Towards Integrating Statistical Model Checking into 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) became increasingly popular, mainly because it does not suffer from one of the major problems that limits traditional model checking, the so called state-space-explosion problem. SMC solves this problem by simulating a stochastic model for finitely many executions. There exist a number of SMC tools, but they require the user to learn a specific modelling language and a particular (temporal) logic to express properties. In this paper we propose a more flexible application of SMC, where both the model and the properties can be defined in a programming language. The technique builds upon the well-known property-based testing approach. We use the programming language C# and its associated tool FsCheck to demonstrate our approach. A stochastic counter serves as illustrating example.**

1. 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 [5] or PLASMA-lab supports a number of different modelling languages, like the Reactive Module Language or Matlab Simulink [4]. 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 [6]. 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.

Our approach supports conventional testing of an SUT with stochastic failures and classical SMC by simulating stochastic models. Conventional testing is realised by utilising state machine properties and comparing faulty systems with a correct model repeatedly by executing these state machine properties within our SMC properties. Classical SMC can be done by utilizing the model as observer automaton and simulating the stochastic model as SUT.

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 dwell time in certain states. The default behaviour for checking PBT state machines is to do random walks through the model by generating (input) command sequences. The generation of these sequences can be controlled with generators. SMC needs 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 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 [16] the required number of samples can be specified and it can report the number of failing samples. This enables a Monte Carlo simulation. In contrast, the focus of our approach is on hypothesis testing, but for demonstration we also show a Monte Carlo method, because it is a common SMC algorithm.

UPPAAL-SMC is a tool for checking real-time properties [5]. It 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) [21] and the Chernoff-Hoeffding bound [9].

The probabilistic model checker PRISM was also extended with SMC functionality [13]. Similar to UPPAAL-SMC it supports priced timed automata, but also discrete- and continuous-time Markov chains, Markov decision processes and probabilistic automata. It is also able to perform the same algorithms as UPPAAL-SMC, i.e. the SPRT and the 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) [19]. 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 [2] presented an extension of VESTA called PVESTA. This extension includes parallel algorithms for SMC and client-server support for VESTA.

Another statistical model checker called Ymer was presented by Younes [22]. 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.

The most similar to our work is from Jegourel et al. [11] and Boyer et al. [4]. First, they 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. 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. [15] presented an algorithm for change detection called cumulative sum, which was also added to the library.

Existing SMC tools often have a rather restricted 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 [7]. Another approach that enables a high-level specification of Systems of Systems was presented by Arnold et al. [3]. 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 are the powerful generators, which are the major ingredients 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 we could not find any work that combines SMC and PBT except that the PBT tools can report the number of passed and failed test-cases which can be seen as a Monte Carlo simulation. Consequently, the contributions of this paper are the following. The main contribution is a new SMC approach that uses the modelling notations from PBT and checks PBT properties instead of logical formulas that are used in conventional SMC approaches. It can also be seen as a novel extension of PBT with SMC functionality providing testers who are already familiar with a PBT tool the option to analyse the stochastic properties of their SUT. Our approach allows the assessment of stochastic failures of an SUT by a comparison with an ideal model. Moreover, it supports classical SMC by performing simulations with state machine properties and evaluating temporal properties as observer automata.

Structure. First, Section 2 will explain the basics of SMC, PBT and FsCheck. Next, in Section 3 we demonstrate how SMC methods can be applied to a small example of a stochastic counter with faulty behaviour. Then, in Section 4 we present details about the implementation of our approach. Finally, we draw our conclusions in Section 5.

2. Background

2.1. 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 [14].

We show two algorithms in order to illustrate our approach for both quantitative and qualitative questions:

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 is 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 [4].

Sequential Probability Ratio Test (SPRT). This sequential method [21] 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 $\log \Lambda_m \leq \log \frac{\beta}{1-\alpha}$ [8].

