Statistical Model Checking of Response Times for Different System Deployments

Bernhard K. Aichernig[1], Severin Kann[2], and Richard Schumi[1]

[1] Institute of Software Technology, Graz University of Technology, Austria
{aichernig, rschumi}@ist.tugraz.at
[2] AVL List GmbH, Graz, Austria
Severin.Kann@avl.com

Abstract. Performance testing is becoming increasingly important for interactive systems. Evaluating their performance with respect to user expectations is complex, especially for different system deployments. Various load-testing approaches and performance-simulation methods aim at such analyses. However, these techniques have certain disadvantages, like a high testing effort for load testing, and a questionable model accuracy for simulation methods. Hence, we propose a combination of both techniques. We apply statistical model checking with a learned timed model and evaluate the results on the real system with hypothesis testing. Moreover, we check the established hypotheses of a reference system on various system deployments (configurations), like different hardware or network settings, and analyse the influence on the performance. Our method is realised with a property-based testing tool that is extended with algorithms from statistical model checking. We illustrate the feasibility of our technique with an industrial case study of a web application.

Keywords: statistical model checking, model-based testing, system deployments, property-based testing, performance, response time, stochastic user profiles, web-service application.

1 Introduction

Analysing the performance of a system for specific usage scenarios is a difficult task. It becomes even more cumbersome, when a system is deployed to customers and should still provide certain performance properties for different hardware or network settings. We propose a performance evaluation method that applies statistical model checking (SMC) on a timed model from a reference system in order to derive hypotheses that allow us to verify system deployments. SMC [1] is an evaluation method that answers qualitative or quantitative questions, which are expressed as properties of a stochastic model or system. In contrast to existing load-testing approaches, we can perform a fast evaluation with a model, and in contrast to model-based methods, we verify the results of the model evaluation on real systems.

Our approach is realised with a property-based testing (PBT) tool that was extended with SMC algorithms [5]. PBT is a random testing method that tries

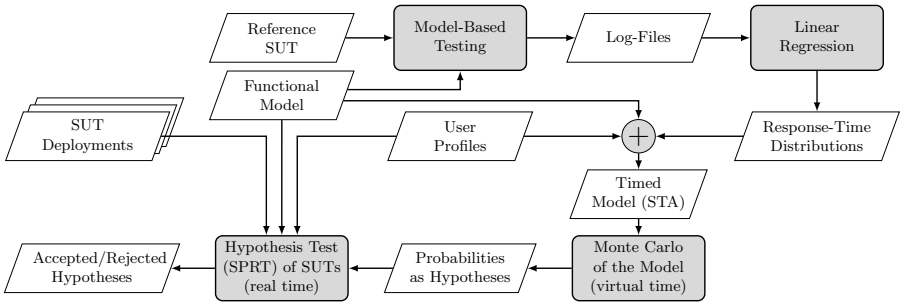

Fig. 1: Overview of the data flow of our method.

to falsify properties that define the expected behaviour of a system-under-test (SUT). PBT tools generate random inputs and check if a property holds.

Previously, we have demonstrated how learned models can be applied to estimate the probability that a system can satisfy certain response-time thresholds, and we verified the resulting probability with hypothesis testing on the SUT. Now, we derive hypotheses about the response time from a reference SUT, in order to check, if deployments of this SUT with a different hardware/network setup have a similar performance, i.e. they satisfy the same hypotheses.

The process of our method is illustrated in Fig. 1. (1) We perform model-based testing with a functional model and capture the response times of requests of a reference SUT as log-data. We run multiple testing processes concurrently in order to obtain response times for simultaneous requests. (2) The log-files are then taken as input for a linear regression, which gives us response-time distributions. (3) These distributions and stochastic user profiles are integrated into the functional model, resulting in a combined stochastic timed automata (STA) [8] model. (4) Next, we perform a Monte Carlo simulation of this model in order to obtain answers for the question: "What is the probability that the response time of each user within a user population is under a certain threshold?". (5) The resulting probabilities serve us as hypotheses in order to check if deployments of the SUT can satisfy the same response-time thresholds as the reference system. We test the deployments with the sequential probability ratio test (SPRT) [38], a form of hypothesis testing that can usually be performed with fewer samples than a Monte Carlo simulation.

Related Work. A number of related approaches in the area of PBT are concerned with testing concurrent software [16,22,31]. The closest related work we found in this area was from Arts [7]. It shows a load-testing approach with QuickCheck that can run user scenarios on an SUT in order to determine the maximum supported number of users. In contrast to our approach, Arts does not consider stochastic user profiles and the user scenarios are only tested on an SUT, but not simulated at model-level.

Related work is also in the area of load testing [9,29]. For example, Draheim et al. [17] demonstrated a load-testing approach that simulates realistic user behaviour with stochastic models. Moreover, a number of related tools, like Neoload perform load testing with user populations [35]. In contrast to our work, load

testing is mostly performed directly on an SUT. With our approach, we want to simulate user populations on the model-level as well. There are also many approaches that focus only on a simulation on the model-level [10,12,14,27,32,40], but with our method we can also directly test an SUT within the same tool.

Another domain with related work, is deployment testing. For example, various related approaches apply a performance analysis of system deployments [28,36,39]. However, in contrast to our work, they do not apply a model that is derived from a reference SUT in order to evaluate the performance of SUT deployments under specific usage scenarios.

The most related tool is UPPAAL SMC [13]. Similar to our approach, it provides SMC of priced timed automata, which can simulate user populations. It also supports testing implementations, but for this a test adapter is required, which, e.g., handles the form-data creation. In contrast, we can use PBT features, like data generators in order to automatically generate form data, and we can model in a programming language. This helps testers, who are already familiar with this language, since they do not have to learn new notations.

