An Experimental Evaluation of Probabilistic Simulation *

Jonathan Bogdoll, Holger Hermanns and Lijun Zhang

Department of Computer Science, Saarland University, Saarbrücken, Germany
{bogdoll,hermanns,zhang}@cs.uni-sb.de

Abstract. *Probabilistic* model checking has emerged as a versatile system verification approach, but is frequently facing state-space explosion problems. One promising attack to this is to construct an abstract model which *simulates* the original model, and to perform model checking on that abstract model. Recently, efficient algorithms for deciding simulation of probabilistic models have been proposed. They reduce the theoretical complexity bounds drastically by exploiting parametric maximum flow algorithms. In this paper, we report on experimental comparisons of these algorithms, together with various interesting optimizations. The evaluation is carried out on both standard PRISM example cases as well as randomly generated models. The results show interesting time-space tradeoffs, with the parametric maximum flow algorithms being superior for large, dense models.

1 Introduction

System performance and dependability becomes more and more important with the ubiquity of computing systems. Discrete-time and continuous-time Markov chains (DTMCs and CTMCs) [18] are widely used to model and analyze performance and dependability of such systems. A related model, which in addition supports nondeterminism, is the model of probabilistic automata (PAs) [17]. For all these three models, tool support is available, in the form of probabilistic model checkers such as Prism [12] or Mrmc [15]. They enable the automatic verification of performance and dependability models for specifications expressed by PCTL [11, 6] or CSL [1, 3] formulas. PCTL is a discrete probabilistic variant of the temporal logic CTL interpreted over DTMCs and PAs, and CSL is its continuous stochastic extension, tailored to CTMCs.

Despite the remarkable versatility of this approach, its power is limited by the infamous state space explosion problem. Several approaches are being pursued to alleviate that problem. Notably, minimizing the system to the bisimulation quotient is a favorable approach [14]. As a more aggressive attack to the problem, simulation relations [13, 4, 5] have been proposed for these models, which, in correspondence to the non-probabilistic setting, preserve relevant fragments

* This work is supported by the NWO-DFG bilateral project VOSS and by the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS.

of the logics PCTL and CSL, respectively. In particular, they provide the principal ingredients to perform abstractions of the models, while preserving *safe* fragments of the respective logics [5, 17].

The kernel of simulation, simulation equivalence, preserves both safe and live fragments of PCTL. Since simulation equivalence is coarser than bisimulation, the induced quotient is thus smaller. This means that as long as one is interested in safety or liveness properties, it is favorable to perform model checking on the simulation equivalence quotient. To strive for the quotient, an algorithm for deciding simulation preorder is needed. Since the bisimulation algorithm is generally faster than the simulation algorithm, one can combine them by constructing the simulation quotient based on the bisimulation quotient.

In many applications the specification can not be easily expressed by the logic PCTL or CSL: it is rather a probabilistic model itself. Examples of this kind include various recent wireless network protocols, such as ZigBee [10], Firewire Zeroconf [7], or the novel IEEE 802.11e, where the central mechanism is selecting among different-sided dies, readily expressible as a probabilistic automaton [16]. For such cases, a decision algorithm for simulation preorder can be applied as a specification checker: The model satisfies the specification if the automaton for the specification *simulates* the automaton for the model. We believe that such specification checking is the only formal validation technique that is in reach for verifying implementations of the above protocols. Given the emergence of ever more wireless standards of that sort, there is an obvious motivation to study the principal technological basis: the decision algorithm for probabilistic simulation. This paper attacks the very problem of efficient decision algorithms for probabilistic simulation.

Let n denote the number of states, and m denote the number of transitions. Baier *et al.* [2] introduced a polynomial decision algorithm for simulation with time complexity $\mathcal{O}(n^7/\log n)$ and space complexity $\mathcal{O}(n^2)$, by tailoring a network flow algorithm to the problem, embedded into an iterative refinement loop. This complexity can be improved to time complexity $\mathcal{O}(m^2 n)$ by exploiting the parametric maximum flow algorithm [8] to solve the maximum flows for the arising sequences of similar networks [21]. This improvement however comes with a penalty in space complexity $\mathcal{O}(m^2)$, since one has to store networks across iterations. Lately, the algorithm developed in [21] has been extended to handle probabilistic automata and their continuous-time extension [20].

The purpose of this paper is to complement the theoretical complexity results with practical evidence concerning which algorithmic approach has the most potential in practical applications. We provide, for the first time, systematic experimental results of the space and time requirements of the available algorithms, also comparing several optimizations and heuristics to accelerate the algorithm. As a base algorithm we use an implementation of decision algorithm [2] without any optimizations. The parametric maximum flow variation is treated as one particular optimization. We also consider the effect of the following optimizations which can be applied selectively:

– Partitioning: By grouping states with identical probabilistic structure into equivalence classes, computations can be performed on representative elements for each class.
– Invariant checking: Some pairs can be removed from the simulation by asserting an invariant on the arc capacities in the corresponding maximum flow network, which is computationally less complex than the maximum flow algorithm. This invariant is referred to as the P-Invariant in the remainder of this paper.
– Significant arcs: As the algorithm progresses, arcs will be deleted from maximum flow networks as a result of pairs being removed from the simulation relation. If deleting such an arc will cause the network's flow to be less than 1, it is called significant. By deciding which arcs are significant in advance, some networks will be discarded as part of the update process and do not have to be considered in the following iteration.

We apply our approach to a variety of case studies taken from the PRISM webpage `http://www.prismmodelchecker.org`. In order to avoid a bias in the selection of models, we also evaluate the algorithms on randomly generated Markov models. This is inspired by [19] where the authors experimentally evaluated algorithms for classical automata constructions on random generated automata. Our experimental approach follows the same strategy. On randomly generated Markov chains, we have two interesting parameters to adjust in our studies: the density of transitions and the density of labels. We study the performance curve for various density combinations.

