



HAL
open science

Checking Response-Time Properties of Web-Service Applications Under Stochastic User Profiles

Richard Schumi, Priska Lang, Bernhard K. Aichernig, Willibald Krenn,
Rupert Schlick

► **To cite this version:**

Richard Schumi, Priska Lang, Bernhard K. Aichernig, Willibald Krenn, Rupert Schlick. Checking Response-Time Properties of Web-Service Applications Under Stochastic User Profiles. 29th IFIP International Conference on Testing Software and Systems (ICTSS), Oct 2017, St. Petersburg, Russia. pp.293-310, 10.1007/978-3-319-67549-7_18 . hal-01678972

HAL Id: hal-01678972

<https://inria.hal.science/hal-01678972>

Submitted on 9 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Checking Response-Time Properties of Web-Service Applications Under Stochastic User Profiles

Richard Schumi¹, Priska Lang², Bernhard K. Aichernig¹, Willibald Krenn²,
and Rupert Schlick²

¹ Institute of Software Technology, Graz University of Technology, Austria
{aichernig, rschumi}@ist.tugraz.at

² Austrian Institute of Technology, Vienna, Austria
{Priska.Lang, Willibald.Krenn, Rupert.Schlick}@ait.ac.at

Abstract. Performance evaluation of critical software is important but also computationally expensive. It usually involves sophisticated load-testing tools and demands a large amount of computing resources. Analysing different user populations requires even more effort, becoming infeasible in most realistic cases. Therefore, we propose a model-based approach. We apply model-based test-case generation to generate log-data and learn the associated distributions of response times. These distributions are added to the behavioural models on which we perform statistical model checking (SMC) in order to assess the probabilities of the required response times. Then, we apply classical hypothesis testing to evaluate if an implementation of the behavioural model conforms to these timing requirements. This is the first model-based approach for performance evaluation combining automated test-case generation, cost learning and SMC for real applications. We realised this method with a property-based testing tool, extended with SMC functionality, and evaluate it on an industrial web-service application.

Keywords: statistical model checking, property-based testing, model-based testing, FsCheck, user profiles, response time, cost learning.

1 Introduction

Statistical model checking (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 analysing simulations of this model. Depending on the SMC algorithm, either a fixed number of samples or a stopping criterion is needed. Property-based testing (PBT) is a random testing technique that tries to falsify a given property, which describes the expected behaviour 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 behaviour is observed.

In previous work [2,3], we have demonstrated how SMC can be integrated into a PBT tool in order to evaluate properties of stochastic models as well as

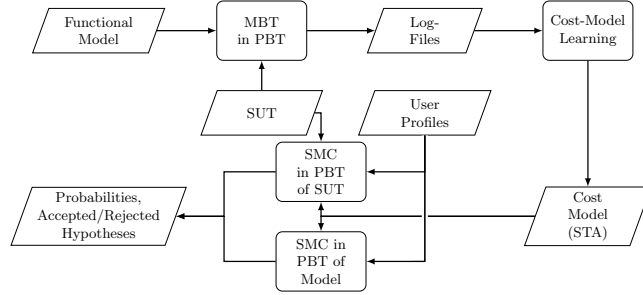


Fig. 1: Overview of the steps for cost-model learning and response-time checking.

stochastic implementations. Based on this previous work, we present a simulation method for stochastic user profiles of a web-service application in order to answer questions about the expected response time of a system-under-test (SUT). Figure 1 illustrates this process.

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 [1], here the focus is on timing.

In the next step, 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 parametrised by the number of active users. 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) [6]. The name *cost model* shall emphasize that our method may be generalized to other type of cost indicators, e.g., energy consumption.

Next, we combine these models with user profiles, containing probabilities for transitions and waiting times, in order to simulate realistic user behaviour and the expected response time. With this simulation we can 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?”.

Additionally, we can check such properties directly on the SUT, e.g., to verify the simulation on the model. It is also possible to skip the model simulation and 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 realistic 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., Monte Carlo simulation. Our aim is to run the SUT only with a limited number of samples in order to check, if the property results of the model are satisfied by the SUT. Therefore, we test the SUT with the sequential probability ratio test [34], a form of hypothesis testing, as this allows us to stop testing as soon as we have sufficient evidence.

