Learning and statistical model checking of system response times

Bernhard K. Aichernig¹ · Priska Bauerstätter² · Elisabeth Jöbstl³ · Severin Kann³ · Robert Korošec³ · Willibald Krenn² · Cristinel Mateis² · Rupert Schlick² · Richard Schumi¹

Published online: 3 January 2019
© The Author(s) 2019

Abstract

Since computers have become increasingly more powerful, users are less willing to accept slow responses of systems. Hence, performance testing is important for interactive systems. However, it is still challenging to test if a system provides acceptable performance or can satisfy certain response-time limits, especially for different usage scenarios. On the one hand, there are performance-testing techniques that require numerous costly tests of the system. On the other hand, model-based performance analysis methods have a doubtful model quality. Hence, we propose a combined method to mitigate these issues. We learn response-time distributions from test data in order to augment existing behavioral models with timing aspects. Then, we perform statistical model checking with the resulting model for a performance prediction. Finally, we test the accuracy of our prediction with hypotheses testing of the real system. Our method is implemented with a property-based testing tool with integrated statistical model checking algorithms. We demonstrate the feasibility of our techniques in an industrial case study with a web-service application.

Keywords Statistical model checking · Property-based testing · Model-based testing · FsCheck · User profiles · Response time · Cost learning · Performance testing

1 Introduction

Performance testing is important, especially for critical systems. It is usually done with sophisticated load techniques that are computationally expensive and even infeasible when various user populations should be analyzed. Alternatively, the performance may be analysed by simulating a model of the system. Simulation allows faster analysis and requires less computing resources, but the quality of the model is often questionable. We present a simulation method based on statistical model checking (SMC) that enables a fast probability estimation with a model and also a verification of the resulting probabilities on the real system.

✉ Richard Schumi
rschumi@ist.tugraz.at

SMC is a simulation method that can answer both quantitative and qualitative questions. The questions are expressed as properties of a stochastic model which are checked by analyzing simulations of this model. Depending on the SMC algorithm, either a fixed number of samples or a stopping criterion is needed.

We implement our method with the help of a property-based test-case generator that is originally intended for functional testing. Property-based testing (PBT) is a random testing technique that tries to falsify a given property, which describes the expected behavior of a function-under-test. In order to test such a property, a PBT tool generates inputs for the function and checks if the expected behavior is observed. PBT tools were originally designed for testing algebraic properties of functional programs, but nowadays, they also support model-based testing.

In previous work (Aichernig and Schumi 2017a, b), we have demonstrated how SMC can be integrated into a PBT tool in order to evaluate properties of stochastic models as well as stochastic implementations. Based on this previous work, we present a simulation method for stochastic user profiles in order to answer questions about the expected response time of a system-under-test (SUT). Figure 1 illustrates this process.

- (1) First, we apply a PBT tool to run model-based testing (MBT) with a functional model concurrently in several threads in order to obtain log-files that include the response times of the tested web-service requests. Since the model serves as an oracle, we also test for conformance violations in this phase. This functional aspect was discussed in earlier work (Aichernig and Schumi 2016a), here the focus is on timing.
- (2) Next, we derive response-time distributions per type of service request via linear regression, which was a suitable learning method for our logs. Since the response time is influenced by the parallel activity on the server, the distributions are parametrized by the number of active users.
- (3) These cost distributions are added to the transitions in the functional model resulting in, so called, cost models. These models have the semantics of stochastic timed automata (STA) (Ballarini et al. 2013). The name *cost model* shall emphasize that our method may be generalized to other type of cost indicators, e.g., energy consumption.

We also combine these models with user profiles, containing probabilities for transitions and input durations, in order to simulate realistic user behavior and the expected response time.

- (4) These combined models can be utilized for SMC, in order to evaluate response-time properties, like “What is the probability that the response time of each user within a user population is under a certain threshold?” or “Is this probability above or below a specific limit?”.

We apply them for a Monte Carlo simulation, in order to estimate the probability of such properties.

- (5) Additionally, we can check such properties directly on the SUT, e.g., to verify the results of the model simulation. In principle, it is also possible to skip the model simulation and (statistically) test response-time properties directly on the SUT. However, running a realistic user population on the SUT is time-consuming and might not be feasible due to very long waiting times. A simulation on the model is much faster. Therefore, also properties that require a larger number of samples can be checked, e.g., using Monte Carlo simulation. We run the SUT only with a limited number of samples in order to check, if the simulation results of the model are satisfied by the SUT. Therefore, we test the SUT with the sequential probability ratio test (Wald 1973), a form of hypothesis testing, as this allows us to stop testing as soon as we have sufficient evidence.

Fig. 1 Overview of the steps for cost-model learning and response-time checking

Related work A number of related approaches in the area of PBT are concerned with testing concurrent software. For example, Claessen et al. (2009) presented a testing method that can find race conditions in Erlang with QuickCheck and a user-level scheduler called PULSE. A similar approach was shown by Norell et al. (2013). They demonstrated an automated way to test blocking operations, i.e., operations that have to wait until a certain condition is met. Another concurrent PBT approach by Hughes et al. (2016) showed how PBT can be applied to test distributed file-synchronisation services, like Dropbox. The closest related work we found in the PBT community is from Arts (2014). He 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 model-based simulation.

There exist various tools for performance testing and load generation (Vinayak Hegde 2014; Rina and Tyagi 2013), which are related to our approach, since they also support the simulation of user populations. For example, Neoload¹ is a performance testing and measurement tool for mobile and web applications that can simulate user populations. A similar open source tool is Apache JMeter (Halili 2008). Initially, it was only build for websites, but recently it also supports other applications areas. Another tool called LoadRunner (Jinyuan 2012) from HP supports the simulation of thousands of users and it works for various software platforms, like .NET or Java.

The most related performance-testing approaches are mainly in the area of load or stress testing. For example, Menascé (2002) presented a load testing approach for web sites that works with user interaction scripts to simulate the user behavior. Another load-testing method was introduced by Draheim et al. (2006). They showed the simulation of realistic user behavior with stochastic models and workload models in order to estimate the performance of web applications. A related stress testing approach was presented by Krishnamurthy et al. (2006). The work shows a synthetic workload-generation technique that is based on request logs, and should mimic real user behavior. In contrast to our work, classical performance or load testing is mostly performed directly on an SUT. With our approach, we want to simulate user populations on the model-level as well.

¹<https://www.neotys.com/neoload/overview>

Related work can also be found in the area of performance engineering or model-based simulation methods Smith (1990), Woodside et al. (2007), Pooley and King (1999), and Book et al. (2005), i.e., various approaches apply a model to predict performance. For example, Becker et al. (2009) presented a prediction method with Palladio component models for the evaluation of component-based software architectures. With their method, they predicted response times of an online music repository for concurrent system usage. Moreover, they compared their prediction against measurements from a real system. Lu et al. (2012) demonstrated a statistical response-time analysis. Their approach takes response-time samples for the construction of a statistical model that is applied to derive upper bounds for response-time estimates. Most of these approaches only apply a model-based analysis, and do not present an automated technique for the evaluation of their model on an SUT. In contrast, with our method we can perform a model-based prediction, and we can also check the accuracy of our predictions by directly testing an SUT within the same tool.

There are also some approaches or tools that can do both, a simulation with a model and testing an SUT. Balsamo et al. (2004) gave an overview of various model-based performance-prediction approaches and tools, and the mention tools, like SPE•ED (Smith and Williams 1997), which works with message sequence charts and supports a model-based simulation, as well as an evaluation of object-oriented systems. A disadvantage of such approaches is that they still require much manual effort, e.g., performance data is often only defined manually. In contrast, we also include an automated approach for response-time learning with linear regression. Moreover, we can exploit PBT features, because our approach is realised within a PBT tool.

The most closely related tool is UPPAAL SMC (Bulychev et al. 2012). Similar to our approach, it provides SMC of priced timed automata, which can simulate user populations. It also supports testing real implementations, but for this, a test adapter needs to be implemented, which, e.g., handles form-data creation. With our method, we can use PBT features, like 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, as they do not have to learn new notations.

To the best of our knowledge, our work is novel: (1) no other work applies PBT for evaluating stochastic properties about the response time of both real systems and stochastic models, (2) no other work performs cost learning on behavior models using linear regression. Grinchtein (2008) learns time-deterministic event-recording automata via active automata learning. Verwer et al. (2010) passively learn probabilistic real-time automata. In contrast, we learn cost distributions and add them to existing automata models for SMC.

Contribution This article is an extended version of an ICTSS conference paper (Schumi et al. 2017). Compared to this previous work we present the following new contributions:

(i) A major new contribution is an extensive description of our cost-model learning approach. The conference paper gave a brief overview only. Here, we focus more on the learning process. We highlight various learning steps, like data cleaning and feature engineering, which are key to obtain a good model. (ii) Another new contribution is the extension of our method to enable an assessment of the prediction power of our learned model. In our previous work, we only checked if the SUT is at least as good as the model in responding within a certain time limit. Now, we also check if the real probability of the SUT is close to the predicted probability of the model by utilising a two-sided hypothesis test. (iii) Moreover, we present an additional industrial case study. We evaluate our approach by applying it to an updated and extended version of the SUT we have used previously.

In relation to our presented process of Fig. 1, our main contributions in this work are located in the cost-model learning and hypotheses-testing phases. However, we also tried to optimize the presentation of all steps of our approach, and we illustrate a more detailed evaluation.

Structure First, Section 2 introduces our SUT and the necessary background regarding SMC, PBT, and their combination. Next, Section 3 explains how we perform cost-model learning. In Section 4, we present our method with an example. In Section 5, we give more details about the process and implementation. Section 6 presents an evaluation based on an industrial web-service application. In Section 7, we discuss limitations of our method and finally, we draw our conclusions in Section 8.

2 Background

2.1 System-under-test

Our approach was evaluated on a web-service application provided by our industrial project partner AVL.² The application originates from the automotive domain and is called Test-factory Management Suite (TFMS).³ It is a workflow tool that supports the process of instrumenting and testing automotive power trains – a core business of AVL. TFMS captures test-bed data, activities, resources, and workflows. A variety of activities can be realized with the system, like test definition, planning, preparation, execution, data management, and analysis (Aichernig and Schumi 2017b).

The application is intended for various kinds of automotive test beds for car components, like engines, gears, power trains, batteries for electric cars or entire cars. For instance, for testing an engine it is mounted to a pallet and also different test equipment is attached. The selection of the test equipment depends on the specific use case. Typical test equipment for an engine might be a measurement device for the power output or the fuel consumption. After a pallet is configured, it is moved to the test bed, where all devices are connected and a test is performed. TFMS manages all steps and needed devices of such a workflow, which is also called a test order. It allows the scheduling of car components that need to be tested, the selection of required test equipment, the definition of the needed wiring for the equipment, and the planning of the sequence of all tasks at a test bed. Moreover, customer specific requirements, like additional management steps or custom restrictions can be freely configured via business-rules.

The system has a client-server architecture which is illustrated in Fig. 2. The “TFMS Server” is the central component of the system. This server is hosted in Microsoft’s IIS (Internet Information Services) and provides several simple object access protocol (SOAP) web services, which are described via the web services description language (WSDL). For data storage, MongoDB is used. TFMS offers different types of clients: one to collect data from the test beds, several office clients for different management activities (e. g., test order management) and a scheduler to plan the execution of test activities on the test beds (Aichernig and Schumi 2017b). TFMS is highly configurable and offers an own client for server configurations (CFG Client). The web services are driven by business rules. A rule engine

²<https://www.avl.com>

³<https://www.avl.com/tfms>

Fig. 2 Client-server architecture of the SUT

takes this business logic in the form of business-rule models and interprets them in order to define the control-flow of the application.