To the best of our knowledge our work is novel: no other work performs SMC on a learned timed model of a reference SUT to derive hypotheses that are verified on SUT deployments in order to check, if they provide comparable response times for given user profiles.

Contribution. This paper builds upon our previous work [37], where we introduced our model-based prediction method that enables an efficient test of the predictions on the SUT. We evaluated this approach on an industrial web-service application. However, it was only tested on one reference system without any deployments. Hence, this work presents the following novel contributions. (1) The major contribution is the new application of our method to analyse the performance of system deployments applying hypotheses that were derived from a reference SUT. This allows software companies to give their customers recommendations for the hardware/network requirements of a system that should satisfy certain performance properties. (2) Another contribution is the additional evaluation of our method by applying it to several deployments of an industrial web-service application. This helps to find possible limitations of our approach and emphasizes its generality.

Structure. First, Sect. 2 introduces the background of SMC, PBT and stochastic timed automata based on previous work [37]. Then, in Sect. 3 we illustrate our method with an example. In Sect. 4, we evaluate our approach with an industrial web-service application as reference SUT, and we check multiple deployments with other hardware/network configurations. Finally, we conclude in Sect. 5.

2 Background

2.1 Statistical Model Checking (SMC)

SMC is a verification method that evaluates certain properties of a stochastic model. These properties are usually defined with (temporal) logics, and they

can describe quantitative and qualitative questions. For example, questions, like "What is the probability that the model satisfies a property?" or "Is the probability that the model satisfies a property above or below a certain threshold?". In order to answer such questions, a statistical model checker produces samples, i.e. random walks on the stochastic model and checks whether the property holds for these samples. Various SMC algorithms are applied, in order to compute the total number of samples needed to find an answer for a specific question, or to compute a stopping criterion. This criterion determines when we can stop sampling, because we have found an answer with a required certainty. In this work, we focus on the following algorithms, which are commonly used in the SMC literature [13,25,26].

Monte Carlo simulation with Chernoff-Hoeffding bound. The algorithm computes the required number of simulations n in order to estimate the probability γ that a stochastic model satisfies a Boolean property. The procedure is based on the Chernoff-Hoeffding bound [20] that provides a lower limit for the probability that the estimation error is below a value ϵ. Assuming a confidence $1-\delta$, the required number of simulations n can be calculated as follows:

$$n \geq \frac{1}{2\epsilon^2} \ln\left(\frac{2}{\delta}\right)$$

The n simulations represent Bernoulli random variables $X_1, \ldots, X_n$ with outcome $x_i = 1$, if the property holds for the i-th simulation run, and $x_i = 0$ otherwise. Let the estimated probability be $\bar{\gamma}_n = (\sum_{i=1}^{n} x_i)/n$, then the probability that the estimation error is below ϵ is greater than our required confidence. Formally, we have: $Pr(|\bar{\gamma}_n - \gamma| \leq \epsilon) \geq 1-\delta$. After the calculation of the number of samples n, a simple Monte Carlo simulation is performed [26].

Sequential Probability Ratio Test (SPRT). This sequential method [38] is a form of hypothesis testing, which can answer qualitative questions. Given a random variable X with a probability density function $f(x,\theta)$, we want to decide, whether a null hypothesis $H_0 : \theta = \theta_0$ or an alternative hypothesis $H_1 : \theta = \theta_1$ is true for desired type I and II errors (α, β). In order to make the decision, we start sampling and calculate the log-likelihood ratio after each observation of x_i:

$$\log \Lambda_m = \log \frac{p_1^m}{p_0^m} = \log \frac{\prod_{i=1}^{m} f(x_i,\theta_1)}{\prod_{i=1}^{m} f(x_i,\theta_0)} = \sum_{i=1}^{m} \log \frac{f(x_i,\theta_1)}{f(x_i,\theta_0)}$$

We continue sampling as long as $\log \frac{\beta}{1-\alpha} < \log \Lambda_m < \log \frac{1-\beta}{\alpha}$. H_1 is accepted when $\log \Lambda_m \geq \log \frac{1-\beta}{\alpha}$, and H_0 when $\log \Lambda_m \leq \log \frac{\beta}{1-\alpha}$ [18].

In this work, we form a hypothesis about the expected response time with the Monte Carlo method on the model. Then, we check with the SPRT if this hypothesis holds on a deployment of the SUT. This is faster than running a Monte Carlo simulation directly on the deployment, since the SPRT requires a far lower number of samples, i.e. test cases.

2.2 Property-Based Testing (PBT)

PBT is a random-testing technique that aims to check the correctness of properties. A property is a high-level specification of the expected behaviour of a function-under-test that should always hold. For example, the length of a concatenated list is always equal to the sum of lengths of its sub-lists:

$$\forall\, l_1, l_2 \in \mathit{Lists}[T] : \mathit{length}(\mathit{concatenate}(l_1, l_2)) = \mathit{length}(l_1) + \mathit{length}(l_2)$$

With PBT, we automatically generate inputs for such a property by applying data generators, e.g., the random list generator. The inputs are fed to the function-under-test and the property is evaluated. If it holds, then this indicates that the function works as expected, otherwise a counterexample is produced.

