Checking and Distributing Statistical Model Checking*

Peter Bulychev[1], Alexandre David[1], Kim G. Larsen[1], Axel Legay[1,2], Marius Mikučionis[1], and Danny Bøgsted Poulsen[1]

[1] Computer Science, Aalborg University, Denmark
[2] INRIA/IRISA, Rennes Cedex, France

Abstract. In this paper we propose a general framework for distributed statistical model checking of networks of priced timed automata. The first contribution is a new algorithm to distribute sequential hypothesis testing without introducing bias in the results. The second contribution is an implementation of this algorithm in Uppaal. The major contribution is an experimental and analytical evaluation of the approach through case studies, including an analysis of the SMC algorithm itself.

1 Introduction

Statistical Model Checking techniques (SMC) [8,12,17], can be seen as a trade-off between testing and formal verification. The core idea of the approach is to conduct some simulations of the system and verify if they satisfy some given property. The results are then used together with statistical algorithms in order to decide whether the system satisfies the property with some probability. Statistical model checking techniques can also be used to estimate the probability that a system satisfies a given property [8,6]. Of course, in contrast to an exhaustive approach, a simulation-based solution does not guarantee a correct result with 100% confidence. However, it is possible to bound the probability of making an error. SMC is getting widely accepted in various research areas and applied to problems that are beyond the scope of classical formal techniques [1,2,10,11,13,19,20].

Unfortunately, extremely huge sized problems and a demand of extremely high confidence may require generation of a large number of simulation runs, each of which may itself be extremely time consuming. To address this *confidence-explosion* problem, we suggest in this paper to take advantage of PC-clusters and GRID computers. In fact, it is well-known that statistical solutions methods that use samples of independent observations are often trivially parallelizable. As observed in [18], SMC algorithms can be distributed through the help of a master/slave architecture where multiple computers are used to generate the simulations. The idea is as follows: one or more slave processes register their

* Work partially supported by VKR Centre of Excellence – MT-LAB, an "Action de Recherche Collaborative" ARC (TP)I and the IDEA4CPS center established on a grant from Danish National Research Foundation.

A. Goodloe and S. Person (Eds.): NFM 2012, LNCS 7226, pp. 449–463, 2012.

ability to generate simulation with a single master process that is used to collect those simulations and perform the statistical test.

However, this process may become complex when considering sequential hypothesis testing (when the number of simulations is not known in advance). The problem is that there might exist a correlation between a time needed to generate a random simulation and the fact that a property is satisfied by this simulation. Thus it is important to guarantee that the technique will not introduce a bias towards the results that are generated by shorter simulations.

In a series of recent works [4], we have extended Uppaal with SMC algorithms applied to Networks of Priced Timed Automata – hence leading to the first implementation of SMC for real-timed stochastic systems. The objective of this paper is to go one step further and propose the first complete study of distributed SMC, in general, and in the framework of Uppaal in particular. Our contributions are:

1. A *distributed implementation* of the estimation algorithm proposed in [8]. Building on classical Monte Carlo techniques [7], an estimation algorithm precomputes the number of simulations needed to estimate the probability to satisfy a property with a given confidence. Such an algorithm which is trivially parallelizable amounts to equally distribute the number of simulations to perform between the slave computers.
2. A *new distributed algorithm* for sequential hypothesis testing where simulations are computed on the fly until a threshold is passed and a decision is taken. Here, it is important to avoid introducing bias in the results, which may be potentially complex and eventually decrease the benefit of using several processors. To counter this, [18] proposed a round-Robin solution where the runs are counted in rounds. We generalise the solution in [18] by introducing *batches* and *buffers*. The batch is used to reduce communication by sending an aggregate result of predefined size (instead of individual results). The buffer is used to improve concurrency since the nodes are more loosely synchronized.
3. A *thorough evaluation* of our implementation through new applications of SMC algorithms. In particular, we apply the distributed SMC engine to an analysis of an instance of the LMAC protocol of unprecedented size. Additionally, a thorough evaluation of the distributed SMC framework itself is made aiming at identifying optimal settings of the parameters for the framework. The evaluation is carried out both experimentally (using the implementation) as well as analytically (using SMC) based on a model of the distributed SMC algorithm itself, and with high consistency in identifications made by the two approaches.

2 Statistical Model-Checking in Uppaal

This section introduces the formalisms used in Uppaal for modeling systems and and specifying properties. Then, we briefly survey existing Statistical Model Checking (SMC) algorithms. Finally, a novel application of SMC is presented.

2.1 Networks of Priced Timed Automata

The new SMC engine of UPPAAL [3] supports the analysis of Priced Timed Automata (PTAs) that are timed automata whose clocks can evolve with different rates, and with no restrictions in guards and invariants. Additionally, we support other features of the UPPAAL model checker's input language such as integer variables, data structures and user-defined functions. We also assume PTAs are input-enabled, deterministic (with a probability measure defined on the sets of successors), and non-zeno. PTAs communicate via broadcast channels and shared variables to generate Networks of Price Timed Automata (NPTA).

