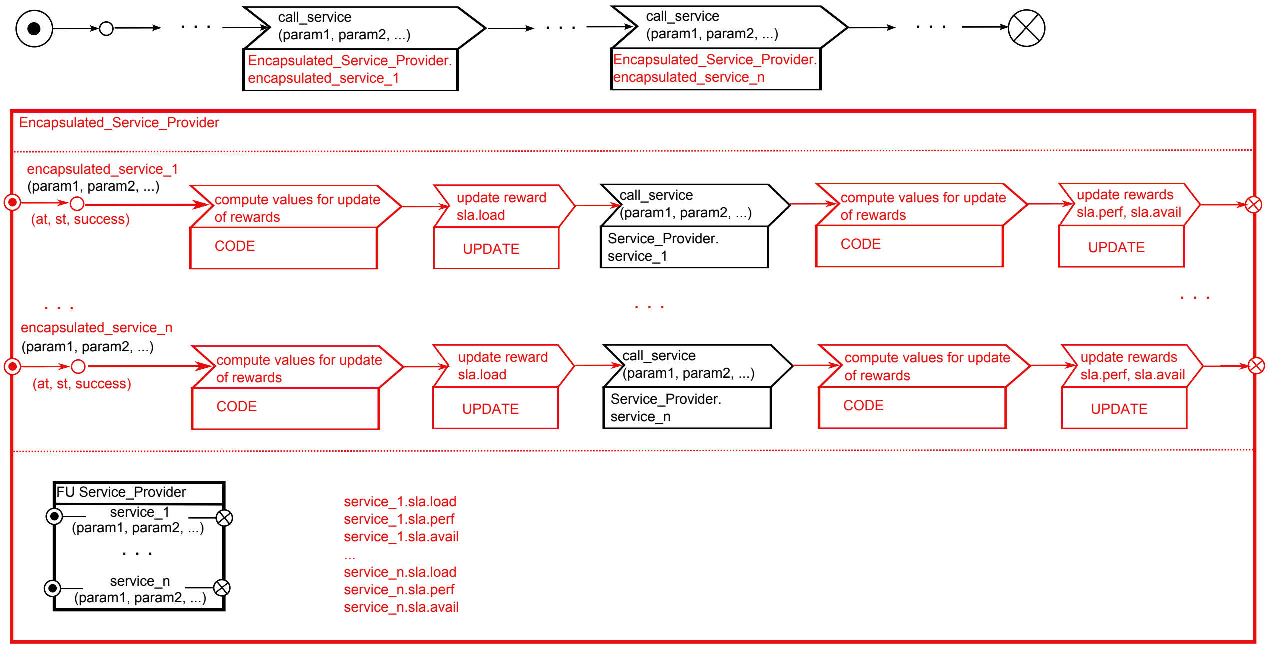

Figure 4: Integration of SLAs into *ProC/B* models

Assume that the upper process chain in Figure 4 calls services of the FU Service_Provider in the original model and we want to analyze the SLAs of Service_Provider. For the specification of the SLA variables and for updating the corresponding measures described in Sect. 3 and 4 we introduce a new layer in the model hierarchy (FU Encapsulated_Service_Provider) that encapsulates the FU Service_Provider. For each service of FU Service_Provider a service with the same set of parameters is provided by Encapsulated_Service_Provider. Those new services forward a call to the corresponding service of FU Service_Provider and compute all the values necessary for updating the rewards. Therefore each of the services has three local variables for storing the arrival time (at), the sojourn time (st) and whether the service call was successful (success). Additionally, three types of rewards are defined for each SLA (sla.load, sla.perf and sla.avail).

The process chains that try to call a service of the FU Service_Provider do not call this service directly anymore, instead they use the corresponding service of Encapsulated_Service_Provider as shown in Figure 4. The service of Encapsulated_Service_Provider computes values to update the rewards and performs those updates before and after the call to the service of Service_Provider. When a service of Encapsulated_Service_Provider is called it stores the arrival time and updates sla.load. After the forwarded call to Service_Provider returns sla.perf and sla.avail are updated. The actual operations for the computation of update values and the updates of the rewards are only indicated by the CODE and UPDATE PCEs in Figure 4.