PBT also supports model-based testing. Models encoded as extended finite state machines (EFSMs) [23] can serve as source for state-machine properties. An EFSM is a 6-tuple (S, s_0, V, I, O, T). S is a finite set of states, $s_0 \in S$ is the initial state, V is a finite set of variables, I is a finite set of inputs, O is a finite set of outputs, T is a finite set of transitions. A transition $t \in T$ is a 5-tuple (s_s, i, g, op, s_t), s_s is the source state, i is an input, g is a guard, op is a sequence of output and assignment operations, s_t is the target state [23]. In order to derive a state-machine property from an EFSM, we have to write a specification comprising the initial state, commands and a generator for the next transition given the current state of the model. Commands encapsulate (1) preconditions that define the permitted transition sequences, (2) postconditions that specify the expected behaviour and (3) execution semantics of transitions for the model and the SUT. A state-machine property states that for all permitted transition sequences, the postcondition must hold after the execution of each transition resp. command [21,33]. Simply put, such properties can be defined as follows:

$$\begin{aligned}
&\mathit{cmd.runModel}, \mathit{cmd.runActual} : S \times I \to S \times O \\
&\mathit{cmd.pre} : I \times S \to \mathit{Boolean}, \mathit{cmd.post} : (S \times O) \times (S \times O) \to \mathit{Boolean} \\
&\forall s \in S, i \in I, \mathit{cmd} \in \mathit{Cmds} : \\
&\quad \mathit{cmd.pre}(i, s) \implies \mathit{cmd.post}(\mathit{cmd.runModel}(i, s), \mathit{cmd.runActual}(i, s))
\end{aligned}$$

There are two functions to execute a command on the model and on the SUT: *cmd.runModel* and *cmd.runActual*. The precondition *cmd.pre* defines the valid inputs for a command. The postcondition *cmd.post* compares the outputs and states of the model and the SUT after the execution of a command.

PBT is a powerful testing technique that allows a flexible definition of generators and properties via inheritance or composition. The first implementation of PBT was QuickCheck for Haskell [15]. Numerous reimplementations followed for other programming languages, like Hypothesis[3] for Python or ScalaCheck [30]. We developed our method with FsCheck[4]. FsCheck is a .NET port of QuickCheck with influences of ScalaCheck. It supports a functional programming style with F# and an object-oriented style with C#. We work with C#, since it is the programming language of our SUT.

[3] `https://pypi.python.org/pypi/hypothesis`
[4] `https://fscheck.github.io/FsCheck`

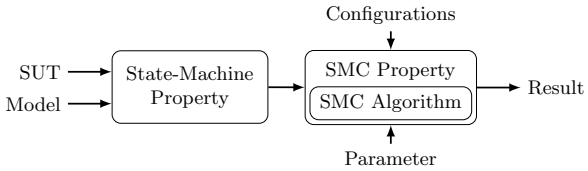

Fig. 2: Data flow diagram of an SMC property.

2.3 Stochastic Timed Automata

Timed automata (TA) were originally introduced by Alur and Dill [6]. Several extensions of TA have been proposed, including stochastically enhanced TA [11] and continuous probabilistic TA [24]. We follow the definition of stochastic timed automata (STA) by Ballarini et al. [8]: An STA can be expressed as a tuple $(L, l_0, A, C, I, E, F, W)$, where the first part is a normal TA (L, l_0, A, C, I, E) and additionally it contains probability density functions $F = (f_l)_{l \in L}$ for the sojourn time and natural weights $W = (w_e)_{e \in E}$ for the edges. L is a finite set of locations, $l_0 \in L$ is the initial location, A is a finite set of actions, C is a finite set of clocks with real-valued valuations $u(c) \in \mathbb{R}_{>0}$, $I : L \mapsto \mathcal{B}(C)$ is a finite set of invariants for the locations and $E \subseteq L \times A \times \mathcal{B}(C) \times 2^C \times L$ is a finite set of transitions between locations, with an action, a guard and a set of clock resets. The transition relation can be described as follows. For a state (l, u), where $l \in L$ is a location and $u \in C \to \mathbb{R}_{\geq 0}$ is a clock valuation, the probability density functions f_l is used to choose the sojourn time d, which changes the state to $(l, u + d)$, where $u + d$ means that the clock valuation is changed $(u + d)(c) = u(c) + d$ for all $c \in C$. After this change, an edge e is selected out of the set of enabled edges $E(l, u + d)$ with the probability $w_e / \sum_{h \in E(l,u+d)} w_h$. Then, a transition to the target location l' of e and $u' = u + d$ is performed. For our models, the underlying stochastic process is a semi-Markov process since the clocks are reset at every transition, but we do not assume exponential delays, and therefore, it is not a standard continuous-time Markov chain.

2.4 Integration of SMC into Property-Based Testing

We have demonstrated that SMC can be integrated into a PBT tool in order to perform SMC of PBT properties [3,5], which were explained in Sect. 2.2. These PBT properties can be evaluated on stochastic models, like in classical SMC, as well as on stochastic implementations. For the integration, we introduced our own new SMC properties, which take a PBT property, configurations for the PBT execution, and parameters for the specific SMC algorithm as input. Then, our properties perform an SMC algorithm by utilizing the PBT tool as simulation environment and they return either a quantitative or qualitative result, depending on the algorithm. Figure 2 shows how we can evaluate a state-machine property within an SMC property. Such a state-machine property can, e.g., be applied for a statistical conformance analysis by comparing an ideal model to a stochastic faulty implementation or it can also simulate a stochastic model. We evaluated our SMC properties by repeating case studies from the SMC literature and we were able to reproduce the results.

