A Modest Approach to Markov Automata

YULIYA BUTKOVA, Saarland University, Germany
ARND HARTMANNS, University of Twente, The Netherlands
HOLGER HERMANNS, Saarland University, Germany and Institute of Intelligent Software, China

Markov automata are a compositional modelling formalism with continuous stochastic time, discrete probabilities, and nondeterministic choices. In this article, we present extensions to Modest, an expressive high-level language with roots in process algebra, that allow large Markov automata models to be specified in a succinct, modular way. We illustrate the advantages of Modest over alternative languages. Model checking Markov automata models requires dedicated algorithms for time-bounded and long-run average reward properties. We describe and evaluate the state-of-the-art algorithms implemented in the mcsta model checker of the Modest Toolset. We find that mcsta improves the performance and scalability of Markov automata model checking compared to earlier and alternative tools. We explain a partial-exploration approach based on the BRTDP method designed to mitigate the state space explosion problem of model checking, and experimentally evaluate its effectiveness. This problem can be avoided entirely by purely simulation-based techniques, but the nondeterminism in Markov automata hinders their straightforward application. We explain how lightweight scheduler sampling can make simulation possible, and provide a detailed evaluation of its usefulness on several benchmarks using the Modest Toolset's modes simulator.

CCS Concepts: • **Theory of computation** → *Quantitative automata*; *Verification by model checking*; • **Computing methodologies** → *Modeling and simulation;*

Additional Key Words and Phrases: Markov automata, probabilistic model checking, statistical model checking, simulation, formal methods, dependability, performance evaluation

ACM Reference format:
Yuliya Butkova, Arnd Hartmanns, and Holger Hermanns. 2021. A Modest Approach to Markov Automata. *ACM Trans. Model. Comput. Simul.* 31, 3, Article 14 (August 2021), 34 pages.
https://doi.org/10.1145/3449355

1 INTRODUCTION

Studying dependability and performance aspects of critical designs or implementations [6] requires an adequate formal mathematical model that captures the core quantitative aspects of

Yuliya Butkova, Arnd Hartmanns, and Holger Hermanns are listed in alphabetical order.

This work has received financial support by DFG Grant No. 389792660 as part of TRR 248 (see perspicuous-computing.science), by ERC Advanced Grant No. 695614 (POWVER), by the Key-Area Research and Development Program Grant No. 2018B010107004 of Guangdong Province, and by NWO VENI Grant No. 639.021.754.

Authors' addresses: Y. Butkova, Saarland University, Saarland Informatics Campus E1 3, 66123 Saarbrücken, Germany; email: butkova@cs.uni-saarland.de; A. Hartmanns, University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands; email: a.hartmanns@utwente.nl; H. Hermanns, Saarland University, Saarland Informatics Campus E1 3, 66123 Saarbrücken, Germany; Institute of Intelligent Software, Huan Shi Da Dao Xi, Nansha, 511400 Guangzhou, China; email: hermanns@cs.uni-saarland.de.

1049-3301/2021/08-ART14 $15.00
https://doi.org/10.1145/3449355

such systems. In particular, we need *stochastic continuous time* to model delays of which we only know averages, e.g., the mean time to failure, *discrete probabilistic choices* to describe instantaneous uncertain decisions, as in, e.g., randomised algorithms, and *nondeterminism* to be able to deal with underspecification, abstraction, unquantified uncertainty (e.g., adversarial inputs or control decisions), and concurrency. **Markov automata (MA)** [36, 38] extend the classical formalisms of **continuous-time Markov chains (CTMC)** and discrete-time **Markov decision processes (MDP)** [70] to encompass all three of these aspects. In contrast to **continuous-time Markov decision processes (CTMDP)**, they are compositional: there is a natural parallel composition operator for networks of MA that provides for both interleaved and synchronising transitions without the need for ad-hoc operations to combine transition rates.

MA are the semantic basis for generalised stochastic Petri nets [37] and dynamic extensions of fault trees [12, 61]. They have been applied to various problems including model-based testing [40] and robot planning under uncertainty [4, 67]. Several publications studied algorithmic problems related to the efficient analysis of MA [3, 21, 23–25, 43, 44, 55]. In this light, it is disappointing that tool support for MA has thus far been rather brittle. The one dedicated tool for compositional modelling with MA, Scoop [79], is unmaintained, as is the corresponding lower-level MA model checker Imca [42]. The one other actively developed tool with comprehensive MA support is Storm [34], which, however, lacks built-in support for high-level compositional modelling.

Using the mathematical formalism of MA directly to build complex models is cumbersome. For their use to be practical, a higher-level *modelling language* is needed. Aside from a parallel composition operator, such languages typically provide variables over finite domains that can be used in expressions to, e.g., enable or disable transitions. Their semantics is then an MA whose states are the valuations of the variables, allowing to compactly describe very large MA. In this article, we present recent extensions to Modest [10, 47], a high-level modelling language for stochastic timed and hybrid systems, that add support for expressing MA models. Rooted in process algebra, Modest provides various composition operators that allow large models to be assembled from small, easy-to-understand components. In Section 3, we illustrate the use of Modest for MA, and we compare its succinctness, expressivity, and readability with alternative languages.