In a nutshell, we observe that state partitioning performs best on models with low to medium transition densities while P-Invariant checking, significant arc detection and parametric maximum flow perform better on models with medium to high transition densities. We also observe that significant arc detection and parametric maximum flow is not commendable in cases where memory usage is a concern.

Organization of the paper. Section 2 recalls the decision algorithm. We discuss various optimization strategies in Section 3. In Section 4 different combinations of the optimizations are compared on regular models, uniform random models and non-uniform random models. Section 5 concludes the paper.

2 Preliminaries

Let AP be a fixed, finite set of atomic propositions. For a finite set S, a distribution μ on S is a function $\mu : S \to [0,1]$ satisfying the condition $\mu(S) \leq 1$. We let $\mathit{Dist}(S)$ denote the set of distributions over the set S. The support of μ is defined by $\mathit{Supp}(\mu) = \{s \mid \mu(s) > 0\}$, and the size of μ is defined by $|\mu| = |\mathit{Supp}(\mu)|$. The distribution μ is called stochastic if $\mu(S) := \sum_{s \in S} \mu(s) = 1$, absorbing if $\mu(S) = 0$, and sub-stochastic otherwise. We use an auxiliary state (not a *real* state) $\perp \notin S$ and set $\mu(\perp) = 1 - \mu(S)$. Further, let $S_\perp$ denote the set $S \cup \{\perp\}$, and let $\mathit{Supp}_\perp(\mu) = \mathit{Supp}(\mu) \cup \{\perp\}$ if $\mu(\perp) > 0$.

Probabilistic Automata [17]. A probabilistic automaton (PA) is a tuple $\mathcal{M} = (S, Act, \mathbf{P}, L)$ where S is a finite set of states, Act is a finite set of actions, $\mathbf{P} \subseteq S \times Act \times Dist(S)$ is a finite set, called the probabilistic transition matrix, and $L : S \to 2^{AP}$ is a labeling function.

For $(s, \alpha, \mu) \in \mathbf{P}$, we use $s \xrightarrow{\alpha} \mu$ as a shorthand notation, and call μ an α-successor distribution of s. The PA $\mathcal{M}$ is a *fully probabilistic system* (FPS) if $Act = \{\alpha\}$ is a singleton and for $s \in S$, there is at most one transition $s \xrightarrow{\alpha} \mu$. A discrete-time Markov chain (DTMC) is an FPS where all distributions are either stochastic or absorbing. For ease of notation, we give a simpler definition for FPSs by dropping the single action: An FPS is a tuple $\mathcal{M} = (S, \mathbf{P}, L)$ where S, L as defined for PAs, and $\mathbf{P} : S \times S \to [0, 1]$ is the probabilistic transition matrix such that $\mathbf{P}(s, \cdot) \in Dist(S)$ for all $s \in S$. The *fanout* of the FPS $\mathcal{M}$ is defined by $\max_{s \in S} |\mathbf{P}(s, \cdot)|$.

Simulation requires that every α-successor distribution of one state have a corresponding α-successor distribution of the other state. The correspondence of distributions is naturally defined with the concept of *weight functions* [13]. For $\mu, \mu' \in Dist(S)$ and $R \subseteq S \times S$, a weight function for (μ, μ') with respect to R, denoted by $\mu \sqsubseteq_R \mu'$, is a function $\Delta : S_\perp \times S_\perp \to [0, 1]$ such that

1. $\Delta(s, s') > 0$ implies $(s, s') \in R$ or $s = \perp$,
2. $\mu(s) = \Delta(s, S_\perp)$ for $s \in S_\perp$,
3. $\mu'(s') = \Delta(S_\perp, s')$ for $s' \in S_\perp$.

The relation $R \subseteq S \times S$ is a *simulation* [17] on $\mathcal{M}$ *iff* for all s_1, s_2 with $(s_1, s_2) \in R$: $L(s_1) = L(s_2)$ and if $s_1 \xrightarrow{\alpha} \mu_1$ then there exists a transition $s_2 \xrightarrow{\alpha} \mu_2$ with $\mu_1 \sqsubseteq_R \mu_2$. We say that s_2 simulates s_1, denoted by $s_1 \precsim_{\mathcal{M}} s_2$, iff there exists a simulation R on $\mathcal{M}$ such that $(s_1, s_2) \in R$. Obviously $\precsim_{\mathcal{M}}$ is the coarsest simulation relation for $\mathcal{M}$.

For $(s_1, s_2) \in R$, we say that s_2 simulates s_1 up to R, denoted $s_1 \precsim_R s_2$, if $L(s_1) = L(s_2)$ and if $s_1 \xrightarrow{\alpha} \mu_1$ then there exists a transition $s_2 \xrightarrow{\alpha} \mu_2$ with $\mu_1 \sqsubseteq_R \mu_2$. Otherwise we write $s_1 \not\precsim_R s_2$. Note that $s_1 \precsim_R s_2$ does not imply $s_1 \precsim_{\mathcal{M}} s_2$ unless R is a simulation, since only the first step is considered for $\precsim_R$.

Algorithm for deciding simulation. The algorithm [2] takes as a parameter a model, which, for now, is an FPS $\mathcal{M}$. To calculate the simulation relation for $\mathcal{M}$, the algorithm starts with the trivial relation $R_{init} = \{(s_1, s_2) \in S \times S \mid L(s_1) = L(s_2)\}$ and removes each pair (s_1, s_2) if s_2 cannot simulate s_1 up to the current relation R, i. e., $s_1 \not\precsim_R s_2$. This proceeds until there is no such pair left, i. e., $R_{new} = R$. Invariantly throughout the loop it holds that R is at least as coarse as $\precsim_{\mathcal{M}}$. Hence, we obtain the simulation preorder $\precsim_{\mathcal{M}} = R$, once the algorithm terminates.