The system consists of multiple modules corresponding to the mentioned clients. Modules can be seen as groups of functionality and they consist of multiple business-rule models which describe what tasks can be performed by a user and how they look like, e. g., what data can be modified. Only one business-rule model can be active at the same time and it determines, which forms can be opened in the current state of the system (Aichernig and Schumi 2017b).

A business-rule model is a state machine defining the behavior of the business objects, so called TFMS Objects. A TFMS object class describes objects of our application domain, like test equipments or test orders. Each object has a state, an identifier, attribute values/data and is stored in the database of our SUT. TFMS works task based. Tasks represent the behavior, i. e., the actions or events a user may trigger, e. g., creating or editing TFMS objects. An example business-rule model and description of tasks and subtasks are presented in Section 4.1.

TFMS is a critical software, because it is essential to efficiently operate test beds. It is deployed to various customers where it is running under different hardware and network settings. Moreover, it is applied for several application fields and under varying usage conditions, i. e., with several users and different user types. It is important for AVL that the system is still fast enough to satisfy even high numbers of concurrent users. Hence, in this work, we are investigating the performance of TFMS for various usage scenarios.

2.2 Statistical model checking

Statistical model checking (SMC) is a verification method that evaluates certain properties of a stochastic model. These properties are usually defined with (temporal) logics, and we may ask quantitative and qualitative questions about their satisfiability. For example, *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 to compute the total number of samples needed in order to find an answer for a specific

question or to compute a stopping criterion. A stopping 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 (Legay et al. 2010; Legay and Sedwards 2014).

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 (Hoeffding 1963) 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 can be calculated as follows:

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

The n simulations represent Bernoulli random variables $X_1, \dots, X_n$ 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 (1), a simple Monte Carlo simulation is performed (Legay and Sedwards 2014).

Sequential probability ratio test This sequential method (Wald 1973) 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 :

$$\log \Lambda_m = \log \frac{p_1^m}{p_0^m} = \log \frac{\prod_{i=1}^m f(x_i, \theta_1)}{\prod_{i=1}^m f(x_i, \theta_0)} = \sum_{i=1}^m \log \frac{f(x_i, \theta_1)}{f(x_i, \theta_0)} \quad (2)$$

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 $\log \Lambda_m \leq \log \frac{\beta}{1-\alpha}$ (Govindarajulu 2004).

In this work, we form a hypothesis about the expected response time with the Monte Carlo method on the model. Then, we check with sequential probability ratio test (SPRT) if this hypothesis holds on the SUT. This is faster than running Monte Carlo directly on the SUT.

2.3 Property-based testing

Property-based testing (PBT) is a random-testing technique that aims to check the correctness of properties. A property is a high-level specification of the expected behavior 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:

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

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 MBT. Models encoded as extended finite state machines (EFSMs) (Kalaji et al. 2009) 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, and T is a finite set of transitions. A transition $t \in T$ can be described as 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, and s_t is the target state (Kalaji et al. 2009). 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 behavior, 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, respectively command (Hughes 2007; Papadakis and Sagonas 2011). Formally, we define such a property as follows:

Let $cmd.runModel$ and $cmd.runActual$ be functions of type $S \times I \rightarrow S \times O$ for executing a command cmd on the model and on the SUT, respectively. Furthermore, we have a precondition $cmd.pre : I \times S \rightarrow Boolean$ defining the valid inputs of the command. Given a postcondition $cmd.post : S \times O \times S \times O \rightarrow Boolean$ that relates the outputs and states after command execution on the SUT and model, then the property to be tested is

$$\forall s \in S, i \in I, cmd \in Cmds : \quad cmd.pre(i, s) \implies cmd.post(cmd.runModel(i, s), cmd.runActual(i, s)) \quad (3)$$

A PBT tool generates random sequences of commands in order to test this property. For generating the input data, it is possible to define custom data generators for each type of input. Put simply, a generator $Gen[A]$ is defined for a type A and provides a function $sample : A$ that returns an instance of this type. For complex data, default and custom generators can be freely combined and even nested. This makes PBT an ideal candidate to test web services with complex input forms.

The first PBT implementation was QuickCheck for Haskell (Claessen and Hughes 2000). Numerous reimplementations followed for other programming languages, like Hypothesis⁴ for Python or ScalaCheck (Nilsson 2014). We build upon our previous work (Aichernig and Schumi 2016a) and demonstrate our approach with FsCheck.⁵ It is a .NET port of QuickCheck influenced by ScalaCheck. It supports a property definition in both, a functional programming style with F# and an object-oriented style with C#. We work with C# as it is the programming language of our SUT.

2.4 Integration of SMC into PBT

We have demonstrated that SMC can be integrated into a PBT tool in order to perform SMC of PBT-properties (Aichernig and Schumi 2017a, b). These PBT-properties can be evaluated on stochastic models, like in classical SMC, as well as on implementations with stochastic behavior. For the integration, we introduced our own new SMC properties for a PBT tool. These SMC properties take a PBT property, configurations for the PBT execution, and parameters for the specific SMC algorithm as input (see Fig. 3).

⁴<https://pypi.python.org/pypi/hypothesis>

⁵<https://fscheck.github.io/FsCheck>

Fig. 3 Data flow diagram of an SMC property

Then, our properties perform an SMC algorithm by utilizing the PBT tool as a simulation environment and return either a quantitative or qualitative result, depending on the algorithm. For example, a state-machine property can be applied for a statistical conformance analysis by comparing an ideal model to a faulty stochastic system. Additionally, it can also simulate a stochastic model.

A simple code example of an SMC property that performs a Monte Carlo simulation is outlined in Algorithm 1. This SMC property takes a PBT property, configurations for the property check and the required number of samples as input. It performs property checks for the number of sample in a loop and returns the probability that the property is satisfied. We evaluated our SMC properties by repeating several case studies from the SMC literature and we were able to reproduce the results (Aichernig and Schumi 2017a, b).

Algorithm 1 Pseudo code of a MonteCarloProperty.

Input: *prop*: PBT property, *config*: configuration for the property check, *n*: number of samples

```

1: passCnt ← 0                                ▷ Counter for the passed property checks
2: for i ∈ {1, ..., n} do
3:   if prop.Check(config) then
4:     passCnt ← passCnt + 1                    ▷ increase pass counter
5: return passCnt/n
  
```

3 Cost-model learning

The response time distributions to be added to the transitions in the functional model are a key part of the method presented in this paper. How can we derive such distributions? Implementing a classical rule-based algorithm is not feasible since appropriate *if-then-else* rules with the associated conditional expressions and calculation formulas for the distribution parameters are hard to be defined a priori in our context. However, we can recognize that we have all the necessary ingredients for a data-driven learning approach, more precisely, for supervised learning with regression. We have log-files with a large number of request examples (instances) for which also the response times (labels) are known. For each request example, the log-file specifies the values of a number of attributes (features) related to the requests. Our regression task is to learn from the (labeled) data given by the log-files, a function which, given the attribute values of a request instance, returns the parameters (μ , σ) of a normal distribution for the response time for that instance.

As we will see in Section 4.2, it turns out that the response times can be fairly well approximated by a linear combination of the request attributes by using the linear regression method. This comes in handy since (i) the statistical properties of the resulting estimators, i.e. the weights of the request attributes, are easier to determine with linear regression than

with other learning algorithms, and (ii) we can use these statistical properties to derive the normal distribution parameters of the response times.

Multiple linear regression The general linear regression model in matrix notation is known as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \quad (4)$$

where $\mathbf{y}$ is the dependent variable (regressand), $\mathbf{X}$ is the design matrix of the independent or explanatory variables (regressors), $\boldsymbol{\beta}$ contains the model parameters (regressor coefficients or weights), and $\boldsymbol{\epsilon}$ is the error term (noise) which captures all other factors which influence the dependent variable other than the regressors (Hastie et al. 2009). In more detail, in case of p regressors the i th observation of the dependent variable is given by

$$y_i = 1\beta_0 + X_{i,1}\beta_1 + \dots + X_{i,p}\beta_p + \epsilon_i \quad (5)$$

with β_0 as the constant or offset term (intercept). The case with more than one independent variable is called multiple linear regression (MLR). Thus, we use MLR to model the relationship between the response time, i.e., the dependent variable, and the attributes, i.e., the independent variables, of a request.

Given a log-file with N examples of requests and their response times, $\mathbf{y}$ is the $N \times 1$ vector of the response times and $\mathbf{X}$ is the $N \times p$ design matrix for p request attributes considered to linearly influence the response time, where y_i is the response time and $X_{i,1}, \dots, X_{i,p}$ are the attributes of the i th request example in the log-file.

We can use $\mathbf{y}$ and $\mathbf{X}$ with the (4) to estimate the model parameters $\boldsymbol{\beta}$ that minimize the error term $\boldsymbol{\epsilon}$. Note that $\boldsymbol{\epsilon}$ is a $N \times 1$ vector and there are various ways to define what “minimize $\boldsymbol{\epsilon}$ ” means. The simplest and most common method is the ordinary least squares (OLS) which minimizes the sum of the squares ϵ_i^2 , $i = 1, \dots, N$.

After we have estimated the parameters $\boldsymbol{\beta} = [\beta_0, \beta_1, \dots, \beta_p]$, we use the formula

$$y = 1\beta_0 + x_1\beta_1 + \dots + x_p\beta_p \quad (6)$$

to predict the response time y of a new, previously unseen, request with attributes $[x_1, x_2, \dots, x_p]$. Please note that the formula (6) is similar to (5) but without a correction error term ϵ which accounts for random variation or other unknown factors. Hence, (6) is an approximation of the real response time which can be evaluated by analyzing the statistics (e.g., standard error, p value, confidence interval) of the estimated model parameters $\boldsymbol{\beta}$ computed when applying the OLS method.

If we consider that the model parameters β_k , $k = 0, \dots, p$ are normally distributed with the mean and standard deviation estimates $(\mu_{\beta_k}, \sigma_{\beta_k})$ given by the model parameters and the corresponding standard errors computed with OLS, then it follows that the predicted response time y is normally distributed with the mean μ_y and standard deviation σ_y given by

$$\mu_y = \sum_{k=0}^p x_k \mu_{\beta_k}, \quad \sigma_y^2 = \sum_{k=0}^p x_k^2 \sigma_{\beta_k}^2 \quad (7)$$

as a linear combination of the normal distributions $\mathcal{N}(\mu_{\beta_k}, \sigma_{\beta_k}^2)$ with weights x_i , $i = 0, \dots, p$ and $x_0 = 1$, according to (6).

The normal distribution $\mathcal{N}(\mu_y, \sigma_y^2)$ with parameters given by (7) is exactly what we are looking for. Thus, given a log-file of request examples with corresponding response times, we learn the parameters $(\mu_{\beta_k}, \sigma_{\beta_k})$ of a model (7) which gives the normal distribution $\mathcal{N}(\mu_y, \sigma_y^2)$ of the response time y for any new request with known attributes $[x_1, \dots, x_p]$ to be associated to the behavioral model as needed.

4 Method

This section shows how we derive cost models from logs and how we can apply these models to simulate stochastic user profiles.