Fig. 1 provides an NPTA with three components A, B, and T as specified using the UPPAAL GUI. One can easily see that the composite system $(A|B|T)$ has the transition sequence:

$$\big((A_0, B_o, T_0), [x = 0, y = 0, C = 0]\big) \xrightarrow{1} \xrightarrow{a!}$$
$$\big((A_1, B_0, T_1), [x = 1, y = 1, C = 4]\big) \xrightarrow{1} \xrightarrow{b!}$$
$$\big((A_1, B_1, T_2), [x = 2, y = 2, C = 6]\big),$$

demonstrating that the final location T_3 of T is reachable. In fact, location T_3 is reachable within cost 0 to 6 and within total time 0 and 2 in $(A|B|T)$ depending on when (and in which order) A and B choose to perform the output actions $a!$ and $b!$. Assuming that the choice of these time-delays is governed by probability distributions, a measure on sets of runs of NPTAs is induced, according to which quantitative properties such as *"the probability of T_3 being reached within a total cost-bound of 4.3"* become well-defined.

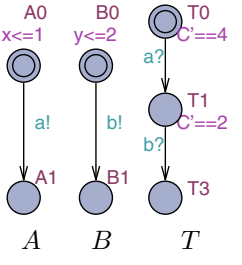


Fig. 1. An NPTA, $(A|B|T)$

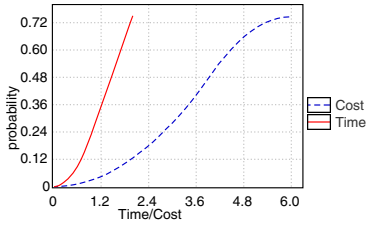


Fig. 2. Cumulative probabilities for `time` and `Cost`-bounded reachability of T_3

In our early works [4], the stochastic semantic of PTA components associates probability distributions on both the delays one can spend in a given state as well as on the transition between states. In UPPAAL uniform distributions are applied for bounded delays and exponential distributions for the case where a component can remain indefinitely in a state. In a network of PTAs the components repeatedly race against each other, i.e. they independently and stochastically decide on their own how much to delay before outputting, with the "winner" being the component that chooses the minimum delay. For instance, in the NPTA of Fig. 1, A wins the initial race over B with probability 0.75.

Properties. For specifying properties of NPTAs, we use cost-constraint temporal properties over runs of the form $\psi = \Diamond_{C \leq c} \varphi$. Here C is an observer clock (that is never reset and should grow to infinity on any infinite run), $c \in \mathbb{R}_{\geq 0}$

and φ is a state-predicate. We say that a run π satisfies $\psi = \Diamond_{C \leq c}\varphi$ if there exists a state (ℓ, v) in π satisfying φ and with $v(C) \leq c$. For an NPTA M we define $\mathbb{P}_M(\psi)$ to be the probability that a random run of M satisfies ψ.

Reconsider the example of Fig. 1, we can evaluate the the probabilities `Pr[time<=2](◇ T.T3)` and `Pr[C<=6](◇ T.T3)` in UPPAAL, obtaining as expected 0.75 for the composition $(A|B|T)$ for both of these probabilities. In fact Fig. 2 gives the time- and cost-bounded reachability probabilities for $(A|B|T)$ for a range of bounds.

2.2 Statistical Model Checking Algorithms

We briefly recall statistical algorithms allowing to answer the following two types of questions : (1) *Qualitative :* Is the probability that a random run of a given NPTA $\boldsymbol{\mathcal{A}}$ satisfies a property $\Diamond_{C \leq c}\varphi$ greater than a certain threshold θ? and (2) *Quantitative :* What is the probability that a random run of $\boldsymbol{\mathcal{A}}$ satisfies $\Diamond_{C \leq c}\varphi$? For both question a run of the system is encoded as a Bernoulli random variable that is true if the run satisfies the property and false otherwise.

Qualitative Question. This reduces to test the hypothesis H: $p = \mathbb{P}_{\boldsymbol{\mathcal{A}}}(\Diamond_{C \leq c}\varphi) \geq \theta$ against $K : p < \theta$. To bound the probability of making errors, we use strength parameters α and β and we test the hypothesis $\mathrm{H}_0 : p \geq p_0$ and $\mathrm{H}_1 : p \leq p_1$ with $p_0 = \theta + \delta_0$ and $p_1 = \theta - \delta_1$ (δ_0 and δ_1 are parameters of the algorithm). The interval $p_0 - p_1$ defines an indifference region, and p_0 and p_1 are used as thresholds in the algorithm. The parameter α is the probability of accepting H_0 when H_1 holds and the parameter β is the probability of accepting H_1 when H_0 holds. The above test can be solved by using Wald's sequential hypothesis testing [16]. This test, which is presented in Algorithm 1, computes a proportion r among those runs that satisfy the property. With probability 1, the value of the proportion will eventually cross $\log(\beta/(1-\alpha))$ or $\log((1-\beta)/\alpha)$ and one of the two hypothesis will be selected.

Algorithm 1. Hypothesis testing

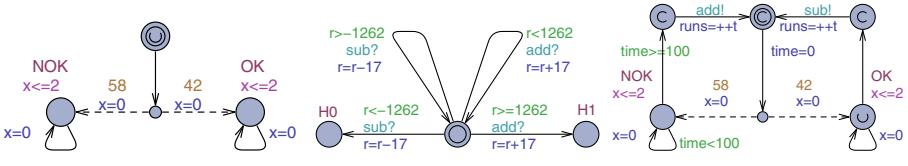
```
   function hypothesis(S:model , ψ: property)
1  r:=0
2  while true do
3      Observe the random variable x corresponding to ◇_{C≤c}φ for a run.
4      r := r + x * log(p1/p0) + (1 − x) * log((1 − p1)/(1 − p0))
5      if r ≤ log(β/(1 − α)) then accept H0
6      if r ≥ log((1 − β)/α) then accept H1
   end
```

Quantitative Question. This reduces to a Monte Carlo approach that computes the number N of runs needed in order to produce an approximation interval $[p - \epsilon, p + \epsilon]$ for $p = Pr(\psi)$ with a confidence $1 - \alpha$. The values of ϵ and α are chosen by the user and N relies on the Chernoff-Hoeffding bound.