The decisive part of the algorithm is the check whether $s_1 \precsim_R s_2$. As the condition $L(s_1) = L(s_2)$ is easy to check, it remains to check whether $\mathbf{P}(s_1, \cdot) \sqsubseteq_R \mathbf{P}(s_2, \cdot)$ holds. This is reduced to a maximum flow computation on the network $\mathcal{N}(s_1, s_2, R)$ constructed out of $\mathbf{P}(s_1, \cdot)$, $\mathbf{P}(s_2, \cdot)$ and R. This network is constructed via a graph containing a copy $\overline{t} \in \overline{S_\perp}$ of each state $t \in S_\perp$ where

$\overline{S_\perp} = \{\overline{t} \mid t \in S_\perp\}$ defined as follows: Let $\nearrow$ (the source) and $\searrow$ (the sink) be two additional vertices not contained in $S_\perp \cup \overline{S_\perp}$. For $\mu, \mu' \in \mathit{Dist}(S)$ and a relation $R \subseteq S \times S$ we define the network $\mathcal{N}(\mu, \mu', R) = (V, E, u)$ with the set of vertices

$$V = \{\nearrow, \searrow\} \cup \mathit{Supp}_\perp(\mu) \cup \overline{\mathit{Supp}_\perp(\mu')}$$

and the set of edges (or arcs) E defined by

$$E = \{(s, \overline{t}) \mid (s,t) \in R \vee s = \perp\} \cup \{(\nearrow, s), (\overline{t}, \searrow)\}$$

where $s \in \mathit{Supp}_\perp(\mu)$ and $t \in \mathit{Supp}_\perp(\mu')$. The capacity function u is defined as follows: $u(\nearrow, s) = \mu(s)$ for all $s \in S_\perp$, $u(\overline{t}, \searrow) = \mu'(t)$ for all $t \in S_\perp$, $u(s, \overline{t}) = \infty$ for all $(s,t) \in E$ and $u(v,w) = 0$ otherwise. Obviously, $\mathcal{N}(s_1, s_2, R)$ is a bipartite network. For two states s_1, s_2, we let $\mathcal{N}(s_1, s_2, R)$ denote the network $\mathcal{N}(\mathbf{P}(s_1, \cdot), \mathbf{P}(s_2, \cdot), R)$.

The crucial relationship exploited in [2] is that $\mathbf{P}(s_1, \cdot) \sqsubseteq_R \mathbf{P}(s_2, \cdot)$ iff the maximum flow in $\mathcal{N}(s_1, s_2, R)$ is 1. Thus we can decide $s_1 \precsim_R s_2$ by computing the maximum flow in $\mathcal{N}(s_1, s_2, R)$. A key observation we made in [21] is that the networks $\mathcal{N}(s_1, s_2, \cdot)$ constructed later in successive iterations are very similar: They differ from iteration to iteration only by deletion of some edges induced by the successive clean up of R. The algorithm, hence, exploits this fact by leveraging maximum flow already computed in the last iteration rather than re-starting maximum flow computation from scratch each time. In more detail, we consider the initial network $\mathcal{N}(s_1, s_2, R_{init})$ for an arbitrary pair $s_1, s_2 \in S$. Recall R_{init} denotes the initial relation $\{(s_1, s_2) \in S \times S \mid L(s_1) = L(s_2)\}$. Let $D_1, \ldots, D_k$ be pairwise disjoint subsets of R_{init}, which correspond to the pairs deleted from R_{init} in iteration i. Let $\mathcal{N}(s_1, s_2, R_i)$ denote $\mathcal{N}(s_1, s_2, R_{init})$ if $i = 1$, and $\mathcal{N}(s_1, s_2, R_{i-1} \setminus D_{i-1})$ if $1 < i \leq k+1$. Let f_i denote the maximum flow of the network $\mathcal{N}(s_1, s_2, R_i)$ for $i = 1, \ldots, k+1$. The problem of checking $|f_i| = 1$ for all $i = 1, \ldots, k+1$ can be checked efficiently by exploiting a variation of the parametric maximum flow algorithm [8] (called algorithm for a sequence of maximum flows in [21]). Based on this, an algorithm with time complexity $\mathcal{O}(m^2 n)$ is introduced for FPSs, CTMCs in [21], and for PAs in [20]. This improvement however comes with a penalty in space complexity: it is increased from $\mathcal{O}(n^2)$ to $\mathcal{O}(m^2)$, due to the need to store of the maximum flow values of the corresponding networks across iterations.

3 Optimization Options

Our implementation of the principal algorithm uses the following optimizations and heuristics to eliminate redundant or trivial computations. All of the optimizations and heuristics presented apply to DTMCs, CTMCs and PAs directly. Throughout this section, we fix an FPS $\mathcal{M}$ and a pair of states s_1, s_2. The same considerations can also be directly applied to CTMCs and PAs. Let n denote the number of states and m denote the number of transitions of $\mathcal{M}$. Let $\mathcal{N}(s_1, s_2, R)$ denote the network as defined earlier. Furthermore, let V denote the set of the vertices, and E denote the set of the edges of $\mathcal{N}(s_1, s_2, R)$.

Compact Maximum Flow. The algorithm used to compute the maximum flow is based on the existing push-relabel based preflow algorithm [9] and tailored specially to the needs of the decision algorithm in order to save memory and to omit computations for cases that never arise in the scenario considered. In a complete maximum flow implementation, the value of the flow is computed. However, for the purpose at hand, it is sufficient to determine whether or not the flow equals 1. To decide the simulation preorder, we consider bipartite networks in which source and sink and all arcs connected with them are not relevant to the computation and can be omitted. Furthermore, the fact that all remaining (not connected to source or sink) arcs have infinite capacity allows us to ignore the concept of arc capacity altogether.