MA models are built for the purpose of assessing quantitative properties of systems such as safety (the probability to reach an unsafe state), reliability (doing so within a time bound), or throughput (the long-run average amount of work completed per time unit). Probabilistic model checking techniques [5] can be applied to MA to effectively compute or approximate such values. While the computation of *unbounded* reachability probabilities and expected accumulated rewards can be reduced to checking the MA's embedded MDP, *time-bounded* probabilities and *long-run average* rewards require dedicated algorithms. Section 4 is dedicated to the currently available algorithms for model checking MA. We summarise their particular characteristics and notable implementation considerations. To complement our extension of the Modest language with suitable analysis facilities, we have implemented the most promising of these algorithms in the mcsta model checker of the Modest Toolset [49]. We use the MA models of the **Quantitative Verification Benchmark Set (QVBS)** [53] to evaluate the performance of our implementation and of the different algorithms. We compare the results with Imca and Storm.

For very large models, exhaustive model checking is currently limited by state space explosion. In the areas of machine learning and probabilistic planning, approaches have been developed that aim to only generate and store in memory those states that are actually necessary to obtain the value of interest at the desired precision. We focus on these *partial exploration* techniques in Section 5. A notable example is **bounded real-time dynamic programming (BRTDP)** [13]. It employs Monte Carlo simulation to guide the partial exploration, which for MA is combined with model checking algorithms to compute value approximations. In some cases, BRTDP allows very

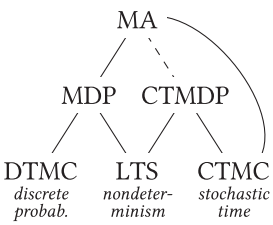


Fig. 1. The MA family tree.

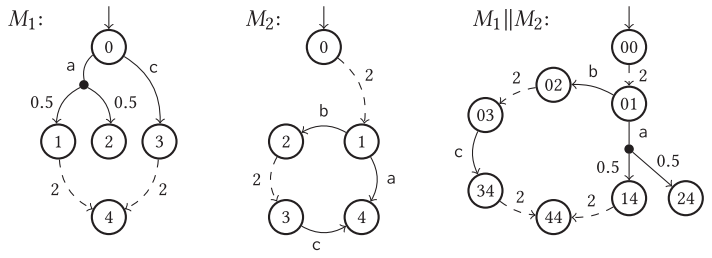


Fig. 2. Example Markov automata.

large models to be checked with decent precision on only few states. In other cases, however, sufficient precision cannot be reached without exploring nearly all states. We experimentally explore the behaviour of the BRTDP approach on MA using all applicable MA models of the QVBS.

If we do not allow storing any more than a constant number of states, then we are in the realm of "pure" **statistical model checking (SMC)** [56, 65, 80] where only the current and next state during a simulation run are ever kept in memory. SMC fully avoids the state space explosion problem. Unfortunately, due to the presence of nondeterminism, MA are not directly amenable to SMC. However, **lightweight scheduler sampling (LSS)** has been introduced to enable SMC for models with nondeterminism [66], albeit without optimality *guarantees*, and lifted to MA for the approximation of optimal unbounded reachability probabilities and expected accumulated rewards [31]. We explain the idea behind, and the limitations of, this approach in Section 6. It is implemented in the Modest Toolset's modes simulator [17]. Using all applicable MA models of the QVBS with *non*-spurious nondeterminism (where nondeterminism being *spurious* means that any choices do not affect the properties of interest, i.e., an equivalent fully stochastic model exists), we present an extensive study of the efficiency of LSS for MA.

Previous work. This article extends upon our conference paper [22] presented at the 16th International Conference on Quantitative Evaluation of Systems (QEST 2019). We have expanded explanations throughout and incorporated a step-by-step introduction to modelling MA with Modest in Section 3 extracted from the larger tutorial in Reference [51]. We consolidated the information and experiments concerning probabilistic model checking in Section 4. We added two new sections: Section 5 presents partial-exploration approaches as classically used in planning and machine learning, focusing on a variant of BRTDP. Section 6 highlights the simulation perspective using LSS. Both of these sections include completely new experimental evaluations.

2 MARKOV AUTOMATA

The mathematical formalism of Markov automata [36, 38] provides nondeterministic choices as in **labelled transition systems** (**LTS**, also known with small variations as finite automata or Kripke structures, see, e.g., Reference [7]), discrete probabilistic decisions as in **discrete-time Markov chains (DTMC)**, and stochastic time as in CTMC. The relationships between these and other formalisms are visualised in Figure 1. The combination of DTMC and LTS leads to the model family of discrete-time **Markov decision processes (MDP)** [70] or probabilistic automata [76], where transitions of the form $s \xrightarrow{a} \mu$ offer in state s a (nondeterministic) decision option (or choice option) labelled by action a that is followed by a probabilistic decision of where to jump according to probability distribution μ. The conceptually closest model in continuous time is that of **continuous-time MDP (CTMDP)** [70], where action-labelled transitions are of the form $s \xrightarrow{a} e$ with e mapping states to rates. Such a transition indicates that probability mass flows from state s to state s' with rate $e(s')$ provided action a is chosen in state s. Markov automata instead

combine MDP and CTMC in an orthogonal manner by providing two types of transitions: $s \xrightarrow{a} \mu$ as in MDP, and $s \overset{\lambda}{\dashrightarrow} s'$ as in CTMC. Choice of action and stochastic timing are thus intertwined in CTMDP but separate in MA—thus the dashed line in Figure 1 from CTMDP to MA. We now define Markov automata formally and describe their semantics.