2.3 Analysing SMC in Uppaal

In this section we will use the SMC engine of Uppaal to our first non-trivial task, namely to analyse itself! More precisely, by suitably modeling the sequential testing algorithm as well as a sample model M, we will be able to use the SMC engine of Uppaal to analyse the performance of SMC on M. Later, in Section 4, this will allow us to evaluate various naive (and even faulty) proposals for distributing SMC.

The sample model M given in Fig. 3a[1] makes an initial probabilistic choice between the two branches, each having a looping transition taken repeatedly with a delay choosen uniformly from $]0, 2]$. Performing sequential testing of the hypothesis H_0: `Pr[<=100](◇ OK)≥ 0.5` some 10 times with $\alpha = 0.05$ as level of significance and with an indifference region of ± 0.01, we consistently (and correctly) dismiss the hypothesis with an average of 408.6 runs and with standard deviation 127.5.



(a) M. (b) Master. (c) Generator.

Fig. 3. Sample model M (a) satisfying `Pr[<=100](◇ OK)`$= 0.42$ and modeling SMC of M (b,c) with respect to H_0: `Pr[<=100](◇ OK)`$\geq$ `0.5` with 0.05 as level of significance and $[0.49, 0.51]$ as indifference region

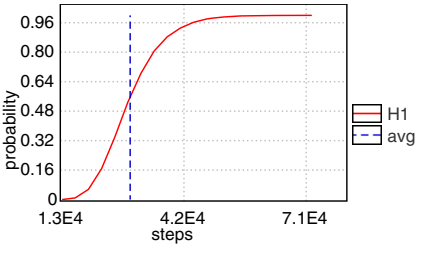
Now, aiming at obtaining a better understanding of sequential testing[2] we may simply model the sequential testing algorithm of M directly and analyse its (expected) performance using Uppaal SMC. The resulting model is given in Fig. 3 and consists of an extension of the sample model M into the component Generator that will repeatedly generate random runs of M (of time-duration 100) and report the outcome to a Master using the channels `add` (when 100 time-units has elepased without `OK` having been observed) and `sub` (used as soon as it is observed that the `OK` branch has been taken, note the absence of the `time>=100` guard on the right side of the Generator model). The Master has the obligation of adjusting appropriately the ratio-variable r according to Alg. 1, and conclude on H_0 or H_1 as soon as the value of r exceeds the given threshold. Given the indifference region $[0.49, 0.51]$ and level of significance 0.05, we find that the approximate values to be used[3] in Alg. 1 are: $-\log(p_1/p_0) =$

[1] M is a timed variant of the model proposed in [17] and used to demonstrate bias in a naive distributed approach to SMC.

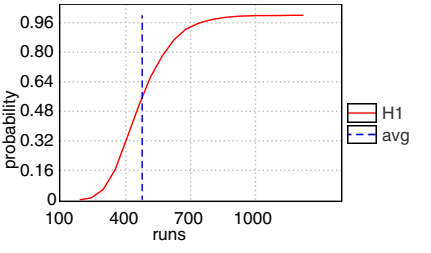
[2] The performance of sequential testing has been subject to significant studies and is well-understood [15]. The aim here is to demonstrate that our Uppaal SMC engine is a useful tool for obtaining such an insight.

[3] Those values are obtained by observing Wald's ratio on several application of the SMC algorithm to the same problem, and then take the average of the observations.

$\log(1-p_0/1-p_1) = 0.01715$ and $\log((1-\beta)/\alpha) = -\log(\beta/(1-\alpha)) = 1.2787$ ($\approx 1.262+0.017$). In the model of Fig. 3 we are using scaled integer constants for these values. Now, looking at the estimation of `Pr[#<=20000](◇ Master.H1)` in Fig. 4, we find – as expected – that the probability of accepting H_1 (H_0) tends to 1 (0) as the number of steps increases. We also see that the average number of runs is estimated to 481.4. The "mismatch" with the experimentally found average 408.6 is due to early termination when the threshold for H_0 is exceeded.



(a) $\Pr[\# \leq 80000](\Diamond\ \texttt{Master.H}_1)$



(b) $\Pr[\texttt{runs} \leq 20000](\Diamond\ \texttt{Master.H}_1)$

Fig. 4. Cumulative probability plots over number of steps and runs

3 Distributed Statistical Model-Checking in Uppaal

SMC suffers from the fact that the high confidence required by an answer may demand a large number of simulation runs, each of which may itself be time consuming. As an example, the first hypothesis test shown later in this section can generate between 14,000 and 2,390,000 runs if the parameters α, β, δ range between 0.01 and 0.001. A possible way to leverage this problem is to use several computers working in parallel using a master/slaves architecture: one or more slave processes register their ability to generate simulation with a single master process that is used to collect those simulations and perform the statistical test. When working with an estimation algorithm, this collection is trivially performed as the number of simulations to perform is known in advance and can be equally distributed between the slaves. When working with sequential algorithms, the situation gets more complicated. Indeed, we need to avoid introducing bias when collecting the results produced by the slave computers. This means that results should not be collected arbitrarily as illustrated by considering the model of Section 2.3 with several instances of the `Generator` template.

