Statistical Model Checking Meets Property-Based Testing

Bernhard K. Aichernig and Richard Schumi
Institute of Software Technology, Graz University of Technology, Austria
{aichernig, rschumi}@ist.tugraz.at

***Abstract*—In recent years, statistical model checking (SMC) has become increasingly popular, because it scales well to larger stochastic models and is relatively simple to implement. SMC solves the model checking problem by simulating the model for finitely many executions and uses hypothesis testing to infer if the samples provide statistical evidence for or against a property. Being based on simulation and statistics, SMC avoids the state-space explosion problem well-known from other model checking algorithms. In this paper we show how SMC can be easily integrated into a property-based testing framework, like FsCheck for C#. As a result we obtain a very flexible testing and simulation environment, where a programmer can define models and properties in a familiar programming language. The advantages: no external modelling language is needed and both stochastic models and implementations can be checked. In addition, we have access to the powerful test-data generators of a property-based testing tool. We demonstrate the feasibility of our approach by repeating three experiments from the SMC literature.**

I. Introduction

Statistical model checking (SMC) is an efficient method to test certain properties of stochastic models. These properties are usually defined in temporal logics, like linear temporal logic (LTL). SMC can be used to answer both qualitative and quantitative questions about these properties by analysing executions of a stochastic model to measure how often the properties are satisfied. A number of tools exist that perform SMC for different kinds of models. For example, UPPAAL-SMC checks priced timed automata [7] or PLASMA-lab supports a number of different modelling languages, like the Reactive Module Language or Matlab Simulink [6], [18]. However, the existing SMC tools are not as flexible as some situations or users may require. They are limited by the modelling language and the properties are limited by the used logics. Therefore, we propose a new SMC approach that builds on property-based testing (PBT).

PBT is a testing technique that tries to falsify a given property by generating random input data and checking the expected behaviour [8]. PBT is very flexible in the sense that properties can range from simple algebraic equations to complex state machine models.

For our SMC approach we introduce new SMC properties that take classical PBT properties as input and check them with an SMC algorithm. Our SMC properties can be applied to both algebraic and state machine properties that can generate command sequences and compare a system-under-test (SUT) to a model after each command execution.

With our approach we can do both, SMC by simulating stochastic models and conformance testing of an SUT with stochastic failures. For classical SMC we define our stochastic models with PBT state machine properties, but we only exploit the model part of these state machine properties, the part for the SUT is neglected.

Additionally, conformance testing can be done by utilising the default state machine properties and comparing faulty systems with a correct model repeatedly by executing these state machine properties within our SMC properties.

PBT provides the tester with generators that enable the generation of test data with certain probability distributions. For example, it is possible to choose between multiple transitions by assigning weights to each of them, or like in UPPAAL-SMC, one may apply a distribution to define the time one may wait in certain states. The default behaviour for checking PBT state machines is to make random walks through the model by generating (input) command sequences. The generation of these sequences can also be controlled with generators. For SMC we need a discrete-event simulation, which can be realised via the random walks in PBT. Hence, PBT has a number of features that are helpful to implement a statistical model checker. For the demonstration of our approach we use the PBT tool FsCheck [1] and C# as a programming language.

Related Work and Contribution: SMC was applied in several case studies and a number of tools exist that implement a variety of SMC algorithms. Simple algorithms like Monte Carlo simulation are already supported by existing PBT tools. For example, with ScalaCheck [28] the required number of samples can be specified and it can report the number of failing samples. This enables a simple Monte Carlo simulation. In contrast, the focus of our approach is on hypothesis testing, but we also included Monte Carlo methods for demonstration purposes, because they are common SMC algorithms.

A tool that provides similar functionality is UPPAAL-SMC [7]. This tool supports SMC for priced timed automata, which can have weights on transitions and probability distributions for the dwell time in states. It supports hypothesis testing and probability comparison and estimation by applying Wald's sequential probability ratio test (SPRT) [34] and Monte Carlo simulation with Chernoff-Hoeffding bound [14].

The probabilistic model checker PRISM was also extended with SMC functionality [23]. Similar to UPPAAL-SMC it supports priced timed automata, but it also supports discrete- and continuous-time Markov chains, Markov decision processes and probabilistic automata. They also support the same algorithms as UPPAAL-SMC, i.e. the SPRT and Monte Carlo simulation with Chernoff-Hoeffding bound.

VESTA is another SMC tool that supports hypothesis testing of properties in probabilistic computation tree logic (PCTL) and continuous stochastic logic (CSL) [32]. For modelling, VESTA uses a language, which is related to PRISM, in order to specify discrete-time and continuous time Markov chains. Furthermore, the tool includes an interface to describe models in probabilistic rewrite theories with the algebraic specification language PMAUDE. AlTurki and Meseguer [3] presented an extension of VESTA called PVESTA. This extension includes parallel algorithms for SMC and client-server support.

Another statistical model checker called Ymer was presented by Younes [36]. It is similar to PVESTA and supports properties in PCTL and CSL and uses the SPRT. For modelling it uses an extension of the PRISM language, which allows the definition of time-homogeneous generalised semi-Markov processes.

An alternative to realise probabilistic models is probabilistic programming [12], [35], [29], which introduces probability distributions into normal programming languages. Different inference techniques, like Bayesian inference [20], are supported. The difference to our approach, or to SMC in general, is that probabilistic programming does not aim to evaluate quantitative properties, but performs probabilistic inference. Most importantly, it does not support PBT.

The most similar to our work is from Jegourel et al. [18] and Boyer et al. [6]. First, they had developed the SMC platform PLASMA, which was later replaced by the PLASMA-lab library. The library can perform SMC for multiple modelling languages. For example, it supports the PRISM language and biological languages, it has plugins for Matlab, SystemC and further plugins can be implemented for other modelling languages. This is a nice feature, because it allows the creation of a custom statistical model checker. However, in order to write a plugin for PLASMA-lab, a user has to be familiar with the architecture of the library and also with the logics for the property definition. The library uses bounded linear temporal logic (BLTL) for the definition of properties and as SMC algorithms it supports simple Monte Carlo, Monte Carlo with Chernoff-Hoeffding bound and SPRT. Furthermore, Legay et al. [26] presented an algorithm for change detection called cumulative sum (CUSUM), which was also added to the PLASMA-lab library.