4.1 Log-data generation

As explained in Section 2.1, our SUT is a web-service application called TFMS that consists of various modules. One simplified model out of the Test Order Manager module serves us as an example. The whole module will later be presented in more detail within the evaluation in Section 6. The example model supports tasks, like creating or editing Business Process Templates, which are objects of the application domain. These objects include attributes (form data) that are stored in a database and have to be set by the users. The state machine in Fig. 4 on the left represents the tasks of this model. (Note that tasks can be enabled in various states and they can also lead to multiple states, as the next state may be selected via an external choice by the user.) To keep it simple, this state machine only represents the tasks of a currently opened object without attributes. In reality, we also have transitions to switch between objects and a variety of attributes. Hence, this functional model is an EFSM. In our previous work, we have demonstrated how such functional models can be derived from business-rule models of the server implementation (Aichernig and Schumi 2017b). In this article, we assume the functional models as given.

Each task consists of subtasks, e.g., for setting attributes or for opening a screen. The subtasks of one task can be seen in the center of Fig. 4. Many subtasks require server interaction. Therefore, they can also be seen as requests.

Based on these functional models, we can perform conventional PBT, which generates random sequences of commands with form data (attributes) in order to check properties that test the functionality of the SUT. We test state-machine properties (3) as explained in Section 2.3, i.e., we perform random walks on the model and check for output and variable equivalence of the model and the SUT in the postconditions. In our case, the output is just the current state after the execution of a command, and since we encode the form data in variables, we test if the variable values are correctly transferred to the SUT and that there is no problem with the database. While these properties are tested, a log is created that captures the response times (costs) of individual requests. The properties are checked concurrently on the SUT in order to obtain response times of multiple simultaneous requests, which represents the behavior of multiple active users.

An example log from a test system is represented in Table 1. Note that this TFMS version was running on a virtual machine with low computing resources. We record response times of tasks, subtasks, simultaneous requests *#ActiveUsers*, the attribute name in case of one

Fig. 4 Cost annotation for the Business Process Template model

Table 1 Example log data of the Business Process Template model

Task	Subtask	#Active-Users	Attribute	#Attributes	ObjSize	Cumulative-ObjSize	Response time [ms]
Create	StartTask	5	–	–	–	7115966	21
Create	SetRefAttr	4	Responsible	–	0	7119938	17
ChangeState	StartTask	2	–	–	0	7119938	22
Create	Commit	5	–	5	3985	7123923	31
ChangeState	Commit	4	–	2	3372	7181842	25

attribute and otherwise the number of attributes *#Attributes*, the generated form-data size *ObjSize*, and the cumulative sum of the data size *CumulativeObjSize* (which represents the database fill level of the SUT). For this initial logging phase the test-case generator chooses the transitions, i.e. the tasks, with uniform distribution. We randomly select the number of testing processes for each test case in order to obtain log data that includes entries for diverse workloads. Our focus is on weak to high average workloads, since we want to find realistic situations that lead to a reduced user satisfaction due to slow responses, and we want to know to what extent such situations occur.

Note that with PBT we can freely choose the number of test cases and the length of each test case. This allows us to control the size of the generated data, which is helpful for our learning method, since we need to try out different data sizes.

4.2 Learning from the log-data

Learning a model from the log-files is a data-driven approach, hence the quality and the accuracy of the log-data are of crucial importance. Unfortunately, it is not possible to obtain log-files through usage of the web-service application (SUT) by real-world users and directly logged by the SUT due to various reasons: (i) it is hard to get permission from SUT customers to use their real-world data from production environments (confidentiality reasons), (ii) the time needed for the log-files generation depends on the usage frequency by the real-world users and is generally long, (iii) logging of critically important attributes, e.g., database fill level, is not acceptable due to invasive character (frequent access to database registers through the logging client might negatively impact the SUT performance). However, we overcome this problem by using a user simulation tool as shown in Section 4.1.

The advantages of the user simulation approach are threefold: (i) we can run the user simulation tool on demand, whenever and with whatever environment setup we need, (ii) we can simulate users with different usage profiles (e.g., some users are typing faster, hence sending requests to SUT with a higher frequency), (iii) we are flexible to record any required attribute, through simulation if necessary, without SUT (re-)coding which would be critical due to release cycle constraints and possible invasive character. For instance, the database fill level might be simulated on the tool side by maintaining an internal variable *CumulativeObjSize* which is incremented, resp. decremented, every time a request is storing to, resp. removing from, the SUT database. The step size used is the objects total size *ObjSize*.

A disadvantage of the user simulation approach is, however, that we might generate biased log-files, e.g., by selecting some parameter setups much more often than others, if we do not carefully define the experimental setup. Moreover, measurement errors might be introduced due to (i) network latency (the simulation tool is not run on the SUT machine in order to do not impact on the SUT performance), (ii) approximations of simulated attributes, e.g., *CumulativeObjSize*, or (iii) tool execution time overhead. However, we keep these errors low, and consider them irrelevant, by (i) running the tool on the local network as close to the SUT as possible, (ii) consistently updating the internal variables for the simulated attributes, and (iii) implementing the simulation tool with real-time requirements in mind in addition to allocating sufficient hardware resources, respectively.

It is worth noting that we have to deal with the following model requirements:

- R.1 *Distributions*. The predictions for response times are required as distributions $\mathcal{N}(\mu_y, \sigma_y^2)$ and not as single values.
- R.2 *Real-time*. Compute “good enough” predictions in real-time (ca. 1 ms).

R.3 Portability. The prediction model is generated in the programming language *python* but needs to be embedded externally into the overall tool (C#-code).

The requirement R.1 is due to the SMC approach followed in this work. R.2 is necessary since the simulation of the SUT is done with a virtual time, i.e., a fraction of the actual time, in order to simulate thousands of requests within hours which would otherwise take days on the SUT. R.3 is a consequence of R.2. More precisely, we can further improve the predicting speed if the algorithm which computes the prediction is natively implemented directly in the module which needs the prediction and not in an external module which would cost additional (precious) time to invoke. As we will see below, the above specific model requirements will affect some choices we have to make during the model learning process.

For learning the cost model described in Section 3, the journey starts with defining the experimental setup. Corresponding log-files, one log-file for each simulated user, are then generated as explained in Section 4.1. The log-files contain many examples of requests, typically up to two million examples all together. For each request, the log-file specifies a number of related attributes including, importantly, the response time, i.e., the time needed by the SUT to process the request and send back a reply. The journey for learning the cost model goes further through several phases as described below.

Data cleaning and pre-processing The log-files are analyzed with help of descriptive statistics and visualizations, e.g., histograms, box plots, pair-wise scatter plots of attributes, etc. If there is any evidence that something unusual happened during the logging phase, e.g., many requests with either unrealistic response times (too long or too short) or missing values for mandatory request attributes due to, e.g., some process which crashed or got stuck, then new log-files are generated. Otherwise, various data cleaning tasks are performed on the current log-files:

- Sporadic requests with missing values for mandatory attributes are removed.
- Non-mandatory request attributes with missing values are filled with default values. E.g., *Attribute* is not required to have a value for every request type.

Thus, we fill missing values of *Attribute* with a default value “NOTSET” as a further category.

- Request examples with an error message received back from the SUT are removed since our goal is to test properties of the SUT under normal usage and network conditions.
- The request examples generated during the first five minutes of the logging phase are removed since during this time various system initialization tasks (e.g., database user authentication, network connection setup, etc.) are performed. These initialization tasks affect in an atypical way the response times of the system and should not be considered for learning (s. Fig. 5).
- Sporadic request examples with unusually long time responses, e.g., due to some temporary network problems, are considered outliers and removed, for instance the data point in Fig. 5 around time 01:20.

Feature selection and engineering Relevant attributes (features) are selected which are believed to influence the response time of the SUT. Again, descriptive statistics and visualizations of the data from the log-files together with a good knowledge about the way how the SUT works and is built are key factors for identifying relevant features. The better we understand the data, the better and more accurate the models that we can build (Tang et al.

Fig. 5 Aggregated response times (marker symbol indicates aggregation size)

2014). For instance, Fig. 5 suggests that the response time depends on some variable changing over time which, excluding the possibility of any memory leak, is very likely to be the database fill level of the SUT.

Table 1 illustrates the request attributes that we found most relevant for learning response times. Thus, we anticipate that our response time prediction model is a function of those variables. However, it turns out that combining some variables for creating new features (feature engineering), instead of using them individually as separate features, leads to better results. For example:

- Using the concatenation *Task.Subtask* as a feature instead of two separate features *Task* and *Subtask* would improve the model performance by ca. 9%. This seems reasonable since different task types have subtasks with the same name but different effects, depending on the corresponding task type. Using *Subtask* as a standalone feature would introduce some noise for the learning process. This noise accounts for being less precise in capturing the different effects of a subtask when the subtask name is associated with different task types.

For instance, both tasks '*Create*' and '*AdminEdit*' have a subtask called '*Commit*'. However, the subtask '*Commit*' associated with the task '*Create*' typically implies more objects being stored in the SUT database, hence it is more “costly” than '*Commit*' associated with '*AdminEdit*'.

- Using as feature the multiplication of *CumulativeObjSize* by a boolean variable (*True* = 1, *False* = 0) indicating whether a request requires SUT database access or not, instead of using *CumulativeObjSize* alone as feature for all requests, further improves the model performance by ca. 10%. This seems also reasonable since using *CumulativeObjSize* for all requests during the learning process, independently whether a request requires database access or not, would clearly introduce noise which accounts for giving weight to a property (database fill level) even if the response time for a request does not depend on that property.

Eventually, we define a list of features including both *raw* features, i.e., request attributes as recorded in the log-files, and *engineered* features, i.e., combinations of raw features. The

values of the features in the log-files might be integers, floats, or strings. Note that missing values do not occur at this time in the data as they have been previously resolved during the data cleaning and pre-processing phase. That is, they have been either filled with default values or removed together with the corresponding rows.

A feature whose values are strings or ordinal numbers encoding some categories is called categorical feature. The possible values of a categorical feature are the categories of that feature. Since the multiple linear regression algorithm that we are going to use next can only handle numerical features for computing appropriate mathematical operations, we need to transform categorical features into equivalent numerical features.

In case of categorical independent features, different coding techniques are available to transfer the feature categories into a linear regression model. (If they are not independent, interaction terms can be added (Jaccard et al. 2003).) The simplest is dummy coding, where for each category i of a categorical feature a (new) binary dummy feature is introduced. For each example in the dataset, the new binary feature is set to 1 if the categorical feature has value i for that example, and set to 0 otherwise. By definition, these binary features are linearly dependent, because the sum of the columns corresponding to all binary dummy features related to the same categorical feature leads to a column of 1s. Therefore, to avoid singularity problems, for each categorical feature it is necessary to have one dummy feature less than the number of categories. The category that has no dummy feature is called the reference group of the model. It has 0s in all dummy features. (For more details, see (Rencher and Christensen 2012)). In Listing 1, *Create_Commit* and *Edit_Commit* are two of the binary dummy features derived from the (engineered) categorical feature *Task_Subtask*, whereas *Attribute_NOTSET* and *Attribute_Responsible* are derived from the (raw) categorical feature *Attribute*.

We selected the multiple linear regression algorithm (see Section 3) to learn the prediction model for response time distributions from the log-files. This is a good match for the specific model requirements R.1–3 highlighted above. Moreover, the learned model makes good enough predictions for the purpose of the work presented in this paper (see Section 6).

We use the programming language *python* 2.7 with the *scikit-learn*⁶ 0.19.1 machine learning package for model prototyping, e.g., testing of various combinations of features, estimating the accuracy of an algorithm using k-fold cross validation, etc. Once we identified the set of features with the highest predictive power, we use the *StatsModels*⁷ 0.8.0 package to generate a deployment model candidate from the *entire* dataset.