Preliminaries. We write $[a, b]$ for the real interval $\{x \in \mathbb{R} \mid a \leq x \leq b\}$, (a, b) for the real interval $\{x \in \mathbb{R} \mid a < x < b\}$, and analogously for half-open intervals. Given a set S, its powerset is 2^S. A (discrete) probability distribution over S is a function $\mu \colon S \to [0, 1]$ such that its support $spt(\mu) \overset{\text{def}}{=} \{s \in S \mid \mu(s) > 0\}$ is countable and $\sum_{s \in spt(\mu)} \mu(s) = 1$. We write $Dist(S)$ for the set of all probability distributions over S, and $\mu_1 \otimes \mu_2$ is the product distribution of μ_1 and μ_2 defined by

$$(\mu_1 \otimes \mu_2)(\langle s_1, s_2 \rangle) = \mu_1(s_1) \cdot \mu_2(s_2).$$

We refer to discrete random choices as *probabilistic* and to continuous ones as *stochastic*. We write $\{x_1 \mapsto y_1, \dots\}$ to denote the function that maps each x_i to y_i, and if necessary in some context, implicitly maps to 0 all x for which no explicit mapping is specified. Thus, we can, e.g., write $\{s \mapsto 1\}$ for the *Dirac distribution* that assigns probability 1 to s.

2.1 Definition

We now mathematically define the formalism of Markov automata as tuples containing their states, transitions, and all other relevant aspects.

Definition 2.1. A ***Markov automaton* (MA)** M is a tuple

$$M = \langle S, s_0, A, P, Q, rr, br \rangle,$$

where
- S is a finite set of *states* with *initial state* $s_0 \in S$,
- A is a finite set of *actions*,
- $P \colon S \to 2^{A \times Dist(S)}$ is the *probabilistic transition* function,
- $Q \colon S \to 2^{\mathbb{Q} \times S}$ is the *Markovian transition* function,
- $rr \colon S \to [0, \infty)$ is the *rate reward* function, and
- $br \colon S \times Tr(M) \times S \to [0, \infty)$ is the *branch reward* function.

The set of all *transitions* of M is $Tr(M) \overset{\text{def}}{=} \bigcup_{s \in S} P(s) \cup Q(s)$; it must be finite. We require that $br(\langle s, tr, s' \rangle) \neq 0$ implies $tr \in P(s) \cup Q(s)$. We say that M is *deadlock-free* if $\forall\, s \in S \colon P(s) \cup Q(s) \neq \varnothing$.

We also write $s \xrightarrow{a}_P \mu$ for $\langle a, \mu \rangle \in P(s)$ and $s \overset{\lambda}{\dashrightarrow}_Q s'$ for $\langle \lambda, s' \rangle \in Q(s)$, and omit the P and Q subscripts if they are clear from the context. In $s \overset{\lambda}{\dashrightarrow}_Q s'$, we call λ the *rate* of the Markovian transition. We refer to every element of $spt(\mu)$ as a *branch* of $s \xrightarrow{a}_P \mu$; a Markovian transition has a single branch only (its target state). We define the *exit rate* of $s \in S$ as $E(s) = \sum_{\langle \lambda, s' \rangle \in Q(s)} \lambda$.

Example 2.2. Figure 2 shows two MA M_1 and M_2 without rewards. We draw probabilistic transitions as solid, Markovian ones as dashed lines. If a transition leads to a single target state, then we omit the intermediate probabilistic branching node. Thus, for $M_1 = \langle S, s_0, A, P, Q, rr, br \rangle$, we have five states in $S = \{0, 1, 2, 3, 4\}$, the initial state being $s_0 = 0$, two actions in $A = \{\mathsf{a}, \mathsf{c}\}$, two probabilistic transitions in $P = \{0 \mapsto \{\langle \mathsf{a}, \{1 \mapsto 0.5, 2 \mapsto 0.5\} \rangle, \langle \mathsf{c}, \{3 \mapsto 1\} \rangle\}\}$, and two Markovian transitions in $Q = \{1 \mapsto \{\langle 2, 4 \rangle\}, 3 \mapsto \{\langle 2, 4 \rangle\}\}$, both with rate 2.

Intuitively, the semantics of an MA is that, in state s, (1) the probability to take Markovian transition $s \overset{\lambda}{\dashrightarrow} s'$ and move to state s' within t model time units is $\lambda / E(s) \cdot (1 - \mathrm{e}^{-E(s) \cdot t})$, i.e., the residence time in s follows the exponential distribution with rate $E(s)$ and the choice of transition is probabilistic, weighted by the rates; and (2) at any point in time, a probabilistic transition $s \xrightarrow{a} \mu$ can be taken with the successor state being chosen according to μ. An MA thus resolves some choices

probabilistically (the choice of successor state of a probabilistic transition, the choice among Markovian transitions) or stochastically (the choice of residence time), while other choices are left open as *nondeterministic* (the timing of probabilistic transitions, and the choice among multiple available probabilistic transitions). Due to the presence of nondeterminism, an MA itself does not induce a probability measure over its possible behaviours. We refer the interested reader to, e.g., [55] for a complete formal definition of this semantics.

An MA without Markovian transitions is an MDP; it is a DTMC if in addition P maps each state to a singleton set. An MA without probabilistic transitions is a CTMC.