The described approach can be used for the integration of SLAs in any FU of a *ProC/B* model and furthermore it can be applied automatically. Of course, some of the reward and variable declarations can be omitted in cases where not all measures for the SLA are of interest. In the following we will clarify the technique by applying it to the model from

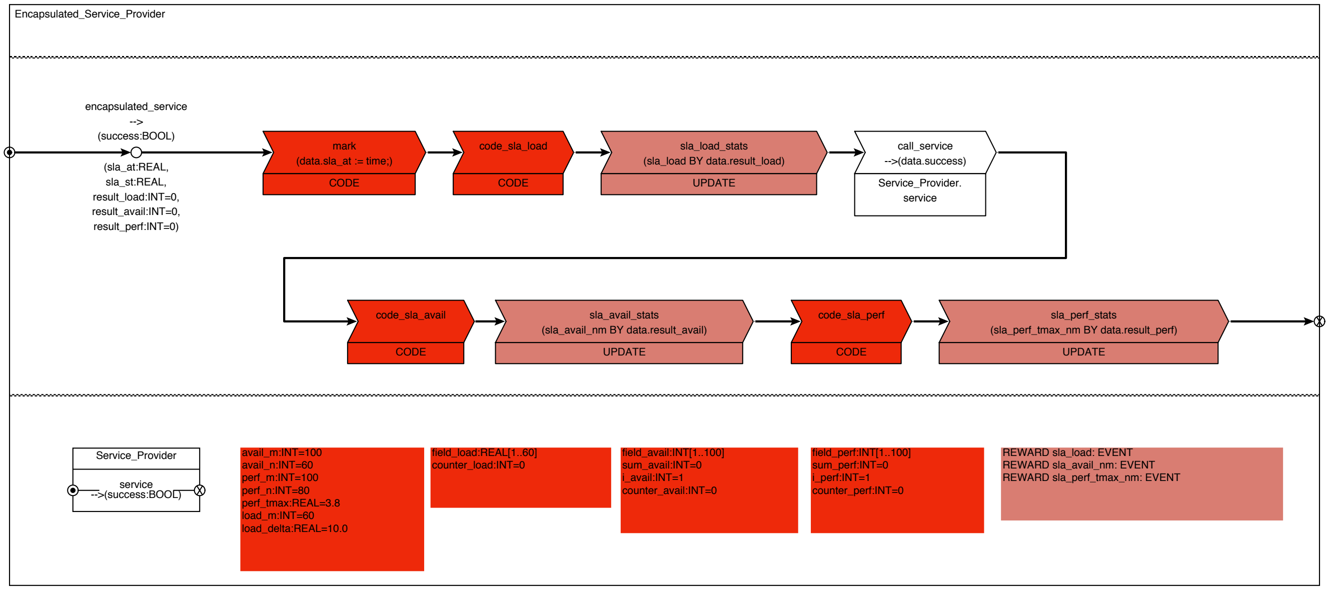

Figure 5: Encapsulated version of FU `Service_Provider`

Sect. 2. We extended the `Service_Provider` example of Figure 1 with an additional layer to validate an SLA by simulation. Figure 5 shows FU `Encapsulated_Service_Provider` added into the model hierarchy to validate the SLAs for `service` in FU `Service_Provider`. Additional elements are marked in red similar to Figure 4. It contains one process chain, the FU `Service_Provider`, global variables and a set of rewards. A possible SLA for the example is given by sla.load $:= (\Delta t = 10, m = 60)$, sla.perf $:= (t_{max} = 3.8, n = 80, m = 100)$ and sla.avail $:= (60, 100)$ which should be satisfied for at least 90% of the service calls. In the following we show how the model can be enhanced to check these specific SLA measures.

The first block of variables in the lower half of the FU expresses the SLA with constant values. They are prefixed with the names of their respective SLA part. The process chain `encapsulated_service` consists of eight PCEs with the call of the evaluated service `call_service` in the middle and it also defines six variables associated with each process. Variables `sla_at`, `sla_st` and `success` hold arrival time, sojourn time and the success flag as defined in Sect. 3. Their concrete values are set in code PCEs along the process chain. Code PCEs are used like normal PCEs but their behavior is described by a segment of code which can be interpreted by the used analysis tool. PCE `mark` sets `sla_at` of each process to the current model time. The load part of the SLA is evaluated in PCE `code_sla_load`. It is located before the service call to measure the arrival rate before the time intervals between processes are changed by the service. A corresponding UPDATE PCE `sla_load_stats` will update the bit sequence in reward `sla_load` with 1 when the time constraints hold and with 0 otherwise. After the actual call of the service in `call_service` the SLA parts sla.avail and sla.perf related to the performance of the encapsulated service are evaluated and the results are used to update the rewards `sla_avail_nm` and `sla_perf_tmax_nm`.