Existing SMC tools often have a rather limited modelling language. In order to reduce the effort in modelling and specification an additional layer of abstraction, i.e. "syntactic sugar", can be added. For example, David et al. presented a simulation method for biological systems for UPPAAL-SMC by translating these systems to timed automata [9]. Another approach that enables a high-level specification of Systems of Systems (SoS) and SoS requirements was presented by Arnold et al. [4]. They show how a contract language can be used to define properties, which they translate to BLTL formulas for PLASMA-lab. In contrast, we do not introduce a new language for the model or property definition and hence do not need translators. With C# we utilize an existing high level programming language that is familiar to many developers in the industry. We show that the models and the properties to be checked can be easily defined in an object-oriented programming language. No new notation or (temporal) logic needs to be learned.

Another advantage is the powerful test-data generators, which are the major ingredient of PBT. These generators can be freely combined and are especially useful for applications, which require a large amount of complex input data, like information systems. Additionally, they support the generation of data with certain probability distributions, which is useful for stochastic models.

To the best of our knowledge, no existing work combines SMC with PBT, except for papers on PBT tools that report the number of passed and failed test-cases using Monte Carlo simulation.

This paper is an extension of our previous short *work-in-progress paper* [2]. In this previous work we outlined the idea of integrating SMC into a PBT tool by introducing new properties and we demonstrated how an SUT with stochastic failures can be evaluated via a conformance check with an ideal model. We showed this idea only for the SPRT and a simple Monte Carlo simulation and we applied it only to a simple example. We did not perform any evaluation.

Compared with our previous work the novel contributions are the following. We present new SMC properties for further SMC algorithms, i.e. Monte Carlo with Chernoff-Hoeffding bound and CUSUM. Another contribution is an optimized PBT approach for classical SMC. The optimisation is that we only exploit the model part of a state machine property in order to avoid the overhead of running both a model and an SUT and it also gives us the possibility to stop during the generation of a sample. Furthermore, we present an evaluation of our approach with three typical SMC examples from the literature.

Structure: First, Section II will explain the basics of SMC, PBT and FsCheck. Next, in Section III we demonstrate how SMC methods can be applied to a small example of a stochastic counter with faulty behaviour. Then, in Section IV we present details about the implementations of our approach. In Section V we evaluate our approach. Finally, we draw our conclusions in Section VI.

II. Background

A. Statistical Model Checking

SMC is a testing method that evaluates certain properties of a stochastic model. These properties are usually defined with (temporal) logics, like BLTL, and they can answer both 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 greater than or below a certain threshold?* In order to answer these kinds

of questions, a statistical model checker produces samples in the form of 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 [25], [24].

Simple Monte Carlo simulation. This is the simplest SMC algorithm. It answers quantitative questions and works as follows. First, a fixed sample number and a property are specified by the user. Then, the statistical model checker simply generates the specified number of samples and counts for how many of them the property holds. Finally, the number of samples that fulfil the property divided by the total number of samples is used to estimate the probability that the model satisfies the property [6].

Monte Carlo simulation with Chernoff-Hoeffding bound. The algorithm computes the number of simulations n needed in order to estimate the probability γ that a stochastic model satisfies a Boolean property. The procedure is based on the Chernoff-Hoeffding bound [15] that provides an upper limit for the probability that the estimation error is below a certain 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 simulations represent discrete random variables $X_1, \ldots, X_n$ with outcome $x_i = 1$ if the property holds 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 sample number n a simple Monte Carlo simulation is performed. This algorithm is implemented in PLASMA-lab [17].

Sequential Probability Ratio Test (SPRT). This sequential method [34] is a form of hypothesis testing, which can be used to 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 type II errors (α and β). In order to make the decision, we start sampling and calculate the log likelihood ratio after each observation of x_i:

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

We continue sampling as long as $\log \frac{\beta}{1-\alpha} < \log \Lambda_m < \log \frac{1-\beta}{\alpha}$. H_1 is accepted when $\log \Lambda_m \geq \log \frac{1-\beta}{\alpha}$ and H_0 when $\log \Lambda_m \leq \log \frac{\beta}{1-\alpha}$ [13].

Cumulative Sum (CUSUM). CUSUM [26] is a sequential analysis technique similar to SPRT, but for detecting the change of an initial probability. Given a finite set of Bernoulli random variables $X_1, \ldots, X_n$, a probability for detecting a change $k \in]0, 1[$ and sensitivity threshold λ, we want to decide between the hypotheses $H_0 : \forall i, 1 \leq i \leq N, p_n < k$ and $H_1 : \exists i, 1 \leq i \leq N$ and a change appears at time $t_i : \forall n, 0 \leq n \leq N$, such that $t_n < t_i \implies p_n < k$ and $t_n \geq t_i \implies p_n \geq k$. We assume that we know the probability under normal conditions p_{init}, which can, for example, be determined with a Monte Carlo simulation. We calculate the log likelihood-ratio s_i and the cumulative sum $S_n = \sum_{i=1}^{n} s_i$.

$$s_i = \begin{cases} \log(\frac{k}{p_{init}}) & \text{if } x_i = 1 \\ \log(\frac{1-k}{1-p_{init}}) & \text{otherwise} \end{cases}$$

We stop sampling when $S_n - \min_{1 \leq i \leq n}(S_n) \geq \lambda$, which means that a change $p_n \geq k$ was detected at time t_n or when no change occurred after a specified number of samples.

B. 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 behaviour of a function-under-test that should always hold. For example, a simple algebraic property might state that the length of a concatenated list is equal to the sum of lengths of its sub-lists:

$$\forall\, l_1, l_2 \in Lists[T] :$$
$$length(concatenate(l_1, l_2)) = length(l1) + length(l2)$$

With PBT we automatically generate inputs for such a property by applying its 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. A counterexample can be quite complex. Therefore, PBT shrinks it by searching for a smaller similar counterexample.

