Accelerated Model Checking of Parametric Markov Chains

Accelerated Model Checking of Parametric Markov Chains

Paul Gainer, Ernst Moritz Hahn, and Sven Schewe

University of Liverpool, UK
{p.gainer,e.m.hahn,sven.schewe}@liverpool.ac.uk

Abstract. Parametric Markov chains occur quite naturally in various applications: they can be used for a conservative analysis of probabilistic systems (no matter how the parameter is chosen, the system works to specification); they can be used to find optimal settings for a parameter; they can be used to visualise the influence of system parameters; and they can be used to make it easy to adjust the analysis for the case that parameters change. Unfortunately, these advancements come at a cost: parametric model checking is—or rather was—often slow. To make the analysis of parametric Markov models scale, we need three ingredients: clever algorithms, the right data structure, and good engineering. Clever algorithms are often the main (or sole) selling point; and we face the trouble that this paper focuses on – the latter ingredients to efficient model checking. Consequently, our easiest claim to fame is in the speed-up we have often realised when comparing to the state of the art.

1 Introduction

The analysis of parametric Markov models is a young and growing field of research. As not only the research direction but also the term 'parametric Markov models' is attractive, it has been used for various generalisations of traditional Markov models. We use Markov chains, where the parameter is used to determine the probabilities and rewards, such that we can reason about the likelihood of obtaining simple temporal properties like safety and reachability as well as standard reward functions, such as long-run average.

What we do *not* intend to do in this paper is to use parameters to change the size of the system or the shape of the Markov chain. (The latter can, of course, be encoded by using parameters to assign a probability of 0 to an edge, effectively removing it. This would, however, come at the cost of efficiency and is not what we want to use the parameters for.)

Using parameters to describe the probabilities of transitions is not quite as easy as it sounds: even when parameters appear in a simple way, like 'p' or '$1-p$', the terms that represent the likelihood of obtaining a temporal property or an expected reward can quickly become quite intricate. One ends up with rational functions. We make a virtue of necessity by using this as a motivation to allow for using rational functions of the occurring parameters to represent the probabilities and payoffs.

To allow for an efficient analysis of such complex parametrised systems, we have taken a look at different strategies for the evaluations of—parametric and non-parametric—Markov chains, and considered their suitability for our purposes. We found the stepwise elimination of vertices from a model to be the most attractive approach to port.

Broadly speaking, this approach works like the transformation from finite automata to regular expressions: a vertex is removed, and the new structure has all successors of this state as—potentially new—successors of the predecessors of this vertex. In the transformation from finite automata to regular expressions, one changes the expressions on the edges, while we adjust the probabilities and, if applicable, the rewards on the edges.

When using this approach with explicit probabilities and rewards, one ends up with a Directed Acyclic Graph (DAG) structure in the evaluation. This DAG structure has been exploited to reduce the cost of re-calculating the probabilities for simple temporal properties or expected rewards, and it proves that it also integrates nicely into our framework, where the probabilities and rewards are provided as rational functions. In fact it integrates so naturally that it seems surprising in hindsight that it has not been discovered earlier.

The natural connection occurs when choosing a similar data structure to represent the rational functions that represent the probabilities and rewards. To make full use of the DAG structure that comes with the elimination, we represent these functions in the form of arithmetic circuits—which are essentially DAGs. We have integrated the resulting representation organically in a small extension of ePMC, and tested it on a range of case studies. We have obtained a speed-up of a hefty factor of 20 to 120 when compared to storing functions in terms of coprime numerator and denominator polynomials.

Related work. For (discrete-time) Markov chains (MCs), Daws [6] has devised a language-theoretic approach to solve this problem. In this approach, the transition probabilities are considered as letters of an alphabet. Thus, the model can be viewed as a finite automaton. Then, based on the state elimination method [22], a regular expression that describes the language of such an automaton is calculated. In a post-processing step, this regular expression is recursively evaluated, resulting in a rational function over the parameters of the model. One of the authors has been involved in extending and tuning this method [16] so as to operate with rational functions, which are stored as coprime numerator and denominator polynomials rather than with regular expressions.

The process of computing a function that describes properties (like reachability probabilities or long-run average rewards) that depend on model parameters is often costly. However, once the function has been obtained, it can very efficiently be evaluated for given parameter instantiations. Because of this, parametric model checking of Markov models has also attracted attention in the area of runtime verification, where the acceptable time to obtain values is limited [3,11].

Other works in the area are centred around deciding the validity of boolean formulas depending on the parameter range using SMT solvers or extending these techniques to models that involve nondeterminism [7,14,5,27].

As an example for a parametric model, consider Figure 1. Knuth and Yao [24] have shown how a six-sided dice can be simulated by repeatedly tossing a coin. The idea is to build a Markov chain with transition probabilities of only 0.5 or 1. Borrowing a model from the PRISM website, we have extended this example to a biased dice, simulated by tossing a biased coin. With probability x we see heads, while with probability $1 - x$ we see tails. This way, we move around in the Markov chain until we obtain a result.

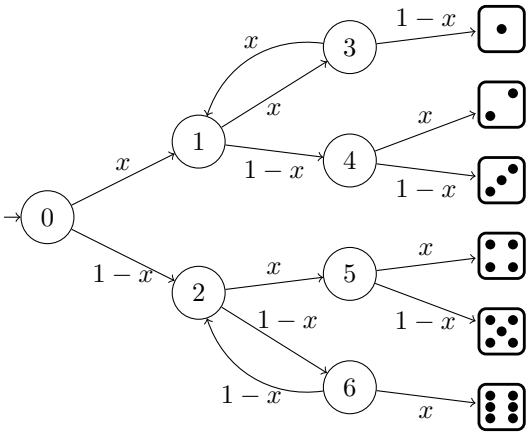

Fig. 1: Simulating a biased dice by a biased coin.