The use of this tailored algorithm greatly reduces the memory usage (by a factor of approximately 4 to 6) in comparison to a more generic implementation while its runtime stays almost unchanged in most cases. It should be noted that this implementation does not use certain known optimizations for the push-relabel based method and is inferior in speed to implementations which use these optimizations.

Parametric Maximum Flow. Premise: By saving the result of previous maximum flow computations and keeping the network consistent with the constraints of a valid flow when deleting arcs as described in [8], the time required to recompute the maximum flow repeatedly on the same network can be reduced.

This adds a $\mathcal{O}(|E|)$ time overhead for updating each network. Additional space in the order of $\mathcal{O}(m^2)$ is needed to store all the networks ($\mathcal{O}(|E|)$ per network) such that they can be passed to the next iteration. Depending on the structure of a maximum flow network, the time needed to compute the flow varies greatly.

State Partitioning. Premise: In large models, many states will be structurally identical. This can be exploited by grouping states with identical probabilistic structure together into an equivalence class. This forms a partition of the state space. The equivalence classes are also referred to as blocks. Given two blocks B_1 and B_2 of the partition, simulation algorithm will yield the same result for any pair (s_1, s_2) with $s_1 \in B_1$ and $s_2 \in B_2$. Thus, it suffices to decide simulation once for an arbitrary pair of states picked from B_1 and B_2.

Two states s_1 and s_2 have an identical probabilistic structure if their successors have pairwise the same labels and the same respective transition probabilities. It is important to note that state partitioning is only correct in the first iteration of the simulation algorithm when the initial relation is defined solely on the basis the labels, thus is an equivalence relation. As soon as the relation is not an equivalence relation any more, state partitioning can no longer be applied.

State partitioning adds an overhead of $\mathcal{O}(n \log n)$ for sorting states and successor sets. This is necessary for being able to compute the partition and to be able to test whether two states should belong to the same block in linear time with respect to the number of transitions in the model. State partitioning uses an extra $\mathcal{O}(n + h^2)$ space, where h is the number of blocks in the partition. In

order to store which block a state belongs to we need $\mathcal{O}(n)$, and in order to store the result of whether one block simulates another block we need $\mathcal{O}(h^2)$.

P-Invariant Checking. Premise: For a relation $R \subseteq S \times S$, we define $R(s) := \{s' \in S \mid (s, s') \in R\}$ and $R^{-1}(s) := \{s' \in S \mid (s', s) \in S\}$. The maximum flow of a network can only be 1 if the following two constraints are met:

1. $\mu(s) \leq \mu'(R(s))$ for all $s \in S$,
2. $\mu'(s') \leq \mu(R^{-1}(s'))$ for all $s' \in S$.

The complexity of verifying these constraints is in the order of $\mathcal{O}(|E|)$ per network and $\mathcal{O}(m^2)$ overall. This operation needs an additional $\mathcal{O}(|V|)$ space while performing the checks. Additionally, if the condition $\mathbf{P}(s_1, S) > \mathbf{P}(s_2, S)$ holds, $(s_1, s_2) \notin R$ is implied. The test of this constraint can be performed in $\mathcal{O}(n)$ time once before the simulation algorithm and it requires $\mathcal{O}(n)$ space during the operation.

Significant Arc Detection. Premise: The P-Invariant constraints are only checked when a network is created. However, it would be desirable to check whether or not the constraints are still fulfilled after a certain arc has been deleted as a result of its corresponding pair having been removed from the relation. This can be done as follows: For a network which satisfies the P-Invariant constraints, an arc is called *significant* iff its removal would cause the network to violate the constraints. The detection of these arcs takes $\mathcal{O}(|V|^2)$ time in addition to that of P-Invariant checking and $\mathcal{O}(|E|)$ space per network for storing the flag for every arc. Removing an arc takes constant time if the arc is significant, otherwise $\mathcal{O}(|E|)$ time to recompute the significance of the remaining arcs.

Significant arc detection is an extension of parametric maximum flow. It requires that networks be stored rather than recomputed from scratch, otherwise it is equivalent to P-Invariant checking.

4 Case studies

The following section examines the performance of the algorithm with the various optimizations turned on and off in respect to different models.

In the case studies, we refer to the different configurations of optimizations considered in this paper by binary numbers constituting combinations of the following strategies: State Partitioning (0001), P-Invariant Checking (0010), Significant Arcs (0100), Parametric Maximum Flow (1000). Reported run-times measure the amount of CPU time (user mode only) spent computing the simulation. Time used on parsing the model prior to simulation and cleaning up memory after simulation is not accounted for. By omitting time spent in system mode, the result is not affected by virtual memory operations. The code was compiled with compiler optimizations turned off to demonstrate the advantage achieved by the heuristics alone. With compiler optimizations turned on, an additional speed-up of up to three times is achieved in some cases. The lowest amount of time/memory is marked in bold print in the tables.