Scikit-learn follows the machine learning tradition where the main supported task is choosing the best model for prediction. That is, the emphasis in the supporting infrastructure in *scikit-learn* is on model selection for best predictions of new, previously unseen samples with cross-validation on test data.

StatsModels follows the statistics tradition where we want to know how well a model fits the data, which features explain or affect the labeled variable, or what the size of the effect is. That is, the emphasis in the supporting infrastructure of *StatsModels* is on analyzing the training data (hypothesis tests) and deriving complex statistical properties of resulting estimators, e.g., standard errors, p values, etc. This points out the distinction between *StatsModels* and *scikit-learn*. Thus, while there is a lot of overlap, e.g., *StatsModels* also does prediction, it is easier to use the cross-validation support of *scikit-learn* for

⁶<http://scikit-learn.org>

⁷<http://www.statsmodels.org>

performing cross-validation for prediction, whereas it is easier to use the statistics support of *StatsModels* for generating the parameters $(\mu_{\beta_k}, \sigma_{\beta_k})$ required by the model (7) from Section 3.

Model predictive power evaluation The dataset used to train a machine learning algorithm is called training dataset. The latter cannot be used to give reliable estimates of the accuracy of the model on new data. This is a big problem because the whole idea of creating the model is to make predictions on new data. However, we use statistical methods called resampling methods to split the dataset up into subsets; some are used to train the model and others are held back and used to estimate the accuracy of the model on unseen data. That is, we split the input dataset into training and test sets; we train the model on the training set and estimate its accuracy on the test set (Hastie et al. 2009).

To reduce the possible effect that the model performs well just by chance with a selected train/test split, we estimate the accuracy of the modeling algorithm using k -fold cross validation. More precisely, we randomly partition the input dataset into k equal sized subsets. Of the k subsets, we hold back a single subset as test set and use the remaining $k-1$ subsets as training set. We repeat the cross-validation process k times, with each of the k subsets used as test set exactly once. If each time we obtain comparable results, we conclude that it is unlikely that they are due to chance. Typical values for k are 5 or 10 but in general k remains an unfixed parameter and depends on the size of the input dataset.

A commonly used metric to evaluate how well a prediction model for regression fits a given (labeled) dataset is the coefficient of determination, denoted R^2 (Wright 1921):

$$R^2 := 1 - \frac{\sum_{i=1}^N (y_i - \hat{y}_i)^2}{\sum_{i=1}^N (y_i - \bar{y})^2} \quad (8)$$

where in our context N is the total number of request examples from the given dataset (pre-processed log-files), y_i is the response time recorded in the dataset for the i^{th} request example (the i^{th} label), $\hat{y}_i$ is the response time that the model would predict for the i^{th} request example, and $\bar{y}$ is the mean value of all N response time values recorded in the dataset (the mean of all labels). The score of R^2 is between 0 and 1. A value of 0 corresponds to a constant model that predicts the mean value of all response time values from the training set. A value of 1 corresponds to the perfect prediction. Intuitively, R^2 gives a number that indicates the proportion of the variance in the dependent variable y that is explained by the model.

To ensure a better model stability and robustness, it is generally recommended to apply feature scaling, e.g., normalization or standardization, to the data before training an MLR algorithm on the data. Recall that normalization is rescaling the range of features to $[0, 1]$ or $[-1, 1]$ and standardization is making the values of each feature in the data have zero-mean and unit-variance. However, whether the model benefits from feature scaling or not strongly depends on the data. Therefore, in order to verify whether standardizing or normalizing the data makes a difference in our case, we perform the model evaluation in turn on the raw, normalized, and standardized data, and compare the results.

We use the *LinearRegression* class from the *scikit-learn* package to train and test an MLR model with the k -fold cross-validation technique, e.g., $k = 5$. We generally obtain R^2 -scores in the range $[0.75, 0.95]$, depending on the complexity of the SUT property to test which determines the experimental setup generating the log-files. We obtain similar R^2 -scores on both train and test sets at all iterations of the k -fold cross-validation which indicates that the model does not suffer from overfitting, i.e., scoring well on the training but badly on the test data.

Importantly, we do not see any significant difference in the above results, whether we make the evaluation on the raw, normalized, or standardized data. This is good news since we can avoid feature scaling without any loss, while contributing at the same time to the fulfillment of the requirements R.2–3, as we will see in the next phase below.

It is worth mentioning that other algorithms (e.g., polynomial regression, random forests) have been tested as well. However, the achieved R^2 -scores were *just slightly* better so that the increased complexity was not worth it. Further investigations would be necessary, especially with regard to the fulfillment of the request R.1. This might be subject of future work if, while using this work in practice, it turns out that better models are needed for the statistical model checking part.

Deployment model generation We use the *OLS* class from the *StatsModels* package to generate a deployment model from the *entire* dataset. We no longer need the cross-validation models or the train/test sets from the previous phase since they have just served to get the confidence that we can get a good model from the log-files. Once we have got this confidence, we can use the entire dataset to generate the model.

The parameters $(\mu_{\beta_k}, \sigma_{\beta_k})$ required by the model (7) belong to the statistics that *OLS* delivers out-of-the-box along with the model. Thus, the requirement R.1 is fulfilled. In order to fulfill the requirements R.2–3, we abstain from applying feature scaling to the data before training the MLR algorithm to learn the deployment model. The reason is that feature scaling would negatively impact both (i) the model performance when it is used to make predictions at run-time (feature scaling has to be applied also at run-time to the data for which we want to make predictions), and (ii) the model portability (the same features transformation used at model training time needs to be performed also at run-time). Moreover, as showed in the model predictive power evaluation phase, feature scaling does not really make a difference in our case. The simple, linear form of the model (7) contributes to the fulfillment of the requirements R.2–3 as well.

Listing 1 shows an excerpt from a deployment model candidate generated with *OLS*. In the left column are the intercept and the regressor variables corresponding to $[x_0, \dots, x_p]$ (including both raw and dummy binary features) from the model (7). The second column shows the estimates of means corresponding to the parameters μ_{β_k} . The third column shows the empirical standard errors of the estimates of means, corresponding to the parameters σ_{β_k} . The fourth column contains the t values, i.e., the ratio of estimate and standard error. The p values in the last column describe the statistical significance of the estimates: low p values indicate high significance.

We use the p values to refine the model by excluding features with low significance. For instance, *#Attributes* in Listing 1 has a high p value, that is, its estimate has low significance. If we generate a new model, this time without the feature *#Attributes*, we obtain a similar R^2 -score but with a simpler model. We typically iterate several times through the model generation phases by adding/removing features until a good simplicity/ R^2 -score tradeoff of the model is reached.

Finally, the right-hand side of Fig. 4 intuitively shows how the generated deployment model is used externally in conjunction with the behavioral model. Briefly, the function *cost*,

$$\text{cost} : \text{Task} \times \text{Subtask} \times \mathbb{N}_{>0} \times \text{Attribute} \times \mathbb{N}_{>0} \times \mathbb{N}_{>0} \times \mathbb{N}_{>0} \rightarrow \mathbb{R}_{>0} \times \mathbb{R}_{>0}$$

is a wrapper around the model (7) which takes as input the values of the *raw* features of a request r for which a prediction is required. It first checks the features *Task* and *Attribute* and if necessary, it fills them with the default value “NOTSET,” as during the model training in

the data cleaning and pre-processing phase. It then computes the values of the engineered features and applies the dummy coding technique to derive the values for the dummy binary features corresponding to the categorical features, as during the feature engineering phase. Note that at this point, we have all the values $[x_0, \dots, x_p]$ from (7). Finally, *cost* uses (i) the formulas (7), (ii) the previously computed values $[x_0, \dots, x_p]$, and (iii) the parameters $(\mu_{\beta_k}, \sigma_{\beta_k})$ given by the deployment model to compute the mean μ_y and the standard deviation σ_y of the normal distribution we consider the response time y of the request r comes from.

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	26.2752	0.1714	153.2138	0.0
#Users	4.4268	0.0149	297.0559	0.0
#Attributes	0.8998	2.1773	0.4132	0.6793
ObjectSize	0.0023	0.0028	0.8253	0.4091
CumulativeObjSize	1.2698e-06	4.3829e-09	289.7210	0.0
AdminEdit_SetRefAttribute	4.9903	0.1359	36.7000	4.9008e-294
AdminEdit_StartTask	0.8427	0.5624	1.4985	0.1340
Create_Commit	-7.9862	0.2186	-36.5286	2.5241e-291
Create_SetRefAttribute	4.5492	0.1452	31.3174	7.4209e-215
Create_StartTask	0.6055	0.5650	1.0716	0.2838
ChangeState_Commit	-4.6380	4.6892	-0.9890	0.3226
ChangeState_SetRefAttribute	4.6955	0.2157	21.7593	7.1371e-105
ChangeState_StartTask	1.0689	0.5769	1.8528	0.0638
Edit_Commit	-2.7279	0.3268	-8.3468	7.0478e-17
Edit_SetRefAttribute	5.2434	0.3042	17.2366	1.5486e-66
Edit_StartTask	4.1145	0.6213	6.6217	3.5560e-11
Attribute_NOTSET	-15.4171	0.5172	-29.8043	7.8762e-195
Attribute_Responsible	-18.6690	0.2332	-80.0309	0.0

Listing 1 Deployment Model generated with OLS (Excerpt)

The function $sample : \mathbb{R}_{>0} \times \mathbb{R}_{>0} \rightarrow \mathbb{R}_{>0}$, takes as input the pair (μ_y, σ_y) from the function *cost* and draws a value from the normal distribution $\mathcal{N}(\mu_y, \sigma_y^2)$. It returns this value as the prediction for the response time y of the request r .

4.3 Model simulation and verification of the SUT

In order to simulate the interaction of users with the SUT, it is also necessary to model the typical behavior of users. A user profile specifies the typical behavior of a class of users, e.g., the frequency of task executions, the pauses between tasks and the time needed to input data.

For our use case, they are represented by weights for tasks, by waiting intervals between tasks/subtasks and additionally by waiting factors for the input duration, e.g., a delay per character for the time to enter a text. The transition probabilities resulting from the task weights are shown on the top of Fig. 6. Note, we also included the probabilities for *select* transitions, which allow a switch between active Business Process Templates. On the bottom

Fig. 6 User profile of the Business Process Template model

of the figure, a representation of this user profile is shown in the JavaScript Object Notation (JSON) format, which was used for storage. It also includes the mentioned waiting intervals and factors.

This user profile is joined with the cost model in order to obtain a combined model that can be applied to simulate a user. A user population is simulated by executing this model concurrently within one of our SMC properties, which were explained in Section 2.4. The combined model has the semantics of a stochastic timed automaton (Ballarini et al. 2013). Note, for these waiting times, we also introduce states in a similar way as for the subtasks, as illustrated in Fig. 4.

In order to estimate the probability of response-time properties, we perform a Monte Carlo simulation with Chernoff-Hoeffding bound. However, this simulation requires too many samples to be efficiently executed on the SUT and so we only run it on the model. For example, checking the probability that the response time of all subtasks is under a threshold of 50 ms for each user of a population of 20 users with parameters $\epsilon = 0.05$ and $\delta = 0.01$, requires 1060 samples and returns a probability of 0.806, when a test-case length of four tasks is considered.