Related Work. A number of related approaches in the area of PBT are concerned with testing concurrent software. For example, Claessen et al. [13] 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. [28]. 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 was demonstrated by Hughes et al. [20]. They showed how PBT can be applied to test distributed file-synchronisation services, like Dropbox. The closest related work we found in the PBT community was from Arts [5]. It shows a load-testing approach with QuickCheck that can run user scenarios on an SUT in order to determine the maximum supported number of users. In contrast to our approach, Arts does not consider stochastic user profiles and the user scenarios are only tested on an SUT, but not simulated at model-level.

Related work can also be found in the area of load testing. For example, Draheim et al. [14] demonstrated a load-testing approach that simulates realistic user behaviour with stochastic models. Moreover, a number of related tools, like Neoload perform load testing with user populations [31]. In contrast to our work, load testing is mostly performed directly on the SUT. With our approach, we want to simulate user populations on the model-level as well. There are also many approaches that focus only on a simulation on the model-level [7,9,11,26], but with our method we can also directly test an SUT within the same tool.

The most related tool is UPPAAL SMC [10]. 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 behaviour models using linear regression. Grinchtein learns time-deterministic event-recording automata via active automata learning [16]. Verwer et al. passively learn probabilistic real-time automata [33]. In contrast, we learn cost distributions and add them to existing automata models for SMC.

Contribution. We present a cost-model learning approach that works with log-files of tests of a PBT tool and derives cost distributions for varying numbers of users. Building upon our previous work, where we integrated SMC into a PBT tool [3], we show how the learned cost models can be applied to simulate the response time of user profiles. With this simulation we can evaluate response-time properties with SMC based on the model. Moreover, we can also check such properties on the real system by applying hypothesis testing and by measuring the real response times instead of the simulated ones. Another contribution is the evaluation of our method by applying it to an industrial web-service application.

Structure. First, Sect. 2 introduces the background of SMC and PBT based on our previous work [3] and it explains cost-model learning. Next, in Sect. 3 we present an example and demonstrate our method. In Sect. 4, we give more details about the process and implementations. Section 5 presents an evaluation with an industrial web-service application. Finally, we conclude in Sect. 6.

2 Background

2.1 Statistical Model Checking (SMC)

SMC is a verification method that evaluates certain properties of a stochastic model. These properties are usually defined with (temporal) logics, and they can describe quantitative and qualitative questions. For example, questions, like *what is the probability that the model satisfies a property* or *is the probability that the model satisfies a property above or below a certain threshold?* In order to answer such questions, a statistical model checker produces samples, i.e. random walks on the stochastic model and checks whether the property holds for these samples. Various SMC algorithms are applied in order to compute the total number of samples needed to find an answer for a specific question or to compute a stopping criterion. This criterion determines when we can stop sampling because we have found an answer with a required certainty. In this work, we focus on the following algorithms, which are commonly used in the SMC literature [24,25].

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 [18] 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:

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

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 , a simple Monte Carlo simulation is performed [25].

Sequential Probability Ratio Test (SPRT). This sequential method [34] 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 :

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

We continue sampling as long as $\log \frac{\beta}{1-\alpha} < \log A_m < \log \frac{1-\beta}{\alpha}$. H_1 is accepted when $\log A_m \geq \log \frac{1-\beta}{\alpha}$, and H_0 when $\log A_m \leq \log \frac{\beta}{1-\alpha}$ [15].

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

2.2 Property-Based Testing (PBT)

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

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

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) [22] can serve as source for state-machine properties. An EFSM is a 6-tuple (S, s_0, V, I, O, T) . S is a finite set of states, $s_0 \in S$ is the initial state, V is a finite set of variables, I is a finite set of inputs, O is a finite set of outputs, T is a finite set of transitions. A transition $t \in T$ 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, s_t is the target state [22]. In order to derive a state-machine property from an EFSM, we have to write a specification comprising the initial state, commands and a generator for the next transition given the current state of the model. Commands encapsulate (1) preconditions that define the permitted transition sequences, (2) postconditions that specify the expected behaviour and (3) execution semantics of transitions for the model and the SUT. A state-machine property states that for all permitted transition sequences, the postcondition must hold after the execution of each transition, respectively command [19,29]. Formally we define such a property as follows:

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

We have two functions to execute a command on the model and on the SUT: $cmd.runModel$ and $cmd.runActual$. The precondition $cmd.pre$ defines the valid

inputs for a command. The postcondition *cmd.post* compares the outputs and states of the model and the SUT after the execution of a command.