The common principle in all code PCEs of this example is to retain the performance of the service over the last m process arrivals in an array prefixed `field_` of size m storing sufficient recent values of the matching indicator. The array is used like a ring buffer, on each process arrival the cycling array position pointer is calculated with a process counter `counter_` modulo m and a new value is inserted replacing an old one. The decision whether the SLA specification is met by a process is based on the array values after the update. The boolean result is stored in a process variable prefixed `result_` to feed the following update PCE.

Listing 1 shows the code placed into PCE `code_sla_avail` to evaluate sla.avail. The local process counter is incremented at first. The conditional block updates one cell of the array `field_avail` to 1 if the service call of the actual process was successful and 0 otherwise.

The availability of the last m processes is based on two conditions. When there are less than m collected values in the array the availability indicator `result_avail` is 1 by definition. In every other case the array contents are summed up and required to be greater than `avail_n`. The same principle is used to determine values for sla.perf in PCE `code_sla_perf`

Listing 1: Source of `code_sla_avail` in *ProC/B* syntax

```
counter_avail := counter_avail + 1;

IF data.success THEN
  field_avail[(counter_avail MOD avail_m) + 1] := 1;
ELSE
  field_avail[(counter_avail MOD avail_m) + 1] := 0;
END IF;

IF counter_avail < avail_m THEN
  data.result_avail := 1;
ELSE
  sum_avail := 0;
  i_avail := 1;

  WHILE i_avail <= avail_m
  LOOP
    sum_avail := sum_avail + field_avail[i_avail];
    i_avail := i_avail +1;
  END LOOP;

  IF sum_avail < avail_n THEN
    data.result_avail := 0;
  ELSE
    data.result_avail := 1;
  END IF;
END IF;
```

Listing 2: Source of `code_sla_perf` in *ProC/B* syntax

```
data.sla_st := time - data.sla_at;

IF data.success THEN
  counter_perf := counter_perf + 1;
  IF data.sla_st <= perf_tmax THEN
    field_perf[(counter_perf MOD perf_m) +1] := 1;
  ELSE
    field_perf[(counter_perf MOD perf_m) +1] := 0;
  END IF;
  IF counter_perf < perf_m THEN
    data.result_perf := 1;
  ELSE
    sum_perf := 0;
    i_perf := 1;
    WHILE i_perf <= perf_m
    LOOP
      sum_perf := sum_perf + field_perf[i_perf];
      i_perf := i_perf + 1;
    END LOOP;
    IF sum_perf < perf_n THEN
      data.result_perf := 0;
    ELSE
      data.result_perf := 1;
    END IF;
  END IF;
END IF;
```

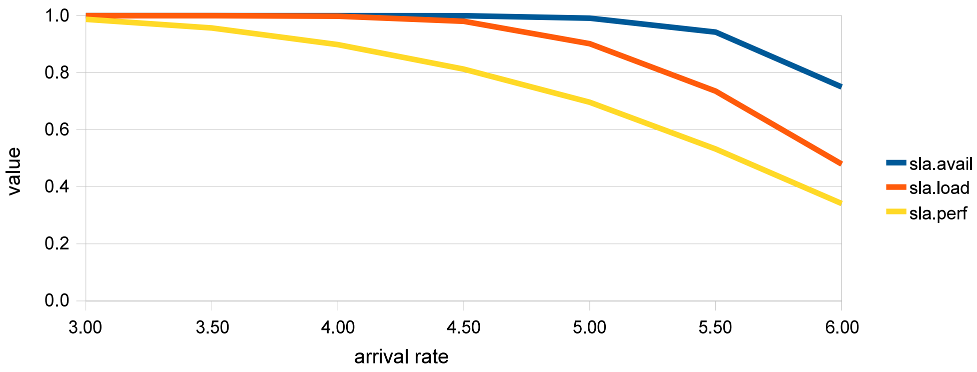

Figure 6: Measurements of the service running on 8 processors

(Listing 2). The first line sets the sojourn time of the process. A significant difference in sla.perf is that only successful service calls are evaluated, so the process counter is only updated if `data.success` is true. The following condition will mark the array depending on `data.sla_st` and t_{max}. Again, when there are not enough collected values the performance indicator `result_perf` is set to 1, otherwise the sum of the array values is considered.

We analyzed the described model by simulation to validate the SLA as described above and made several experiments with different arrival rates and simulated each setting for 100,000 units model time. The results are shown in Table 1 and are also visualized in Figure 6. The row *confidence 90%* shows the width of the *90%* confidence interval of the mean value.