Fortunately, hypothesis testing requires fewer samples and is, therefore, better suited for the evaluation of the SUT. The probability that was computed on the model serves as a hypothesis to check, if the SUT is at least as good. We apply it as alternative hypothesis and select a probability of 0.556 as null hypothesis, which is 0.25 smaller, because we want to be able to reject the hypothesis that the SUT has a smaller probability. Additionally, we want to check if the probability of the SUT is not much higher than the probability computed on the model, in order to assess the prediction power of our model. Hence, we also test a probability of 1, which is about 0.2, as a null hypothesis with a second SPRT and the same alternative hypotheses. By running SPRT (with 0.01 as type I and II error parameters) for each user of the population, we can check these hypotheses.

The alternative hypotheses were accepted for both SPRTs and for all users which means that the model's prediction was accurate. This verification phase on the SUT is indeed efficient: on average only 17.55 and 11 samples were needed for the first and second SPRTs, respectively.

5 Architecture and implementation

In this section, we show the integration of the cost models and user profiles into a collective model, and we illustrate how such models can be simulated with PBT.

We already presented an existing implementation of MBT with FsCheck (Aichernig and Schumi 2016a), which supports automatic form-data generation and EFSMs. Based on this work, we implemented the following extensions in order to support our new method. The first extension is a parser that reads the learned response-time distributions and integrates them into the model. In the previous implementation, we had command instances, which represent the tasks and generators for different data types (for form data). Now, we introduce new cost generators for sampling costs or response times, which can be applied in the same way as normal generators for form data. During the test-case generation, the generated costs can be evaluated within the commands. This helps to check response-time properties.

Algorithm 2 represents the implementation of a cost generator. The inputs are a task, a subtask, an attribute, an array of encapsulated attributes (for requests that transfer multiple attributes) and a cost function, which returns the parameters μ and σ of the normal distribution. Additionally, there are global variables *ActiveUserNum* and *CumulativeObjSize*,

which are shared by all users. The generator is expressed as a function that is called during the generation process and it works as follows. First, a sequence generator is applied to generate values for the *encapsulatedAttr* array and the *select* function further processes the generated values and constructs a new generator that is then returned. This function takes an anonymous function, which takes the values as input and returns a new value that can have a different type. It can be applied to convert a generator of certain type A to one of type B by processing the generated values of the first generator:

$$Gen[A].select : (A \rightarrow B) \rightarrow Gen[B]$$

Inside this function, the number of active users is increased to simulate a request. (The access to *ActiveUserNum* should be locked to avoid race conditions.) Then, a value is sampled according to the normal distribution and assigned to the *delay* variable. The sample is created with the parameters μ and σ from the cost function that was explained before. (Note that *encapsulatedAttr.length* represents the number of attributes that are set by a subtask (#Attributes), and *sizeOf(data)* is the size of the generated attribute data, i.e., the *ObjSize* argument of the cost function.)

Algorithm 2 Pseudo code of a cost generator.

Inputs: Task *t*, Subtask *st*, Attribute *a*, Attribute[] *encapsulatedAttr*, function *cost* : (...) $\rightarrow (\mu, \sigma)$

Global Variable: *ActiveUserNum* $\in \mathbb{N}$, $\triangleright$ number of users that have an open request
CumulativeObjSize $\in \mathbb{N}$ $\triangleright$ sum of the data sizes of the created objects

```

1: function Generator
2:   return Gen.sequence(encapsulatedAttr).select(data  $\rightarrow$  {
3:     ActiveUserNum  $\leftarrow$  ActiveUserNum + 1  $\triangleright$  should be locked (Mutex)
4:     delay  $\leftarrow$  sample(cost(t, st, ActiveUserNum, a, encapsulatedAttr.length,
                          sizeOf(data), CumulativeObjSize))  $\triangleright$  sample normal
                          distribution
5:     sleep(delay)  $\triangleright$  thread should sleep
6:     ActiveUserNum  $\leftarrow$  ActiveUserNum - 1  $\triangleright$  should be locked (Mutex)
7:   return delay}

```

Next, the thread is put to sleep for the duration that was generated with the cost function. Then, the number of users is decreased again. Finally, the generated delay is returned so that it can be checked outside the generator. Note, this generator function also applies the generated delay. This is done, because we need to know the number of active users for the generation of a sample and in order to know which user is active it is necessary to directly execute this behavior, so that we have active users during the generation step. Multiple users are executed concurrently in different threads in an independent way. However, their shared variable *ActiveUserNum* causes a certain dependency between the user threads, because when one user increases this variable, then this affects the response-time distributions of the other users.

The user profiles are also parsed and their user behavior is added into the combined model. The user input-durations that represent the time needed for filling web forms can be integrated in a similar way as the cost functions by introducing input-duration generators. Their implementation details are omitted, as they work in the same way as cost generators except that they do not change the number of active users and they use a uniform distribution instead of a normal distribution.

With both these generators, we are able to implement the sequence of subtasks of tasks as represented in Fig. 7. Input-duration generators represent the time that a user needs for

Fig. 7 Generator sequence of a task, that are executed with a sequence generator

the input (e.g., for filling forms) and cost generators simulate the response times of different requests. These generators are instantiated with different parameters depending on the request type. Algorithm 2 shows the necessary parameters for the instantiation. It is important to point out that the model can be simulated with a virtual time, i.e., a fraction of the actual time. The selection of the tasks according to the given weights was implemented with a frequency generator. A frequency generator takes a set of weights and generator *Gen* pairs and selects one of the generators according to the weights.

$$Gen.frequency : \mathcal{P}(\mathbb{R}_{>0} \times Gen) \rightarrow Gen$$

This generator was applied in order to choose commands, which handle the execution of tasks. The generator for commands does not only generate commands, but also their required data. We implemented a test-case generation process that works for both, conventional PBT in order to produce our logs, as well as for the response-time simulation with the cost models. For the log creation, we apply generators for the form data and execute the resulting test cases on the SUT. For the model simulation, we apply the cost and input-duration generators and check if the produced test cases fulfill our response-time properties. Algorithm 3 outlines this process. It requires a state-machine specification *spec*, which includes a generator for the next state and the initial state of the model. First, the initial model is retrieved from a function of the *spec*. Then, there is an iteration over the size parameter and in each iteration the *next* function of the *spec* is called to obtain a command generator for the current model state. A command *cmd* is sampled according to this generator (Line 4) and executed on the model *cmd.runModel* in order to retrieve a new model, which incorporates the applied state change. This model (state) is updated in each iteration with the *next* function which behaves as follows: First, a set of pairs of weights and task generators is retrieved from the *getEnabledTasksWithWeights* function of the model. Based on this set, a frequency generator is built (Line 8). The function *selectMany* of this generator is called to further process the selected value. This function can be applied to a generator in order to build a new generator. It needs an anonymous function as argument, which takes a value of the generator as input and has to return a new generator.

$$Gen[A].selectMany : (A \rightarrow Gen[B]) \rightarrow Gen[B]$$

Within this function, a sequence generator is called that generates the response times and times for the user input, based on the generator sequence of the task, which is, e.g., shown in Fig. 7. (Alternatively, this generator can also be applied to produce form data, when an evaluation of the SUT is performed instead of a model simulation.) The *selectMany* function is applied again on this generator and within this function a command generator is created for the given task and data.

Algorithm 3 Pseudo code of the test-case generation.

Input: *spec*: state-machine specification of a PBT tool, *size*: parameter for test-case length

```

1: model  $\leftarrow$  spec.initialModel()
2: for  $i \in \{1, \dots, \text{size}\}$  do
3:   gen  $\leftarrow$  spec.next(model)                                 $\triangleright$  next returns a command generator
4:   cmd  $\leftarrow$  gen.sample()                                     $\triangleright$  command is generated
5:   model  $\leftarrow$  cmd.runModel(model)                           $\triangleright$  command is executed
6: function spec.next(model)
7:   set  $\leftarrow$  model.getEnabledTasksWithWeights()               $\triangleright$  set of (weight, Gen[Task])
8:   return Gen.frequency(set).selectMany(task  $\rightarrow$ 
9:     Gen.sequence(task.Generators).selectMany(data  $\rightarrow$ 
10:      CmdGenerator(task, data)))                              $\triangleright$  generate form data or costs
                                                                 $\triangleright$  generator for a command
```

6 Evaluation

We evaluated our method for a web-service application from the automotive domain, which was explained in Section 2.1 and we applied it to two major modules of this application, the Test Order Manager and the Test Equipment Manager. Their descriptions are based on a previous work (Aichernig and Schumi 2017b), where we performed classical PBT for these modules and we also presented the functional models in detail. Now, we present a performance evaluation of this system. We focus on the response times and the number of samples needed, and also present run times of the simulation and testing process.

Settings The evaluation was performed in a distributed environment at AVL. The TFMS server (version 1.8) was running on a virtual machine with Windows Server 2012, 15 GB RAM and 7 Intel Xeon E5-2690v4 2.6 GHz CPUs. The test clients that simulated the users were executed in a separate virtual machine with Windows Server 2008, 6 GB RAM and 3 Intel Xeon E5-2690v4 2.6 GHz CPUs. The logs for the cost-model learning were created on these test clients and they were applied to evaluate our models. For both, the test-case generation for the logs and the simulation with SMC, we applied the PBT tool FsCheck version 2.8.2.

6.1 Test Order Manager

The Test Order Manager is the main module of our SUT and it 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 8 shows the tasks of an example test order.

Fig. 8 Example test order model

Each task represents the invocation of a form, entering data for form fields and saving the form. The Test Order Manager contains further sub-models for the creation of test orders, like Business Process Template, but they are similar to this model, and are therefore omitted.

We applied our method in order to check the following property: *What is the probability that the response time of all subtasks of a task sequences with a fixed length, i.e. a test case, is under a specific threshold?* We check this property for a given user that is part of a user population of a specific size. For this evaluation a user profile was created in cooperation with domain experts from AVL. This profile was similar to the one shown in Section 4, and is illustrated in Listing 2. Note that better user profiles could be obtained by monitoring a live system with real users. Unfortunately, this was not possible in our case, because we did not receive approval from TFMS customers.

```
{TaskWeights:{
  TestOrder:{ToCreate:35, MakeReady:1, Finish:1, Activate:1, Duplicate:1, Reject:1,
    AdminEdit:1, EditCreated:65, EditStandardWorkInWork:65,
    EditStandardWorkExecuted:65, CancelInCreated:1,
    CancelInStandardWorkInWork:1, CancelInStandardWorkExecuted:1,
    CancelInFinished:1, CancelInInvalid:1, Invalidate:1, Select:5,
    SelectREM:1},
  BusinessProcessTemplate:{BptCreate:35, AdminEdit:1, Edit:65, ChangeState:1,
    Select:5, SelectREM:5}},
TaskWaitIntervalStart:500, TaskWaitIntervalEnd:1500, SubTaskWaitIntervalStart:300,
SubTaskWaitIntervalEnd:500, WaitPerReference:10, WaitPerCharacter:30}
```

Listing 2 User profile of the Test Order Manager

The multiple linear regression model was similar to the one of Section 4 as well and is shown in Listing 3.

We applied the profile to form user populations of different sizes, and we checked the proposed property for test cases with increasing lengths via a Monte Carlo simulation with Chernoff-Hoeffding bound with parameters $\epsilon = 0.05$ and $\delta = 0.01$. (This requires 1060 samples per data point.) The results for an empty database (*CumulativeObjSize* = 0) and for a database size that represents about 14,000 test orders (*CumulativeObjSize* = 80, 000, 000) are shown in Figs. 9 and 10. Note, we selected the user-population sizes (5,25,45) by starting from a trivial size of five users and selected a step size that showed a significant difference. We chose thresholds that led to interesting probabilities, but normally these thresholds should be based on customer requirements.