Organisation of the paper. After formalising our setting in Section 2, we describe how we exploit DAGs in the representation of rational functions, and exploit them using synergies with the DAG-style state elimination technique, in Section 3. We then describe how to expand this technique to determine long-run average rewards in Section 4. In Section 5, we evaluate our approach on a range of benchmarks, and discuss the results briefly in Section 6.

2 Preliminaries

2.1 Parametric Markov chains with state rewards

Let $V = \{v_1, \ldots, v_n\}$ denote a set of variables over $\mathbb{R}$. A *polynomial* g over V is a sum of monomials

$$g(v_1, \ldots, v_n) = \sum_{i_1, \ldots, i_n} a_{i_1, \ldots, i_n} \, v_1^{i_1} \ldots v_n^{i_n},$$

where each $i_j \in \mathbb{N}$ and each $a_{i_1, \ldots, i_n} \in \mathbb{R}$. A *rational function* f over a set of variables V is a fraction $f(v_1, \ldots, v_n) = \frac{f_1(v_1, \ldots, v_n)}{f_2(v_1, \ldots, v_n)}$ of two polynomials f_1, f_2 over V. We denote the set of rational functions from V to $\mathbb{R}$ by $\mathcal{F}_V$.

Definition 1. *A* parametric Markov chain *(PMC) is a tuple $\mathcal{D} = (\mathcal{S}, \overline{s}, \mathbf{P}, V)$, where $\mathcal{S}$ is a finite set of states, $\overline{s}$ is the initial state, $V = \{v_1, \ldots, v_n\}$ is a finite set of parameters, and $\mathbf{P}$ is the probability matrix $\mathbf{P} \colon \mathcal{S} \times \mathcal{S} \to \mathcal{F}_V$. A* path *$\omega$ of a PMC $\mathcal{D} = (\mathcal{S}, \overline{s}, \mathbf{P}, V)$ is a non-empty finite, or infinite, sequence $s_0, s_1, s_2, \ldots$ where $s_i \in \mathcal{S}$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for $i \geqslant 0$. We let Ω denote the set of infinite paths. With Pr_s, we denote the parametric probability measure over Ω assuming that we start in state s, with $\mathsf{Pr} = \mathsf{Pr}_{\overline{s}}$. We use Exp_s, Exp to*

Algorithm 1 Parametric Reachability Probability for PMCs

procedure StateElimination($\mathcal{D}, \mathcal{B}$)
  requires: A PMC $\mathcal{D} = (\mathcal{S}, \overline{s}, \mathbf{P}, V)$ and set of target states $\mathcal{B} \subseteq \mathcal{S}$, where $\mathsf{reach}_D(\overline{s}, s)$ holds for all $s \in \mathcal{S}$.
  $E \leftarrow \mathcal{S}$
  while $E \neq \emptyset$ **do**
    $s_e \leftarrow \mathsf{choose}(E)$
    $E \leftarrow E \setminus \{s_e\}$
    for all $s \in \mathsf{post}_{\mathcal{D}}(s_e)$ **do**
      $\mathbf{P}(s_e, s) \leftarrow \mathbf{P}(s_e, s)/(1 - \mathbf{P}(s_e, s_e))$
    end for
    $\mathbf{P}(s_e, s_e) \leftarrow 0$
    for all $(s_1, s_2) \in \mathsf{pre}_{\mathcal{D}}(s_e) \times \mathsf{post}_{\mathcal{D}}(s_e)$ **do**
      $\mathbf{P}(s_1, s_2) \leftarrow \mathbf{P}(s_1, s_2) + \mathbf{P}(s_1, s_e)\mathbf{P}(s_e, s_2)$
    end for
    if $s_e \neq \overline{s} \wedge s_e \notin \mathcal{B} \wedge \mathsf{post}_{\mathcal{D}}(s_e) \neq \emptyset$ **then**
      $\mathsf{Eliminate}(\mathcal{D}, s_e)$ *// remove s_e and incident transitions from $\mathcal{D}$*
    end if
  end while
  return $\sum_{s \in \mathcal{B}} \mathbf{P}(\overline{s}, s)$
end procedure

denote according expectations. With $X(\mathcal{D})_{s,i} \colon \Omega \to \mathcal{S}$, $X(\mathcal{D})_{s,i}(s_0, s_1, \ldots) = s_i$ we denote the random variable expressing the state occupied at step $i \geq 0$, and let $X(\mathcal{D})_i = X(\mathcal{D})_{\overline{s},i}$.

Definition 2. *Given a PMC $\mathcal{D} = (\mathcal{S}, \overline{s}, \mathbf{P}, V)$, the* underlying graph *of $\mathcal{D}$ is given by $\mathcal{G}_{\mathcal{D}} = (\mathcal{S}, E)$ where $E = \{(s, s') \mid \mathbf{P}(s, s') > 0\}$. A* bottom strongly connected component (BSCC) *is a set $A \subseteq \mathcal{S}$ such that in the underlying graph each state $s_1 \in A$ can reach each state $s_2 \in A$ and there is no $s_3 \in \mathcal{S} \setminus A$ reachable from s_1.*

Given a state s, we denote the set of all immediate predecessors and successors of s in the underlying graph of $\mathcal{D}$ by $\mathsf{pre}_{\mathcal{D}}(s)$ and $\mathsf{post}_{\mathcal{D}}(s)$, respectively, excluding s itself. We write $\mathsf{reach}_D(s, s')$ if s' is reachable from s in the underlying graph of D.