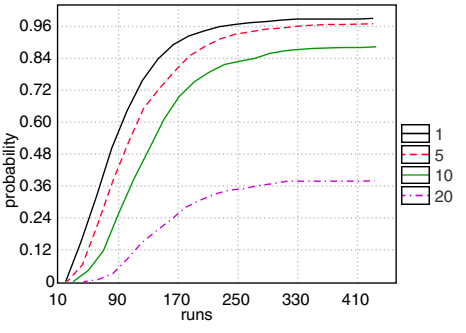


Fig. 5. Probability distributions obtained with 1 (top), 5, 10, and 20 (bottom) generator nodes

Checking the property `Pr[runs<=20000](◇ Master.H1)` Fig. 5 shows that different distributions can be obtained with different numbers of generator nodes, hence revealing a bias in the results. In fact the probability of accepting H_1 tends (incorrectly) to 0 when the number of `Generator` components increases.

A solution, which was proposed in [17], consists in observing that Wald's ratio r is updated as a function of the Bernouilli random variable x as $r+ = x*r_{acc}+(1-x)*r_{rej}$ with r_{acc} and r_{rej} being constants depending on the tested hypothesis. To reduce blocking and still update r, the non-biased algorithm updates two safe approximations for r (r_1 and r_2). If x is unknown then it updates with $r_1+ = r_{rej}$ and $r_2+ = r_{acc}$, and then testing if $r_1 \leq I$ to accept H_0 or if $r_2 \geq S$ to accept H_1[4]. When all outcomes of a round are known then we can reset $r_1 := r_2 := r$. This allows us to accept H_0 even if some accepting outcomes are missing or conversely to accept H_1 if some rejecting outcomes are missing.

We generalize [17] by aggregating the outcomes x by *batches* (of size B) and also by implementing a *buffer* (of size K) of incoming results. The batch is used to reduce communication by sending B aggregate results. The buffer is used to improve concurrency since the nodes are more loosely synchronized and they can be K runs ahead of the slowest node. Fig. 6 illustrates our algorithm at the master node that receives asynchronous messages from all other nodes in a buffer. A message is an aggregate result containing the outcome of B runs. The master may take a decision as soon as $r_1 \leq I$ or $r_2 \geq S$. When all outcomes at the bottom line of the buffer are known we reset $r_1 := r_2 := r$ with the exact updated value of r with those outcomes, and free the bottom line of the buffer. In practice, our algorithm is calibrated to count the runs up to a certain depth in the buffer. Indeed, the outcomes are weighted by B so few missing aggregated outcomes can prevent the algorithm from deciding. We have implemented this algorithm with asynchronous communications (using OpenMPI). There can be at most K pending messages due to the size of the buffer. If a slave tries to send more messages, then the communication will block waiting for a "slot" to be free. The experiment performed in the remainder of the paper has been carried out on varying numbers of nodes on a cluster with dual Xeons 5650 (hexa-cores at 2.66GHz) interconnected with infiniband.

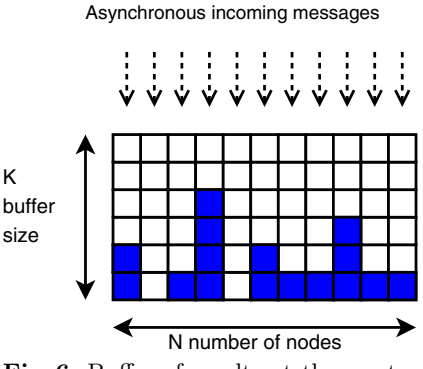


Fig. 6. Buffer of results at the master node

We first make two types of experiments to exhibit the performance characteristics of our algorithm. The experiments are carried out using the train-gate example available as a demo of UPPAAL. This model comprises a number of

[4] $I = \log(\beta/(1-\alpha))$ and $S = \log((1-\beta)/\alpha)$ as stated in Alg. 1.

trains crossing a bridge with only one track. A gate controller stops and restart the trains to ensure mutual exclusion on the bridge and absence of starvation for the trains. Our first experiment concerns 6 trains and the property of being in a state where train 5 is crossing while all the other trains are stopped.

```
Pr[<=100](<> Train(5).Cross and
        (forall (i : id_t) i != 5 imply Train(i).Stop)) >= 0.46188
```

The runs are relatively short with few components so they will be cheap to compute and we expect the throughput of messages to be high. In addition, the hypothesis we are testing is not deterministic, which means that the outcomes and computation times of the runs will vary. The property is checked with high confidence (99.999%) and small indifference regions (+/- 0.00001) to have a precise and reliable result – and to stress our distributed algorithm.

Our second experiment considers a "large" instance with 20 trains, where we check if the model satisfies mutual exclusion on the bridge, expressed by the property

```
Pr[<=1000]([] forall (i : id_t) forall (j : id_t)
         Train(i).Cross and Train(j).Cross imply i == j) >= 0.9999
```

Here, the runs are random but bounded by the same large bound and since the inner property `[]forall(i : id_t)forall(j : id_t)...` holds by model-checking, all the runs will all reach their bounds. In addition, we have 20 trains and the runs are long (1000 time units) so they are relatively expensive to generate. This means that all the runs are implicitely synchronized and small deviations are amortized by the long runs. The throughput of messages will be low, which means a low overhead compared to the actual useful work of generating the runs.