PBT also supports model-based testing. Models encoded as extended finite state machines (EFSMs) [19] 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 [19]. In order to create a state machine property out of 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 [16], [30]. Formally we can define a state machine property as follows:

$$cmd.runModel, cmd.runActual : S \times I \to S \times O$$
$$cmd.pre : I \times S \to \mathbb{B}, cmd.post : S \times O \times S \times O \to \mathbb{B}$$
$$\forall s \in S, i \in I, cmd \in Cmds :$$
$$cmd.pre(i, s) \implies cmd.post(cmd.runModel(i, s), cmd.runActual(i, s))$$

We have two functions to execute a command on the model and on the SUT $cmd.runModel$, $cmd.runActual$. 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. It scales well for a high number of tests, as it is random testing and it can generate a large number of tests in an acceptable time [33]. The first implementation of PBT was QuickCheck for Haskell [8]. Numerous reimplementations followed for other programming languages, like Hypothesis[1] for Python or ScalaCheck [28]. We demonstrate our approach with FsCheck [1].

C. FsCheck

The PBT tool FsCheck is a .NET port of QuickCheck with influences of ScalaCheck. Similar to other PBT tools it supports algebraic properties as well as state machine properties and it is equipped with generators for common data types, which can be combined or extended in order to build custom generators.

Furthermore, FsCheck has extensions for unit testing, which support a convenient definition and execution of properties like normal unit tests. We have successfully applied FsCheck for the automated testing of web services in industry [1].

FsCheck can evaluate if an SUT conforms to a model by generating command sequences and comparing the state of the SUT with the expected state of the model after the command execution. In order to perform such an evaluation, FsCheck needs a state machine specification that it converts into a state machine property. This property is able to perform the generation of command sequences and the comparison of the SUT with the model. In order to define such a state machine specification, it is necessary to implement an interface for FsCheck that includes the model and the SUT and their initial states, a generator that selects transitions for a given model state, and classes for the commands. More details about state machine specifications in FsCheck were shown in our previous work [1].

III. Example

In this section we demonstrate our approach with a simple example of a counter, which is commonly used in the PBT community in order to illustrate model-based testing with state machine properties.

[1]https://pypi.python.org/pypi/hypothesis

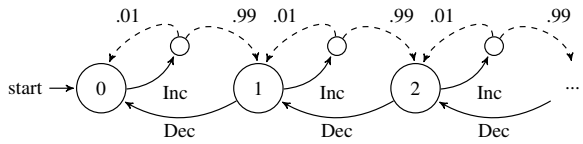

Fig. 1. Stochastic model example of a counter from the PBT community.

Figure 1 shows the state machine of our counter implementation. It can be seen that we added stochastic faulty behaviour to the increment function (Inc) of the counter. This behaviour was achieved by adding a probabilistic choice: the function can either do a normal increment (99%) or do nothing (1%). The decrement function (Dec) works as usual.

Listing 1 shows the implementation of the counter with the stochastic behaviour. Internally the counter uses an integer to store the state. We utilise a System.Random object, which is a pseudo-random number generator from the .NET framework, to implement the stochastic behaviour. In the Inc function we call $random.Next(100)$, which gives us numbers from 0 to 99. This number is used to produce value 0 with probability 0.01 and value 1 with probability 0.99.

The main property we wanted to check for this example is how likely it is that the counter with the stochastic behaviour behaves like a normal counter. In order to check such properties we implemented new properties that are based on the properties from PBT with the difference that they perform an SMC algorithm instead of normal property checks. Our new SMC properties take a normal PBT property and parameters for an SMC algorithm as input and apply the algorithm on the input property. More details about the implementation of these properties are discussed in Section IV.

The following listing shows an example property for Monte Carlo simulation:

```
Property p = new CounterMachine().ToProperty();
new MonteCarloProperty(p, config, 1000).QuickCheck();
```

It can be seen that we first define an FsCheck state machine property and after that we check it by performing a simple Monte Carlo simulation with 1000 runs. This is done by defining a MonteCarloProperty that takes the state machine property and configuration parameters as input and executing the *QuickCheck* method. The output of this method was that the property holds in 98.7% of the cases.

Another example of a property for the SPRT is shown in the following listing:

```
new SPRTProperty(p, config, 0.95, 0.9, 0.01, 0.01)
```

```
1 public class Counter{
2   private int n; System.Random random;
3   public Counter(System.Random r) { this.random = r; }
4   public void Inc(){
5     n += random.Next(100) > 0 ? 1 : 0;
6   }
7   public void Dec(){ n--; }
8   public int Get(){ return n; }
9 }
```

Listing 1. Stochastic counter implementation for FsCheck.

The four arguments of the SPRT method are: the probability for the null hypothesis, the probability for the alternative hypothesis and the type I and type II error parameters. The example demonstrates an SPRTProperty, which can check if the probability that the stochastic counter works like a normal counter is closer to 0.95 or 0.9. When we check this property for samples of length 10, we obtain the result that the null hypothesis H_0 was accepted.

Similar to the SPRT, a property for CUSUM can be defined as follows:

```
new CusumProperty(p,config,0.945, 0.85, 5, 5000)
```

This property also requires four arguments: the initial probability, the probability to detect a change, the sensitivity threshold and a maximum sample number to stop, when no change was detected. When we run this CusumProperty with samples of length 10, then, as expected, no change is detected. After the model was adapted so that the probability of a correct increment is decreased after 1000 Inc commands, we were able to observe this change after 345 samples. CUSUM is useful for testing systems with random failures where the probability of failure changes after a while. Knowing when the change occurs helps in localising the fault.

The concrete testing of properties that we want to check happens in the state machine specification of FsCheck. We propose the following optimised approach for SMC. In conventional PBT a state machine property has a part for the model and for the SUT, which are both executed and compared. Due to the overhead of running both the model and the SUT, we utilise only the model part of these properties to simulate stochastic models, the SUT part is ignored. We instrument the model with observer functions that monitor the state of the model during execution. With these observer functions we form the conditions that are checked during run-time (run-time verification). These conditions can be directly inserted in the $runModel$ function of a command, which is responsible to perform the execution of an action on the model. If we observe that the property is already fulfilled, we can terminate the sample execution with a stop command. When we observe that it fails, an exception is thrown, which also stops the generation of further commands.