Given a PMC $\mathcal{D} = (\mathcal{S}, \overline{s}, \mathbf{P}, V)$ we are interested in computing the function that represents the probability of reaching some set of target states $\mathcal{B} \subset \mathcal{S}$.

$$\mathsf{Reach}(\mathcal{D}, \mathcal{B}) = \mathsf{Pr}\left[\exists i \geq 0. X(\mathcal{D})_{\overline{s},i} \in \mathcal{B}\right]$$

Our base algorithm to obtain this value is described in Algorithm 1. A state $s_e \in \mathcal{S}$ is selected, and then eliminated by considering each pair $(s_1, s_2) \in \mathsf{pre}_{\mathcal{D}}(s_e) \times \mathsf{post}_{\mathcal{D}}(s_e)$ and updating the existing probability $\mathbf{P}(s_1, s_2)$ by the probability of reaching s_2 from s_1 via s_e. Heuristics to determine the order in which states are chosen for elimination by the choose function are discussed in Section 5.5.

Definition 3. *A* parametric reward function *for a PMC* $\mathcal{D} = (\mathcal{S}, \overline{s}, \mathbf{P}, V)$ *is a function* $r \colon \mathcal{S} \to \mathcal{F}_V$.

The reward function labels states in $\mathcal{D}$ with a rational function over V that corresponds to the reward that is gained if that state is visited. Given a PMC $\mathcal{D} = (\mathcal{S}, \overline{s}, \mathbf{P}, V)$ and a reward function $r \colon \mathcal{S} \to \mathcal{F}_V$, we are interested in the *parametric expected accumulated reward* defined as

$$\mathsf{Acc}(\mathcal{D}, r) = \mathsf{Exp}\left[\sum_{i=0}^{\infty} r(X(\mathcal{D}))_{\overline{s},i}\right]$$

or a variation [25], the *parametric expected accumulated reachability reward* given $\mathcal{B} \subseteq \mathcal{S}$ defined as

$$\mathsf{Acc}(\mathcal{D}, r, \mathcal{B}) = \mathsf{Exp}\left[\sum_{i=0}^{\{j \mid X(\mathcal{D})_{\overline{s},j} \in \mathcal{B}\}} r(X(\mathcal{D}))_{\overline{s},i}\right].$$

This can, however, be transformed to the former.

Algorithm 1 can be extended to compute the parametric expected accumulated reward. In addition to updating the probability matrix for each predecessor and successor pair, we also update the reward function as follows:

$$r(s_1) \leftarrow r(s_1) + \mathbf{P}(s_1, s_e)\frac{\mathbf{P}(s_e, s_e)}{1 - \mathbf{P}(s_e, s_e)} r(s_e).$$

The updated value for $r(s_1)$ reflects the reward that would be accumulated if a transition would be taken from s_1 to s_e, where the expected number of self-loops would be $\frac{\mathbf{P}(s_e,s_e)}{1-\mathbf{P}(s_e,s_e)}$. Upon termination, the algorithm returns the value $r(\overline{s})$.

3 Representing Formulas using Directed Acyclic Graphs

In existing tools for parametric model checking of Markov models, rational functions have traditionally been represented in the form $f(v_1, \ldots, v_n) = \frac{f_1(v_1,\ldots,v_n)}{f_2(v_1,\ldots,v_n)}$, where $f_1(v_1, \ldots, v_n)$ and $f_2(v_1, \ldots, v_n)$ [15,8,26] are coprime. As a result, for some cases the representations of such functions are very short. Often, during the state elimination phase, large common factors can be cancelled out, such that one can operate with relatively small functions throughout the whole algorithm. There are, however, many cases without—or with very few—large common factors. The nominator-denominator representations then become larger and larger during the analysis. In this case, the analysis is slowed down severely, mostly by the time taken for the cancellation of common factors. Cancelling out such factors is non-trivial, and indeed a research area in itself. In addition, if formulas become large, this can also lead to out-of-memory problems.

To overcome this issue, we propose the representation of rational functions by arithmetic circuits. These arithmetic circuits are directed acyclic graphs (DAG).

Terminal nodes are labelled with either a variable of the set of parameters V, or with a rational number. Non-final nodes are labelled with a function to be applied on the nodes it has edges to. In our setting, we require two unary functions, additive inverse and multiplicative inverse, and two binary functions, addition and multiplication. All functions used are represented using a single DAG, and a function is represented by a reference to a node of this common DAG.

This representation has two advantages. Firstly, all operations are practically constant time: to apply an operator on two functions, one simply introduces a new node labelled with the according operator, with edges pointing to the two nodes to connect. In particular, we do not have to use expensive methods to cancel out common factors. Secondly, because we are using a DAG and not a tree, common sub-expressions can be shared between different formulas, which is not possible when representing rational functions in terms of two polynomials represented as a list of monomials.

For illustration, let us consider the example from Figure 1. We analyse the probability that the final result is ⚅. This probability can be described by the function

$$\frac{-x^2 + 2x - 1}{x - 2}.$$

In our DAG-based representation, we would represent the function as in Figure 2;

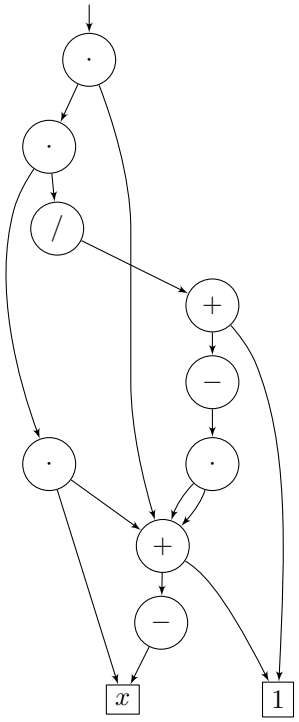

Fig. 2: Probability of rolling ⚅.