PBT is a powerful testing technique that allows a flexible definition of generators and properties via inheritance or composition. The first implementation of PBT was QuickCheck for Haskell [12]. Numerous reimplementations followed for other programming languages, like Hypothesis³ for Python or ScalaCheck [27]. We demonstrate our approach with FsCheck [1]. FsCheck is a .NET port of QuickCheck with influences of 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.

2.3 Stochastic Timed Automata

Timed automata (TA) were originally introduced by Alur and Dill [4]. Several extensions of TA have been proposed, including stochastically enhanced TA [8] and continuous probabilistic TA [23]. We follow the definition of Stochastic Timed Automata (STA) by Ballarini et al. [6]: An STA can be expressed as a tuple $(L, l_0, A, C, I, E, F, W)$, where the first part is a normal TA (L, l_0, A, C, I, E) and additionally it contains probability density functions (PDFs) $F = (f_l)_{l \in L}$ for the sojourn time and natural weights $W = (w_e)_{e \in E}$ for the transitions. L is a finite set of locations, $l_0 \in L$ is the initial location, A is a finite set of actions, C is a finite set of clocks with real-valued valuations $u(c) \in \mathbb{R}_{>0}$, $I : L \mapsto \mathcal{B}(C)$ is a finite set of invariants for the locations and $E \subseteq L \times A \times \mathcal{B}(C) \times 2^C \times L$ is a finite set of transitions between locations, with an action, a guard and a set of clock resets. The transition relation can be described as follows. For a location l and a clock valuation u the PDF f_l is used to choose the sojourn time d , which changes the state to $(l, u + d)$, where $u + d$ means that the clock valuation is changed $(u + d)(c) = u(c) + d$ for all $c \in C$. After this change, an edge e is selected out of the set of enabled edges $E(l, u + d)$ with the probability $w_e / \sum_{h \in E(l, u + d)} w_h$. Then, a transition to the target location l' of e and $u' = u + d$ is performed. For our models the underlying stochastic process is a semi-Markov process as the clocks are reset at every transition, but we do not assume exponential waiting times and therefore the process is not a standard continuous-time Markov chain.

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 [2,3], which were explained in Sect. 2.2. These PBT-properties can be evaluated on stochastic models, like in classical SMC, as well as on stochastic implementations. For the integration we introduced our own new SMC properties, which take a PBT property, configurations for the PBT execution, and parameters for the specific SMC algorithm as input. Then, our properties perform an SMC algorithm by utilizing the PBT tool as simulation environment and they return either a quantitative or qualitative result,

³ <https://pypi.python.org/pypi/hypothesis>

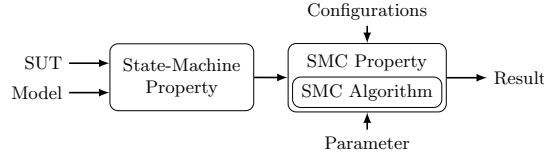


Fig. 2: Data flow diagram of an SMC property.

depending on the algorithm. Figure 2 shows how we can evaluate a state-machine property within an SMC property. Such a state-machine property can, e.g., be applied for a statistical conformance analysis by comparing an ideal model to a stochastic faulty implementation or it can also simulate a stochastic model. We evaluated our SMC properties by repeating case studies from the SMC literature and we were able to reproduce the results.

2.5 Cost-Model Learning

We aim at learning response times or other costs from log-files in order to associate them to behavioural models. This problem can be seen as a classical regression problem. (Note that other types of costs or systems can require different learning methods, like Splines or tree-based models [17,35].) The simplest regression method is the linear least squares regression, which minimizes the difference between the observed and estimated values (called residuals). An advantage of this method is that it may also help in detecting confounding variables, e.g., when the full model does not predict well. For removing these noisy or highly correlated variables, different feature selection algorithms are available [32].

Multiple Regression. The general linear regression model is known as

$$y = X\beta + \epsilon$$

where y is the dependent variable (regressand), X is the design matrix of the independent variables (regressors), β contains the partial derivatives and ϵ is the error term. In more detail, in case of i regressors the cost function for the n^{th} observation is

$$y_n = \beta_0 + x_{n1}\beta_1 + \dots + x_{ni}\beta_i + \epsilon_n$$

with β_0 as the constant term.