2.2 Parallel Composition

The co-existence of action-labelled probabilistic transitions of the form $s \xrightarrow{a} \mu$ and of Markovian transitions of the form $s \overset{\lambda}{\rightarrowtail} s'$ separates actions from timing. It enables parallel composition operators with action synchronisation for MA without the need to prescribe an ad-hoc operation for combining rates (as would be necessary to compose CTMC or CTMDP).

Definition 2.3. Given two MA

$$M_i = \langle S_i, s_{0_i}, A_i, P_i, Q_i, rr_i, br_i \rangle,$$

$i \in \{1, 2\}$, a finite set A of actions, and a *synchronisation relation*

$$sync \subseteq (A_1 \uplus \{\bot\}) \times (A_2 \uplus \{\bot\}) \times A,$$

their parallel composition is

$$M_1 \parallel M_2 \stackrel{\text{def}}{=} \langle S_1 \times S_2, \langle s_{0_1}, s_{0_2} \rangle, A, P, Q, rr, br \rangle,$$

where P is the smallest function that satisfies the inference rules

$$\frac{s_1 \xrightarrow{a_1}_{P_1} \mu \qquad \langle a_1, \bot, a \rangle \in sync}{\langle s_1, s_2 \rangle \xrightarrow{a}_P \mu \otimes \{s_2 \mapsto 1\}} \ (prob_1) \qquad \frac{s_2 \xrightarrow{a_2}_{P_2} \mu \qquad \langle \bot, a_2, a \rangle \in sync}{\langle s_1, s_2 \rangle \xrightarrow{a}_P \{s_1 \mapsto 1\} \otimes \mu} \ (prob_2)$$

$$\frac{s_1 \xrightarrow{a_1}_{P_1} \mu_1 \qquad s_2 \xrightarrow{a_2}_{P_2} \mu_2 \qquad \langle a_1, a_2, a \rangle \in sync}{\langle s_1, s_2 \rangle \xrightarrow{a}_P \mu_1 \otimes \mu_2} \ (prob_{sync}),$$

Q is the smallest function that satisfies the inference rules

$$\frac{s_1 \overset{\lambda}{\rightarrowtail}_{Q_1} s_1'}{\langle s_1, s_2 \rangle \overset{\lambda}{\rightarrowtail}_Q \langle s_1', s_2 \rangle} \ (mar_1) \qquad \frac{s_2 \overset{\lambda}{\rightarrowtail}_{Q_2} s_2'}{\langle s_1, s_2 \rangle \overset{\lambda}{\rightarrowtail}_Q \langle s_1, s_2' \rangle} \ (mar_2),$$

and for all states $\langle s_1, s_2 \rangle$, we have $rr(\langle s_1, s_2 \rangle) = rr_1(s_1) + rr_2(s_2)$. Function br sums the values of br_1 and br_2 for the combinations of branches in synchronisation (inference rule $prob_{sync}$), and otherwise preserves the original branch rewards.

Rules $prob_1$ and $prob_2$ allow the individual MA to proceed independently of each other if allowed by *sync*; rule $prob_{sync}$ covers the case where both automata synchronise on a pair of actions as determined by the *sync* relation. Rules mar_1 and mar_2 state that Markovian transitions are always performed independently. An element of *sync* is a *synchronisation vector*; we also write $\langle a_1, a_2 \rangle \mapsto a$ for vector $\langle a_1, a_2, a \rangle$. This form of parallel composition can be generalised to more than two automata in the straightforward way with longer synchronisation vectors. It is very flexible, allowing, in particular, the traditional CCS-style binary and CSP-style multi-way synchronisation patterns [57, 69] to be encoded. Originally established by CADP [39], it is today used for MA in the JANI format [18]. We refer to a general parallel composition of several MA as a *network* of MA.

Example 2.4. Figure 2 includes the parallel composition of the example MA M_1 and M_2, where we write nm for state $\langle n, m\rangle$. The two automata synchronise on the shared actions a and c, i.e., we have $sync = \{\langle \mathsf{a}, \mathsf{a}\rangle \mapsto \mathsf{a}, \langle \bot, \mathsf{b}\rangle \mapsto \mathsf{b}, \langle \mathsf{c}, \mathsf{c}\rangle \mapsto \mathsf{c}\}$.

2.3 Semantics

We defined MA as *open* systems [14]: probabilistic transitions can interact with, wait for, and be blocked by other MA in parallel composition. For verification, we make the usual *closed system* and *maximal progress* assumptions: probabilistic transitions face no further interference and take place without delay. If multiple probabilistic transitions are available in a state, however, then the choice between them remains nondeterministic. Since the probability that a Markovian transition is taken in zero time is 0, the maximal progress assumption allows us to remove all Markovian transitions from states that *also* have a probabilistic transition. In such *closed MA*, we can thus distinguish between Markovian states (where $P(s) = \emptyset$) and probabilistic states (where $Q(s) = \emptyset$). The behaviour of a closed, deadlock-free MA M is defined via its paths:

Definition 2.5. Let M be a closed, deadlock-free MA as above. A *path* π of M is an infinite sequence,

$$\pi = s_0\ t_0\ tr_0\ s_1 \ldots \in (S \times [0, \infty) \times Tr(M))^\omega,$$

such that, for all $i \in \{0, \ldots\}$, we have
— $tr_i \in P(s_i) \cup Q(s_i)$,
— $Q(s_i) = \emptyset$ implies $t_i = 0$,
— $tr_i = \langle a, \mu\rangle \in P(s_i)$ implies $\mu(s_{i+1}) > 0$, and
— $tr_i = \langle \lambda, s'\rangle \in Q(s_i)$ implies $s' = s_{i+1}$.
$\Pi(M)$ is the set of all paths of M. We write $\Pi_{fin}(M)$ for the set of all *path prefixes* π_{fin} ending in a state. The last state of π_{fin} is denoted $last(\pi_{fin})$. Let $\pi_{\leq j} \stackrel{\text{def}}{=} s_0\ t_0 \ldots s_j$. The *duration* $\mathrm{dur}(\pi_{fin})$ of a path prefix is the sum of its residence times t_i. A path's *reward* is

$$\mathrm{rew}(\pi) \stackrel{\text{def}}{=} \sum_{i=0}^{\infty} t_i \cdot rr(s_i) + br(s_i, tr_i, s_{i+1});$$

it may be ∞, and is defined analogously for prefixes (where it is always finite).

A path comprises states s_i, times t_i spent in s_i, and transitions tr_i taken from s_i to s_{i+1}. It is a resolution of all nondeterministic, probabilistic, and stochastic choices. To define a probability measure, we resolve nondeterminism only:

Definition 2.6. Let M be a closed, deadlock-free MA as above. A (deterministic, history-dependent) *scheduler* is a function

$$\sigma \colon \Pi_{fin}(M) \to Tr(M)$$

such that $\forall \pi_{fin} \in \Pi_{fin}(M) \colon \sigma(\pi_{fin}) \in P(last(\pi_{fin})) \cup Q(last(\pi_{fin}))$. We write $\mathfrak{S}(M)$ for the set of all schedulers of M. A *time-dependent* scheduler is in $S \times \mathbb{R} \to Tr(M)$; a *memoryless* scheduler is in $S \to Tr(M)$. Given a time bound $b \in [0, \infty)$, every time-dependent scheduler σ_t defines a corresponding scheduler σ by

$$\sigma(\pi_{fin}) = \sigma_t(\langle last(\pi_{fin}), b - \mathrm{dur}(\pi_{fin})\rangle)$$

for all path prefixes π_{fin}. Since it will be used for time-bounded properties (see below), its decisions for negative values of $b - \mathrm{dur}(\pi_{fin})$—i.e., when the time bound has expired—are in practice irrelevant. Every memoryless scheduler σ_{ml} similarly defines a corresponding scheduler σ by

$$\sigma(\pi_{fin}) = \sigma_{ml}(last(\pi_{fin})).$$

A *randomised scheduler* is a function

$$\sigma_{rand} \colon \Pi_{fin}(M) \to Dist(Tr(M))$$

such that $\forall \pi_{fin} \in \Pi_{fin}(M) \colon spt(\sigma(\pi_{fin})) \subseteq tr \in P(last(\pi_{fin})) \cup Q(last(\pi_{fin}))$. Every scheduler σ defines a corresponding randomised scheduler σ_r by, for all path prefixes π_{fin},

$$\sigma_r(\pi_{fin}) = \{\sigma(\pi_{fin}) \mapsto 1\}.$$

Randomised schedulers are the most general type of schedulers; they choose a history-dependent probability distribution over the available transitions in every state instead of deterministically picking one of the transitions. However, we define schedulers as deterministic by default, since randomised schedulers are not needed for the types of properties we mainly consider in this article (but rather for, e.g., multi-objective problems [71]). The only randomised scheduler we use is the *uniform scheduler* σ_U, defined by

$$\sigma_U(\pi_{fin}) = \left\{ tr \mapsto \frac{1}{|P(last(\pi_{fin})) \cup Q(last(\pi_{fin}))|} \mid tr \in P(last(\pi_{fin})) \cup Q(last(\pi_{fin})) \right\},$$

i.e., it is the randomised memoryless scheduler that resolves nondeterminism in a state s by uniformly at random selecting one of the transitions in $P(s)$.

We note that CTMDP with "early" schedulers [73], which select one fixed action for every path prefix, can be encoded as closed MA. Since actions and stochastic time are combined in CTMDP, their schedulers could also choose different actions depending on how much time has elapsed in a state (as long as a transition with their chosen action is not yet taken): these are the more general "late" schedulers, which have no direct counterpart in MA.

2.4 Properties

If we "apply" a scheduler to an MA, then it removes all nondeterminism, and we are left with a fully stochastic process whose paths can be measured and assigned probabilities according to the rates and distributions in the (remaining) MA. Formally, these probability measures over sets of measurable paths are built via cylinder sets; we refer the interested reader to, e.g., Reference [55] for a fully formal definition. For all of the following types of properties, we are interested in the maximum (supremum) and minimum (infimum) values when ranging over all schedulers $\sigma \in \mathfrak{S}(M)$:

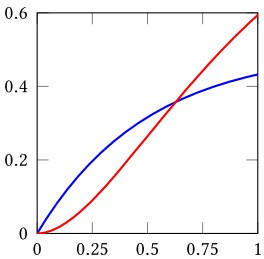
Reachability probabilities: Given goal states $G \subseteq S$, compute the probability of the set of paths that include a state in G. Memoryless schedulers suffice to achieve optimal results (i.e., the maximum and minimum probabilities).

Time-bounded reachability: Additionally restrict to paths where the duration of the prefix to the first state in G is below a bound $b \in [0, \infty)$. Time-dependent schedulers suffice to obtain the optimal values for time-bounded reachability properties.