2.2. Property-Based Testing

Property-based testing (PBT) is a random testing technique that evaluates a function or a system by checking a given property. A property is a high-level specification of behaviour that should hold for a range of data points. For example, a property might state that a function should have a certain expected behaviour. When the function runs through as expected, then the property passed, otherwise a counter example is returned. Simple properties can be expressed as predicates, i.e. functions with Boolean return values that should be true when the property is fulfilled. These functions should work for any input values, hence a high number of random inputs are generated for the parameters. Another important aspect of PBT is shrinking, which is used to find a similar simpler counterexample, when a property fails. In order to shrink a counterexample, a PBT tool searches for smaller failing counterexamples. The search method can be specified individually for different data types [17], [18], [10]. A simple example of an algebraic property is that the reverse of the reverse of a list must be equal to the original list:

$$\forall xs \in Lists[T] : reverse(reverse(xs)) = xs$$

A PBT tool will generate a series of random lists xs, execute the reverse function and evaluate the property.

PBT can also be applied to models in the form of extended finite state machines (EFSMs) [12]. An EFSM can formally be described as a 6-tuple (S, s_0, V, I, O, T). S is a finite set of states, $s_0 \in S$ is an 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, $t \in T$ 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 [12].

In order to use such an EFSM for PBT the permitted transition sequences have to be defined with preconditions and also the effect of each transition has to be defined with postconditions. Pre-, postconditions and the execution semantics of transitions are encapsulated in so-called commands $Cmds$. A property of an EFSM is that for each permitted path in the model, the postcondition of each transition, respectively command of the path must hold. In order to check this property a PBT tool produces random transition sequences and checks the postconditions after each transition. Given two functions to execute the model and the actual SUT

$$cmd.runModel, cmd.runActual : S \times I \to S \times O$$

a property of an EFSM can be specified via a pre- and a postcondition as follows:

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

The Boolean function $cmd.pre$ is the precondition. It defines the valid inputs and states of a command. The post condition $cmd.post$ relates the new states and the outputs of the SUT and the model after the execution of the command.

PBT constitutes a flexible and scalable model-based testing technique, because it is random testing and it has been shown that it generates a large number of tests in reasonable time [20]. The first PBT tool was QuickCheck [6] for Haskell. Since then, it has been ported to many other programming languages, e.g., ScalaCheck [16] and Hypothesis[1] for Python. Our approach is shown for FsCheck.

2.3. FsCheck

FsCheck is a PBT tool for .NET based on QuickCheck and influenced by ScalaCheck. Like ScalaCheck it extends the basic QuickCheck functionality with support for state-based models. With FsCheck, properties can be defined both in a functional programming style with F# and in an object-oriented style with C#. Similar to QuickCheck it has default generators for basic data types and more complex ones can be defined via composition. Furthermore, FsCheck has extensions for unit testing, which support a convenient definition and execution of properties like for 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. 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. For its definition we need to implement an interface comprising a model, an SUT, an initial state, a command generator, and classes for the commands. More details about these specifications were shown in our previous work [1].

3. Example

In this section we demonstrate our approach with an example of a counter, which is commonly used in the PBT community to demonstrate model-based testing. 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.

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

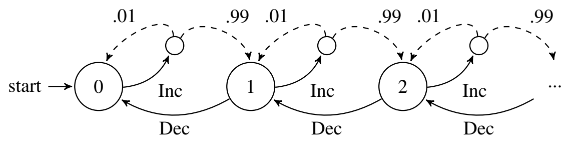

Figure 1. Stochastic model example of a counter from the PBT community.

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 stochastic counter 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 the 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 4. 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 our MonteCarloProperty was that the property holds in 98.7% of the cases when we consider a sample length of two commands.

Another example property for the SPRT is shown here:

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

Four arguments are needed for the SPRT method: the probability for the null hypothesis H_0, the probability for the alternative hypothesis H_1 and the type I and type II error parameters. The example shows 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 (closer to 0.95) was accepted.

The concrete testing of properties that we want to check happens in the state machine specification of FsCheck. A state machine property of FsCheck can be evaluated within

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

Listing 1. Stochastic counter implementation for FsCheck.