Optionally, it is also possible to make a conformance test between an ideal model and an SUT with stochastic failures. We already presented this approach in our previous work [2]. In this setting we provided the default state machine property of FsCheck as input to our SMC properties. As mentioned, the default state machine property runs both a model and an SUT and checks, if the state of the SUT conforms to the model. For example, we can use the stochastic counter as SUT and a regular counter as model and test if we can find a difference for a certain number of generated command sequences. This approach is useful if an SUT has failures that occur irregularly or if a black-box system is tested that cannot be easily instrumented with additional observer functions. In the following we will apply our optimised SMC approach, because it is better suited for classical SMC.

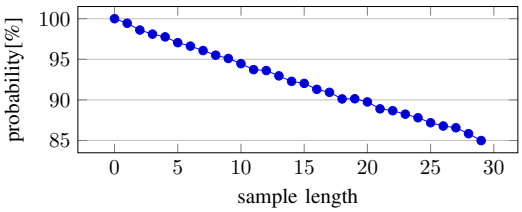

Fig. 2. Simulation results for the property: how likely is it that the stochastic counter behaves like a normal counter?

Figure 2 shows the evaluation results of our counter. We performed a Monte Carlo simulation with 100,000 samples for each data point in order to compute the probability that the stochastic counter acts like a normal counter. It can be seen that the probability decreases with increasing sample length. This behaviour met our expectations because with a larger sample length, i.e. longer random walks on the model, it is more likely that we see a faulty increment.

IV. Implementation

In this section we illustrate how we implemented our SMC approach by introducing our own new SMC properties that are based on PBT properties. Furthermore, we want to highlight the advantages like flexibility and user convenience of our proposed approach. The mathematical background of our implemented SMC algorithms were already briefly discussed in Section II-A.

We propose new properties for each SMC algorithm. These properties are based on properties from PBT with the difference that they perform an SMC algorithm instead of a normal test that only checks if a property holds or fails. We want to know the probability that the property holds, or we want to assess if the probability of the property is closer to a null hypothesis or closer to an alternative hypothesis. Our new SMC properties take a normal PBT property, a configuration object for the check of the PBT property and parameters for an SMC algorithm as constructor arguments. They provide a $QuickCheck$ function that performs the SMC algorithm by simulating the input PBT property, which is used to generate samples and also to evaluate them. When the simulation is finished the result is presented to the user. The SMC properties for the different SMC algorithms have the same structure, but they require different parameters for the algorithms and different stopping criteria for the simulation.

Listing 2 shows the MonteCarloProperty class, which can perform a simple Monte Carlo simulation. It can be seen that the constructor takes a property, a configuration object for checking the property and the total sample number for the simulation (Line 6). The $QuickCheck$ function (Line 11) performs the actual simulation. First, we initialise a counter for the number of passing samples. Then, we run a for-loop that creates samples with the specified sample number. A sample is generated by applying the $Check.One$ method, which takes the PBT property and a $config$ object as input. A $config$ object contains FsCheck configurations like Boolean flags to control the output/exception behaviour of properties and the required number of tests. The $Check.One$ method also evaluates, if the property was fulfilled. If it fails, an exception is thrown.

```
public class MonteCarloProperty{
  protected Property property;
  protected Config config;
  protected int samples;

  public MonteCarloProperty(Property p, Config c, int samples){
    this.property = p;
    this.config = c;
    this.samples = samples;
  }
  public void QuickCheck(){
    int passCnt = 0;
    for (int i = 0; i < samples; i++){
      try{
        Check.One(config, property);
        passCnt++;
      }catch{}
    }
    Console.Write("Property true for: " + passCnt + " samples\n");
    Console.Write("Property false for: "+(samples-passCnt)+" samples\n");
    Console.Write("Property holds "+((double)passCnt/samples*100)+"%\n");
  }
}
```

Listing 2. Implementation of a simple Monte Carlo simulation.

```
class ChernoffProperty : MonteCarloProperty{
  public ChernoffProperty(Property p, Config c, double epsilon, double delta) : base(p,c,0) {
    this.samples = (int)Math.Ceiling((1 / (2*Math.Pow(epsilon, 2))) * Math.Log(2 / delta));
    Console.WriteLine("sampleNumber: " + this.samples);
  }
}
```

Listing 3. Implementation of a ChernoffProperty for Monte Carlo simulation with Chernoff-Hoeffding bound derived from a MonteCarloProperty.

```
class SPRTProperty{
  Property property;
  Config config;
  double p0;
  double p1;
  double log_a;
  double log_b;

  public SPRTProperty(Property p, Config c, double p0, double p1, double alpha, double beta){
    this.property = p;
    this.config = c;
    this.p0 = p0;
    this.p1 = p1;
    this.log_a = Math.Log(beta / (1 - alpha));
    this.log_b = Math.Log((1 - beta) / alpha);
  }
  public void QuickCheck(){
    double s_i = 0;
    do{
        bool success = false;
        try{
          Check.One(config, property);
          success = true;
        } catch {}
        double ratio = success ? (p1 / p0) : ((1 - p1) / (1 - p0));
        s_i = s_i + Math.Log(ratio);
    } while (log_a < s_i && s_i < log_b);
    if (s_i >= log_b){
      Console.WriteLine("H1 accepted.");
    }
    else if (s_i <= log_a){
      Console.WriteLine("H0 accepted.");
    }
  }
}
```

Listing 4. Implementation of hypothesis testing with an SPRTProperty.

Therefore, we have put the method call inside a try-catch block (Lines 14-17) and we only count a sample as passed, if the method finishes successfully. After the desired number of samples was evaluated the results are presented to the user.

Listing 3 shows how we constructed a ChernoffProperty, which performs Monte Carlo simulation with Chernoff-Hoeffding bound, by deriving from the MonteCarloProperty. These two properties are very similar. The only difference is that for the simple Monte Carlo simulation the user directly specifies the total sample number. For the version with Chernoff-Hoeffding bound, the user has to specify the required accuracy and confidence with the parameters epsilon and delta. Then, the algorithm computes the required sample number and performs a Monte Carlo simulation. We can basically reuse the behaviour from the base class. We only implement the calculation of the sample number in the constructor (Line 3). Then, when the normal $QuickCheck$ method from the base class is called, the computed sample number of the derived class is used and the simulation is performed with the required accuracy and confidence.