Expected accumulated rewards: Compute the expected value of the random variable that assigns to π the value $\mathrm{rew}(\pi_{fin})$ with π_{fin} being the shortest prefix of π with a state in G. If measurable sets of paths exist that do not reach a state in G, then this value is not well-defined. The standard definition, which we follow, is to consider such paths to have "infinite reward"; as a consequence, the maximum (minimum) expected accumulated reward is ∞ if the minimum (maximum) probability to reach G is less than 1. Memoryless schedulers suffice.

Long-run average rewards: Compute the expected value of the random variable that assigns to path π the value $\lim_{i \to \infty} \mathrm{rew}(\pi_{\leq i})/\mathrm{dur}(\pi_{\leq i})$. Again, memoryless schedulers suffice to achieve optimal values here.

Example 2.7. Consider MA $M_1 \parallel M_2$ of Figure 2 and the probability to reach state $\langle 4, 4\rangle$ within 1 time unit. In state $\langle 0, 1\rangle$, we have to decide whether to choose action a or b. The optimal decision depends on the amount of time t that has passed in state $\langle 0, 0\rangle$. In the plot on the right, we show the probability of reaching state $\langle 4, 4\rangle$ within the time limit (y-axis) depending on the remaining time $1 - t$ (x-axis). The blue (initially upper) line represents the reachability probability for the memoryless scheduler that always chooses a and the red (initially lower) one is for the scheduler that always takes action b. A time-dependent scheduler can make better decisions than either of these two by determining the values of t for which a results in a higher probability than b and vice-versa. The optimal scheduler thus chooses a if and only if $1 - t \leq 0.63$ approximately.



2.5 Variables

We can extend MA with discrete *variables*: An MA with variables (MA^{V}) is an MA like in Definition 2.1 that additionally contains a finite set of variables. We call its states *locations*, its transitions *edges*, and their branches *destinations*. Every edge additionally has a *guard* and every destination has a set of *updates*. A guard is a Boolean expression over the variables that determines whether the edge is enabled, and a set of assignments modifies the values of the variables. Tools usually work with the semantics of an MA^{V} in terms of an MA: The MA^{V} M_V corresponds to the MA M with states $\langle \ell, v\rangle$, each consisting of a location ℓ of M_V and a valuation v that assigns a value to every variable. The transitions out of $\langle \ell, v\rangle$ are those edges out of ℓ in M_V whose guard is satisfied in v. The target state of a branch of a transition is $\langle \ell', v'\rangle$ with ℓ' the target location in M_V and v' obtained by executing the destination's assignments on v. Our parallel composition operator extends to MA with variables by using the conjunction of guards and the union of assignments for synchronising transitions. If we allow variables to be shared between MA^{V}, then parallel composition does not distribute over semantics; we need to compose the MA^{V} before converting them to MA.

2.6 Our Tools for MA

The Modest Toolset [49] is a comprehensive suite of tools for quantitative modelling and verification. Its primary input languages are Modest [10, 47], which we introduce in more detail in Section 3 below, and the Jani model interchange format [18]. MA are supported in the toolset's mosta, moconv, mcsta, and modes tools. mosta visualises the symbolic semantics of models (i.e., of networks of MA^{V} before and after parallel composition as shown throughout Section 3.1) and is useful for model debugging. moconv transforms models between modelling languages (it can, e.g., convert Modest to Jani) and performs syntactic rewriting and optimisations. mcsta is a fast [19, 46] explicit-state model checker that can use secondary storage to alleviate state space explosion [50]; we present and evaluate its MA-specific algorithms in Sections 4.1 and 4.2 and its implementation of BRTDP for partial exploration in Sections 5.1 and 5.2. modes [17] is a statistical model checker with automated rare event simulation capabilities. It implements the **lightweight scheduler sampling approach (LSS)** [66] for nondeterministic models, including MA [31]. We explain and study LSS for MA in Section 6. The Modest Toolset is written in C#, works cross-platform on 64-bit Linux, Mac OS, and Windows systems, and is freely available at modestchecker.net. All its tools share a common infrastructure for parsing and syntactic transformations. mcsta and modes additionally build on the same state space exploration engine that compiles models to bytecode at runtime for memory efficiency and performance.

3 MODELLING WITH MARKOV AUTOMATA

Tools for the automated analysis of MA like mcsta and modes need a syntax in which the model and the properties of interest are specified. As noted in Section 1, such a modelling language needs to provide a parallel composition operator (akin to the operator introduced in the previous section) such that large MA can be built from small specifications, and will typically support modelling with variables.

3.1 Modest for Markov Automata

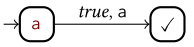
Modest [10, 47] is the modelling and description language for stochastic timed systems. At its core, it is a process algebra: it provides various operations such as parallel and sequential composition, parameterised process definitions, process calls, and guards to flexibly construct complex models out of small and reusable components. Its syntax, however, borrows heavily from commonly used programming languages, and it provides high-level conveniences such as loops and an exception handling mechanism. As such, Modest tends to be more verbose than classic process algebras but also more readable and beginner-friendly. To specify complex behaviour in a succinct manner, Modest provides variables of standard basic types (e.g., `bool`, `int`, or bounded `int`), arrays, and user-defined recursive datatypes akin to functional programming languages. Its syntax for expressions is aligned with C-like programming languages for ease of use.