Table 1. Time and memory used for Leader Election models under various optimizations. Memory statistics represent peak values throughout the process of deciding simulation preorder, excluding memory used by the relation map which is present in all configurations (Map size)

States	439	1031	2007	3463	439	1031	2007	3463
Trans.	654	1542	3006	5190	654	1542	3006	5190
Unit	Time (sec)		Time (min)		Space (kB)			
Map size					47.158	259.763	983.900	2928.669
0000	6.62001	196.25106	47.409	421.233	*754.500*	*4156.195*	*15742.382*	*46858.687*
0001	0.22081	2.07773	0.234	1.181	754.515	4156.210	15742.398	46858.703
0010	0.14801	0.69684	0.049	0.209	754.500	4156.195	15742.382	46858.687
0011	0.09101	0.39202	0.026	0.113	754.516	4156.211	15742.398	46858.703
1000	6.59761	196.70669	47.632	422.430	3910.007	20711.601	81310.734	266355.210
1001	0.19201	2.04513	0.235	1.180	2589.883	13113.180	53140.984	182841.039
1110	0.10681	0.59084	0.043	0.170	4015.472	21263.984	83497.390	273674.011
1111	*0.06600*	*0.32102*	*0.022*	*0.084*	2651.290	13412.281	54388.648	187375.586

4.1 Regular case studies

Leader Election Models. The leader election family of models have a very simple structure, namely that of one state in each model with a large number, denoted by k, of successors while the remaining states have only one successor. As such, these models are a prime example for a successful application of partitioning. Due to the structural similarity of the models, the number of blocks of the state partition is 4 for all leader election models and the number of times that the maximum flow algorithm is actually invoked is drastically decreased. For the simulation of three leaders and $k = 8$ (1031 states, 1542 transitions) with uniform distribution of three different labels, the maximum flow algorithm is invoked 369859 times without any optimization, and 228109 times with state partitioning.

The time advantage achieved by this becomes apparent in Table 1 (0000 vs. 0001). Due to the simplistic structure of the models, parametric maximum flow yields only a small advantage on the leader election models as recomputing from scratch is not very complex. In general, using the parametric maximum flow algorithm by itself is not desirable for sparse models because the advantage is negligible in comparison to the time and memory overhead. Table 1 illustrates the additional amount of time and memory required for parametric maximum flow (1000) versus the approach without any optimizations (0000).

Additionally, maximum flow usage statistic shows that the maximum flow algorithm is invoked more often (although by a relatively small margin) with parametric maximum flow enabled than not. This is due to the fact that certain trivial networks are discarded during construction without ever computing their maximum flow. However if a network was not initially trivial but becomes trivial after an arc is deleted, this is only detected upon reconstruction of the same network, but not upon updating and recomputing the network if it was saved.

Significant arc detection works against this by effectively performing P-Invariant checking every time an arc is removed from a network.

P-Invariant checking and significant arc detection have little effect in reducing the number of times that the maximum flow algorithm is used on models similar to leader election when used alone. This is due to the fact that almost all states (all except for the first) have exactly one successor and consequently almost all networks have either one arc or none at all. Those with no edges at all are filtered out in advance and those with one edge have $\sum_{s' \in S} P(s, s') = 1$ for both s_1 and s_2 so that P-Invariant checking cannot achieve any additional filtering. The small reduction in maximum flow usage is due to the first state which has more than one successor but is unfortunately negligible.

We also note that the time advantage achieved by P-Invariant checking and significant arc detection is exceptionally large compared to the reduction in maximum flow usage. This is because a small number of networks which appear in the leader election models and are filtered out by these optimizations, are inefficient to compute under the maximum flow implementation used in this study. Therefore, the time spent computing maximum flow decreases significantly even though the algorithm is still used almost as much.

Overall, it is notable that the minimum time for simulating leader election is consistently achieved by the configuration 1111. It can be said that in general, the combination of all presented optimizations is beneficial for extremely sparse models such as leader election. If memory usage is a concern, 0011 should be preferred over 1111 as it works without ever storing more than one maximum flow problem in memory at a time (cf. Table 1) while only slightly inferior to 1111 in speed.

Molecular Reactions. For CTMCs we consider the Molecular Reactions as a case study. In particular, we focus on the reaction $Mg + 2Cl \longleftrightarrow Mg^{+2} + 2Cl^{-}$. Models for other reactions found on the PRISM web-site are very similar in structure and do not offer any additional insight.

While the structure of this family of models is relatively simple, few optimizations show any notable effect. All states have between 1 and 4 successors with the average being around 3.8 for all models, but the transition rates are different between almost all states. As a consequence, state partitioning fails entirely. With a few minor exceptions, all blocks of the partition contain exactly one state, which means that no speed-up can be achieved at all. In particular, the reduction in maximum flow usage is always below 1%.

Although the optimizations are not very effective, you will note that in comparison to the leader election models, the algorithm terminates very quickly on this family of models (See also Table 1 and Table 2): 7 hours for Leader Election with 3463 States and 5190 Transitions (cf. 0000), 9 seconds for Molecular Reaction with 4032 States and 15750 Transitions (cf. 0000). This is because the simulation relation is empty except for the identity relation for all these models which is known after just two iterations of the algorithm. The leader election family on the other hand needs four iterations and does not have a trivial simulation relation, which makes the process of deciding simulation preorder more

Table 2. Time and memory used for Molecular Reaction models under various optimizations. Memory statistics represent peak values throughout the process of deciding simulation preorder, excluding memory used by the relation map which is present in all configurations (Map size)

States	676	1482	2601	4032	5776	676	1482	2601	4032	5776
Trans.	2550	5700	10100	15750	22650	2550	5700	10100	15750	22650
Unit	Time (ms)		Time (sec)			Memory (MB)				
Map size						0.11	0.52	1.61	3.88	7.95
0000	226.0	1158.9	3.622	9.261	19.840	*0.88*	*4.28*	*13.19*	*31.72*	*65.12*
0001	234.8	1169.3	3.751	9.487	20.650	1.33	6.42	19.79	47.60	97.70
0010	204.0	976.1	3.059	*7.660*	16.960	0.88	4.28	13.19	31.72	65.12
0011	212.0	1039.3	3.375	8.321	18.552	1.33	6.42	19.79	47.60	97.70
1000	227.2	1139.3	3.610	9.039	19.458	1.09	5.50	17.16	40.99	85.49
1001	232.8	1181.7	3.788	9.571	20.386	1.52	7.44	23.08	55.73	114.53
1110	*194.8*	*954.1*	*3.027*	7.761	*16.754*	0.90	4.37	13.53	32.54	66.88
1111	215.2	1077.7	3.349	8.744	19.107	1.46	7.21	22.29	53.92	110.80

complex. (Additionally, the leader election family also has some networks for which the maximum flow is hard to compute.) This is also why the memory values are all relatively close to each other (see Table 2), specifically the configurations which use parametric maximum flow (1***). Intuitively this is true because almost every pair is immediately discarded and does not have to be saved for later iterations. This implies that parametric maximum flow does not hold any benefit for this type of model.