Discrete values are handled via categorical variables that can take on one of a limited number of possible values, called levels. In case of categorical independent variables, to transfer the factors into a linear regression model different coding techniques are available. (If they are not independent, interaction terms can be added [21].) The simplest is dummy coding, where each level of a factor has its own binary dummy variable (indicator variable), set to 1 if the observation has factor level i , 0 otherwise. By definition, these variables are linearly dependent, because the sum of all columns related to the same factor leads to a column of ones, which is the constant intercept term. Therefore, to avoid singularity problems, for each factor it is necessary to have one dummy variable less than

the number of factor levels. The factor level that has no dummy variable is the so called reference group of the model. It has zeros in all dummy variables. In case of numerical and categorical regressors, we have a combination that yields $y = X\beta + Z\gamma + \epsilon$, where X is the design matrix for the categorical variables and Z contains the measured values (covariables). γ contains analogous to β the partial derivatives. For more details, see [30].

3 Method

In this section, we show how we derive cost models from logs and how we can apply these models to simulate stochastic user profiles. This approach is demonstrated by an example of an industrial incident manager [1].

This SUT is a web-based tool that supports tasks, like creating, editing or closing incident objects, which are elements of the application domain, e.g., bug reports. 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. 3 on the left represents the tasks of an incident object. To keep it simple, this state machine only represents the tasks of a currently opened incident 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 [1]. For this paper, we assume existing functional models, although they are created in the same way as before. 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 middle of Fig. 3. 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). While the properties are tested on the SUT, 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 behaviour of multiple active users. An example log from a non-productive test system with low computing resources (virtual machine) is represented in Table 1. We record response times of tasks, subtasks, attributes, states (From, To) and simultaneous requests ($\#ActiveUsers$). For this initial logging phase the transitions are chosen with uniform distribution. For learning the cost models, we first did some descriptive statistics and feature selection by

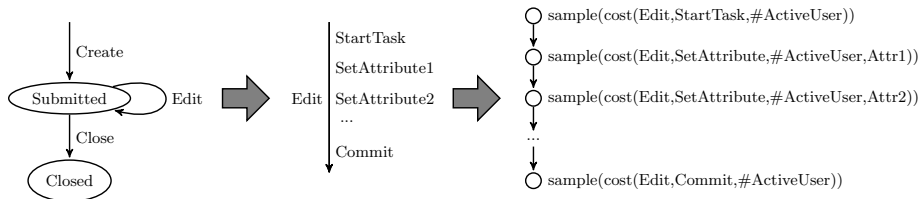


Fig. 3: Cost model of the incident manger.

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

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

applying common wrapper models to the logs, e.g., stepwise regression. Selecting the most important variables yields the linear multiple regression (LMR) model:

$$ResponseTime \sim \#ActiveUsers + Task + Subtask + Attribute$$

For categorical variables (tasks, subtasks and attributes), the dummy coding, as explained in Sect. 2.5, was applied. Listing 1.1 shows the results of the LMR. For this system, the log-file contains 293.361 observations (subtasks). The calculations are done in R version 3.3.2 with the *lm* function from the stats package.⁴ In the left column are the intercept and the regressor variables. The second column shows the estimates of means with empirical standard errors in the third. The fourth column contains the *t* values that are 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. They are marked with * if $0.01 < p \leq 0.05$ and *** if $p \leq 0.001$. In our LMR model, nearly all variables are significant and are, therefore, used to obtain different probability distributions for costs. These costs can be expressed as functions that take a task, subtask, the number of active users (i.e. a natural number without zero $\mathbb{N}_{>0}$) and an attribute as input and return empirical parameters for probability distributions. For our observed response times, we selected the normal distribution and since the real parameters are unknown, the cost function gives us the parameter μ for the mean (estimate of the LMR output) and σ for the std. deviation (std. error of the LMR output). Both of these parameters are positive real numbers $\mathbb{R}_{>0}$.

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

$$sample : \mathbb{R}_{>0} \times \mathbb{R}_{>0} \rightarrow \mathbb{R}_{>0}$$

We use these parameters for a sample function returning a response-time value, which is chosen according to this normal distribution. The right-hand side of Fig. 3 shows the application of these functions for a task. For each subtask, we introduced a state with the *sampled* sojourn time.