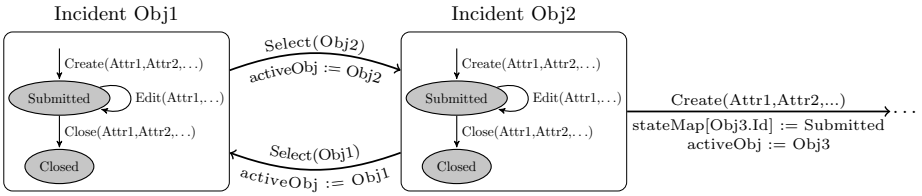

Fig. 3: Functional EFSM model of the incident manager [4].

3 Method

In this section, we present the necessary steps to apply our method, based on the overview presented in Fig. 1. The steps are illustrated with an example of an incident manager that was introduced in our previous work [2,4].

Model-Based Testing. First, we perform model-based testing within PBT in order to produce log-data. This initial testing phase is conducted with a functional EFSM model. The SUT is a web-based tool that supports tasks, like creating, editing or closing incident objects, which can, e.g., be bug reports. These objects include attributes (form data) that are stored in a database. We automatically generate data for the attributes with PBT generators. An example model of the incident manager is illustrated in Fig. 3. This model is a hierarchical state machine [19]. There are sub-state machines for each incident object and select transitions can switch between these objects. We have a variable *activeObj* that identifies the currently open incident and a map (*stateMap*) that has an object identifier as key and stores the state of all incidents. Each sub-state machine shows the tasks that can be performed for an incident object. In reality, each of these tasks represents a page of the incident manager with required form fields (attributes). Hence, the transitions are parametrised with attributes [4].

With this functional model, we perform classical PBT, which generates random sequences of commands with form data. We run several testing processes concurrently in order to produce log-data that includes response times of simultaneous requests. Note that the tasks of the sub-state machines in Fig. 3 consist of multiple subtasks that are not shown in this figure for clarity. For example, there are subtasks for opening the page (*StartTask*), for setting attributes (*SetAttribute*), and for saving the page (*Commit*). Most of these subtasks are requests, hence, we record them in our logs. An example log from a non-productive test system with low computing resources (virtual machine) is represented in Table 1. The log contains response times of tasks, subtasks, attributes, states (*From, To*)

Table 1: Example log-data of the incident manager.

Task	**From**	**To**	**Subtask**	**#ActiveUsers**	**Attribute**	**ResponseTime[ms]**
Create	Global	Submitted	StartTask	7	-	334
Create	Global	Submitted	SetAttribute	8	Assignee	77
Edit	Submitted	Submitted	StartTask	5	-	286
Create	Global	Submitted	Commit	6	-	918
Edit	Submitted	Submitted	SetAttribute	4	TestOrder	347

and simultaneous requests (*#ActiveUsers*). For this initial testing phase, we selected transitions with a uniform distribution.

Linear Regression. We apply a multiple linear regression in order to obtain distributions for response times. First, we performed data cleaning and preprocessing, where we, e.g., filter outliers. For example, we are not interested in unusual long response times that are caused by network disruptions. Our aim is to maximise the user satisfaction. Hence, we are mainly interested in average response times under normal conditions, but not in worst-case scenarios. Next, we select the log variables, which have a high influence on the response time. This phase is called *feature selection*. This and all other steps are facilitated with the help of data visualisations, e.g., scatter plots or correlation matrices. Finally, we can run the linear regression by applying R, which is a standard statistics tool.[5] A more detailed description of the regression can be found in our previous work [37] and in the work of Rencher and Christensen [34].

As a result, we obtain estimates of the mean response time and standard errors for the regression variables. We combine these values with a linear combination to obtain parameters for the normal distribution for specific variable assignments. This combination is done inside the function *rtime* (response time). This function takes a task, a subtask the number of active users and an optional attribute as input and returns the parameters μ and σ of the normal distribution.

$$rtime : Task \times Subtask \times \mathbb{N}_{>0} \times Attribute \to \mathbb{R} \times \mathbb{R}$$

Monte Carlo Simulation. In order to apply SMC for a realistic usage scenario, we integrate a given user profile and response-time distributions from the linear regression into the functional model. An example user profile for the incident manager is shown in Listing 1.1. It includes weights for tasks, user input (waiting) time intervals between tasks/subtasks that represent the time that a user needs for the input and data specific waiting factors, e.g., a delay per character, or a delay per reference for the number of options of a drop-down menu.

The extension of the initial functional model with a user profile and response-time distributions gives us a combined model that is a stochastic timed automaton (Sect. 2.3). Figure 4 illustrates this automaton for one incident object. Note, we only show the combined model of one sub-state machine of the hierarchical EFSM in Fig. 3 for brevity. All locations (states) l in this combined model include a sojourn time that is defined with a probability density function f_l. The tasks of the functional model where separated into subtasks in order to represent the response times of individual requests. Each subtask comprises an

[5] `https://www.r-project.org`

```
{TaskWeights: { Create: 70, Edit: 45, Close: 25, Select: 30 },
TaskWaitMin:500,TaskWaitMax:1500, SubTaskWaitMin:300,SubTaskWaitMax:500,
WaitPerReference: 10, WaitPerCharacter: 30 }
```

Listing 1.1: User profile in the JavaScript Object Notation (JSON) format.