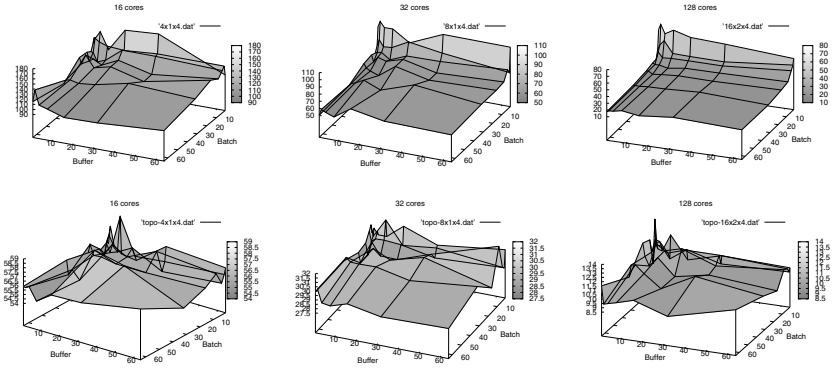
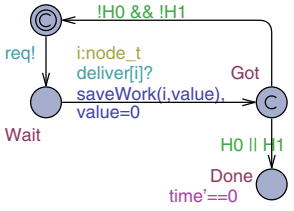
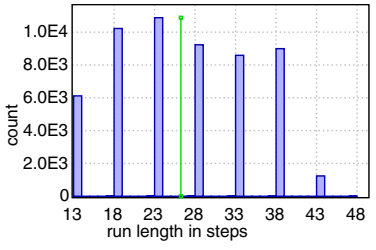


Fig. 7. Verification times on 16, 32, and 128 cores in function of B and K for the "small" model (first row) and the "large" model (second row)

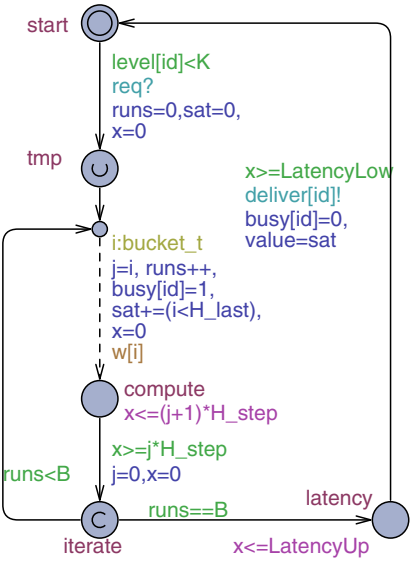
Figure 7 shows our results for different number of cores. The solution in [18] corresponds to the particular case with K and B are equal to one, exhibiting in all the experiments the worst verification time, and with performance deteriorating



(a) Master.



(b) Histogram of verification steps.



(c) Slave.

Fig. 8. Timed automata model of a statistical model checking process

with increasing number of cores (i.e. for 128 cores performance loss is a factor of 4). Though the impact of the buffer size is less, the experiments indicate that a buffer of size 2-4 will suffice. The results also demonstrate linear scalability of our distributed implementation: for $B = 32$ and $K = 2$ the verification times for 16, 32 and 128 cores are 108, 56 and 19 seconds (respectively).

4 Analyzing Distributed SMC in Uppaal

In this section we model the implemented distributed algorithm of sequential hypothesis testing and we check it using the SMC engine of Uppaal. The goal is to estimate the verification time and processor utilization, check for bias in the distributed algorithm, and explore the parameters of our distributed SMC algorithm in an analytical manner.

Modeling. We model the master and slave processes described in Section 3 as shown in Fig. 8. The master sends a broadcast request `req!` to verify batches of runs (of size B). We use a standard modeling pattern to synchronize on the corresponding `req?` as soon as possible. The master gathers the results with its `saveWork` function and loops again if neither H_0 nor H_1 is accepted. Listing 1.1 shows this `saveWork` function that implements the distributed hypothesis testing algorithm of Section 3. Uppaal uses floating point numbers that are not available in the modeling language. Instead we encode fixed point arithmetics with integers and we use precomputed tables for logarithm values. Once the master accepts H_0 or H_1, it moves to the location `Done` and stops the clock `time`.