An SPRTProperty which performs the SPRT method is shown in Listing 4. This property can decide if a null hypothesis specified with the parameter p0 or an alternative hypothesis given with p1 will be accepted. In contrast to the previous algorithms we do not know the sample number in the beginning. We have an indifference region, in which we have not found a decision yet and where we have to continue sampling. The thresholds for this indifference region are calculated in the constructor with the desired type I and type II error parameters alpha and beta (Lines 14-15). Inside the $QuickCheck$ method we perform the simulation. A sample is checked in the same way as in a MonteCarloProperty (Lines 20-24). For each sample we calculate the log likelihood ratio and sum it up with the previous ratios (Lines 25-26). We stop when the sum is outside the thresholds. Depending on which threshold was met, either H_0 or H_1 is accepted.

Listing 5 shows a CusumProperty that performs the CUSUM algorithm, which is similar to SPRT. As parameters this property requires an initial probability p_init, a probability k for detecting a change, a sensitivity threshold lambda and a maximum number of samples for stopping when no change was detected (Line 4). The first steps of the algorithm are the same as for the SPRT, because we also need to calculate the log likelihood ratio (Lines 12-17). The difference is that we calculate the minimum of the ratio sums and check if the difference to the current value is greater than lambda, in order to detect a change (Lines 19-22).

The architecture of our SMC properties makes it easy to check all kinds of PBT properties. Although our focus is on stochastic models and state machine properties, it is also possible to check the stochastic behaviour of other kinds of properties. For example, one might want to check properties of a stochastic function or a call to an operation with stochastic failures. Our properties can easily be implemented in other PBT tools. As already explained in Section II-B, there exist various PBT tools for different programming languages. It is not much effort to apply our approach for other tools since the structure is simple and works for other languages as well.

The definition of our stochastic models and properties in a high level programming language provides some benefits like flexibility. For example, the models can be easily extended to include observer functionality like counting certain incidents. Counters can then be evaluated within the FsCheck specification in order to decide if a sample fails. We looked at existing SMC approaches and noticed that they are quite limited in some areas, e.g., when one wants to check models with different numbers of instances or when instances should be created dynamically. In a high-level programming language it is quite easy to create a fixed number of instances via a loop or even dynamically add instances during the execution of a model. Furthermore, we noticed that often very long formulas are required for the properties within the models of existing SMC approaches because the used notations often do not support loop functionality. We will give examples and further details about these issues in Section V. It should be noted that we used a new experimental version of the FsCheck state machine specification. The advantage of this version is that it makes it possible to generate fixed length samples and that it supports stop commands, which allows us to stop during the command generation. These two features are quite important for our implementation because we have to ensure that our generated samples are long enough, but it is also important that we can stop, when we know the result of a sample. More details about this new experimental version can be found in the documentation.[2]

V. Evaluation

In this section we evaluate our SMC approach by applying it to three existing case studies from the SMC community and discuss differences to PLASMA-lab. We report performance results, because they formed part of an original PLASMA-lab case study. However, our primary focus is not performance, but the usability and flexibility of our modelling style.

A. Dining Philosophers Case Study

We applied our first case study to a probabilistic version of the *dining philosophers* by Pnueli and Zuck [31]. We based our implementation on a case study which was presented on the PLASMA-lab website.[3] A similar example was also shown for PRISM [21].

The implementation for this example was straightforward. We have a simple philosopher state machine, which is illustrated in Figure 3. A philosopher first decides if he wants to remain thinking or if he becomes hungry. In the States 1 – 7 he is hungry and in States 8 and 9 he is eating. The guards $lfree$ and $rfree$ determine if the left and right forks are free. Our model is basically a circle of individual philosophers which all have a right and left neighbour, but it also contains observation and control functionality like a counter for steps and Boolean variables to check, if someone was eating in

[2]https://fscheck.github.io/FsCheck/StatefulTestingNew.html

[3]https://project.inria.fr/plasma-lab/examples/dining-philosophers

```
public class CusumProperty{
  Property property; Config config; double p_init; double k; double lambda; int max_n;

  public CusumProperty(Property p, Config c, double p_init, double k, double lambda, int max_n){
    this.property = p; this.config = c; this.p_init = p_init; this.k = k;
    this.lambda = lambda; this.max_n = max_n;
    Assert.True(p_init != k,"p_init and k cannot have the same value!");
  }
  public void QuickCheck(){
    double s_i = 0; double m_n = 0;
    for (int i = 1; i <= max_n; i++){
      bool success = false;
      try{
        Check.One(config, property);
        success = true;
      } catch {}
      double ratio = success ? (k / p_init) : ((1 - k) / (1 - p_init));
      s_i = s_i + Math.Log(ratio);
      m_n = i == 1 ? s_i : Math.Min(m_n, s_i);
      if (s_i - m_n >= lambda){
        Console.WriteLine("Change detected after "+i+" samples!");return;
      }
    } Console.WriteLine("No change detected after "+max_n+" samples!");
  }
}
```

Listing 5. Implementation of change detection with a CusumProperty.

the past. A generator serves as a scheduler that randomly selects a philosopher that should be executed or generates a stop command when we know the outcome of a sample. The generator is part of the FsCheck state machine specification that serves as our simulation environment. Only one command class is needed for this specification, which is responsible for the execution of the model and also performs the evaluation of our properties.

We checked the same quantitative properties as used for the PLASMA-lab case study.

1) Is any of the philosophers hungry within 1000 steps and after that will any philosopher eat within 1000 steps?
2) What is the probability that a given philosopher will eat within 30 steps (for a table size of 150)?