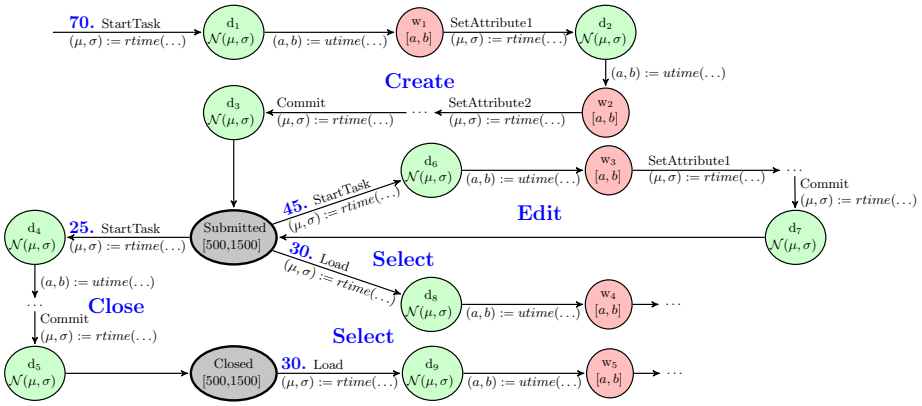

Fig. 4: Stochastic timed automata model for one incident object.

edge that calls the *rtime* function to receive the parameters (μ, σ) and a location (d_i) that applies these parameters in a normal distribution for f_l. All other locations describe f_l with a uniform distribution given by an upper and lower bound $[a, b]$. The locations *Submitted* and *Closed* have bounds from the user input time intervals between tasks of the user profile and for the other locations (w_i) the bounds are calculated in a separate edge with a function *utime* (user time). This function takes into account the user-time intervals between subtasks and the data-dependent time, e.g., the wait per character, from the user profile, and returns according bounds. The task weights of the user profile are attached to the edge weights w_e and they are shown before a edge name (in bold). It can be seen that each transition or task of the initial functional model is now represented as a sequence of edges with a silent edge at the end. Note, we also included the *select* tasks, which were explained earlier. A *create* task is also possible in the *submitted* and *closed* location, but we omit additional edges for this, in order to keep the figure simple. Note that we also omit parameters and their assignments for the *rtime* and *utime* functions. The parameters for *rtime* were already explained before and *utime* takes the generated attribute data as input and returns associated intervals for the user-input time.

With this combined model, we can evaluate user profiles by simulating their expected response times. Furthermore, we can analyse a user population consisting of multiple users, by running several models concurrently. We execute this model to answer questions, like "What is the probability that the response time of each *Commit* subtask of a user within a population is under a certain threshold?". Such questions can be answered with a Monte Carlo simulation with Chernoff-Hoeffding bound. For example, checking the probability for a response-time threshold of one second for each user of a population of 10 users with parameters $\epsilon = 0.05$ and $\delta = 0.01$, requires 1060 samples and returns a probability of 0.593, when a test-case length of three tasks is considered. Fortunately, the SPRT requires fewer samples and is, therefore, better suited for the evaluation of the reference SUT or SUT deployments.

Hypothesis Testing with the SPRT. The probability that was computed on the model with a Monte Carlo simulation serves as a hypothesis in order to check, if SUT deployments are at least as good as the reference SUT. However, first we evaluate the hypothesis on the reference SUT. We apply the computed probability as alternative hypothesis and select a probability of 0.493 as null hypothesis, which is 0.1 smaller, because we want to be able to reject the hypothesis that the SUT has a smaller probability. By running the SPRT (with 0.01 as type I and II error parameters) for each user of the population, we can check these hypotheses. The alternative hypothesis was accepted for all users and on average 76.8 samples were needed for the decision. After we have evaluated the hypotheses on the reference SUT, we can reuse the hypotheses to check if deployments of this SUT provide a similar performance. For example, an evaluation of a deployment with less RAM might result in the acceptance of the same hypotheses. The acceptance of the same hypotheses means that the deployment provides the same or a similar performance as the reference SUT for our usage scenario, otherwise the deployment has worse response times.

Note that our method was implemented in the same way, as described in our previous work [37], by introducing custom generators for the response- and user-input time. For brevity, we omit the details of the implementation.

4 Evaluation

System-Under-Test. We evaluated our method by applying it to a web-service application from the automotive domain, which was provided by our industrial partner AVL[6]. The application is called Testfactory Management Suite (TFMS) and it enables various management activities of test beds, like test definition, planning, preparation, execution and data management/analysis for testing engines. TFMS is based on a client/server architecture. The server is connected to an external MongoDB database. Client and server communicate via (SOAP) web services hosted on a Microsoft Internet Information Server (IIS). There are several client types that support different activities, e.g., one client for managing test orders for test beds. As part of the software quality verification process, there is a test framework that simulates a client. This framework facilitates the creation of requests to the server, and hence, supports our testing method which works from a client's perspective [4].

TFMS consists of several modules which group together objects of the application domain and associated activities. For our evaluation, we focused on one main module, the Test Order Manager. This module enables the configuration and execution of test orders, which are basically a composition of steps that are necessary for a test sequence at an automotive test bed. Figure 5 shows an example sub-state machine for the tasks of one test order object. The complete model of the Test Order Manager is also a hierarchical state machine, like Fig. 3, but it is even more complex and therefore not presentable. Each task of the state

[6] `https://www.avl.com`

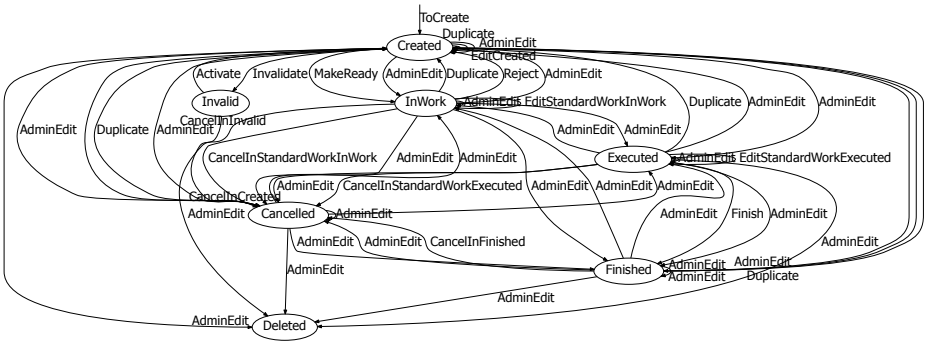

Fig. 5: Example sub-state machine of one test order object.

machine represents the invocation of a page, entering data for form fields and saving the page, e.g., the page of one task is shown in Fig. 6. The Test Order Manager contains further sub-state machines for the creation of test orders, like Business Process Templates, but they are similar to this state machine, and are therefore omitted [2].