Listing 1.1. Master code.

```
// buffer portion for early termination:
const int P = (K<=4)?K : ((K<=8)?5 : ((K<=16)?8 : ((K<=32)?10 : 12)));
bool H0 = false, H1 = false; // for hypothesis H0 and H1
int batch[N][K]; // buffer of batches (K batches for N nodes)
long satisfied =0, unsatisfied =0; // information about filled lines
long sat=0, unsat=0, unknown=N*P*B; // early results in unfilled lines
long logRatio = 0, ratioLow = 0, ratioUp = 0; // scaled by p.scale
void saveWork(const node_t node, const int value) {
    if (level [node]<=P) { // entered the early results portion
        sat += value; unsat += B-value; unknown -= B;
    }
    batch[node][level [node]] = value; level [node]++; // store
    if (level [node]==1) { // entered at the lowest level
        bool filled = forall (i:node_t) level [i]>0;
        if (filled ) { // line at the lowest level has been filled
            int L;
            for (i: node_t) { // shift all queues one by one
                satisfied += batch[i][0]; // count as firm results
                unsatisfied += B-batch[i][0];
                sat -= batch[i][0]; // discount from early results
                unsat -= B-batch[i][0]; unknown += B;
                level [i]--; // remove from buffer
                for (L=0; L<level[i]; ++L) {
                    batch[i][ L] = batch[i][ L+1]; // shift
                    if (L==P) { // entered the early results portion
                        sat += batch[i][L+1]; unsat += B-batch[i][L+1];
                    }
                }
                batch[i][ level [i]]=0; // cleanup
            }
            logRatio = p.valAcc*satisfied + unsatisfied *p.valRef;
            if (logRatio <= p.logInf) H0 = true;
            if (logRatio >= p.logSup) H1 = true;
        }
    }
    ratioLow = p.valAcc*(satisfied +sat+unknown) +
               p.valRef*(unsatisfied +unsat);
    ratioUp = p.valAcc*(satisfied +sat) +
              p.valRef*(unsatisfied +unsat+unknown);
    if (ratioUp <= p.logInf) H0 = true;
    if (ratioLow >= p.logSup) H1 = true;
}
```

Slave processes proceed to compute their batches if their communication buffers are not full (`level[id] < K`) or wait for the condition to hold. The `compute` location models the computation time of a run, chosen according to the distribution shown in Fig. 8b. This is encoded using probabilistic edges with

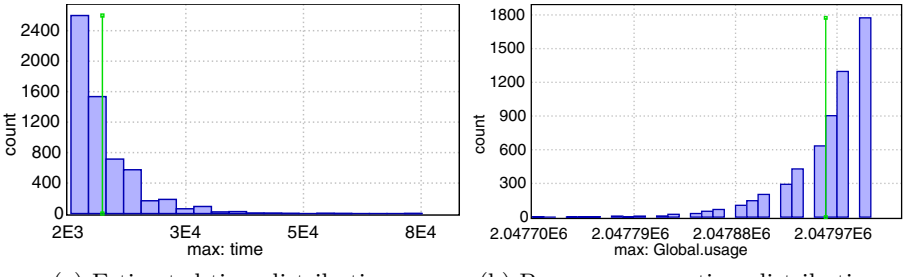
weights matching the distribution. The distribution comes from a real verification of the property in Section 3:

```
Pr[<=100](<> Train(5).Cross and
        (forall (i : id_t) i != 5 imply Train(i).Stop)) >= 0.46188
```

The last weighted edge (case `i=H`) is reserved for the runs that did not satisfy the property.

Verification. In the hypothesis we test, the actual probability is very close to 0.46188. Since the real probability falls in the indifference region of our test, we would expect that a non-biased implementation would accept H_0 or H_1 equally often. Estimating the probability of confirming the hypothesis H_0 with the query `Pr[<=10000000](<> master.H0)` gives the probability 0.503 ± 0.005 with 99.9% confidence, confirming that our algorithm is not biased as well as the validity of our model.

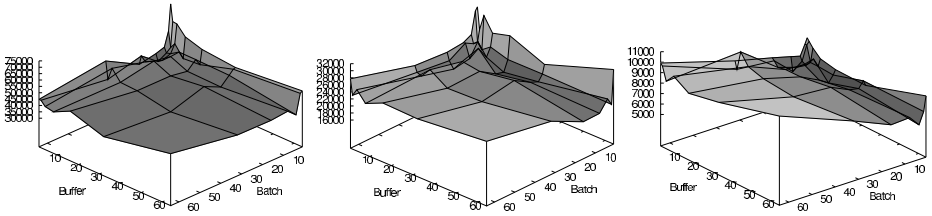
Similarly, we obtain the distribution of the verification time by the query `Pr[<=10000000](<> master.Done)` for a model with number of nodes $N = 128$, batch size $B = 64$, and buffer size $K = 4$. The result is 9557.6 time units in average and the distribution histogram is depicted in Fig. 9a. To estimate the processor usage time, we add another process with a single location with the invariant `usage'==sum(i:node_t)busy[i]`. Here, `usage` is a clock that grows with a rate equal to the number of busy nodes.



(a) Estimated time distribution. (b) Processor usage time distribution.

Fig. 9. Time estimation from 6000 runs of DSMC model

The question is now to find a good settings for the parameters of our algorithm (B and K). We perform a parameter sweep to estimate the verification time for values of B and K taking values in $1, 2, 4, 8, 16, 32, 64$ for three topologies with the number of processing nodes $N = 16, 32$, or 128. The results are depicted in Fig. 10, where it is visible that the extremely small batch size requires more time. Large batch sizes can also be detrimental in a large cluster setting (Fig. 10c where too many runs are requested in bulk than actually needed to establish the result). Buffer size of one has a huge penalty of blocking with small blocks, but it is barely noticeable otherwise. This confirms the experimental findings of Section 3.



(a) 16 nodes. (b) 32 nodes. (c) 128 nodes.

Fig. 10. Estimated verification times in model time units

5 Lightweight Media Access Control

LMAC is a Lightweight Media Access protocol (studied in [4,5]) used for scheduling communication in wireless sensor networks where the topology is determined by physical location and radio connectivity of the individual nodes. One of the goals of the LMAC protocol is to minimize the number of collisions in the network and to reconfigure the network to avoid further collisions.