As expected, a decrease in the probability that the property holds can be observed, when the test-case length or the population size increases. Moreover, the size of the database has an important influence on the response times. We can see that the response times increase when the database size rises. The advantage of the simulation on the model-level is that it runs much faster than on the SUT. With a virtual time of 1/10 of the actual time, we can perform simulations that would take days on the SUT within hours.

It is also important to check the probabilities that we received through model simulation on the SUT. This was done as explained in Section 4 by applying the SPRT with the same parameters. Table 2 shows the results. Due to the high computation effort, we only check a limited selection of data points of Fig. 9.

The table shows the hypotheses and evaluation results for different thresholds, different numbers of users and for the two database fill levels (*CumulativeObjSize*). As explained in Section 4.3, we perform two SPRTs, one to check if the SUT is not much worse than the model, and one to check if the SUT is not much better than the model. The alternative hypothesis H_1 is produced via the model simulation and is the same in both SPRTs, but the null hypotheses are different (smaller or larger). As result, we report the accepted

	Estimate	Std. Error	t-value	Pr(> t)
(Intercept)	25.570	0.194	131.200	0.0
#Users	4.980	0.016	299.814	0.0
#Attributes	2.473	2.477	0.998	0.318
ObjectSize	0.000	0.003	0.054	0.956
CumulativeObjectSizeForRisingTasks	1.291e-06	4.972e-09	259.800	0.0
AdminEdit_SetRefAttribute	4.890	0.154	31.592	1.302e-218
AdminEdit_StartTask	0.728	0.639	1.138	0.254
BptCreate_Commit	-7.968	0.248	-32.027	1.343e-224
BptCreate_SetRefAttribute	4.353	0.165	26.349	8.694e-153
BptCreate_StartTask	0.371	0.642	0.577	0.563
CancelInCreated_Commit	3.474	1.455	2.387	0.016
CancelInCreated_StartTask	1.278	0.678	1.885	0.059
CancelInFinished_Commit	3.483	1.663	2.094	0.036
CancelInFinished_StartTask	-4.583	1.038	-4.414	1.013e-05
CancelInStandardWorkExecuted_Commit	2.102	1.941	1.082	0.273
CancelInStandardWorkExecuted_StartTask	-3.765	1.428	-2.636	0.008
CancelInStandardWorkInWork_Commit	3.337	1.517	2.199	0.027
CancelInStandardWorkInWork_StartTask	-4.357	0.792	-5.500	3.794e-08
ChangeState_Commit	-1.231	5.335	-0.230	0.817
ChangeState_SetRefAttribute	4.484	0.245	18.267	1.677e-74
ChangeState_StartTask	1.055	0.656	1.608	0.107
Duplicate_Commit	56.725	0.243	233.013	0.0
Duplicate_SetRefAttribute	4.779	0.227	21.032	4.099e-98
Duplicate_StartTask	2.405	0.653	3.683	0.000
EditCreated_Commit	2.253	0.315	7.149	8.720e-13
EditCreated_SetRefAttribute	4.627	0.289	16.004	1.267e-57
EditCreated_StartTask	0.855	0.680	1.256	0.208
EditStandardWorkExecuted_Commit	3.400	1.712	1.985	0.047
EditStandardWorkExecuted_StartTask	-4.032	1.123	-3.588	0.000
EditStandardWorkInWork_Commit	4.678	1.521	3.075	0.002
EditStandardWorkInWork_StartTask	-3.125	0.793	-3.937	8.232e-05
Edit_Commit	-2.849	0.371	-7.661	1.847e-14
Edit_SetRefAttribute	5.042	0.346	14.572	4.389e-48
Edit_StartTask	4.038	0.706	5.713	1.108e-08
Finish_Commit	4.223	1.773	2.381	0.017
Finish_StartTask	-2.666	1.208	-2.206	0.027
MakeReady_Commit	4.294	1.451	2.958	0.003
MakeReady_SetRefAttribute	5.717	0.335	17.050	3.794e-65
MakeReady_StartTask	1.395	0.671	2.076	0.037
Reject_Commit	4.792	1.515	3.162	0.001
Reject_StartTask	-4.272	0.784	-5.442	5.264e-08
SelectREM_Load	-9.469	0.637	-14.848	7.449e-50
SelectREM_Open	-5.106	0.637	-8.007	1.178e-15
Select_Load	-9.177	0.636	-14.410	4.675e-47
Select_Open	-5.456	0.636	-8.567	1.064e-17
ToCreate_Commit	-4.903	0.335	-14.617	2.272e-48
ToCreate_SetRefAttribute	6.886	0.141	48.576	0.0
ToCreate_StartTask	-2.340	0.714	-3.278	0.001
Attribute_NOTSET	-15.211	0.588	-25.847	4.094e-147
Attribute_ParentFolder	-31.078	0.220	-140.743	0.0
...				

Listing 3 Linear regression model of the Test Order Manager

Fig. 9 Test Order Manager simulation results of the model

Fig. 10 Test Order Manager simulation results of the model with filled DB

Table 2 Test Order Manager results of the SUT evaluation with the SPRT

Threshold [ms]	#Users	Cumulative ObjSize	H_1	1. H_0	Result	#Samples	2. H_0	Result	#Samples	Run time [min:s]
50	5	0	0.728	0.478	H_1	20.6	0.978	$1H_1 4H_0$	14.6	17:56
50	25	0	0.653	0.403	H_1	11.76	0.902	H_0	29.48	51:30
50	45	0	0.456	0.206	H_1	7.68	0.705	H_0	17	22:02
100	5	0	0.997	0.747	H_1	16	–	–	–	11:22
100	25	0	0.995	0.745	H_1	16	–	–	–	11:56
100	45	0	0.986	0.736	H_1	17.84	–	–	–	22:08
100	5	80,000,000	0.428	0.178	H_1	27	0.678	H_1	37.6	28:46
100	25	80,000,000	0.425	0.175	H_1	35.56	0.675	H_1	29.16	38:54
100	45	80,000,000	0.419	0.169	$7H_0 38H_1$	62.82	0.669	H_1	18.22	55:37
150	5	80,000,000	1	0.75	$2H_0 3H_1$	12.8	–	–	–	10:25
150	25	80,000,000	0.999	0.749	$16H_0 9H_1$	9.64	–	–	–	22:40
150	45	80,000,000	0.972	0.722	H_0	5.78	–	–	–	6:52
200	5	80,000,000	1	0.75	H_1	16	–	–	–	11:32
200	25	80,000,000	1	0.75	H_1	12.72	–	–	–	11:56
200	45	80,000,000	1	0.75	H_1	13.82	–	–	–	32:46

hypotheses and how often they were accepted, when it was not always the same hypothesis. Moreover, we show the number of samples that were needed for the SPRT (#Samples) and the run time of this evaluation. We only perform one SPRT if the predicted probability of the model is close to one or zero, because then we are already close enough to the min./max. probability.

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 and we only had limited time in the test environment. Compared to the execution on the model, a smaller number of samples is needed, as the SPRT stops, when it has sufficient evidence.

We can see that in many cases, the alternative hypotheses were accepted, which means that the predicted probability was close enough to the real probability of the SUT. In some cases the null hypothesis was accepted, which means that our model was too optimistic or pessimistic in these cases. We will discuss this later in Section 7.

Moreover, it is apparent that the smaller number of required samples of the SPRT (max. ca. 62) compared to Monte Carlo simulation (1060 samples) allowed us to analyze the SUT within a feasible short time. For example, in the worst case it took only about an hour to apply the SPRT.

6.2 Test Equipment Manager

The Test Equipment Manager is another important module of our SUT. This module enables the administration of equipment that is relevant for the test beds, like measurement devices, sensors, actuators, and various input/output modules. All these test equipment can be created, edited, calibrated, and maintained. A hierarchy of test equipment types is used to classify the test equipment. Test configurations, which are compositions of different test equipment, can also be administrated. The connection of devices via channels can be controlled with this module too.

We performed the same evaluation for the Test Equipment Manager as for the Test Order Manager. The user profile (Listing 4) and the linear regression model (Listing 5) were also similar to the one shown in Section 4.

```
{ TaskWeights: {
  TestEquipmentType: { TetCreate: 20, TetEditGeneral: 80, TetAdminEdit: 1,
                      TetChangeState: 1, Select: 5, SelectREM: 1 },
  TestEquipment: { CreateTestEquipment: 20, TeEditGeneral: 80, MarkAsDefect: 1,
                  Select: 5, SelectREM: 5 },
  TaskWaitIntervalStart: 500, TaskWaitIntervalEnd: 1500, SubTaskWaitIntervalStart: 300,
  SubTaskWaitIntervalEnd: 500, WaitPerReference: 10, WaitPerCharacter: 30 }
```

Listing 4 User profile of the Test Equipment Manager

The results of the Monte Carlo simulation of the model for an empty database (*CumulativeObjSize* = 0) and for a database size that represents about 9,200 test equipment objects (*CumulativeObjSize* = 30,000,000) are presented in Figs. 11 and 12. We can see that the curves for an empty database are similar to that of the Test Order Manager. The curves for a filled database are quite different. This difference is caused by a higher number of subtasks that are dependent on the database size in this module.

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	24.170	0.735	32.873	8.301e-236
#Users	12.583	0.079	158.534	0.0
#Attributes	-18.201	11.522	-1.579	0.114
ObjectSize	0.007	0.014	0.507	0.611
CumulativeObjectSizeForRisingTasks	2.962e-06	1.604e-08	184.665	0.0
CreateTestEquipment_SetRefAttribute	33.426	0.789	42.360	0.0
CreateTestEquipment_StartTask	1.713	2.550	0.671	0.501
MarkAsDefect_Commit	12.167	6.812	1.786	0.074
MarkAsDefect_StartTask	12.630	2.618	4.823	1.414e-06
SelectREM_Load	-1.909	2.539	-0.751	0.452
SelectREM_Open	-4.122	2.539	-1.623	0.104
Select_Load	-1.096	2.538	-0.432	0.665
Select_Open	-1.701	2.538	-0.670	0.502
TeEditGeneral_Commit	-2.073	14.436	-0.143	0.885
TeEditGeneral_StartTask	11.302	2.641	4.279	1.876e-05
TetAdminEdit_Commit	-21.109	24.536	-0.860	0.389
TetAdminEdit_SetRefAttribute	7.646	0.500	15.279	1.187e-52
TetAdminEdit_StartTask	8.616	2.582	3.336	0.000
TetChangeState_Commit	-36.918	6.842	-5.395	6.837e-08
TetChangeState_StartTask	6.045	2.627	2.300	0.021
TetCreate_Commit	-12.211	10.486	-1.164	0.244
TetCreate_SetRefAttribute	12.538	0.450	27.808	1.416e-169
TetCreate_StartTask	0.856	2.553	0.335	0.737
TetEditGeneral_Commit	-46.019	6.847	-6.720	1.817e-11
TetEditGeneral_StartTask	9.978	2.622	3.805	0.000
Attribute_NOTSET	-29.440	1.840	-15.993	1.653e-57
Attribute_ParentType	20.184	0.530	38.078	0.0
...				

Listing 5 Linear regression model of the Test Equipment Manager

We also evaluated the results of the Monte Carlo simulation in the same way as before by applying the SPRT. Table 3 shows the results. For the empty database, we see that the alternative hypothesis was accepted in most of the cases, but for the filled database, the null hypothesis was accepted more often. The model seems to be too optimistic for this database size. We think the reason for this is that we have much more subtasks that are dependent on the database size. In addition, more data is transferred over the network in comparison with the Test Order Manager. This causes more network interference and makes the cost-model learning more difficult. Nevertheless, it was again possible to evaluate the SUT by applying the SPRT with an acceptable number of samples (max. ca. 20) and with a decent run time (max. ca. 13 min).