When operating with arithmetic circuits, there are a number of ways to reduce their memory footprint, which will, however, lead to a higher running time. The simplest one is that, while creating a new node to represent a function, it might turn out that there already exists a node with exactly the same operator, and exactly the same operand. In this case, it is better to drop the newly created node and use a reference to the existing node to counter the growth of the DAG. In case we use hash maps for the lookup, we can also still keep the overhead close to constant time. Another optimisation is to use simple algebraic equivalences. This includes computing the values of constant functions. E.g. instead of creating a node representing $2 + 3$ we introduce a new terminal node labelled 5, and if we are about the create a new node for $y + x$ but we already have a node for $x + y$ we reuse this node instead. We also take the additive and multiplicate neutral elements into account (rather than creating a new node for $0 + x$, we return the one for x, and the like). Another optimisation method is to evaluate functions of the DAG at random points and then to

identify functions if the result of this evaluation is the same. Using the Schwartz-Zippel Lemma [29,30,9], we can then bound the probability that we mistakenly identify two nodes although they do not represent the same function. We can again minimise the overheads incurred by this method by using hash maps.

Arithmetic circuits sometimes become very large, consisting of millions of nodes. This way, they cannot serve as a concise, human-readable description of the analysis result. Compared to performing a non-parametric analysis, it is however often still beneficial to obtain a function representation in this form. Even for large dags, obtaining evaluating parameter instantiations is very fast, and linear in the number of DAG nodes. This is useful in particular if a large number of points is required, for instance for plotting a graph. In this case, results can be obtained much faster than using non-parametric model checking, as demonstrated in Section 5. In particular, for any instantiation, values can be obtained in the same, predictable, time. This is quite in contrast to value iteration, where the number of iterations required to obtain a certain precision varies with the concrete values of parameters.

For this reason, parametric model checking is particularly useful for online model checking or runtime verification [3]. Here, one can precompute the DAG before running the actual system, while concrete values can be instantiated at runtime, with a running time that can be precisely calculated offline. Using arithmetic circuits expands the range of systems for which this method is applicable. Evaluation of parameter instantiations can be performed using exact arithmetic or floating-point arithmetic. From our experience, the quality of the floating-point results using DAGs is often better than the one using the representation of rational functions as coprime numerator and denominator, which has been used so far in known implementations. The reason is that, in the latter approach, one often runs into numerical problems such as cancellation, which often forces the use of expensive exact arithmetic to be used for evaluation. The DAG-based method seems to be more robust against such problems.

It has recently come noted by the verification community that the usual way in which value iteration is implemented is not safe, and solutions have already been proposed [1]. While this solves the problem, it requires more complex algorithms and leads to increased model checking time. In case arithmetic circuits are used, it is easy to obtain conservative upper and lower bounds for parameter instantiations. One only has to use interval arithmetic and provide implementations for the basic operations used (addition, multiplication, additive and multiplicative inverse). The increase in the time to evaluate functions is small. In our experiments, the largest interval diameter we have obtained is around 10^{-13}.

4 Computation of Fractional Long-Run Average Values

Consider a PMC $\mathcal{D} = (\mathcal{S}, \overline{s}, \mathbf{P}, V)$ together with two reward functions $r_u \colon \mathcal{S} \to \mathcal{F}_V$ and $r_l \colon \mathcal{S} \to \mathcal{F}_V$. The problem we are interested in is computing the value

fractional long-run average reward [2,10]

$$\mathsf{LRA}(\mathcal{D}, r_u, r_l, s)=\mathsf{Exp}\left[\lim_{n\to\infty} \frac{\sum_{i=0}^{n} r_u(X(\mathcal{D})_{s,i})}{\sum_{i=0}^{n} r_l(X(\mathcal{D})_{s,i})}\right], \mathsf{LRA}(\mathcal{D}, r_u, r_l)=\mathsf{LRA}(\mathcal{D}, r_u, r_l, \overline{s}).$$

In a simple case, $r_l(\cdot) = 1$, which means that we compute the *long-run average reward*

$$\mathsf{Exp}\left[\lim_{n\to\infty} \frac{1}{n+1} \sum_{i=0}^{n} r_u(X(\mathcal{D})_{s,i})\right],$$

where each step is assumed to take the same amount of time. Solution methods for this property has been implemented (but to the best of our confidence, not been published) for parametric models in PRISM and Storm. The fractional long-run average reward is more general and allows to express values like the average energy usage per task performed more easily. Given a reward structure $r\colon \mathcal{S} \to \mathcal{F}_V$, we define the *recurrence reward* as

$$\mathsf{Return}(\mathcal{D}, r, s) = \mathsf{Exp}\left[\sum_{i=0}^{\min\{j \mid X(\mathcal{D})_{s,j}=s \wedge j>0\}} r(X(\mathcal{D})_{s,i})\right], \mathsf{Return}(\mathcal{D}, r) = \mathsf{Return}(\mathcal{D}, r, \overline{s}).$$

It is known [4] that this value is the same for all states of a BSCC. Furthermore, for $r_l(\cdot) = 1$ we have

$$\frac{\mathsf{Return}(\mathcal{D}, r_u, s)}{\mathsf{Return}(\mathcal{D}, r_l, s)} = \mathsf{LRA}(\mathcal{D}, r_u, r_l, s),$$

which immediately extends to the general case.