The original model has been developed in [5] where topologies of 4-5 nodes are studied exhaustively using classical Uppaal and a number of topologies are identified as problematic, containing perpetual collisions. In this paper we provide new insight as to the likelihood of perpetual collisions in different topologies. This insight could not be delivered by the use of classical Uppaal and the experiment conducted is of unprecedented size.

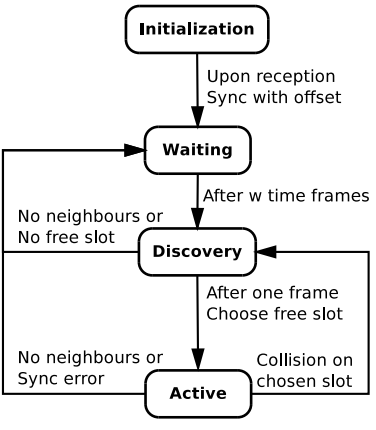


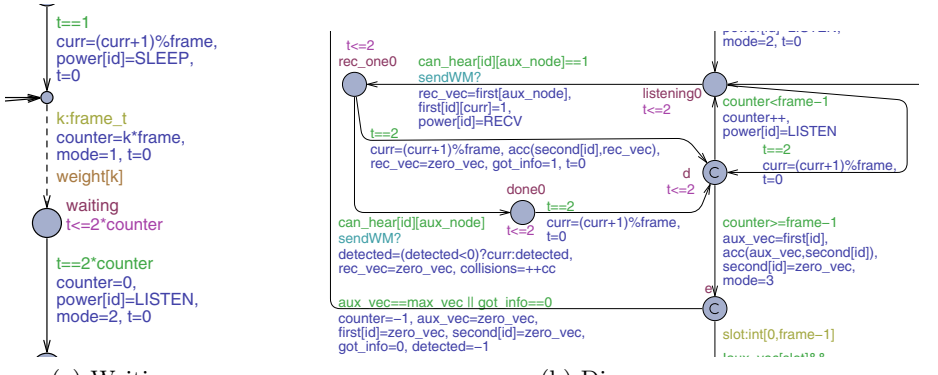
Fig. 11. LMAC protocol phases

In LMAC communication media access time is discretized into time frames and each time frame is divided into time slots. The goal of the protocol is to allocate the time slots to each node efficiently. The challenge is that there is no central node distributing and assigning slots and nodes cannot themselves listen while transmitting, hence neighbours are responsible for detecting and informing each other about collisions.

After waiting phase, the node moves to a discovery phase and listens for an entire time frame and notes which time slots are used by its neighbours. The collision counting expression `collisions=++cc;` is added on the edge from `rec_one0` to `done0` in Fig. 12b. After one time frame of discovery phase, the node chooses seemingly unused time slot and moves to an active phase. The node falls back to waiting phase if there are no neighbours (no signal received) or all slots are occupied. During active and discovery phases the node listens and notes any collisions (several receptions during the same slot). During active phase the node transmits information about collisions it has detected during its time slot and listens for collisions and information about collisions during other

time slots. From the active phase the node may fall back to discovery phase if it is notified about the collisions on its time slot and falls back to the waiting phase if it detects that neighbours are gone.

Figure 11 shows the four phases of the protocol. Initially all nodes except the gateway are listening and waiting for a radio signal from its neighbourhood during the initialization phase. The communication is triggered by a dedicated gateway node. Upon reception of signal, the node notes the relative time offset of the signal and moves to waiting phase, during which it chooses to wait for a random amount of time frames. The random delay is modeled using probabilistic branching (see Fig. 12a) with geometrical weights (`weight` array).



(a) Waiting. (b) Discovery.

Fig. 12. LMAC phases in the model

Starting from the model[5] of [5], we removed the verification optimizations constraining the parallelism, annotated it with power consumption and collision counting (as cost variables). The model contains twice as many slots as nodes, whereas one slot per node is enough to schedule flawless communication in any topology if nodes were aware of each others choices.

First we examine the distribution of the first collisions over time. The first row of Fig. 13 is a result of a query `Pr[<=1000](◇ collision>0)` and it shows that most collisions happen early in time and in a ring topology some collisions may be discovered later (possibly when the first signal propagation meets at the opposite of the ring). In the second row of Fig. 13 the distribution of possible number of collisions is examined using a query `Pr[collisions<=100](◇ time>=1000)`: in a chain and a ring topologies the collisions are unlikely to occur ($> 90\%$ probability of 0 collisions), but in a star it is almost guaranteed to occur (only 8% probability of 0 collisions). The third row of Fig. 13 shows the probability distribution of collision counts after twice as long period of time (using query `Pr[collisions<=100](◇ time>=2000)`). Notice that the shape of distributions has not changed, but the small bumps have shifted to the right at exactly twice the number of collisions and almost identical probability density, which implies that those particular collisions are accumulating proportionally to the progress

[5] Thanks to Ansgar Fehnker and Angelika Mader.