We performed our evaluation in a virtual machine with 4 GB RAM and one CPU on a Macbook Pro (late 2013 version) with 8 GB RAM and a 2.6 GHz Intel Core i5. The first property was evaluated for different numbers of philosophers by applying a Monte Carlo simulation with Chernoff-Hoeffding bound. The parameter settings were as follows: $\epsilon = 0.003$ and $\delta = 0.01$, which results in a sample number of 294,351. The property was checked with our approach and with PLASMA-lab version 1.4.0 with the same parameters and thus the same number of samples. The results are shown in Table I. For philosopher tables with a small size our approach is slower than PLASMA-lab, but for a larger number of philosophers our approach performs better. We assume the reason for this is that we can check the property in a more efficient way. We do not always check if all philosophers become hungry or are eating, we only check the currently executed philosopher.

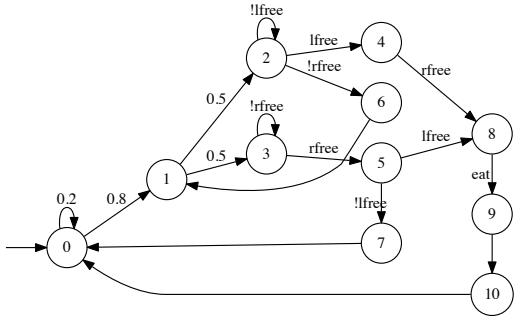

Fig. 3. State machine of a philosopher as presented for PLASMA-lab.

We checked the second property with a Monte Carlo simulation with 30 million samples. The property was true for 29 samples, which means the probability is 9.6×10^{-7}. The run time was 110 minutes. The results are similar to those of PLASMA-lab. In contrast to them we used a smaller sample number and we did not implement parallelisation.

Additionally, we checked two qualitative properties:

1) Is the probability that a given philosopher will eat within 50 steps closer to 0.1 or 0.15 (for a table size of 20)?
2) Can a change in the probability that a given philosopher eats within 50 steps be detected, when the number of philosophers rises? (We start with a certain number of philosophers and add a philosopher every 300 samples.)

We checked the first property with the SPRT with value 0.01 for the type I and type II error parameters (α and β). The result was that the alternative hypothesis H_1 was accepted, which means that the probability that a given philosopher will eat within 50 was closer to 0.15.

The second property was evaluated with the CUSUM algorithm with different initial numbers of philosophers. Initially, we performed a Monte Carlo simulation to obtain the probability p_{init} for a constant number of philosophers. Then, we adapted the original model so that a new philosopher

TABLE I
DINING PHILOSOPHERS RUNTIME COMPARISON FOR RISING TABLE SIZE FOR PROPERTY 1. THE PROPERTY WAS ALWAYS TRUE.

#Philosophers	Runtime [s]	PLASMA-lab Runtime [s]
3	969	6
10	991	14
30	1031	47
100	1145	256
300	1538	2151
1000	2676	17057

TABLE II
DINING PHILOSOPHERS CUSUM EVALUATION WITH DIFFERENT INITIAL NUMBER OF PHILOSOPHERS AND PARAMETERS.

Initial #Philosophers	p_{init}	k	Change detected at	
			Sample	#Philosophers
5	0.712	0.612	319	6
10	0.387	0.287	961	13
15	0.250	0.150	1072	18
20	0.157	0.057	1337	24

was added every 300 samples. We wanted to detect when the probability is 10% below the initial probability, which gives us the threshold $k = p_{init} - 0.1$. For the sensitivity threshold, we selected the value eight, which was enough to prevent false positives and we chose 5000 as a maximum sample number. The results are shown in Table II. It can be seen that a change can be detected quite fast, for example, for five philosophers we can detect a change after 319 samples, when the number of philosophers was increased to six. For a higher initial philosopher number the CUSUM algorithm takes longer, because the probability change was smaller when one philosopher was added.

We compared our modelling style in the programming language to the models defined for PLASMA-lab and noticed several differences. PLASMA-lab needs separate models for settings with different numbers of philosophers. We have only one model that contains a parameter for the table size. Furthermore, our model supports a dynamic change of the number of philosophers. Another observation is the long formulas in the PLASMA-lab models. The models contain formulas that include variables for each philosopher, e.g.:

```
label "hungry" = ((p1>0)&(p1<8))|...|((pn>0)&(pn<8));
```

For a model with 300 philosophers this quickly becomes impracticable. This can be avoided with an abstraction layer added to PLASMA-lab, e.g., by introducing a custom DSL [4]. In contrast, modelling in a programming language allows us to formulate these (quantified) formulas with loops. Consequently, in our object-oriented framework it is very easy to create models and to adjust them to different settings. For example, the philosopher case study was implemented within an hour and it can be easily adjusted to different settings.

On the other hand, PLASMA-lab provides a nice graphical user interface that helps the user to become familiar with the SMC techniques. Moreover, it provides a helpful simulation feature for debugging, which makes it possible to execute a model step by step and inspect all variables.

B. Randomised Consensus Case Study

The second case study is the *randomised consensus shared coin protocol* by Aspnes and Herlihy [5]. Our model is inspired by a PRISM case study [22]. It is also a PLASMA-lab case study, which is presented at its website.[4] The protocol describes an algorithm for achieving consensus among a number of processes that can communicate with shared memory. The protocol needs a constant parameter k that is required for the

[4]https://project.inria.fr/plasma-lab/examples/consensus-protocol

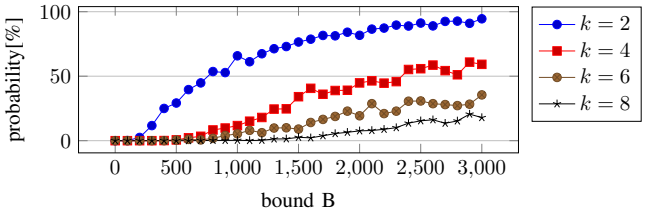

Fig. 4. Simulation results for the property: can the protocol finish within B steps for different k values and a process number of 10?

computation and influences the probability that the protocol finishes within a certain number of steps B.

The results of our case study are presented in Figure 4. Each data point in this figure was computed with a Monte Carlo simulation with 1000 samples. The outliers could be avoided with a bigger sample number. We performed simulations with PLASMA-lab in order to compare our results. The results of our SMC approach are consistent with the results obtained when running PLASMA-lab.