In Section 2, we have discussed how state elimination can be used to obtain values for the expected accumulated reward values. For this, we have repeatedly eliminated states so as to bring the PMC of interest into a form in which reward values can be obtained in a trivial way. It is easy to see that the transformations for the expected accumulated rewards also maintains the recurrence rewards. After having handled each state of our model, we have two possible outcomes. In the simpler case, the remaining model consists of the initial state $\overline{s}$ with a self-loop with probability one and $r_u = u_{\overline{s}}$, $r_l = l_{\overline{s}}$. In this case, we have $\mathsf{LRA}(\mathcal{D}, r_u, r_l) = \frac{u_{\overline{s}}}{l_{\overline{s}}}$. In the other case, the remaining model consists of the initial state $\overline{s}$ which has a probability of p_i to move to one of the other n remaining states s_i , $i = 1, \ldots, n$, which all have a self-loop with probability one and $r_u(s_i) = u_i$, $r_l(s_i) = l_i$. In this case, we have $\mathsf{LRA}(\mathcal{D}, r_u, r_l) = \sum_{i=1,\ldots,n} p_i \frac{u_i}{l_i}$.

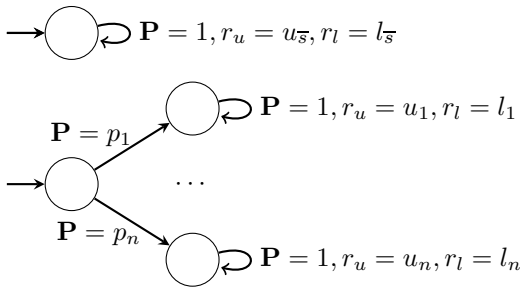

Fig. 3: Computation of long-run average values.

5 Experiments

We now consider four case studies that illustrate the efficiency and scalability of our approach. Three models [21,23,28] are taken from the PRISM benchmark suite[1], and the last is taken from the authors' work on synchronisation protocols [12,13]. All experiments were conducted on a PC with an Intel Core i7-2600 (tm) processor at 3.4GHz, equipped with 16GB of RAM, and running Ubuntu 16.04. For each case study we compare the performance times obtained for model analysis when using the parametric engine of the model checker ePMC [17][2], using either polynomial fractions or DAGs to represent the functions corresponding to transition probabilities and state rewards. Basically, the DAG is implemented as an array of 64-bit integers. Functions are represented as indices to this array. 4 bits describe the type of the node. For terminal nodes, the remaining bits denote the parameter or number used. For non-terminal nodes, 2×30 bits are used to refer to the operands within the DAG. We also compare our results to those obtained using the parametric engine of PRISM [26], and the parametric and sampling engines of Storm [8][3].

Given a parametric model, and a set of valuations for its parameters, we are interested in the total time taken to check some property of interest for every valuation for the parameters. Since our primary concern is the efficiency of multiple evaluations of an existing model, we omit model construction times and restrict our analysis to the total time taken for the evaluation of all parameter valuations. For the parametric engines of ePMC, PRISM, and Storm, we record the total time taken for both state elimination and the evaluation of the resulting function for all parameter valuations. For the sampling engine of Storm, we record the total time taken for value iteration, using default settings to determine convergence. For Storm, we set the precision to 10^{-10} rather than the default of 10^{-6}. This had a very minor influence on the runtime, and allowed a better comparison to ePMC, the results of which have a precision of $< 10^{-13}$.

5.1 Crowds Protocol

The Crowds protocol [28] provides anonymity for a crowd consisting of N Internet users, of whom M are dishonest, by hiding their communication via random routing, where there are R different path reformulates. The model is a PMC parametrised by $B = \frac{M}{M+N}$, the probability that a member of the crowd is untrustworthy, and P, the probability that a member sends a package to a randomly selected receiver. With probability $1-P$ the packet is directly delivered to the receiver. The property of interest is the probability that the untrustworthy members observe the sender more than they observe others.

Table 1 shows the performance statistics for different values of N and R, where each entry shows the total time taken to check all pairwise combinations of

[1] `http://www.prismmodelchecker.org/benchmarks/`

[2] `http://iscasmc.ios.ac.cn/?p=1241`, `https://github.com/liyi-david/ePMC`

[3] `http://www.stormchecker.org/`

N	R	States	Trans.	PRISM	ePMC	ePMC(D)	ePMC(DS)	Storm(P)	Storm(S)
5	3	1198	2038	722	737	13	13	681	26
5	5	8653	14953	745	806	15	15	723	64
5	7	37291	65011	818	900	19	17	735	153
10	3	6563	15143	732	771	15	14	690	26
10	5	111294	261444	1146	910	23	16	712	63
10	7	990601	2351961	–T–	–T–	103	42	737	159
15	3	19228	55948	761	825	16	16	703	26
15	5	592060	1754860	–T–	–M–	42	28	709	64
15	7	8968096	26875216	–M–	–M–	–M–	–M–	777	174
20	3	42318	148578	814	805	15	14	709	26
20	5	2061951	7374951	–M–	–M–	108	90	720	67

Table 1: Performance statistics for crowds protocol.