⁴ <https://www.r-project.org/>

	Estimate	Std. Error	t value	Pr(> t)
1 (Intercept)	405.4160	52.9412	7.658	1.90e-14 ***
2 X.ActiveUsers	33.7867	0.3094	109.187	< 2e-16 ***
3 Task_IncidentCloseTask	44.7672	52.9505	0.845	0.3979
4 Task_IncidentCreateTask	365.8872	52.9359	6.912	4.79e-12 ***
5 Task_IncidentEditTask	135.8733	52.9421	2.566	0.0103 *
6 Task_Select	-220.7655	52.9422	-4.170	3.05e-05 ***
7 Action_SetAttribute	-133.2684	2.3094	-57.706	< 2e-16 ***
8 Action_StartTask	-341.9593	1.4774	-231.460	< 2e-16 ***
9 Attribute_Assignee	-486.6695	2.8706	-169.539	< 2e-16 ***

Listing 1.1: Excerpt of the linear multiple regression output.

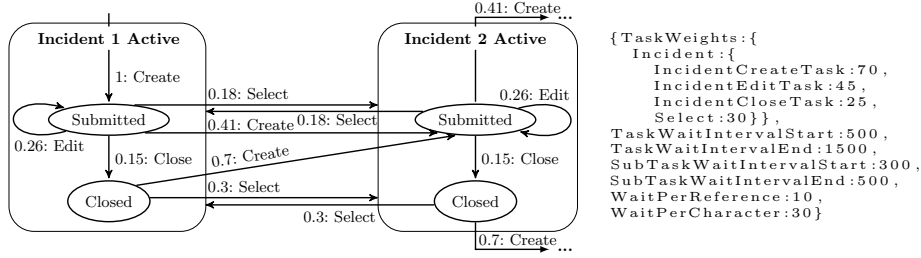


Fig. 4: User profile of the incident manager.

In addition to the cost models, also user profiles are needed for the simulation. 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 time, 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 left-hand side of Fig. 4. Note, we also included the probability for *select* transitions, which allow a switch between active incident objects. On the right-hand side, 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 Sect. 2.4. The combined model has the semantics of a stochastic timed automaton, as explained in Sect. 2.3. The weights of the tasks can be expressed with the transition weights W . The probability density functions F for the sojourn time can be defined with parameters μ and σ of the normal distribution or with intervals for the uniform distribution, which we used for the waiting times of user profiles. Note, for these waiting times, we also introduce states in a similar way as for the subtasks, as illustrated in Fig. 3.

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 a *Commit* subtask is under a threshold of one second for each user of a population of 10 users with parameters $\epsilon = 0.05$ and $\delta = 0.01$, requires 1060 samples and returns a probability of 0.593, when a test-case length of three tasks is considered. Fortunately, 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 0.493 as null hypothesis, which is 0.1 smaller, because we want to be able to reject the hypothesis that the SUT has a smaller probability. By running SPRT (with 0.01 as type I and II error parameters) for each user of the population, we can check these hypotheses. The alternative hypothesis was accepted for all users and on average 76.8 samples were needed for the decision.

Algorithm 1 Pseudo code of a *CostAttribute* class.

Local Variables and Inputs: Task t , Subtask st , Attribute a , function $cost : (\dots) \rightarrow (\mu, \sigma)$
Global Variable: $ActiveUserNum \in \mathbb{N}$ ▷ number of users that have an open request

```

1: function GENERATOR
2:    $ActiveUserNum \leftarrow ActiveUserNum + 1$  ▷ should be locked (Mutex)
3:    $delay \leftarrow sample(cost(t, st, ActiveUserNum, a))$  ▷ sample normal distribution
4:    $sleep(delay)$  ▷ thread should sleep
5:    $ActiveUserNum \leftarrow ActiveUserNum - 1$  ▷ should be locked (Mutex)
6:   return  $Gen.Constant(delay)$ 

```

4 Model-Simulation Architecture and Implementations

Here, we detail 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 [1], which supports automatic form-data generation and EFSMs. Based on this work, we implemented the following extensions in order to support our method. The first extension is a parser that reads the cost distributions and integrates them into the model. In the previous implementation, we had command instances, which represent the tasks and attributes which include generators for different data types (for the generation of form data). Now, we introduce new *CostAttributes* for costs or response times, which can be applied in the same way as normal form-data attributes. The generators of attributes are called during the test-case generation and the generated values can be evaluated within the commands. This helps to check response-time properties.