Let us now introduce the Modest language syntax step-by-step by using it to model our example MA shown in Figure 2, starting with M_1. Modest models are structured into *processes*, with each process consisting of *declarations* and a *behaviour*. The declarations introduce all named objects like actions, variables, exceptions, nested processes, and so on, that are available for use in the behaviour and inside nested processes. A process' behaviour defines an MA with those variables.[1] To model M_1 as a Modest process, we thus start by declaring the actions and a Boolean variable to later distinguish between states 1 and 2:

```
action a, c;
bool f = false; // to distinguish between states 1 and 2
```

The simplest behaviour in Modest is to perform a (previously declared) action:

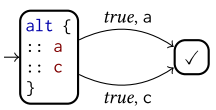
```
a
```



Semantically, this behaviour represents the MA with variables shown above on the right, where the one edge has guard expression *true*. Every location ℓ is uniquely identified by a behaviour such that the MA with ℓ as its initial location is the semantics of the behaviour. The checkmark ✓ is a special behaviour called *successful termination* that is not part of the syntax of Modest, and whose semantics is a state with no outgoing edges. It receives special treatment by several other Modest constructs. Modest also contains a `stop` construct with the same semantics but without the special treatment.

Initially, automaton M_1 offers a choice between two probabilistic transitions. The `alt` construct combines multiple behaviours into a nondeterministic choice between them, thus the initial choice in M_1 can be represented as follows:

```
alt {
:: a
:: c
}
```

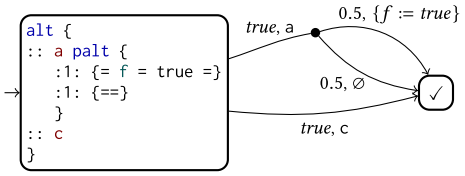


[1] Actually, the semantics of Modest [47] is defined in terms of **stochastic hybrid automata (SHA)**, of which MA are a special case; we restrict to that case in this article.

The semantic effect of the `alt` construct is simply to merge the initial states of the semantics of its child behaviours, the start of each of which is indicated by `::`. Note that both edges lead to the same location here; this is because the semantics of both behaviours `a` and `c` end in the identical location ✓.

Now, in M_1, the transition labelled `a` actually has two branches. The branching of probabilistic transitions can be represented in Modest with the `palt` construct. Since it does not create a new transition, but only defines branches, it has to be prefixed by the transition's action:

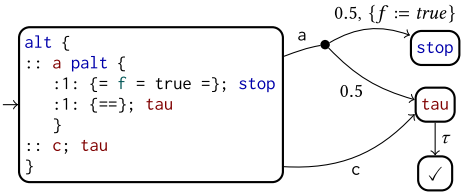
```
alt {
:: a palt {
   :1: {= f = true =}
   :1: {==}
   }
:: c
}
```



Probabilities are specified as *weights* between colons `:`, i.e., the actual probability in the semantics is calculated as the given weight divided by the sum of all weights in the `palt` construct. The assignments for every branch are specified in `{= =}` blocks, and they are executed *atomically*, so, e.g., the assignment block `{= x = y, y = x =}` performs an in-place swap of variables `x` and `y`. To create an edge labelled `a` with a single destination and assignments u, we can omit the `palt` and just write `a {= u =}`. Observe that, in the semantics of our example above, all destinations still lead to the same location. However, the semantics of this MA^{V} contains two states in location ✓: one where `f` is *true*, which is the target of the branch for the uppermost destination, and one where it is *false*. We will from now on omit *true* guards and empty assignment sets in MA^{V}.

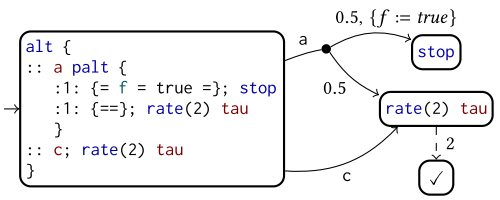
Continuing to model M_1 in Modest, we now add the Markovian transitions to state 4. We need two new constructs: for sequential composition, and for rates. First, the semantics of the sequential composition construct $P; Q$, for two behaviours P and Q, is to first behave like P, and upon successful termination of P (i.e., upon reaching location ✓), behave like Q. We thus get the following:

```
alt {
:: a palt {
   :1: {= f = true =}; stop
   :1: {==}; tau
   }
:: c; tau
}
```



`tau` is the predefined *silent action*, which does not take part in synchronisation (i.e., in a binary parallel composition, it is governed by synchronisation vectors $\langle \tau, \bot \rangle \mapsto \tau$ and $\langle \bot, \tau \rangle \mapsto \tau$ and cannot occur in any other vectors). To turn the τ-labelled probabilistic edge into a Markovian one, we simply specify rates:

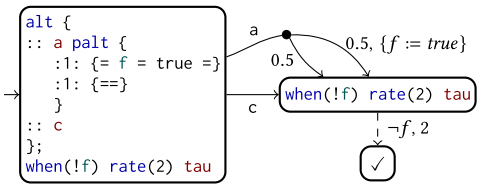
```
alt {
:: a palt {
   :1: {= f = true =}; stop
   :1: {==}; rate(2) tau
   }
:: c; rate(2) tau
}
```



The `rate` construct is a recent addition to Modest with the specific purpose of allowing MA models to be expressed in a convenient and natural way. Modest enforces the separation of probabilistic and Markovian transitions by requiring edges for which a rate is specified to have action `tau`. If this restriction is not met, then the model is recognised as a CTMDP.