Additionally, we applied the SPRT in order to check the following property: is the probability that the protocol finishes within 500 steps closer to 0.2 or 0.3, when we consider k=2 and ten processes? We used the value 0.01 for the type I and type II error parameters (α and β) and the result was that the null hypothesis H_0 was accepted, which means that the value is closer to 0.2.

C. Bluetooth Case Study

We performed the third case study for a device discovery phase of Bluetooth, which is a wireless telecommunication standard [27]. This standard tries to avoid interference problems by applying a frequency hopping scheme. For this scheme the devices use pseudo-random jumps between common sets of frequencies. Figure 5 illustrates the phases of the scheme. It can be seen that there is a scan state, in which devices are listening for requests. When a request is received by a device, then it enters a *reply* state, where it answers a request after two time slots. (A time slot has a duration of 312.5μs.) Then, the device must wait for a random number of time slots. After this waiting time, the device goes back to the *scan* or the *sleep* state. In the *scan* state a device can also start a *sleep* state to reduce the energy consumption, when no request was received. The case study was originally presented for PRISM [11] and later also for UPPAAL-SMC [10]. We based our implementation on the PRISM model. Compared with our previous case studies, the model was more complex, because it has a number of different modules which interact through synchronisations. PRISM models support synchronized actions, which enable

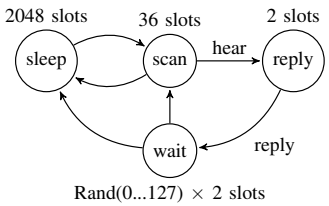

Fig. 5. Bluetooth device discovery as presented for PRISM [11].

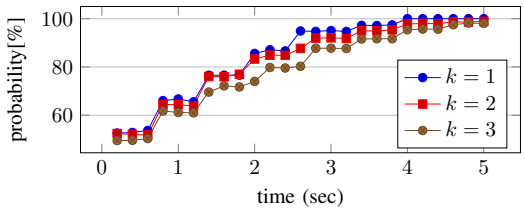

Fig. 6. Bluetooth evaluation results for the property: what is the probability that we can observe k replies within a specified time?

two or more modules to perform actions simultaneously. The model comprises modules for a sender, a receiver and for the frequency calculation. For our model implementation we had to add functionality for the synchronisation. This was done by executing the corresponding actions on all modules when they were part of the synchronisation and we also had to make sure that the variable updates during these actions had no influence on the guards of the other executed actions. The rest of the implementation was similar to the one for the dining philosopher case study.

We checked the following properties:

1) What is the probability that we can observe k replies within a specified time?
2) What is the probability that the receiver sleeps at most s times until we observe k replies?

We performed a Monte Carlo simulation with 10000 samples to check the first property. The results are shown in Figure 6. It can be seen that the data points have a stair-like structure. This is because of the sleep phases, which occur at certain probabilities and cause a sharp increase of the required time.

The second property was checked with a Monte Carlo simulation with Chernoff-Hoeffding bound with $\epsilon = 0.01$ and $\delta = 0.01$, which gives us a sample number of 26,492. Table III shows the results for this property. We performed the evaluation until we observed k replies and we checked in how many cases we can observe this number of replies before the receiver sleeps s times. As expected we see an increase of the probability of observing k replies, when the number of allowed sleep phases rises. The results we obtained for both properties were corresponding to the results of the case study from PRISM. Hence, our approach could also reproduce the simulation of a more complex stochastic model.

Furthermore, we performed an evaluation with the SPRT

TABLE III
BLUETOOTH PROPERTY: WHAT IS THE PROBABILITY THAT THE RECEIVER SLEEPS AT MOST s TIMES UNTIL WE OBSERVE k REPLIES?

Max Sleep Count s	**Probability of Finishing [%]**		
	$k = 1$	$k = 2$	$k = 3$
0	52.733	51.853	50.812
1	65.895	64.593	61.426
2	77.28	75.664	71.308
3	86.147	84.569	79.613
4	94.821	92.126	87.411
5	97.505	94.957	91.462
6	100	97.947	96.018
7	100	98.773	97.780
8	100	99.649	99.245

TABLE IV
BLUETOOTH PROPERTY: IS THE PROBABILITY THAT WE CAN OBSERVE k REPLIES WITHIN A CERTAIN TIME CLOSER TO x OR y?

Time (sec)	$H_0 : p_0 = x$	$H_1 : p_1 = y$	**Accepted Hypothesis**		
			$k = 1$	$k = 2$	$k = 3$
1	0.6	0.65	H_1	H_1	H_0
2	0.8	0.85	H_1	H_1	H_0
3	0.85	0.95	H_1	H_1	H_0

in order to check the property: is the probability that we can observe k replies within a certain time closer to x or y? We used the value 0.01 for α and β and we checked the property for different time limits and values for x and y. The results are shown in Table III.

VI. CONCLUSION

We have demonstrated that statistical model checking can be quite easily integrated into a property-based testing framework. We have implemented four commonly used SMC algorithms in the form of SMC properties and evaluated them on standard examples from the literature: the dining philosophers, a randomised consensus shared coin protocol and a Bluetooth device discovery protocol. The results are encouraging. The case studies revealed that our approach enables the definition of stochastic models and properties in a high-level programming language, which provides some benefits in the modelling style and is easier to use for developers who are not familiar with (temporal) logics.

The elegance of our integration is due to the fact that our new SMC properties take a classical property to be checked as input parameter. This results in a very flexible SMC approach where, e.g., state-machine properties as well as algebraic-properties can be checked.

In our previous work [2], we demonstrated that our SMC properties also support the conformance testing of implementations with stochastic failures against a correct model. This allows the assessment of the failure probability. In contrast to this statistical conformance analysis, here we focused on classical SMC and presented an optimized approach that utilizes only the model part of a state-machine property.

In the near future, we intend to utilize this method for load- and performance testing. For example, we will analyse the average response time of an industrial web-service application [1] under different user profiles. Our vision is to transfer this technology to our industrial partners. The fact that SMC algorithms can be represented as SMC properties inside a PBT framework should make statistical model checking accessible to test engineers already familiar with PBT.