Test Setup. We evaluated a TFMS server (version 1.8) that was running on a virtual machine with Windows Server 2012. Note that the example given in Sect. 3, was done with TFMS 1.7. Our reference SUT (D_0) had 15 GB of RAM and 7 Intel Xeon E5-2690v4 2.6 GHz CPUs. A similar virtual machine with 6 GB RAM and 3 CPUs was used to run the test clients. We defined a set of deployments by varying values for the CPUs, the RAM size, the network bandwidth, and the network delay. These deployments (D_i) are shown in Table 2.

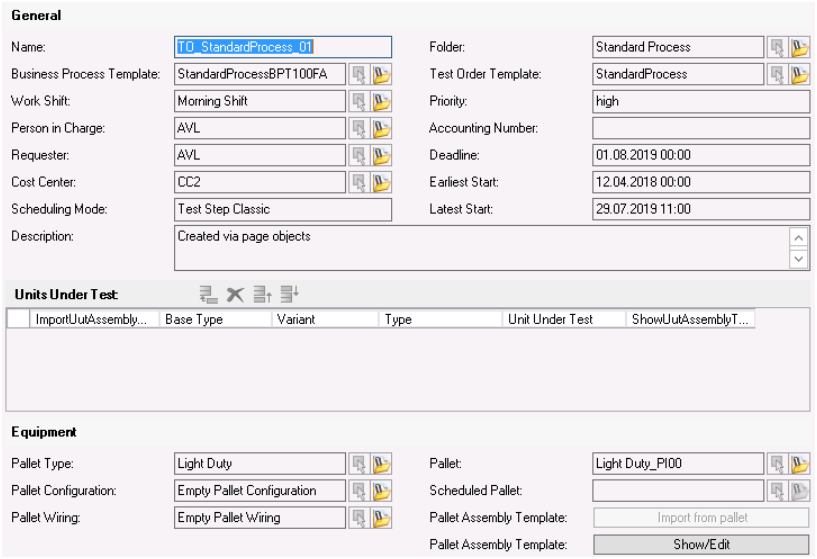

Fig. 6: Screen shot of a Test Order Manager web form for one task.

Table 2: Different system deployments with various hardware/network settings.

Deplyoment	Hardware		Network	
	#CPUs	RAM [GB]	Bandwidth [Mbps]	Delay [ms]
D_0	7	15	1000	0
D_1	7	4	1000	0
D_2	2	15	1000	0
D_3	7	15	500	0
D_4	7	15	100	0
D_5	7	15	50	0
D_6	7	15	1000	25
D_7	7	15	1000	10

Since the server was running on a virtual machine, the hardware settings could easily be changed. A tool called Network Emulator for Windows helped us to configure the network setup of the test client, e.g., it allowed us to decrease the network bandwidth. The testing phase and also the model evaluation were performed with the PBT tool FsCheck 2.8.2.

Monte Carlo Simulation. We applied our method in order to answer the following question: "What is the probability that the response time of all requests within a task sequence of a fixed length, i.e. a test case, is under a specific threshold for each user within a population?". For this evaluation, a user profile was created in cooperation with domain experts from AVL. This profile was similar to the one of Listing 1.1, and is hence, omitted. Also the stochastic timed automata model was similar to that of Fig. 4, but more complex, since the Test Order Manager comprises multiple sub-state machines for different object types. We also omit this model for brevity. We applied the model in the same way as described in Sect. 3, in order to evaluate user populations of different sizes, and we checked various response-time thresholds with a fixed test-case size of four tasks.

The model was analysed with a Monte Carlo simulation with Chernoff-Hoeffding bound with parameters $\epsilon = 0.05$ and $\delta = 0.01$, which requires 1060 samples (per data point). Figure 7 shows the results. As expected, a decrease in the probability of our given question can be observed, when the number of users increases or the threshold decreases. Note that an advantage of the evaluation of the model is that the model execution can be accelerated with a virtual time. We apply a virtual time of 1/10 of the actual time, which speeds up the model execution by a factor of ten (compared to the SUT).

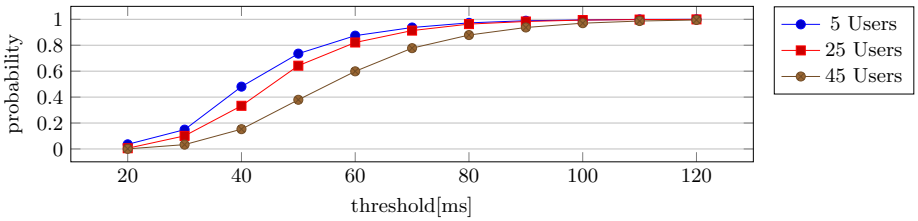

Fig. 7: Test Order Manager Monte Carlo simulation results of the model.

Table 3: Different SPRTs for various numbers of users and thresholds.