Algorithm 1 represents the implementation of these attributes. The inputs are a task, a subtask, an attribute and a cost function, which returns parameters for the normal distribution. Additionally, there is a global variable *ActiveUserNum*, which is shared by all users. The main function of a *CostAttribute* is its generator, which works as follows. First, the number of active users is increased to simulate a request. (The access to *ActiveUserNum* should be locked to avoid race conditions.) Next, 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. The next step is a sleep for the time that was sampled. Then, the number of users is decreased. Finally, the generated delay is returned within a constant generator so that it can be checked outside the generator. A constant generator is applied, because the default generators do not support normal distributions, but the *Attribute* has to return an object of type *Gen* for this method. 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 behaviour, 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.

For the user profiles there is a parser as well and the user behaviour is also included in the combined model. The waiting times of a user can be integrated in

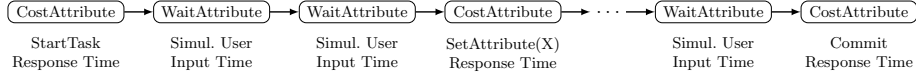


Fig. 5: Attribute sequence of a task.

a similar way as the costs by introducing *WaitAttributes*. Their implementation details are omitted, as they work in the same way as *CostAttributes* 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 attributes, we are able to implement the sequence of subtasks of tasks as represented in Fig. 5. *WaitAttributes* represent the time that a user needs for the input and *CostAttributes* simulate the response time. Note, the simulation of the model can be done 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 attributes. Algorithm 2 outlines the process of the test-case generation. The algorithm requires a state-machine specification *spec*, which includes a generator for the next state and the initial state of the model. First, 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 3) and executed on the model *cmd.runModel* in order to retrieve a new model, which incorporates the applied state change. This new model is needed in the next iteration for the *Next* function, which works 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 build (Line 7). 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.

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

Algorithm 2 Pseudo code of the test-case generation.

```

Input: spec: state-machine specification of a PBT tool, size: parameter for test-case length
1: for  $i \in \{1, \dots, size\}$  do
2:    $gen \leftarrow spec.Next(model)$  ▷ Next returns a command generator
3:    $cmd \leftarrow gen.Sample()$  ▷ command is generated
4:    $model \leftarrow cmd.runModel(model)$  ▷ command is executed
5: function SPEC.NEXT(model)
6:    $set \leftarrow model.getEnabledTasksWithWeights()$  ▷ set of (weight, Gen[Task])
7:   return  $Gen.Frequency(set).selectMany(task \rightarrow$ 
8:      $task.Attributes.Generator().selectMany(data \rightarrow$  ▷ generate attribute data
9:      $CmdGenerator(task, data))$  ▷ generator for a command

```

Within this function, a generator is called that generates the attribute data of the task. The *selectMany* function is applied again on this generator and within this function a command generator is created for the given task and data.

5 Evaluation

We evaluated our method by applying it to a web-service application from the automotive domain, which was provided by our industrial partner AVL.⁵ We focus on the response times and the number of samples needed, but omit the run-times of the simulation and testing process. The application is called Testfactory Management Suite (TFMS) version 1.7 and it enables various management activities of test fields, like test definition, planning, preparation, execution and data management/analysis for testing engines. Note that there already is a new version of TFMS with better performance, but it was not available for this work.

For our evaluation we focused on one module of the application, the Test Order Manager (TOM). This module enables the configuration and execution of test orders, which are basically a composition of steps that are necessary for a test sequence at a test field [1]. Figure 6 shows the tasks of an example test order. Each task represents the invocation of a page, entering data for form fields and saving the page. The TOM module contains further sub-models for the creation of test orders, but they are similar to this model, and are therefore omitted.