ACKNOWLEDGEMENT

The research leading to these results was funded by the Austrian Research Promotion Agency (FFG), project number 845582, Trust via cost function driven model based test case generation for non-functional properties of systems of systems (TRUCONF). The authors would like to thank Florian Lorber, Silvio Marcovic, Martin Tappler and the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

REFERENCES

[1] B. K. Aichernig and R. Schumi, "Property-based testing with FsCheck by deriving properties from business rule models," in *2016 IEEE Ninth International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 13th Workshop on Advances in Model Based Testing (A-MOST 2016)*. IEEE Computer Society, 2016, pp. 219–228.

[2] ——, "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*. IEEE Computer Society, 2016, pp. 71–76.

[3] M. AlTurki and J. Meseguer, "PVESTA: a parallel statistical model checking and quantitative analysis tool," in *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*, ser. LNCS, vol. 6859. Springer, 2011, pp. 386–392.

[4] A. Arnold, B. Boyer, and A. Legay, "Contracts and behavioral patterns for SoS: The EU IP DANSE approach," in *Proceedings 1st Workshop on Advances in Systems of Systems, AiSoS*, ser. EPTCS, vol. 133. Open Publishing Association, 2013, pp. 47–66.

[5] J. Aspnes and M. Herlihy, "Fast randomized consensus using shared memory," *Journal of Algorithms*, vol. 11, no. 3, pp. 441–461, 1990.

[6] B. Boyer, K. Corre, A. Legay, and S. Sedwards, "PLASMA-lab: A flexible, distributable statistical model checking library," in *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, ser. LNCS, vol. 8054. Springer, 2013, pp. 160–164.

[7] P. E. Bulychev, A. David, K. G. Larsen, M. Mikucionis, D. B. Poulsen, A. Legay, and Z. Wang, "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.*, ser. EPTCS, vol. 85. Open Publishing Association, 2012, pp. 1–16.

[8] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*. ACM, 2000, pp. 268–279.

[9] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, and S. Sedwards, "Statistical model checking for biological systems," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 3, pp. 351–367, Jun. 2015.

[10] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and Z. Wang, "Time for statistical model checking of real-time systems," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. LNCS, vol. 6806. Springer, 2011, pp. 349–355.

[11] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker, "A formal analysis of Bluetooth device discovery," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 6, pp. 621–632, Oct. 2006.

[12] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*. ACM, 2014, pp. 167–181.

[13] Z. Govindarajulu, *Sequential statistics*. World Scientific, 2004.

[14] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, "Approximate probabilistic model checking," in *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, ser. LNCS, vol. 2937. Springer, 2004, pp. 73–84.

[15] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.

[16] J. Hughes, "QuickCheck testing for fun and profit," in *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007*, ser. LNCS, vol. 4354. Springer, 2007, pp. 1–32.

[17] C. Jegourel, "Rare event simulation for statistical model checking," Theses, Universite de Rennes 1, France, Nov. 2014.

[18] C. Jégourel, A. Legay, and S. Sedwards, "A platform for high performance statistical model checking - PLASMA," in *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, ser. LNCS, vol. 7214. Springer, 2012, pp. 498–503.

[19] A. S. Kalaji, R. M. Hierons, and S. Swift, "Generating feasible transition paths for testing from an extended finite state machine (EFSM)." in *Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation (ICST'09)*. IEEE Computer Society, 2009, pp. 230–239.

[20] D. Koller, D. A. McAllester, and A. Pfeffer, "Effective Bayesian inference for stochastic programs," in *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.* AAAI Press / The MIT Press, 1997, pp. 740–747.

[21] M. Kwiatkowska, G. Norman, and D. Parker, "Verifying randomized distributed algorithms with PRISM," in *Proc. of the Post-CAV'00 Workshop on Advances in Verification (WAVe'2000)*, 2000.

[22] M. Kwiatkowska, G. Norman, and R. Segala, "Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM," in *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, ser. LNCS, vol. 2102. Springer, 2001, pp. 194–206.

[23] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. LNCS, vol. 6806. Springer, 2011, pp. 585–591.

[24] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, ser. LNCS, vol. 6418. Springer, 2010, pp. 122–135.

[25] A. Legay and S. Sedwards, "On statistical model checking with PLASMA," in *2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014*. IEEE Computer Society, 2014, pp. 139–145.

[26] A. Legay and L.-M. Traonouez, "Statistical model checking with change detection," Sep. 2015, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01242138

[27] P. McDermott-Wells, "What is Bluetooth?" *IEEE Potentials*, vol. 23, no. 5, pp. 33–35, Dec 2005.

[28] R. Nilsson, *ScalaCheck: The Definitive Guide*, ser. IT Pro. Artima Incorporated, 2014.

[29] A. V. Nori, C. Hur, S. K. Rajamani, and S. Samuel, "R2: an efficient MCMC sampler for probabilistic programs," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.* AAAI Press, 2014, pp. 2476–2482.

[30] M. Papadakis and K. Sagonas, "A PropEr integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Erlang'11*. ACM, 2011, pp. 39–50.

[31] A. Pnueli and L. Zuck, "Verification of multiprocess probabilistic protocols," *Distributed Computing*, vol. 1, no. 1, pp. 53–72, 1986.

[32] K. Sen, M. Viswanathan, and G. A. Agha, "VESTA: a statistical model-checker and analyzer for probabilistic systems," in *Second International Conference on the Quantitative Evaluaiton of Systems (QEST 2005), 19-22 September 2005, Torino, Italy*. IEEE Computer Society, 2005, pp. 251–252.

[33] Y. Wada and S. Kusakabe, "Performance evaluation of a testing framework using QuickCheck and Hadoop," *Journal of Information Processing*, vol. 20, no. 2, pp. 340–346, 2012.

[34] A. Wald, *Sequential analysis*. Courier Corporation, 1973.

[35] S. S. J. Wang and M. P. Wand, "Using Infer.NET for statistical analyses," *The American Statistician*, vol. 65, no. 2, pp. 115–126, 2011.

[36] H. L. S. Younes, "Ymer: a statistical model checker," in *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, ser. LNCS, vol. 3576. Springer, 2005, pp. 429–433.