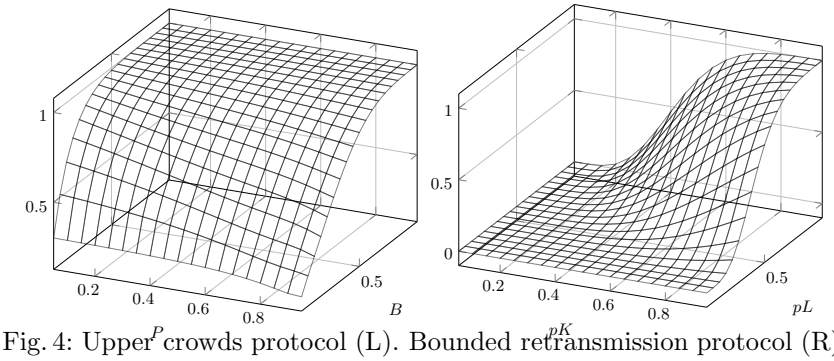

Fig. 4: Upper crowds protocol (L). Bounded retransmission protocol (R).

values for B, P taken from $0.002, 0.004, \ldots, 0.998$. There is a substantial increase in the performance of ePMC when using non-simplified DAGs (ePMC(D)), and using DAGs (ePMC(DS)) simplified by evaluating random points (cf. Section 3), instead of polynomial fractions (ePMC) to represent functions. Here, ePMC clearly outperforms the parametric engines of both PRISM and Storm. In some instances, ePMC turns out to be the fastest choice, while the sampling engine of Storm proves to be faster for other instances. Processes that exceeded the time limit of one hour are indicated by –T–, and processes that ran out of memory are indicated by –M–. In Fig. 4 (left) we plot the results for $N = 5$ and $R = 7$.

5.2 Bounded Retransmission Protocol

The bounded retransmission protocol [21] divides a file, which is to be transmitted, into N chunks. For each chunk, there are at most MAX retransmissions over two lossy channels K and L that send data and acknowledgements, respectively.

N	MAX	States	Trans.	PRISM	ePMC	ePMC(D)	ePMC(DS)	Storm(P)	Storm(S)
64	4	4139	5543	1029	1016	36	38	991	160
64	5	4972	6695	1145	1118	36	33	1021	188
256	4	16427	22055	–M–	–M–	48	40	3332	403
256	5	19756	26663	–M–	–M–	35	15	–T–	318
512	4	32811	44071	–M–	–M–	29	19	–T–	491
512	5	39468	53287	–M–	–M–	28	23	–T–	596

Table 2: Performance statistics for bounded retransmission protocol.

The model is a PMC parametrised by pK and pL, the reliability of the channels. We are interested in the probability that the sender reports an unsuccessful transmission after more than 8 chunks have been sent successfully.

The performance statistics for different values of N and MAX are shown in Table 2, where each entry shows the total time taken to check all pairwise combinations of values for pK, pL taken from $0.002, 0.004, \ldots, 0.998$. Here, ePMC with DAGs again has the best performance: the running time remains approximately constant when using this data structure, even for much larger problem instances. In contrast, the running time for both engines of Storm scale linearly. Both the parametric engines of PRISM and ePMC, with polynomial fraction representation, run out of memory for all larger problem instances.

Figure 4 (right) plots the results obtained for $N = 256$ and $MAX = 4$. As we see the probability of interest first increases with increasing channel reliability, but then decreases again. The reason is that, on the one hand, if the channel reliability is low, then we do not send many chunks successfully. On the other hand, if the channel reliability is high, then it is unlikely that the transmission will fail in the end.

5.3 Cyclic Polling Server

This cyclic server polling model [23] is a model of a network, described as a continuous-time Markov chain. There are two parameters, μ and γ. The model consists of one server and N clients. When a client is idle, then a new job arrives at this client with a rate of μ/N. The server 'polls' the clients in a cyclic manner. At each point of time, it observes a single client. If there is a job waiting for a given client, the server servers its job (provided there is one) with a rate of μ. When the client it observes is idle, then the server moves on to observe the next client with a rate of γ. Even though our method targets discrete-time models, we can handle this model by computing the embedded DTMC.

In this case study, we consider the probability that, in the long run, Station 1 is idle. That is, the expected limit average of the time that Station 1, or, due to symmetry, any other station, is idle. We compute this long-run average value using the method described in Section 4. Probabilities are displayed as a

N	States	Trans.	PRISM	ePMC	ePMC(D)	ePMC(DS)	Storm(P)	Storm(S)
4	96	272	1166	888	14	14	953	50
5	240	800	–T–	–T–	28	25	–T–	121
6	576	2208	3550	–T–	108	102	–T–	305
7	1344	5824	1399	–T–	759	736	–T–	801
8	3072	14848	1052	–T–	–T–	–T–	–T–	1991
9	6912	36864	–T–	–T–	–M–	–M–	–T–	–T–

Table 3: Performance statistics for cyclic polling server.

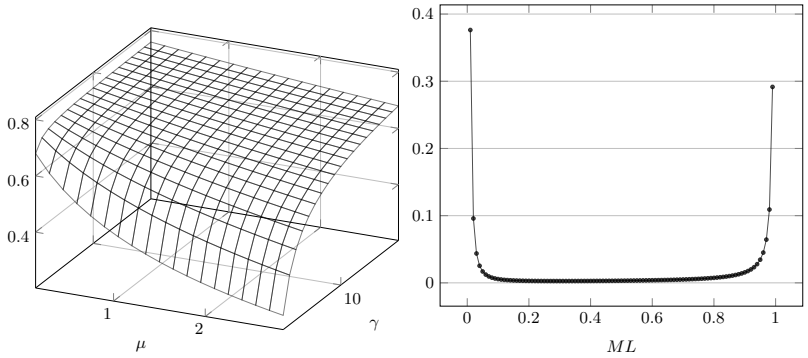

Fig. 5: Cyclic polling server (L). Synchronisation model (R).

function of the parameters in Figure 5, and Table 3 shows how the various tools perform on this benchmark. With increasing γ the likelihood that Station 1 is idle increases: if we increase γ, then the server will more quickly find stations to be served. As the long-run average idle time only depends on the rate between μ and γ, the likelihood that Station 1 is idle falls with increasing μ.