We applied our method in order to compute the probability that the response time of a *Commit* subtask is under a threshold of 1.65 s. Hence, we check the probability that all the response times of these subtasks within a sequence of tasks with fixed lengths are under this threshold. Note that we focus on this subtask as it is the most computationally expensive one. For this evaluation a user profile was created in cooperation with domain experts from AVL. This profile was similar to the one shown in Sect. 3, and is therefore omitted. The LMR model was similar as well and also omitted, the only difference was that due to the increased complexity of this module, we had more log-data (929.584 observations). 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

⁵ <https://www.avl.com>

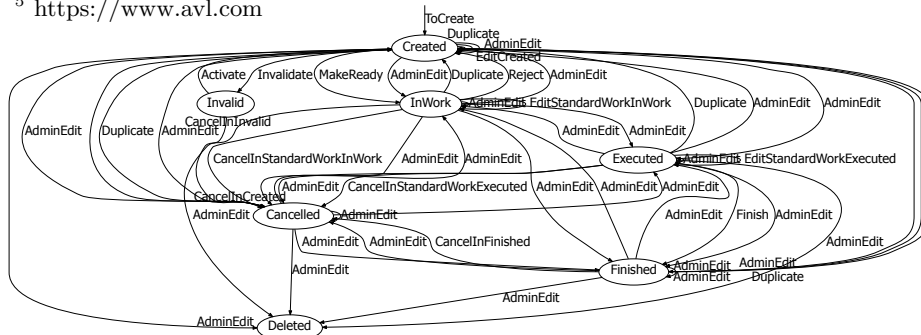


Fig. 6: Example test order model.

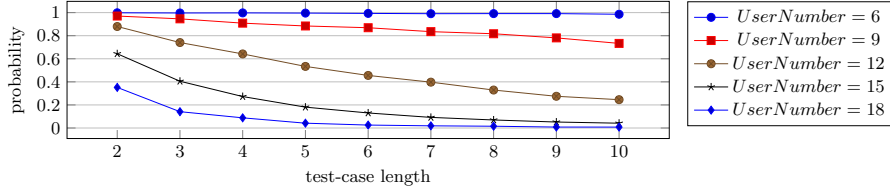


Fig. 7: Simul. result: how likely is it that the response time is under a threshold?

$\epsilon = 0.05$ and $\delta = 0.01$. This requires 1060 samples per data point. Figure 7 shows the results. Note that test cases of length one have always probability one as the initial task for sub-model selection has no requests and, hence, zero response time. As expected, a decrease in the probability of the property can be observed, when the test-case length or the population size increases. The advantage of the simulation on the model-level is that it runs much faster than on the SUT. With a virtual time of 1/100 of the actual time, we can perform simulations that would take weeks 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 Sect. 3 by applying the SPRT with the same parameters. Table 2 shows the results. Due to the high computation effort, we did not check all data points of Fig. 7. Our focus was on test cases with length three as this was a common length of user scenarios. The table shows the hypotheses and evaluation results for different numbers of users. 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. Compared to the execution on the model, a smaller number of samples is needed, as the SPRT stops, when it has sufficient evidence. The result column shows that the alternative hypothesis was always accepted. This means that the probabilities of response-time properties on the SUT were at least as good as on the model. The smaller required number of samples of the SPRT (max. 66) compared to Monte Carlo simulation (1060 samples) allowed us to analyse the SUT within a feasible time.

6 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 such probabilities

Table 2: Results of the evaluation on the SUT by applying the SPRT.

#Users	H_0	H_1	Result	#Samples Needed
6	0.897	0.997	H_1	66.6
9	0.846	0.946	H_1	42.0
12	0.641	0.741	H_1	60.8
15	0.305	0.405	H_1	17.3
18	0.042	0.142	H_1	7.6

can be tested directly 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. We showed that we can 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 [17,35] 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. Additionally, we want to apply our method to compare the performance of different versions of the SUT, i.e. non-functional regression testing.

Acknowledgments. This work was funded by the Austrian Research Promotion Agency (FFG), project TRUCONF, No. 845582. We are grateful to Martin Tappler, the team at AVL, especially Elisabeth Jöbstl, and the anonymous reviewers for their valuable inputs.