SPRT No.	#Users	Threshold [ms]	H_0	H_1
1	5	50	0.478	0.729
2	25	50	0.400	0.650
3	45	50	0.201	0.451
4	5	100	0.746	0.996
5	25	100	0.744	0.994
6	45	100	0.738	0.988

Hypothesis Testing with the SPRT. Next, we applied the probabilities of the Monte Carlo simulation as hypotheses (H_1) for SPRTs of the different deployments. We selected six data points of Fig. 7 with interesting thresholds and different user numbers in order to form the hypotheses shown in Table 3. We evaluate all deployments as explained in Sect. 3 by applying the SPRT with the same parameters. Figure 8 summarises the results in three groups: one for the deployments (and SPRTs), where all clients accepted H_1, one where there was no clear consensus among the clients, and one where all clients accepted H_0. It can be seen that H_1 was accepted by most of the deployments, which means that they provide a similar performance. For one deployment (D_5) only SPRT 1–4 were successful, SPRT 5–6 were inconclusive, i.e. 48 % of the clients accepted H_1 for SPRT 5 and 44 % for SPRT 6. For two deployments, H_0 was accepted, which means that their response time was worse than that of the reference SUT. In summary, it can be said that a change in the server hardware did not significantly affect the performance, as H_1 was accepted for all deployments with a changed hardware. Also, a change in the network bandwidth had only a weak influence on the performance. A clear change in the performance was only observed for deployments with a higher network delay.

Additionally, we evaluated the number of needed samples of the SPRTs. Note that in order to obtain an average number of needed samples, we run the SPRT concurrently for each user of the population and calculate the average of these runs. Multiple independent SPRT runs would produce a better average, but the computation time was too high. Figure 9 shows the average number of needed samples for the SPRTs of different deployments. It can be seen that certain SPRTs are quite easy to check, e.g., SPRT 3 only needs about 6–13 samples, other SPRTs take more than twice as many samples. However, a maximum of

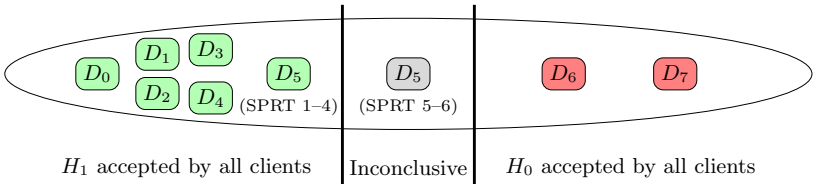

Fig. 8: SPRT results of the different deployments.

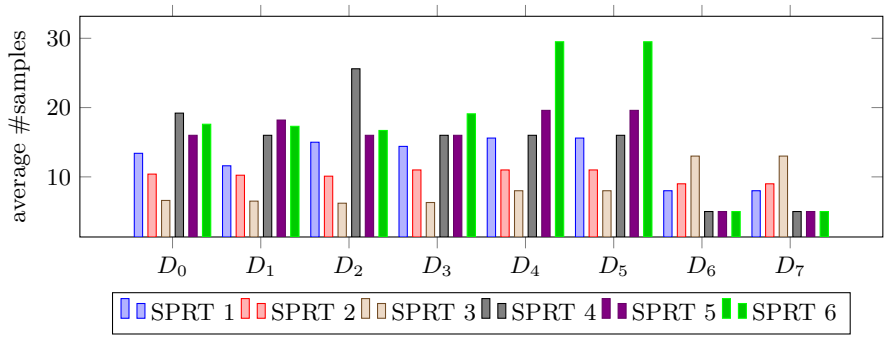

Fig. 9: Average number of samples (test cases) for the SPRTs of our deployments.

about 30 samples is still very low compared to the 1060 samples of the Monte Carlo simulation. This low number of samples allows us to evaluate multiple SUT deployments within a feasible time.

5 Conclusion

We have demonstrated that we can apply SMC with learned timed models in order to answer questions about the expected response time of given usage scenarios, like "What is the probability that the response time of each user within a population, is under a specific threshold?". Moreover, we have illustrated that we can verify the results of such evaluations with hypothesis testing on a real system. Additionally, we checked deployments of an SUT by reusing the hypotheses of a reference SUT.

A major benefit of our approach is that it enables an efficient performance comparison of a reference system with system deployments for specific usage scenarios. This is especially helpful, when customer recommendation for the hardware or network settings are needed for a deployment that should satisfy certain user expectations. Another advantage of our method is that it is realised within a PBT tool, which increases the accessibility for testers from industry, because the models and properties can be defined in a high-level programming language. Hence, there is no need to learn new notations.

We have evaluated our method with an industrial case study of a web-service application, and it showed promising results. We analysed various deployments of an SUT with different hardware and network settings. This analysis showed that deployments with different server hardware provide a comparable performance as the reference system for our given usage scenarios. Only deployments with higher network delays showed a significant performance loss.

In the future, we plan to combine our technique with different learning methods. Since a linear regression requires still a high manual effort, we aim to evaluate learning methods that support a higher degree of automation. Moreover, an analysis of the applicability of our method for other performance indicators than response times, e.g., for energy, has a great potential for future work.

Acknowledgements. This research was funded by the Austrian Research Promotion Agency (FFG), project TRUCONF, No. 845582. We are grateful to the project team and to the anonymous reviewers for their remarks.