For the current configuration, classic parametric model checking does not seem to be advantageous. Using our DAG-based implementation, however, is much more efficient than classic parametric model checking, but it is space consuming. With the chosen number of parameter instantiations, our method does not quite compete with non-parametric model checking.

5.4 Oscillator Synchronisation

The models of [12,13] encode the behaviour of a population of N coupled nodes in a network. Each node has a clock that progresses, cyclically, through a range of discrete values $1, \ldots, T$. At the end of each clock cycle a node transmits a message to other nodes in the network. Nodes that receive this message adjust their clocks to more closely match those of the firing node. The model is a PMC, parametrised by the likelihood ML that a firing message is lost in the communication medium.

N	T	States	Trans.	PRISM	ePMC	ePMC(D)	ePMC(DS)	Storm(P)	Storm(S)
4	6	218	508	280	304	6	4	6	20
4	7	351	822	302	399	7	4	6	28
4	8	535	1257	542	1520	9	5	13	37
5	6	449	1179	354	499	10	5	11	39
5	7	799	2094	1694	–T–	11	6	34	60
5	8	1333	3533	–T–	–T–	17	8	137	90
6	6	841	2491	1070	–T–	12	7	48	74
6	7	1639	4820	–T–	–T–	19	8	239	130
6	8	2971	8871	–T–	–T–	33	10	2311	211

Table 4: Performance statistics for synchronisation model.

The property of interest is the expected power consumption of the network (in Watt-hours) to reach a state, where the clocks of all nodes are synchronised.

Table 4 shows the results for different values of N and T, where each entry shows the total time taken to check all values of ML taken from $10^{-5}, 2 \cdot 10^{-5}, \ldots, 1 - 10^{-5}$.

Figure 5 (right) plots the results obtained for $N = 6$ and $T = 8$. For extremal values of ML, the network is expected to use much more energy to synchronise, because the expected time required for this to occur increases. Very high values of ML result in nearly all firing messages being lost, and hence nodes cannot communicate well enough to coordinate, while very low values of ML lead to perpetually asynchronous states for the network, an artefact of the discreteness of the clock values [12].

In this case study, the DAG-based method, in particular with random points evaluation, performs best, followed by the sampling-based method of Storm. We note that the time required for each value iteration is relatively high, while the cost of evaluating a point for the DAG-based method is quite low. Therefore, the advantage of our approach would have been even more pronounced, if we had evaluated more instantiations in the experiments above. The method of choice thus depends mostly on whether such a high number of instantiations is required.

We have performed value iteration with a (local) precision of 10^{-10} for Storm. This does, however, not guarantee any global precision [1]. Obtaining guaranteed results using value iteration is relatively expensive while, as discussed in Section 3, extending our approach to obtain conservative guarantees is relatively simple—and inexpensive—to achieve by using basic interval arithmetic.

5.5 Heuristics

An important consideration when performing state elimination is the order, in which different states are eliminated from the graph. Using different elimination orders to evaluate the same model can result in functions, whose representations

Model		Elimination Heuristic					
		NumNew	MinProd	TargetBFS	Random	BFS	ReverseBFS
Crowds	(N=10,R=5)	19	103	5	14	22	6
BRP	(N=512,MAX=5)	4	11	4	–M–	5	4
Cyclic	(N=7)	7	9	8	8	8	8
Synch	(N=6,T=8)	18	18	17	19	18	17

Table 5: Performance statistics for different heuristics.

(nominator-denominator or DAGs) vary greatly in size, and hence also in the corresponding memory footprint and analysis time. Heuristics for efficient state elimination have been studied in automata theory, to obtain shorter regular expressions from finite-state automata [19,18], and in graph theory, for efficient peeling of a probabilistic network [20]. We employ the following heuristics, consisting of both existing schemes taken from the literature, and novel schemes that prove to be effective for some models.

- NumNew: each state is weighted by the number of new transitions that are introduced to the model when that state is eliminated. That is, we consider each predecessor-successor pair for that state, and add one to the weight if the transition from the predecessor to the successor was not already defined in the underlying graph before state elimination. States with the lowest weight are eliminated first. The aim here is to minimise the total number of transitions as elimination progresses.
- MinProd: similarly to NumNew, we consider each predecessor-successor pair. However, one is added to the weight irrespective of whether that transition already existed in the underlying graph. Again states with the lowest weight are considered first.
- TargetBFS: states are eliminated in the order in which they are discovered when conducting a breadth-first search backwards from the target states.
- Random: a state is selected uniformly at random for elimination from the set of remaining states.
- BFS: states are eliminated in the order in which they are discovered when conducting a breadth-first search from the initial state(s) of the model.
- ReverseBFS: similar to BFS, except states are eliminated in reverse order.

In Table 5, we compare the different heuristics described. We have applied each of them for each considered model, and provide the time in seconds required for medium-sized instances. As seen, it turns out that TargetBFS is in general a good choice. In one case, however, NumNew turns out to be faster.