The only optimization which shows some promise for this type of model is P-Invariant checking (0010). Only surpassed by configuration 1110 in a few cases, it has the greatest performance boost of all, although it is relatively small when compared to the approach without any optimizations (0000). While P-Invariant checking consistently reduces maximum flow computation by about 99.2%, the largest part of the run-time is taken up by the remaining set of pairs which are not discarded until the second iteration. Significant arc detection, which builds upon P-Invariant checking and parametric maximum flow computation, does not hold any benefit for this model due to the failure of parametric maximum flow. While faster than pure P-Invariant checking in some cases as a result of the left-over pairs not discarded in the first iteration, the speed-up is not consistent and only in the range of about 1.5% to 5.25%.

Dining Cryptographers. We use the Dining Cryptographers model from the PRISM web-site to study the performance of our algorithm on PAs. In this study, we reduce the set of configurations to 0000, 0001, 0010 and 0011, excluding significant arcs and parametric maximum flow which have not yet been implemented.

Table 3 shows that state partitioning (0001) is clearly the best choice for this model. While the average size of the partition is relatively small, a speedup of about 50% is achieved on average.

Table 3. Time and memory used on Dining Cryptographers models.

Cryptographers	3	4	5	3	4	5
States	381	2166	11851	381	2166	11851
Trans.	780	5725	38778	780	5725	38778
Actions	624	4545	30708	624	4545	30708
	Time			Space (MB)		
Map size				0.03488	1.11959	33.49911
0000	71.00 ms	2.037 s	86.788 s	*0.36649*	*11.91495*	*357.08298*
0001	*40.00 ms*	*0.977 s*	*39.839 s*	0.36916	11.95903	357.72893
0010	79.01 ms	2.248 s	89.793 s	0.36649	11.91495	357.08298
0011	42.00 ms	1.068 s	42.056 s	0.36916	11.95903	357.72893

It is notable that P-Invariant checking is actually slower on this model than approach 0000. This is because of the structure of the models. Since every action has either one or two equally likely successors, a pair will almost never be discarded due to violating the P-Invariant constraint which can be seen as follows. All networks have at most two vertices on the left and two on the right. Consider a network and assume first that there is at least a vertex which has no arcs connected to it. In this case the network is discarded as trivial since the maximum flow must be below 1. Now assume that each vertex in the network has at least an arc connected to it. In this case it is easy to see that the maximum flow of the network is 1. Consequentially, the benefit of P-Invariant checking is very low in the first iteration, which accounts for the bulk of the total runtime and the computational overhead prevails.

For the same reason as described above, the combination of state partitioning and P-Invariant checking does not outperform state partitioning on its own.

4.2 Randomly Generated Models

Uniform models. In addition to regular case studies, we consider randomly generated DTMCs with uniform distributions, that is, all transitions from a state s have equal probabilities. If not stated explicitly, we also use three different labels which are uniform distributed. Furthermore, these random models can be described by three parameters n, a and b such that $|S| = n$ and $a \leq |post(s)| \leq b \ \forall s \in S$. We will reference random uniform model by the parameters n, a, b. Table 4 illustrates required time, memory and number of invocations of the maximum flow algorithm with respect to different model sizes for random uniform models.

This study is particularly remarkable because it demonstrates the strength of parametric maximum flow. In comparison to other cases studied above, leading configurations in the study at hand use parametric maximum flow. This is due to the density of the model, i.e. the larger number of successors per state in comparison to the other case studies in this paper. It is also remarkable that, in contrast to other case studies above, all optimizations hold some (even though limited) benefit.

Table 4. Comparison of all optimizations on uniform random models $400, 1, B$ with varying numbers of B. Values are in milliseconds.

B	10	20	30	40	50	60	70	80
0000	7.93	36.60	83.81	140.34	224.68	372.66	650.67	718.48
0001	3.13	28.04	66.64	117.97	185.61	303.72	521.30	573.94
0010	6.90	34.37	81.47	151.68	229.28	395.62	649.67	671.28
0011	*2.77*	*26.43*	*60.64*	97.14	196.08	276.15	473.63	520.97
1000	8.00	34.80	78.97	126.47	195.01	319.29	543.03	612.57
1001	3.17	27.37	64.54	109.37	166.64	272.08	*449.16*	510.20
1010	7.10	34.54	80.57	138.21	211.98	349.39	573.07	637.74
1011	2.77	26.50	61.24	*96.44*	183.28	*268.75*	455.56	*493.56*
1100	9.47	40.30	89.01	137.24	214.05	356.22	601.07	685.98
1101	3.90	31.04	72.24	117.64	181.38	296.05	490.80	555.77
1110	7.37	36.87	84.64	132.37	207.65	344.99	583.04	660.61
1111	2.90	27.47	63.24	99.57	*174.71*	278.78	469.56	509.00

State partitioning performs well on the lower end of the range, yielding a speed-up of about 80% at best and about 20% at worst. While a larger speed-up may be desirable, this is a very good result since it means that state partitioning will never slow down the process on this kind of model.

P-Invariant checking is beneficial in most cases, particularly towards the upper end of the range, but in a few cases ($40 \leq B \leq 65$) it is actually slower than approach 0000 and it is also slower than state partitioning in general. Consequentially, P-Invariant checking should not be applied on its own. Coupled with state partitioning however (see configuration 0011), P-Invariant checking performs better and is in fact one of the best configurations in the study at hand.