6.3 Run times of the method

Our method consists of several phases that have different computation times. Here, we give an overview of the timings of these phases in order to illustrate the overall run time of our method and to demonstrate its effectiveness.

In the first step, we generate log data with model-based testing. This initial testing phase took about an hour for both our tested modules, i.e., about 63 min. for the Test Order Manager and about 65 min. for the Test Equipment Manager. The next step was the cost-model learning, which took only about 70 to 100 seconds including the time for data cleaning and preprocessing.

Fig. 11 Test Equipment Manager simulation results of the model

Fig. 12 Test Equipment Manager simulation results of the model with filled DB

The model-simulation times are illustrated in Table 4. Note that these timings were measured on the client machine that was described in the setting. It can be seen that they were very similar for the empty and the filled database. The reason for this is that the major part of the simulation time were the user-input times from the user profiles. For the same reason, we only see a small increase in the simulation time, when the number of users becomes higher. In summary, the simulation time was about 9 to 13 min. for the Test Order Manager and 6 to 9 min. for the Test Equipment Manager.

The last columns of Tables 2 and 3 show the run times of the SPRTs. Note that during the execution of a sample, we stopped when we already observed a higher response time than our threshold, and we only have one run time for both SPRTs, since we check them in one execution. The run times of the Test Order Manager were about 1h in two cases. All other cases were mostly shorter than half an hour and the best cases were about 10 min. The run times of the Test Equipment Manager were shorter due to less complexity. They were always below 15 min and in the best cases about 3 min.

Executing the Monte Carlo simulation that we applied for the model directly on the SUT would take about one day. By applying the SPRT, we can perform such an evaluation within less than an hour in the worst case.

7 Discussion

The evaluation showed that our simulation approach allows us to estimate the probability that a user can perform task sequences without having to wait longer than a specific threshold for a system response. Moreover, we demonstrated that we can check if the estimated probability is close to the real probability of the SUT with an acceptable number of samples. In some cases however, the models were not able to estimate the probability accurately enough: they were either too optimistic or too pessimistic. This is an indication that the prediction errors, i.e., differences between predicted and actual response times, were probably too large. Such an issue would have a straightforward explanation if the R^2 -score obtained at model training time (see Section 4.2) was rather low. However, critical prediction errors can occur also when the R^2 -score seems to be high enough. These situations are more difficult to detect in practice, especially with a distributed environment setting, and might be due to several reasons:

- (1) *Measurement errors.* Some noise factors, e.g., variable network latency, memory cache misses, blocking effects of the SUT, etc., might have unevenly increased artificially the actual response times recorded in the log-files. We could still obtain a reasonably high R^2 -score if by chance we were able to identify some linear dependencies in the log-data. However, the predictions at run-time are not as good as indicated during the

Table 3 Test Equipment Manager results of the SUT evaluation with the SPRT

Threshold[ms]	#Users	Cumulative ObjSize	H_1	1. H_0	Result	#Samples	2. H_0	Result	#Samples	Run time [min:s]
50	5	0	0.973	0.723	H_1	19.6	–	–	–	10:52
50	25	0	0.936	0.686	H_1	16.2	–	–	–	9:46
50	45	0	0.671	0.421	H_1	11.111	0.921	H_0	18.578	13:11
100	5	0	1	0.75	H_1	16	–	–	–	5:46
100	25	0	0.998	0.748	H_1	16	–	–	–	6:10
100	45	0	0.962	0.712	H_1	16	–	–	–	7:49
100	5	30,000,000	0.013	0.125	H_0	14	0.625	H_1	9	3:39
100	25	30,000,000	0.114	–	–	–	0.364	H_1	14.4	2:59
100	45	30,000,000	0.014	–	–	–	0.264	H_1	16.244	3:11
150	5	30,000,000	0.999	0.749	H_0	5	–	–	–	3:19
150	25	30,000,000	0.82	0.57	H_0	6	0.82	H_1	5	3:55
150	45	30,000,000	0.137	0.387	H_0	14.266	–	–	–	3:15
200	5	30,000,000	1	0.75	$3H_0 2H_1$	12.8	–	–	–	4:41
200	25	30,000,000	0.997	0.747	H_0	5.68	–	–	–	4:58
200	45	30,000,000	0.496	0.246	H_0	13.555	0.746	H_1	7.688	6:00

Table 4 Average simulation time [min:s] of the model for the Test Order Manager and the Test Equipment Manager for an empty and filled database

#Users	Test-case length	Simul. time (empty DB)		Simul. time (filled DB)	
		Test Order Manager		Test Equipment Manager	
		Test Order Manager	Test Equipment Manager	Test Order Manager	Test Equipment Manager
5	3	9:24	6:40	9:23	6:40
25	3	9:31	6:51	9:41	6:51
45	3	9:37	7:08	9:45	7:08
5	4	12:46	8:56	12:45	8:57
25	4	12:52	9:09	12:58	9:09
45	4	13:02	9:37	13:04	9:37

model predictive power evaluation phase simply because the same noise factors did not apply also at run-time.

Measurement errors are significantly lower with a non-distributed environment setup where our method generally achieve better results. For this paper, we selected however the less favorable case of the distributed environment setup.

- (2) *Sampling bias.* The simulation for generating the log-files might unintentionally be designed and set up in such a way that all relevant scenarios were not equally likely to be simulated. That is, the log-files do not contain equally many examples of all relevant scenarios. Thus, the model “learns” only the dominant dependencies available in the log-files and fails to make good predictions for samples with dependencies not represented in the log-files.

Additionally, some false dependency might be derived from the (biased) log-files which does not hold in general. For instance, if the number of concurrently active users is monotonically increased instead of being randomly selected during the simulation for generating the log-files, then a misleading positive correlation between the number of active users and the database size arises which does not hold in general. While carefully analyzing the log-data, e.g., by means of diagrams like correlation matrices and scatter plots, helps to reduce the sampling bias risk, we generally cannot avoid it completely.

A threat to the validity might be that one case study with only a specific system cannot show the applicability or generality of our approach. In order to resolve this threat, we have now also applied our method to another application domain, i.e. for a performance comparison of different MQTT brokers (Aichernig and Schumi 2018). However, evaluations in further application areas would still be interesting future work.

An interesting observation, which might be seen also as a weakness of our approach, is that SMC seems to be inefficient when the given threshold of the response-time property to be tested is far below or far above the actual response time. In these cases, the probability of the response-time property does not vary in a significant way with the user population size. SMC wastefully computes the probability for various user population sizes, even if a single run with a fixed user population size, say one user, would be sufficient to get a similar result. This phenomenon can be clearly observed in Fig. 10, where the probability curves of different user population sizes are very close to each other for low and high thresholds, whereas they only go apart for thresholds close to the actual response times where the user population size seems to make a difference.

Finally, efforts to improve the prediction model accuracy, e.g., through non-linear learning methods, might be subject of future work if, while using the presented method in practice, it turns out that better prediction models are needed.

8 Conclusion

We have demonstrated that we can exploit PBT features in order to check response-time properties under different user populations both on a model-level and on an SUT. With SMC, we can evaluate stochastic cost models and check properties like, *what is the probability that the response time of a user within a population is under a certain threshold?* We also showed that we can test the accuracy of such probability estimations on the SUT without the need for an extra tool. A big advantage of our method is that we can perform simulations, which require a high number of samples on the model in a fraction of the time

that would be required on the SUT. Moreover, we can check the results of such simulations on the SUT by applying the SPRT, which needs fewer samples. Another benefit lies in the fact that we simulate inside a PBT tool. This facilitates the model and property definition in a high-level programming language, which makes our method more accessible to testers from industry.

We have evaluated our method by applying it to an industrial web-service application from the automotive industry and the results were promising. First, we presented the learning process for our cost models in detail. Then, we showed that we can apply these cost models to derive probabilities for response-time properties for different population sizes and that we can evaluate these probabilities on the real system with a smaller number of samples. In principle, our method can be applied outside the web domain, e.g., to evaluate run-time requirements of real-time or embedded systems. However, for other applications and other types of costs alternative cost-learning techniques (Hastie et al. 2009; West et al. 2006) may be better suited.

In the future, we plan to apply our cost models for stress testing as they help to find subtasks or attributes that are more computationally expensive than others.

Moreover, we intend to apply our method to evaluate different versions of the SUT, i.e., to perform non-functional regression testing.

Funding information Open access funding provided by Graz University of Technology. This work was funded by the Austrian Research Promotion Agency (FFG), project no. 845582 Trust via cost function driven model-based test case generation for non-functional properties of systems of systems (TRUCONF).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

- Aichernig, B.K., & Schumi, R. (2016a). Property-based testing with FsCheck by deriving properties from business rule models. In *Ninth IEEE international conference on software testing, verification and validation workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016* (pp. 219–228). IEEE Computer Society.
- Aichernig, B.K., & Schumi, R. (2016b). Towards integrating statistical model checking into property-based testing. In *2016 ACM/IEEE international conference on formal methods and models for system design, MEMOCODE 2016, Kanpur, India, November 18-20, 2016* (pp. 71–76). IEEE.
- Aichernig, B.K., & Schumi, R. (2017a). Statistical model checking meets property-based testing. In *2017 IEEE International conference on software testing, verification and validation, ICST 2017, Tokyo, Japan, March 13-17, 2017* (pp. 390–400). IEEE Computer Society.
- Aichernig, B.K., & Schumi, R. (2017b). Property-based testing of web services by deriving properties from business-rule models. *Software & Systems Modeling*.
- Aichernig, B.K., & Schumi, R. (2018). How fast is MQTT? - Statistical model checking and testing of IoT protocols. In *Quantitative evaluation of systems - 15th international conference, QEST 2018, Beijing, China, September 4-7, 2018, proceedings, volume 11024 of lecture notes in computer science* (pp. 36–52). Springer.
- Arts, T. (2014). On shrinking randomly generated load tests. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang, Gothenburg, Sweden, September 5, 2014* (pp. 25–31). ACM.