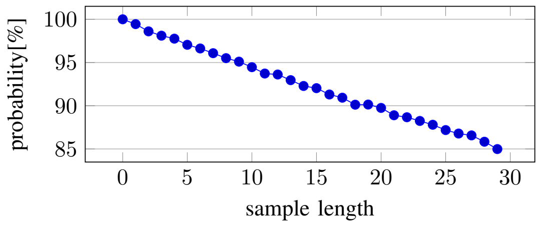

Figure 2. Simulation results for the property: how likely is it that the stochastic counter behaves like a normal counter?

our SMC properties. We use our stochastic counter as SUT and a regular counter as model and check if we can find a difference for a generated command sequence, which represents a sample. This approach is useful if an SUT has failures that occur irregularly. With our introduced properties it is possible to compute the probability of certain failures. Moreover, they enable the assessment of hypotheses, like if a certain type of error is more likely than another. This approach might also be suitable for reliability and performance testing, because the required criteria can be easily specified in the model and checked when the SUT is executed. For example, web applications, where performance (response time) is an important issue when the number of users increases, might provide an interesting case study for future work. This setup of checking a stochastic SUT against a specification EFSM could also be used for classical SMC: the temporal properties would be expressed as observer-automata specifications that are checked against the stochastic models interpreted as SUT.

Figure 2 shows the results of the evaluation of the counter example. We performed a Monte Carlo simulation with 100,000 samples for each data point in order to compute the probability that the stochastic counter behaves 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, respectively longer random walks on the model, it is more likely that we observe a faulty increment that fails incrementing.

4. 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.

We propose new properties for each SMC algorithm. The difference to normal PBT properties is 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 is closer to a null hypothesis or 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 function $QuickCheck$ that performs the SMC algorithm by simulating the input PBT property, which is used to generate samples and also to check them. The SMC

```
1  public class MonteCarloProperty{
2    protected Property property; protected Config config; protected int samples;
3
4    public MonteCarloProperty(Property p, Config c, int samples){
5      this.property = p; this.config = c; this.samples = samples;
6    }
7    public void QuickCheck(){
8      int passCnt = 0;
9      for (int i = 0; i < samples; i++){
10       try{
11         Check.One(config, property);
12         passCnt++;
13       }catch{}
14     }
15     Console.Write("Property␣holds␣"+((double)passCnt/samples*100)+"%\n"); } }
```

Listing 2. Implementation of a simple Monte Carlo simulation.

```
1  class SPRTProperty{
2    Property property; Config config; double p0; double p1; double log_a; double log_b;
3
4    public SPRTProperty(Property p, Config c, double p0, double p1, double alpha, double beta){
5      this.property = p; this.config = c; this.p0 = p0;  this.p1 = p1;
6      this.log_a = Math.Log(beta / (1 − alpha));
7      this.log_b = Math.Log((1 − beta) / alpha);
8    }
9    public void QuickCheck(){
10     double s_i = 0;
11     do{
12         bool success = false;
13         try{
14           Check.One(config, property);
15           success = true;
16         } catch {}
17         double ratio = success ? (p1 / p0) : ((1 − p1) / (1 − p0));
18         s_i = s_i + Math.Log(ratio);
19     } while (log_a < s_i && s_i < log_b);
20     if(s_i >= log_b){ Console.WriteLine("H1␣accepted."); }
21     else if(s_i <= log_a){ Console.WriteLine("H0␣accepted."); } } }
```

Listing 3. Implementation of hypothesis testing with an SPRTProperty.

properties for the different SMC algorithms have the same structure, but require different parameters for the algorithms and have 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 4). The $QuickCheck$ function (Line 7) 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 number of tests that should be performed. The $Check.One$ method also evaluates, if the property was fulfilled. If it fails, an exception is thrown. Therefore, we have put the method call inside a try-catch block (Lines 10–13) 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.

An SPRTProperty which performs the SPRT method is shown in Listing 3. 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 algorithm, 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 6–7). Inside the $QuickCheck$ method we perform the simulation. A sample is checked in the same way as in a MonteCarloProperty (Lines 13–16). For each sample we calculate the log-likelihood ratio and sum it up with the previous ratios (Lines 17–18). We stop when the sum is outside the thresholds. Depending on which threshold was met, either H_0 or H_1 is accepted.