At the relatively modest arrival rate of 3.5 all parts of the SLA are met for more than 95% of all processes. A slight increase to 4.0 shows a first decline of sla.perf to 89.8% which violates the SLA. At an arrival rate of 5.0 the performance part drops below 70% although sla.load is still within specification. Obviously the service offered in this model is unable to fulfill the SLA specification.

Simulation models can also be used as a testbed to modify the service until it can provide the SLA. The original service uses eight processors to execute calculations (cf. Figure 2). As there is a performance problem with the processors, two additional cores are added to the system. Table 2 contains measurements of a 10 processor system taken under equal conditions as before. The speed gain of additional processors removes the performance problems and helps the service to meet all requirements set by the SLA. At an arrival rate of 5.0 the performance delivered by the service is accepted while

Table 1: Service Levels with 8 processors

	mean interarrival time	3.00^{-1}	3.50^{-1}	4.00^{-1}	4.50^{-1}	5.00^{-1}	5.50^{-1}	6.00^{-1}
sla.avail	mean	0.999894	0.999826	0.999868	0.999656	0.991132	0.942338	0.750041
	standard deviation	0.010311	0.013172	0.011501	0.018545	0.093749	0.233103	0.432989
	confidence 90%	0.02%	0.04%	0.03%	0.04%	0.18%	0.44%	1.14%
sla.load	mean	1.000000	0.999949	0.998018	0.980323	0.901595	0.735159	0.479551
	standard deviation	0.000000	0.007156	0.044472	0.138888	0.297862	0.441248	0.499582
	confidence 90%	0.00%	0.00%	0.05%	0.18%	0.48%	0.93%	1.61%
sla.perf	mean	0.987235	0.956688	0.898489	0.812611	0.696342	0.532435	0.341438
	standard deviation	0.112259	0.203559	0.302004	0.390224	0.459837	0.498947	0.474192
	confidence 90%	0.07%	0.13%	0.19%	0.29%	0.47%	0.84%	1.44%

Table 2: Service Levels with 10 processors

	mean interarrival time	3.00^{-1}	3.50^{-1}	4.00^{-1}	4.50^{-1}	5.00^{-1}	5.50^{-1}	6.00^{-1}
sla.avail	mean	1.000000	1.000000	1.000000	1.000000	0.999892	0.999773	0.998509
	standard deviation	0.000000	0.000000	0.000000	0.000000	0.010393	0.015055	0.038582
	confidence 90%	0.00%	0.00%	0.00%	0.00%	0.02%	0.03%	0.05%
sla.load	mean	1.000000	0.999937	0.997751	0.981663	0.907546	0.725280	0.486045
	standard deviation	0.000000	0.007920	0.047367	0.134168	0.289666	0.446373	0.499805
	confidence 90%	0.00%	0.01%	0.06%	0.17%	0.45%	0.94%	1.58%
sla.perf	mean	0.999030	0.995105	0.983182	0.954922	0.909562	0.849661	0.792968
	standard deviation	0.031131	0.069795	0.128589	0.207474	0.286808	0.357403	0.405179
	confidence 90%	0.02%	0.04%	0.08%	0.12%	0.16%	0.20%	0.21%

sla.load matches the mark of 90% closely. sla.avail is also advanced as service requests are no longer blocked at the processing unit and resources of FU `Transaction` are released earlier.

6 CONCLUSIONS

In this paper we presented an approach to model and validate service level agreements for service oriented architectures using hierarchical process chain models. The validation is done by simulation giving the service provider an effective and cost-efficient opportunity to consider different scenarios. E.g., as illustrated by the example of Sect. 5, the service provider might concentrate on his equipment and possible investments or he might focus on negotiating customer demands.

We showed how such scenarios can be validated using the *ProC/B* modeling environment. Currently the SLA specifications and corresponding rewards have to be manually integrated into existing *ProC/B* models. Future work is intended to automate this process. Furthermore, SLA enhanced simulation models can also be used to support monitorability (cf. (Skene et al. 2007, Raimondi, Skene, and Emmerich 2008)) helping to identify relevant points for measurements.

REFERENCES

Bause, F., H. Beilner, M. Fischer, P. Kemper, and M. Völker. 2002. The ProC/B toolset for the modelling and analysis of process chains. In *Computer Performance Evaluation / TOOLS*, ed. T. Field, P. G. Harrison, J. T. Bradley, and U. Harder, Volume 2324 of *Lecture Notes in Computer Science*, 51–70: Springer.