- Ballarini, P., Bertrand, N., Horváth, A., Paolieri, M., Vicario, E. (2013). Transient analysis of networks of stochastic timed automata using stochastic state classes. In *Quantitative evaluation of systems - 10th international conference, QEST 2013, Buenos Aires, Argentina, August 27–30, 2013. Proceedings, volume 8054 of lecture notes in computer science* (pp. 355–371). Springer.
- Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M. (2004). Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5), 295–310.
- Becker, S., Koziolok, H., Reussner, R.H. (2009). The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1), 3–22.
- Book, M., Gruhn, V., Hülder, M., Köhler, A., Kriegl, A. (2005). Cost and response time simulation for web-based applications on mobile channels. In *Proceedings fifth international conference on quality software (QSIC 2005), 19–20 September 2005, Melbourne, Australia* (pp. 83–90). IEEE Computer Society.
- Bulychev, P.E., David, A., Larsen, K.G., Mikucionis, M., Poulsen, D.B., Legay, A., Wang, Z. (2012). UPPAAL-SMC: statistical model checking for priced timed automata. In *Proceedings 10th workshop on quantitative aspects of programming languages and systems, QAPL 2012, Tallinn, Estonia, 31 March and 1 April 2012., volume 85 of EPTCS* (pp. 1–16). Open Publishing Association.
- Claessen, K., & Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN international conference on functional programming (ICFP'00), Montreal, Canada, September 18–21, 2000* (pp. 268–279). ACM.
- Claessen, K., Palka, M.H., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.T. (2009). Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceeding of the 14th ACM SIGPLAN international conference on functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009* (pp. 149–160). ACM.
- Draheim, D., Grundy, J.C., Hosking, J.G., Lutteroth, C., Weber, G. (2006). Realistic load testing of web applications. In *Proceedings of the 10th European conference on software maintenance and reengineering (CSMR 2006) Bari, Italy, 22–24 March 2006* (pp. 57–70). IEEE.
- Govindarajulu, Z. (2004). *Sequential statistics*. World Scientific.
- Grinchtein, O. (2008). *Learning of timed systems*. PhD thesis. Uppsala University, Sweden.
- Halili, E.H. (2008). *Apache JMeter: a practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd.
- Hastie, T., Tibshirani, R., Friedman, J.H. (2009). *The elements of statistical learning: data mining, inference, and prediction*, 2nd edn. Springer.
- Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301), 13–30.
- Hughes, J. (2007). QuickCheck testing for fun and profit. In *Practical aspects of declarative languages, 9th international symposium, PADL 2007, Nice, France, January 14–15, 2007, volume 4354 of lecture notes in computer science* (pp. 1–32). Springer.
- Hughes, J., Pierce, B.C., Arts, T., Norell, U. (2016). Mysteries of Dropbox Property-based testing of a distributed synchronization service. In *2016 IEEE International conference on software testing, verification and validation, ICST 2016, Chicago, IL, USA, April 11–15, 2016* (pp. 135–145). IEEE Computer Society.
- Jaccard, J., Turrissi, R., Jaccard, J. (2003). *Interaction effects in multiple regression*. SAGE.
- Jinyuan, C. (2012). The application of load runner in software performance test. *Computer Development & Applications*, 5, 014.
- Kalaji, A.S., Hierons, R.M., Swift, S. (2009). Generating feasible transition paths for testing from an extended finite state machine (EFSM). In *Second international conference on software testing verification and validation, ICST 2009, Denver, Colorado, USA, April 1–4, 2009* (pp. 230–239). IEEE Computer Society.
- Krishnamurthy, D., Rolia, J.A., Majumdar, S. (2006). A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering*, 32(11), 868–882.
- Legay, A., & Sedwards, S. (2014). On statistical model checking with PLASMA. In *2014 Theoretical aspects of software engineering conference, TASE 2014, Changsha, China, September 1–3, 2014* (pp. 139–145). IEEE Computer Society.
- Legay, A., Delahaye, B., Bensalem, S. (2010). Statistical model checking: an overview. In *Runtime verification - first international conference, RV 2010, St. Julian's, Malta, November 1–4, 2010. Proceedings, volume 6418 of lecture notes in computer science* (pp. 122–135). Springer.
- Lu, Y., Nolte, T., Bate, I., Cucu-Grosjean, L. (2012). A statistical response-time analysis of real-time embedded systems. In *Proceedings of the 33rd IEEE real-time systems symposium, RTSS 2012, San Juan, PR, USA, December 4–7, 2012* (pp. 351–362). IEEE Computer Society.
- Menascé, D.A. (2002). Load testing of web sites. *IEEE Internet Computing*, 6(4), 70–74.
- Nilsson, R. (2014). *ScalaCheck: the definitive guide*. IT Pro Artima Incorporated.

- Norell, U., Svensson, H., Arts, T. (2013). Testing blocking operations with QuickCheck's component library. In *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, Boston, Massachusetts, USA, September 28, 2013* (pp. 87–92). ACM.
- Papadakis, M., & Sagonas, K. (2011). A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Erlang'11, Tokyo, Japan, September 23, 2011* (pp. 39–50). ACM.
- Pooley, R., & King, P. (1999). The unified modelling language and performance engineering. *IEEE Proceedings-Software*, 146(1), 2–10.
- Rencher, A., & Christensen, W. (2012). *Methods of multivariate analysis. Wiley series in probability and statistics*, 3rd edn. Wiley.
- Rina, & Tyagi, S. (2013). A comparative study of performance testing tools. *International Journal of Advanced Research in Computer Science and Software Engineering Research*, 3(5), 1300–1307.
- Schumi, R., Lang, P., Aichernig, B.K., Krenn, W., Schlick, R. (2017). Checking response-time properties of web-service applications under stochastic user profiles. In *Testing software and systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings, volume 10533 of lecture notes in computer science* (pp. 293–310). Springer.
- Smith, C.U. (1990). Software performance engineering tutorial. In *16th International Computer Measurement Group Conference, December 10-14, 1990, Orlando, FL, USA, proceedings* (pp. 1311–1318). Computer Measurement Group.
- Smith, C.U., & Williams, L.G. (1997). Performance engineering evaluation of object-oriented systems with SPE-ED. In *Computer performance evaluation: modelling techniques and tools, 9th International Conference, St. Malo, France, June 3-6, 1997, proceedings, volume 1245 of lecture notes in computer science* (pp. 135–154). Springer.
- Tang, J., Alelyani, S., Liu, H. (2014). Feature selection for classification: a review. In *Data classification: algorithms and applications* (pp. 37–64). CRC Press.
- Verwer, S., de Weerd, M., Witteveen, C. (2010). A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In *Grammatical inference: theoretical results and applications, 10th international colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings, volume 6339 of lecture notes in computer science* (pp. 203–216). Springer.
- Vinayak Hegde, P.M.S. (2014). Web performance testing: methodologies, tools and challenges. *International Journal of Scientific Engineering and Research (IJSER)*, 2.
- Wald, A. (1973). *Sequential analysis*. Courier Corporation.
- West, B.T., Welch, K.B., Galecki, A.T. (2006). *Linear mixed models*. CRC Press.
- Woodside, C.M., Franks, G., Petriu, D.C. (2007). The future of software performance engineering. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA* (pp. 171–187). IEEE Computer Society.
- Wright, S. (1921). Correlation and causation. *Journal of Agricultural Research*, 20, 557–585.

Bernhard K. Aichernig is a tenured associate professor at Graz University of Technology, Austria. He investigates the foundations of software engineering for realizing dependable computer-based systems. Bernhard is an expert in formal methods and testing. His research covers a variety of areas combining falsification, verification, and abstraction techniques. Current topics include the Internet of Things, model learning, and statistical model checking. Since 2006, he participated in four European projects. From 2004–2016, Bernhard served as a board member of Formal Methods Europe, the association that organizes the Formal Methods symposia. From 2002 to 2006, he had a faculty position at the United Nations University in Macao S.A.R., China. Bernhard holds a habilitation in Practical Computer Science and Formal Methods, a doctorate, and a diploma engineer degree from Graz University of Technology.

Priska Bauerstätter is a Junior Scientist at AIT Austrian Institute of Technology GmbH. She is working on different statistical and machine learning tasks. The main focus is on statistical modeling. Priska holds a bachelor degree in statistics from the University of Vienna and is working on her Master Thesis which focuses on finding good prediction models for thyroid cancer data.

Elisabeth Jöbstl currently works as test manager at AVL List GmbH coordinating all test activities regarding AVL's simulation software products. Earlier, she held the position of the test lead for the test field management software TFMS serving as industrial case study in this article. Apart from management tasks, she lays a strong focus on technical topics such as test automation and code quality. Elisabeth holds a doctorate as well as master and bachelor degrees in computer science from Graz University of Technology. Her research topics aim at automated test-case generation via model-based testing, mutation testing, and the use of appropriate conformance relations.

Severin Kann is an integration engineer at AVL List GmbH in Graz. He is part of a project team dealing with testing of non-functional requirements in an industrial environment. Other fields of activity are automation of load and performance tests for multi-user systems as well as improvement and maintenance of test infrastructure. Severin is interested in formal systems. He studies computer science at the Graz University of Technology and is working on his Bachelor Thesis which focuses on refining formal specifications and test models in Event-B towards model-based testing.

Robert Korošec studied Physics at the Graz University of Technology. In 1996, he joined AVL List, and since then, he has been working in different expert and managing functions in software development units of the “Instrumentation & Test Systems” business unit. Currently Dipl. Ing. Korošec is heading the department “Software Program Management and Test Center”, which manages agile development programs and performs the qualification of software and mechatronic products of AVL’s “Instrumentation & Test System” portfolio for the market. The focus of work is on integrating, piloting, releasing, and maintaining SW intensive solutions used in engineering and test centers of global OEMs in the automotive industry. Current topics of interest are model-based testing of software intensive solutions and applying systems engineering methods to scaled agile development programs.

Willibald Krenn is the Thematic Coordinator of the Dependable Systems Engineering group at the AIT Austrian Institute of Technology GmbH. He is also giving lectures at the Technical University of Vienna. Prior to moving to Vienna, Willibald has been working as a Compiler Engineer at Maxeler Technologies in London and as a post-doctoral researcher at Graz University of Technology. His research interests focus on dependable systems engineering, e.g., automated test-case generation, automated machine-code verification, compiler construction, and rigorous design approaches. He is active in several national and international research projects and regularly serves as a program committee member in a number of conferences.

Cristinel Mateis is an independent Machine Learning practitioner with AIT Austrian Institute of Technology GmbH. He is interested in breaking new grounds to solve complex business and technical problems by using recent advancements in Machine Learning and by leveraging his many years of industry experience backed by his research background in Artificial Intelligence. Cristinel holds a master degree in Computer Engineering from the University of Calabria and a Ph.D. in Knowledge Representation and Reasoning from Vienna University of Technology.

Rupert Schlick is a scientist at AIT Austrian Institute of Technology GmbH, Austria, working in the Center for Digital Safety & Security. His work evolves around model-based technologies and software validation and verification techniques for dependable, cyber-physical, and safety-critical systems. The main focus is on model-based testing and the tool suite MoMuT, with current research interests going towards addressing different non-functional aspects, like security, safety, and performance in addition to functional testing. Rupert holds a Master degree in Technical Mathematics, Information and Data Processing and collected close to a decade of experience in the software industry before going into research. He was and is involved in several funded research projects on national and European level and acts as consortium leader in the nationally funded project TRUCONF, which made the presented work possible.

Richard Schumi is a Ph.D. student at Graz University of Technology. His research interests are in model-based testing, property-based testing, and statistical model checking. He works for his Ph.D. in a local project which is concerned with testing non-functional properties of systems of systems. Richard holds a master and bachelor degree in computer science from Graz University of Technology. He is also a qualified engineer in network and information technology.

Affiliations

Bernhard K. Aichernig¹ · Priska Bauerstätter² · Elisabeth Jöbstl³ · Severin Kann³ · Robert Korošec³ · Willibald Krenn² · Cristinel Mateis² · Rupert Schlick² · Richard Schumi¹

Bernhard K. Aichernig
aichernig@ist.tugraz.at

Priska Bauerstätter
Priska.Bauerstaetter@ait.ac.at

Elisabeth Jöbstl
Elisabeth.Joebstl@avl.com

Severin Kann
Severin.Kann@avl.com

Robert Korošec
Robert.Korosec@avl.com

Willibald Krenn
Willibald.Krenn@ait.ac.at

Cristinel Mateis
Cristinel.Mateis@ait.ac.at

Rupert Schlick
Rupert.Schlick@ait.ac.at

¹ Institute of Software Technology, Graz University of Technology, Graz, Austria

² Austrian Institute of Technology, Vienna, Austria

³ AVL List GmbH, Graz, Austria