The architecture of our introduced SMC properties makes it easy to check all kinds of PBT properties. Although our main 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 introduced properties can easily be implemented in other PBT tools. As already explained in Section 2.2, there exist various PBT tools for different programming languages. It is not much effort to implement 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. For example, 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.

It should be mentioned that we used a new experimental version of the FsCheck state machine specification.[2] This version supports the generation of fixed length samples and stop commands that enable a termination during the command generation. These two features are important for our implementation, because we have to ensure that our generated samples are long enough and also that we can stop, when we know the outcome of a sample.

5. Conclusion

We have demonstrated that statistical model checking can be quite easily integrated into a property-based testing framework. We have implemented two commonly used SMC algorithms in the form of SMC properties and illustrated our approach by applying it to a stochastic counter implementation. Furthermore, we highlight some benefits of our approach in the modelling style. 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. This integration method enables the evaluation of faulty implementations with stochastic failures, by comparing them to a correct implementation, as well as the evaluation of stochastic models with observer automata. For load- and performance testing such an analysis is also a promising option to check if certain non-functional requirements are met by an SUT.

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. We are in the process of evaluating this technique by applying it typical case studies from the SMC literature.

Acknowledgment

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.

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

References

[1] B. K. Aichernig and R. Schumi, “Property-based testing with FsCheck by deriving properties from business rule models,” in *ICSTW*. IEEE, 2016, pp. 219–228.

[2] M. AlTurki and J. Meseguer, “PVESTA: A parallel statistical model checking and quantitative analysis tool,” in *CALCO*, ser. LNCS, vol. 6859. Springer, 2011, pp. 386–392.

[3] A. Arnold, B. Boyer, and A. Legay, “Contracts and behavioral patterns for SoS: The EU IP DANSE approach,” in *AiSoS*, ser. EPTCS, vol. 133. Open Publishing Association, 2013, pp. 47–66.

[4] B. Boyer, K. Corre, A. Legay, and S. Sedwards, “PLASMA-lab: A flexible, distributable statistical model checking library,” in *QEST*, ser. LNCS, vol. 8054. Springer, 2013, pp. 160–164.

[5] 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 *QAPL*, ser. EPTCS, vol. 85. Open Publishing Association, 2012, pp. 1–16.

[6] K. Claessen and J. Hughes, “QuickCheck: A lightweight tool for random testing of Haskell programs,” in *ICFP*. ACM, 2000, pp. 268–279.

[7] 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.

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

[9] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, “Approximate probabilistic model checking,” in *VMCAI*, ser. LNCS, vol. 2937. Springer, 2004, pp. 73–84.

[10] J. Hughes, “QuickCheck testing for fun and profit,” in *PADL*, ser. LNCS, vol. 4354. Springer, 2007, pp. 1–32.

[11] C. Jégourel, A. Legay, and S. Sedwards, “A platform for high performance statistical model checking - PLASMA,” in *TACAS*, ser. LNCS, vol. 7214. Springer, 2012, pp. 498–503.

[12] A. S. Kalaji, R. M. Hierons, and S. Swift, “Generating feasible transition paths for testing from an extended finite state machine (EFSM).” in *ICST*. IEEE, 2009, pp. 230–239.

[13] M. Z. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of probabilistic real-time systems,” in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 585–591.

[14] A. Legay, B. Delahaye, and S. Bensalem, “Statistical model checking: An overview,” in *RV*, ser. LNCS, vol. 6418. Springer, 2010, pp. 122–135.

[15] 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

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

[17] M. Papadakis and K. Sagonas, “A PropEr integration of types and function specifications with property-based testing,” in *Erlang*. ACM, 2011, pp. 39–50.

[18] C. Runciman, M. Naylor, and F. Lindblad, “SmallCheck and lazy SmallCheck: Automatic exhaustive testing for small values,” in *Haskell*. ACM, 2008, pp. 37–48.

[19] K. Sen, M. Viswanathan, and G. A. Agha, “VESTA: A statistical model-checker and analyzer for probabilistic systems,” in *QEST*. IEEE, 2005, pp. 251–252.

[20] 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.

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

[22] H. L. S. Younes, “Ymer: A statistical model checker,” in *CAV*, ser. LNCS, vol. 3576. Springer, 2005, pp. 429–433.