6 Conclusion and Future Work

We have implemented an approach for the evaluation of parametric Markov chains that exploits the synergies of using DAGs in a state-elimination based

analysis and using DAGs in an encoding of rational functions as arithmetic circuits. Our experimental evaluation suggests that these two approaches integrated so seamlessly that they often provide a notable speedup. The nicest observation is that this seems so natural in hindsight that it is almost more surprising that this has not been attempted before than that it works so well. We therefore hope to have discovered one of these simple and natural approaches that will stand the test of time.

The next step in exploiting our approach could be an integration into applications. One of the applications we have in mind is to use it in the context of parameter extraction, which we expect to work similar to Model extraction, for online Model checking. The growing knowledge of the model can be used to refine or adjust the parameters in this application. Our application can help to provide the speed required to make the approach scale, and to keep the analysis and, if required, the visualisation[4] of the effect of the learnt parameters (and the confidence area around them) efficient.

We also note that interval arithmetic could be used to evaluate *boxes*—hyperrectangles $[a_1, b_1] \times \cdots [a_n, b_n]$ of parameter ranges—so as to obtain bounds on the lower and upper values taken by any occurring function value in the box. This approach could be used instead of using SMT solvers (as in [7,14]) to decide PCTL properties. A similar approach to avoid using SMT solvers has been proposed [27], which is however not based on computing a function depending on the parameters but on value iteration. We assume that the DAG-based approach will perform better when a high coverage of the parameter space is required.

It would also be straightforward to parallelise evaluation of points using SIMD approaches such as GPGPU.

7 Acknowledgements

This work was supported by the Sir Joseph Rotblat Alumni Scholarship at Liverpool, EPSRC grants EP/M027287/1 and EP/N007565/1, and by the Marie Skłodowska Curie Fellowship *Parametrised Verification and Control*.

References

1. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: CAV. LNCS, vol. 10426, pp. 160–180. Springer (2017)
2. Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Hofferek, G., Jobstmann, B., Könighofer, B., Könighofer, R.: Synthesizing robust systems. Acta Inf. 51(3-4), 193–220 (2014)
3. Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. CACM 55(9), 69–77 (2012)
4. Cox, D.R.: Renewal Theory. Methuen & Co. Ltd., London (67)
5. Cubuktepe, M., Jansen, N., Junges, S., Katoen, J., Papusha, I., Poonawala, H.A., Topcu, U.: Sequential convex programming for the efficient verification of parametric mdps. In: TACAS. pp. 133–150 (2017)

[4] We obtain the probablities and expected rewards as functions of the parameters. This can be visualised, just as we have visualised this in Section 5.

6. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: ICTAC. pp. 280–294. Springer (2004)
7. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Katoen, J., Ábrahám, E., Bruintjes, H.: Parameter synthesis for probabilistic systems. In: MBMV. pp. 72–74 (2016)
8. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is coming: A modern probabilistic model checker. In: CAV. pp. 592–600. Springer (2017)
9. DeMillo, R.A., Lipton, R.J.: A probabilistic remark on algebraic program testing. Inf. Process. Lett. 7, 193–195 (1978)
10. von Essen, C., Jobstmann, B.: Synthesizing systems with optimal average-case behavior for ratio objectives. In: iWIGP. pp. 17–32 (2011)
11. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: ICSE. pp. 341–350. ACM (2011)
12. Gainer, P., Linker, S., Dixon, C., Hustadt, U., Fisher, M.: Investigating parametric influence on discrete synchronisation protocols using quantitative model checking. In: QEST. pp. 224–239. Springer (2017)
13. Gainer, P., Linker, S., Dixon, C., Hustadt, U., Fisher, M.: The power of synchronisation: Formal analysis of power consumption in networks of pulse-coupled oscillators. arXiv preprint arXiv:1709.04385 (2017)
14. Hahn, E.M., Han, T., Zhang, L.: Synthesis for PCTL in parametric Markov decision processes. In: NFM. pp. 146–161 (2011)
15. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Param: A model checker for parametric Markov models. In: CAV. pp. 660–664. Springer (2010)
16. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. STTT 13(1), 3–19 (2011)
17. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: IscasMc: a web-based probabilistic model checker. In: FM. pp. 312–317. Springer (2014)
18. Han, Y.S.: State elimination heuristics for short regular expressions. FI 128(4), 445–462 (2013)
19. Han, Y.S., Wood, D.: Obtaining shorter regular expressions from finite-state automata. TCS 370(1-3), 110–120 (2007)
20. Harbron, C.: Heuristic algorithms for finding inexpensive elimination schemes. SC 5(4), 275–287 (1995)
21. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: TYPES. pp. 127–165. Springer (1993)
22. Hopcroft, J.E.: Introduction to automata theory, languages, and computation. Pearson Education India (2008)
23. Ibe, O.C., Trivedi, K.S.: Stochastic Petri net models of polling systems. J-SAC 8(9), 1649–1657 (1990)
24. Knuth, D., Yao, A.: Algorithms and Complexity: New Directions and Recent Results, chap. The complexity of nonuniform random number generation. Academic Press (1976)
25. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: SFM. pp. 220–270. Springer (2007)
26. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. pp. 585–591. Springer (2011)
27. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.: Parameter synthesis for Markov models: Faster than ever. In: ATVA. pp. 50–67 (2016)
28. Reiter, M.K., Rubin, A.D.: Crowds: Anonymity for web transactions. TISSEC 1(1), 66–92 (1998)
29. Schwartz, J.T.: Fast probabilistic algorithms for verification of polynomial identities. J. ACM 27(4), 701–717 (Oct 1980)
30. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: ISSAC. pp. 216–226. EUROSAM '79, Springer-Verlag, London, UK, UK (1979)