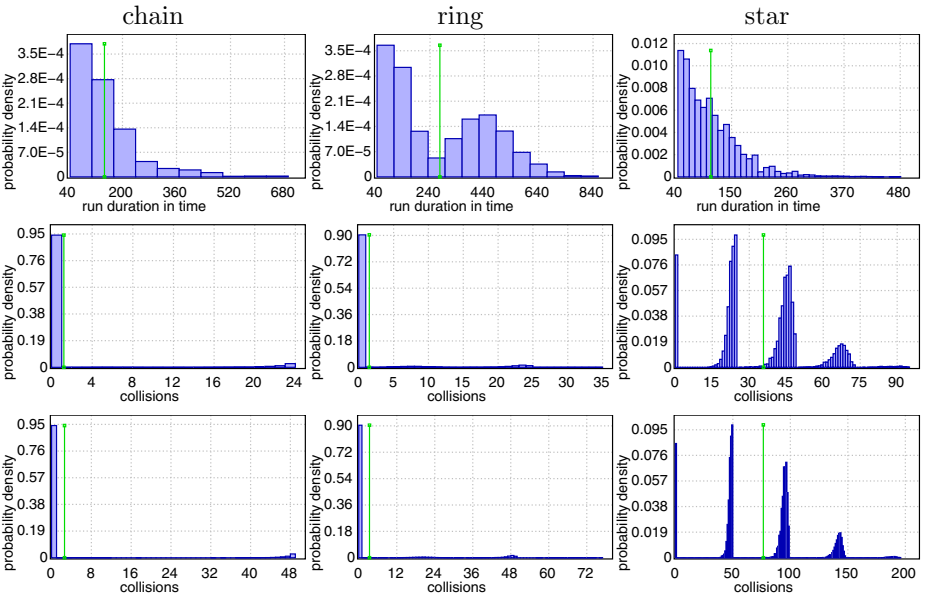


Fig. 13. Collision statistics in three different topologies, in rows: probability of a collision over time, probabilities of a number of collisions up to 1000 and up to 2000 time units

of time, and in other words it means that collisions are reoccurring perpetually without recovery. We checked these three properties on a 128 cores cluster with high precision (with $\alpha = \beta = 0.0001$ and $\varepsilon = 0.0005$) in about 30 minutes, which generated around 19 million runs.

We have demonstrated how Uppaal SMC can be used to identify problematic topologies and distributed implementation can provide a high degree of accuracy in spotting the reoccurring collisions.

6 Conclusion and Future Work

In this paper we have developed, implemented, applied and evaluated a general and scalable framework for distributed statistical model checking. We have thoroughly investigated the distribution of sequential algorithms where bias can be introduced when collecting the samples produced by slave computers. In particular, we have identified best choices of batch and buffer sizes both experimentally and analytically, with agreement in the findings of the two approaches. In the future, we plan to implement and distribute other SMC algorithms, principaly the Bayesian algorithms introduced in [20,9].

Finally, it is worth mentioning that we have tried to use other distributed SMC model checkers such as Ymer [18] or PVesta [14]. Aside from the fact that the Gui of those two tools is quite restricted, we observed that Ymer does not work anymore and that PVesta only distributes those algorithms where the number of simulations is precomputed in advance.

References

1. Bogdoll, J., Ferrer Fioriti, L.M., Hartmanns, A., Hermanns, H.: Partial Order Methods for Statistical Model Checking and Simulation. In: Bruni, R., Dingel, J. (eds.) FMOODS/ FORTE 2011. LNCS, vol. 6722, pp. 59–74. Springer, Heidelberg (2011)
2. Clarke, E.M., Faeder, J.R., Langmead, C.J., Harris, L.A., Jha, S.K., Legay, A.: Statistical Model Checking in *BioLab*: Applications to the Automated Analysis of T-Cell Receptor Signaling Pathway. In: Heiner, M., Uhrmacher, A.M. (eds.) CMSB 2008. LNCS (LNBI), vol. 5307, pp. 231–250. Springer, Heidelberg (2008)
3. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for Statistical Model Checking of Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
4. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical Model Checking for Networks of Priced Timed Automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer, Heidelberg (2011)
5. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007)
6. Grosu, R., Smolka, S.A.: Monte Carlo Model Checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
7. Hammersley, J.M., Handscomb, D.C.: Monte Carlo Methods. Methuen (1975)
8. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate Probabilistic Model Checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004)
9. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A Bayesian Approach to Model Checking Biological Systems. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009)
10. Legay, A., Delahaye, B.: Statistical model checking: An overview. CoRR, abs/1005.1327 (2010)
11. El Rabih, D., Pekergin, N.: Statistical Model Checking Using Perfect Simulation. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 120–134. Springer, Heidelberg (2009)
12. Sen, K., Viswanathan, M., Agha, G.: Statistical Model Checking of Black-Box Probabilistic Systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
13. Sen, K., Viswanathan, M., Agha, G.: On Statistical Model Checking of Stochastic Systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)
14. Sen, K., Viswanathan, M., Agha, G.A.: Vesta: A statistical model-checker and analyzer for probabilistic systems. In: QEST, pp. 251–252. IEEE Computer Society (2005)
15. Wald, A.: Sequential tests of statistical hypotheses. Annals of Mathematical Statistics 16(2), 117–186 (1945)
16. Wald, R.: Sequential Analysis. Dove Publisher (2004)
17. Younes, H.L.S.: Verification and Planning for Stochastic Processes with Asynchronous Events. PhD thesis, Carnegie Mellon (2005)
18. Younes, H.L.S.: Ymer: A Statistical Model Checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005)
19. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. STTT 8(3), 216–228 (2006)
20. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: HSCC, pp. 243–252. ACM (2010)