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. 28(1), 6:1–6:39 (2018)
2. Aichernig, B.K., Schumi, R.: Property-based testing with FsCheck by deriving properties from business rule models. In: ICSTW. pp. 219–228. IEEE (2016)
3. Aichernig, B.K., Schumi, R.: Towards integrating statistical model checking into property-based testing. In: MEMOCODE. pp. 71–76. IEEE (2016)
4. Aichernig, B.K., Schumi, R.: Property-based testing of web services by deriving properties from business-rule models. Software & Systems Modeling (Dec 2017)
5. Aichernig, B.K., Schumi, R.: Statistical model checking meets property-based testing. In: ICST. pp. 390–400. IEEE (2017)
6. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
7. Arts, T.: On shrinking randomly generated load tests. In: Erlang'14. pp. 25–31. ACM (2014)
8. Ballarini, P., Bertrand, N., Horváth, A., Paolieri, M., Vicario, E.: Transient analysis of networks of stochastic timed automata using stochastic state classes. In: QEST. LNCS, vol. 8054, pp. 355–371. Springer (2013)
9. Banga, G., Druschel, P.: Measuring the capacity of a web server under realistic loads. World Wide Web 2(1-2), 69–83 (1999)
10. Becker, S., Koziolek, H., Reussner, R.H.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82(1), 3–22 (2009)
11. Blair, L., Jones, T., Blair, G.S.: Stochastically enhanced timed automata. In: FMOODS. vol. 177, pp. 327–347. Kluwer (2000)
12. Book, M., Gruhn, V., Hülder, M., Köhler, A., Kriegel, A.: Cost and response time simulation for web-based applications on mobile channels. In: QSIC. pp. 83–90. IEEE (2005)
13. Bulychev, P.E., David, A., Larsen, K.G., Mikucionis, M., Poulsen, D.B., Legay, A., Wang, Z.: UPPAAL-SMC: statistical model checking for priced timed automata. In: QAPL. EPTCS, vol. 85, pp. 1–16. Open Publishing Association (2012)
14. Chen, X., Mohapatra, P., Chen, H.: An admission control scheme for predictable server response time for web accesses. In: WWW. pp. 545–554. ACM (2001)
15. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP. pp. 268–279. ACM (2000)
16. Claessen, K., Palka, M.H., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.T.: Finding race conditions in Erlang with QuickCheck and PULSE. In: ICFP. pp. 149–160. ACM (2009)
17. Draheim, D., Grundy, J.C., Hosking, J.G., Lutteroth, C., Weber, G.: Realistic load testing of web applications. In: CSMR. pp. 57–70. IEEE (2006)
18. Govindarajulu, Z.: Sequential statistics. World Scientific (2004)
19. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. 8(3), 231–274 (1987)
20. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association 58(301), 13–30 (1963)

21. Hughes, J.: QuickCheck testing for fun and profit. In: PADL. LNCS, vol. 4354, pp. 1–32. Springer (2007)
22. Hughes, J., Pierce, B.C., Arts, T., Norell, U.: Mysteries of Dropbox: Property-based testing of a distributed synchronization service. In: ICST. pp. 135–145. IEEE (2016)
23. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine. In: ICST. pp. 230–239. IEEE (2009)
24. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Verifying quantitative properties of continuous probabilistic timed automata. In: CONCUR. LNCS, vol. 1877, pp. 123–137. Springer (2000)
25. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: RV. LNCS, vol. 6418, pp. 122–135. Springer (2010)
26. Legay, A., Sedwards, S.: On statistical model checking with PLASMA. In: TASE. pp. 139–145. IEEE (2014)
27. Lu, Y., Nolte, T., Bate, I., Cucu-Grosjean, L.: A statistical response-time analysis of real-time embedded systems. In: RTSS. pp. 351–362. IEEE (2012)
28. Malik, H., Shakshuki, E.M.: Classification of post-deployment performance diagnostic techniques for large-scale software systems. Procedia Computer Science 37, 244–251 (2014)
29. Menascé, D.A.: Load testing of web sites. IEEE Internet Computing 6(4), 70–74 (2002)
30. Nilsson, R.: ScalaCheck: The Definitive Guide. IT Pro, Artima Incorporated (2014)
31. Norell, U., Svensson, H., Arts, T.: Testing blocking operations with QuickCheck's component library. In: Erlang'13. pp. 87–92. ACM (2013)
32. Nourikhah, H., Akbari, M.K., Kalantari, M.: Modeling and predicting measured response time of cloud-based web services using long-memory time series. The Journal of Supercomputing 71(2), 673–696 (2015)
33. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Erlang'11. pp. 39–50. ACM (2011)
34. Rencher, A., Christensen, W.: Methods of Multivariate Analysis. Wiley (2012)
35. Rina, Tyagi, S.: A comparative study of performance testing tools. Intern. Journal of Adv. Research in Comp. Sci. and SW Engineering, IJARCSSE 3(5), 1300–1307 (2013)
36. Roloff, E., Diener, M., Carissimi, A., Navaux, P.O.A.: High performance computing in the cloud: Deployment, performance and cost efficiency. In: CloudCom. pp. 371–378. IEEE Computer Society (2012)
37. Schumi, R., Lang, P., Aichernig, B.K., Krenn, W., Schlick, R.: Checking response-time properties of web-service applications under stochastic user profiles. In: ICTSS. LNCS, vol. 10533, pp. 293–310. Springer (2017)
38. Wald, A.: Sequential analysis. Courier Corporation (1973)
39. Yu, J., Han, J., Schneider, J., Hine, C.M., Versteeg, S.: A petri-net-based virtual deployment testing environment for enterprise software systems. Comput. J. 60(1), 27–44 (2017)
40. Zhang, F., Bu, L., Wang, L., Zhao, J., Chen, X., Zhang, T., Li, X.: Modeling and evaluation of wireless sensor network protocols by stochastic timed automata. Electronic Notes in Theoretical Computer Science 296, 261–277 (2013)