While faster in a few cases, significant arc detection does not yield a consistent performance boost in any configuration. Significant arc detection is most powerful in gradual simulation decision processes where few arcs are deleted in one iteration. The simulation relations in this study however are decided in only three to four iterations, indicating that most pairs of states are deleted from the relation in the first iteration already, but significant arc detection can only speed up the decision on pairs which are not deleted immediately. It stands to reason that significant arc detection would perform better in models with a larger minimum number of successors per state.

Parametric maximum flow shows good results in this study. Clocking in at speeds faster than P-Invariant checking in many cases, this is the kind of model for which parametric maximum flow is beneficial. At its worst, parametric maximum flow is about 4% slower than approach 0000. At its best, it is faster by 18%.

The best configuration for this model is a tie between 1001 and 1011. While 1111 sometimes achieves times better than 1001 or 1011, it also requires more memory and has about the same average performance as either 1001 or 1011.

Consider also Figure 1 which compares the performances of all configurations[1] on uniform random models with different numbers of labels. All optimizations except state partitioning (0001) and configurations making use of it have monotone falling curves because more labels means that the initial relation will be smaller. Configurations using state partitioning however are affected in a different manner, displaying a very low value at one label, a maximum at two labels and a monotone curve after that. The reason for this behavior is that having only one label works in favor of the partitioning algorithm, enabling it to partition the state space into fewer blocks.

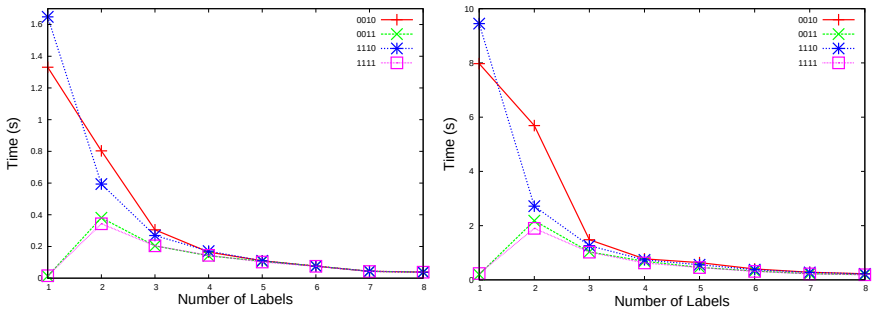

Fig. 1. Comparison of configurations on random uniform models $200, 1, 25$ (left)and $200, 1, 50$ (right) with respect to varying numbers of labels. Values are averaged over 4 independently generated models of the same class.

Non-uniform models. In addition to random uniform models, we also briefly consider randomly generated DTMCs with varying degrees of structure. For this purpose, we define several structural features called biases which loosely represent the probability that a certain feature is present or not. We define the following biases:

- Probability Bias, $pb \in [0; 1]$, defines whether or not the transition probabilities are distributed uniformly ($pb = 0$) or randomly ($pb = 1$)
- Fanout Bias, $fb \in [-1; 1]$, defines if a state is more likely to have the minimum ($fb < 0$) or maximum ($fb > 0$) number of successors

It must be noted that, in case of $pb > 0$, the generated probabilities are not random values. Rather, the partition of the successor set into subsets of successors, each of which have different transition probabilities, is random. This means that the distribution for state s is equal to the distribution for state s' w.r.t. transition probabilities iff $|post(s)| = |post(s')|$ and the successor sets are partitioned into subsets of equal sizes. As a consequence, the state partitioning optimization

[1] To get a readable picture, we plot only the representative configurations, i.e., configurations showing extreme performances. This holds also for Figure 2.

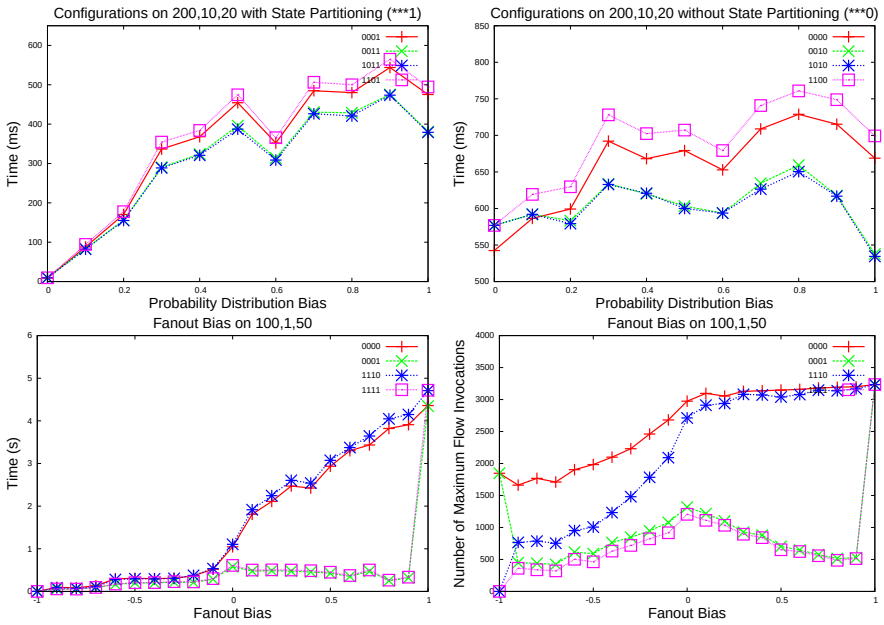

Fig. 2. Comparison on random nonuniform models with probability bias and fanout bias

is still likely to find useful partitions, even though the same optimization would be useless for models with truly randomized transition probabilities.