Bause, F., P. Buchholz, J. Kriege, and S. Vastag. 2008a. A framework for simulation models of service-oriented architectures. In *SIPEW*, ed. S. Kounev, I. Gorton, and K. Sachs, Volume 5119 of *Lecture Notes in Computer Science*, 208–227: Springer.

Bause, F., P. Buchholz, J. Kriege, and S. Vastag. 2008b. Simulating process chain models with OMNeT++. In *Proc. of 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008)*. Marseille.

Collaborative Research Center 559. Modelling of large logistics networks. http://www.sfb559.uni-dortmund.de.

Dwyer, M. B., G. S. Avrunin, and J. C. Corbett. 1999. Patterns in property specifications for finite-state verification. In *ICSE*, 411–420.

Konrad, S., and B. H. C. Cheng. 2005. Real-time specification patterns. In *ICSE*, ed. G.-C. Roman, W. G. Griswold, and B. Nuseibeh, 372–381: ACM.

Kuhn, A. 1995. *Prozessketten in der Logistik - Entwicklungstrends und Umsetzungsstrategien*. Dortmund: Verlag Praxiswissen.

Kuhn, A. 1999. *Prozesskettenmanagement - Erfolgsbeispiele aus der Praxis*. Dortmund: Verlag Praxiswissen.

Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis*. Wiley.

Ludwig, H., A. Keller, A. Dan, R. P. King, and R. Franck. 2003. Web service level agreement (WSLA) language specification. http://www.research.ibm.com/wsla.

Menascé, D. A., H. Ruan, and H. Gomaa. 2007. QoS management in service-oriented architectures. *Perform. Eval.* 64 (7-8): 646–663.

Peltz, C. 2003. Web services orchestration and choreography. *IEEE Computer* 36 (10): 46–52.

Raimondi, F., J. Skene, and W. Emmerich. 2008. Efficient online monitoring of web-service SLAs. In *SIGSOFT FSE*, ed. M. J. Harrold and G. C. Murphy, 170–180: ACM.

Sarjoughian, H., S. Kim, M. Ramaswamy, and S. Yau. 2008. A simulation framework for service-oriented computing systems. In *Proceedings of the 2008 Winter Simulation Conference*, ed. S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, and J. W. Fowler, 845–853. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Skene, J., D. D. Lamanna, and W. Emmerich. 2004. Precise service level agreements. In *ICSE*, 179–188: IEEE Computer Society.

Skene, J., A. Skene, J. Crampton, and W. Emmerich. 2007. The monitorability of service-level agreements for application-service provision. In *WOSP*, ed. V. Cortellessa, S. Uchitel, and D. Yankelevich, 3–14: ACM.

Teyssié, C. 2006. UML-based specification of QoS contract negotiation and service level agreements. In *ICN/ICONS/MCL*, 12: IEEE Computer Society.

Trienekens, J. J. M., J. J. Bouman, and M. van der Zwan. 2004. Specification of service level agreements: Problems, principles and practices. *Software Quality Journal* 12 (1): 43–57.

AUTHOR BIOGRAPHIES

FALKO BAUSE holds a Doctoral degree in computer science from the TU Dortmund. His main research interests are in the area of system engineering with emphasis on Stochastic Petri Nets. He defined the Queueing Petri Net formalism, which combines Queueing Networks with Stochastic Petri Nets and coauthored a book with the title "Stochastic Petri Nets – An Introduction to the Theory". His e-mail address is <falko.bause@udo.edu>.

PETER BUCHHOLZ received the Diploma degree (1987), the Doctoral degree (1991) and the Habilitation degree (1996) all from the TU Dortmund, where he is currently a professor for modeling and simulation. His current research interests are efficient techniques for the analysis of stochastic models, formal methods for the analysis of discrete event systems, the development of modeling tools, as well as performance and dependability analysis of computer and communication systems. His e-mail address is <peter.buchholz@udo.edu>.

JAN KRIEGE received the Diploma degree in computer science from the TU Dortmund in 2006. His research interests include the modeling and analysis of logistics networks and computer and communication systems. His e-mail address is <jan.kriege@udo.edu>.

SEBASTIAN VASTAG is a research assistant at the chair for quantitative techniques in computer science at the TU Dortmund, Germany. He received the Diploma degree in computer science in 2006. His research topics are modeling, analysis and validation of logistic process models. His e-mail address is <sebastian.vastag@udo.edu>.