References

1. Aichernig, B.K., Schumi, R.: Property-based testing with FsCheck by deriving properties from business rule models. In: ICSTW. pp. 219–228. IEEE (2016)
2. Aichernig, B.K., Schumi, R.: Towards integrating statistical model checking into property-based testing. In: MEMOCODE. pp. 71–76. IEEE (2016)
3. Aichernig, B.K., Schumi, R.: Statistical model checking meets property-based testing. In: ICST. pp. 390–400. IEEE (2017)
4. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* 126(2), 183–235 (1994)
5. Arts, T.: On shrinking randomly generated load tests. In: Erlang’14. pp. 25–31. ACM (2014)
6. Ballarini, P., Bertrand, N., Horváth, A., Paolieri, M., Vicario, E.: Transient analysis of networks of stochastic timed automata using stochastic state classes. In: QEST. LNCS, vol. 8054, pp. 355–371. Springer (2013)
7. Becker, S., Koziolok, H., Reussner, R.H.: The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82(1), 3–22 (2009)
8. Blair, L., Jones, T., Blair, G.S.: Stochastically enhanced timed automata. In: FMOODS. vol. 177, pp. 327–347. Kluwer (2000)
9. Book, M., Gruhn, V., Hülder, M., Köhler, A., Kriegel, A.: Cost and response time simulation for web-based applications on mobile channels. In: QSIC. pp. 83–90. IEEE (2005)

10. Bulychev, P.E., David, A., Larsen, K.G., Mikucionis, M., Poulsen, D.B., Legay, A., Wang, Z.: UPPAAL-SMC: statistical model checking for priced timed automata. In: QAPL. EPTCS, vol. 85, pp. 1–16 (2012)
11. Chen, X., Mohapatra, P., Chen, H.: An admission control scheme for predictable server response time for web accesses. In: WWW. pp. 545–554. ACM (2001)
12. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP. pp. 268–279. ACM (2000)
13. Claessen, K., Palka, M.H., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.T.: Finding race conditions in Erlang with QuickCheck and PULSE. In: ICFP. pp. 149–160. ACM (2009)
14. Draheim, D., Grundy, J.C., Hosking, J.G., Lutteroth, C., Weber, G.: Realistic load testing of web applications. In: CSMR. pp. 57–70. IEEE (2006)
15. Govindarajulu, Z.: Sequential statistics. World Scientific (2004)
16. Grinchtin, O.: Learning of Timed Systems. Ph.D. thesis, Uppsala Univ. (2008)
17. Hastie, T., Tibshirani, R., Friedman, J.H.: The elements of statistical learning: data mining, inference, and prediction, 2nd Edition. Springer (2009)
18. Hoeffding, W.: Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association* 58(301), 13–30 (1963)
19. Hughes, J.: QuickCheck testing for fun and profit. In: PADL. LNCS, vol. 4354, pp. 1–32. Springer (2007)
20. Hughes, J., Pierce, B.C., Arts, T., Norell, U.: Mysteries of Dropbox: Property-based testing of a distributed synchronization service. In: ICST. pp. 135–145. IEEE (2016)
21. Jaccard, J., Turrisi, R.: Interaction Effects in Multiple Regression. SAGE (2003)
22. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine. In: ICST. pp. 230–239. IEEE (2009)
23. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Verifying quantitative properties of continuous probabilistic timed automata. In: CONCUR. LNCS, vol. 1877, pp. 123–137. Springer (2000)
24. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: RV. LNCS, vol. 6418, pp. 122–135. Springer (2010)
25. Legay, A., Sedwards, S.: On statistical model checking with PLASMA. In: TASE. pp. 139–145. IEEE (2014)
26. Lu, Y., Nolte, T., Bate, I., Cucu-Grosjean, L.: A statistical response-time analysis of real-time embedded systems. In: RTSS. pp. 351–362. IEEE (2012)
27. Nilsson, R.: ScalaCheck: The Definitive Guide. IT Pro, Artima Incorporated (2014)
28. Norell, U., Svensson, H., Arts, T.: Testing blocking operations with QuickCheck’s component library. In: Erlang’13. pp. 87–92. ACM (2013)
29. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Erlang’11. pp. 39–50. ACM (2011)
30. Rencher, A., Christensen, W.: Methods of Multivariate Analysis. Wiley (2012)
31. Rina, Tyagi, S.: A comparative study of performance testing tools. *Intern. Journal of Adv. Research in Comp. Sci. and SW Engineering, IJARCSSE* 3(5) (2013)
32. Tang, J., Alelyani, S., Liu, H.: Feature selection for classification: A review. In: *Data Classification: Algorithms and Applications*, pp. 37–64. CRC Press (2014)
33. Verwer, S., de Weerd, M., Witteveen, C.: A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In: ICGI. LNCS, vol. 6339, pp. 203–216. Springer (2010)
34. Wald, A.: Sequential analysis. Courier Corporation (1973)
35. West, B., Welch, K., Galecki, A.: Linear Mixed Models. CRC Press (2006)