Consider Figure 2 (first row) which plots the time needed for simulation for $200, 10, 20$ models with different values of probability bias. On the left, we have all configurations which use state partitioning (***1). On the right, we have all remaining configurations. We observe that state partitioning (left) performs best with uniform distributions and gets progressively slower for higher values of the bias. Intuitively this is because the partitioning algorithm is able to create fewer blocks when more distributions are uniform. All other configurations are only insignificantly affected by the bias (right). In these cases, only the complexity of computing the maximum flow depends on the distributions, which accounts for a comparatively small portion of the run-time in models with a low number of successors per state. In both subsets, the configurations using P-Invariant checking (**1*) perform better compared to the remaining configurations for higher values of the bias, because nonuniform distributions are more likely to violate the P-Invariant constraints.

In Figure 2 we also compare the impact of different fanout biases on the set of representative configurations. We observe, as one might expect, that a higher fanout bias increases the run-time of the algorithm. An exception to this are configurations which use state partitioning (***1), which are only insignificantly affected by the bias, except for the special case of $fb = 1$. For this value, all states are in the same block and thus state partitioning cannot improve the run-

time. The right plot shows that the increase in run-time is not directly linked to the number of times the maximum flow algorithm is invoked. In particular, the maximum (disregarding corners) for configurations which use state partitioning (***1) is at $fb = 0$, the value which represents the highest entropy and the highest number of blocks. For other configurations (***0), the maximum is reached by $fb > 0$, in which case only a statistically insignificant number of maximum flow computations is trivial. However, the run-time of the algorithm still rises because the complexity of the individual maximum flow computations increases. We conclude that this result depends to a high degree on the complexity of maximum flow computation more than the number of such computations, which means that it will vary greatly for different ranges of numbers of successors.

5 Conclusions

This paper has investigated an experimental approach to algorithm design, especially for Markov models. Starting off with a published simulation algorithm, we experimented with different models to determine ways of further improving upon this algorithm. At the end of this empirical process we have several promising concepts, implemented as optimizations to the fundamental algorithm. Using a collection of well-chosen case studies as well as randomly generated models we studied the practical performance of the concepts.

One of the most interesting observations of our experimental studies is the not uncommon imbalance between theoretical complexity and runtime in practice. While the parametric maximum flow based method [21, 20] offered a tremendous drop in theoretical complexity, its practical implementation comes with an overhead that makes it considerably weaker in many practical applications than more straightforward approaches. Its strength are large, dense models which require several iterations to terminate. These cases seem seldom in models commonly used for case studies. The gap between theoretical and practical efficiency is not caused by "the constant factors" but by the fact that the corner cases that blow up the worst case complexity are rare in practice.

We were surprised to find that simpler and more intuitive approaches like state partitioning and P-Invariant checking actually produced promising results in general in our practical studies, in comparison to our theoretically proven algorithm. In particular, state partitioning works very well on models with low to medium transition densities and near-uniform or uniform probability distributions. On the other hand, P-Invariant checking performs very well on models with non-uniform probability distributions.

As future work we plan to make the tool available such that the optimizations and achievements are at hand for deciding simulation preorders for Markov chains and probabilistic automata. We also plan to extend the implementation to compute weak simulation for Markov chains. Additionally, we plan to develop heuristics to determine internally where to selectively apply optimizations to achieve an even better performance. Another direction is to compute the pre-

orders symbolically, i. e., using MTBDDs (multi-terminal BDDs) to fight state space explosion problems.

References

1. A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous time Markov chains. In *CAV*, pages 269–276, 1996.
2. C. Baier, B. Engelen, and M. E. Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *J. Comput. Syst. Sci.*, 60(1):187–231, 2000.
3. C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003.
4. C. Baier, J.-P. Katoen, H. Hermanns, and B. Haverkort. Simulation for continuous-time Markov chains. In *CONCUR*, pages 338–354, 2002.
5. C. Baier, J.-P. Katoen, H. Hermanns, and V. Wolf. Comparative branching-time semantics for Markov chains. *Inf. Comput*, 200(2):149–214, 2005.
6. A. Bianco and L. de Alfaro. Model Checking of Probabilistic and Nondeterministic Systems. In *FSTTCS, LNCS 1026:499-513*. Springer, 1995.
7. H. C. Bohnenkamp, P. van der Stok, H. Hermanns, and F. W. Vaandrager. Cost-optimization of the ipv4 zeroconf protocol. In *DSN*, pages 531–540, 2003.
8. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, 1989.
9. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
10. C. Groß, H. Hermanns, and R. Pulungan. Does clock precision influence ZigBee's energy consumptions? In *OPODIS*, pages 174–188, 2007.
11. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.
12. A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS*, pages 441–444, 2006.
13. B. Jonsson and K. G. Larsen. Specification and refinement of probabilistic processes. In *LICS*, pages 266–277, 1991.
14. J.-P. Katoen, T. Kemna, I. Zapreev, and D. N. Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In *TACAS*, pages 87–101, 2007.
15. J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *QEST*, pages 243–244, 2005.
16. S. Mangold, Z. Zhong, G. R. Hiertz, and B. Walke. Ieee 802.11e/802.11k wireless lan: spectrum awareness for distributed resource sharing. *Wireless Communications and Mobile Computing*, 4(8):881–902, 2004.
17. R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.
18. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.
19. D. Tabakov and M. Y. Vardi. Experimental evaluation of classical automata constructions. In *LPAR*, pages 396–411, 2005.
20. L. Zhang and H. Hermanns. Deciding simulations on probabilistic automata. In *ATVA*, pages 207–222, 2007.
21. L. Zhang, H. Hermanns, F. Eisenbrand, and D.N. Jansen. Flow faster: Efficient decision algorithms for probabilistic simulations. In *TACAS*, pages 155–169, 2007.