In the model above, the behaviour `rate(2) tau` occurs twice. We can eliminate this duplication by moving it out of the `alt` construct. At this point, let us also introduce the `when` construct to specify guards: instead of using `stop` to make the model deadlock in the upper destination, we use `f` to cause the deadlock in the semantics of the MA$^{\text{V}}$. The result is

```
alt {
:: a palt {
   :1: {= f = true =}
   :1: {==}
   }
:: c
};
when(!f) rate(2) tau
```



The semantics of the MA$^{\text{V}}$ on the right above is *almost* isomorphic to M_1; the difference is that states 1 and 3 are merged, since they have the same behaviour.

In Figure 3, we show the full Modest model of the parallel composition of MA M_1 and M_2 of Figure 2. It includes the model that we built for M_1 above as the body of the named process `M1`. Such processes can have parameters (specified between the parentheses in the declaration, not shown here) and local variables. A process call like `M1()` behaves exactly like the behaviour of `M1`, with all formal parameters taking the values of the actual arguments, and new variable instances for all parameters and local variables to separate them from any other calls to `M1`. The semantics of the parallel composition construct `par` is the n-ary parallel composition of its child behaviours, with synchronisation vectors that implement CSP-style synchronisation for all actions declared with the `action` keyword (in this model, that is the vectors given in Example 2.4), and as described above for τ. The model also declares two properties, `P_Min` and `P_Max`, which ask for the probability to reach state $\langle 4, 4\rangle$—made observable via the global variable `succ`, which is of bounded integer type[2] with range $\{0, 1, 2\}$—within time bound `B` akin to Example 2.7. `B` is an open parameter for which values can be specified at verification time.

We have now covered most of the basic constructs of Modest. There are many features not used in this small model; we refer the interested reader to our extended tutorial on Modest for MA [51], where we continue by tackling two different realistic case studies with Modest in a step-by-step fashion. Additional Modest MA models are also part of the **Quantitative Verification Benchmark Set (QVBS)** [53] at qcomp.org.

3.2 Alternative Modelling Languages

Modest is not the only modelling language for MA. We briefly contrast it to the currently available alternative modelling languages with support for MA here.

State space files for Imca. The first MA-specific algorithms were implemented in the Imca tool [42]. Its only input language is a text-based explicit state space format as illustrated for our example of $M_1 \parallel M_2$ in Figure 6. This is clearly not a useful modelling language, but a format to be automatically generated by tools.

Guarded commands with Storm. The Storm model checker [34] provides many input languages, with MA being supported through a state space format similar to Imca's, via Jani, as the semantics of generalised stochastic Petri nets [37] in GreatSPN format [1], and through an extension

[2]MA model checking requires finite state spaces; thus all variables must be bounded. Indicating the bounds in the types is good practice to avoid accidentally creating infinite-state models and may improve performance, but it is not a requirement for mcsta as long as only finitely many distinct values are ever assigned to the variables occurring in the model.

```
const real B;
int(0..2) succ = 0;

action a, b, c;

property P_Min = Pmin(<>[T<=B](succ == 2));
property P_Max = Pmax(<>[T<=B](succ == 2));

process M1()
{
   bool f = false;

   alt {
   :: a palt {
      :1: {==}
      :1: {= f = true =}
      }
   :: c
   };
   when(!f) rate(2) {= succ++ =}
}

process M2()
{
   rate(2) tau;
   alt {
   :: a {= succ++ =}
   :: b; rate(2) tau; c {= succ++ =}
   }
}

par {
:: M1()
:: M2()
}
```

Fig. 3. Modest model for $M_1 \parallel M_2$.

```
ma

const double B

module M1
  s1: [0..4];

  [a] s1=0 -> 0.5:(s1'=1)
              + 0.5:(s1'=2);
  [c] s1=0 -> 1:(s1'=3);
  <>  s1=1 | s1=3 -> 2:(s1'=4);
endmodule

module M2
  s2: [0..4];

  <>  s2=0 -> 2:(s2'=1);
  [a] s2=1 -> 1:(s2'=4);
  [b] s2=1 -> 1:(s2'=2);
  <>  s2=2 -> 2:(s2'=3);
  [c] s2=3 -> 1:(s2'=4);
endmodule

"P_Min": Pmin=? [F<=B (s1=4 & s2=4)];
"P_Max": Pmax=? [F<=B (s1=4 & s2=4)];
```

Fig. 5. Prism dialect supporting MA.

```
global succ:{0..2} = 0
DONE = done.DONE[]
M1  = a.psum(0.5 -> M1a[] ++ 0.5 -> DONE[])
      ++ c.M1a[]
M1a = <2>.setGlobal(succ, succ + 1)
      .DONE[]
M2  = <2>.(a.M2a[] ++ b.<2>.c.M2a[])
M2a = setGlobal(succ, succ + 1).DONE[]
init M1[] || M2[]
comm (a, a, a), (c, c, c)

reachCondition (succ = 2)
```

Fig. 4. MAPA process algebra.

```
#INITIALS
s00

#GOALS
s44

#TRANSITIONS
s00 !
* s01 2
s01 b
* s02 1
s01 a
* s14 0.5
* s24 0.5
s14 !
* s44 2
s02 !
* s03 2
s03 c
* s34 1
s34 !
* s44 2
```

Fig. 6. Imca state space format.

of the Prism guarded command language. We show our example in the latter in Figure 5. This is a very simple and small language that is easy to learn, however its only higher-level construct to structure and compose models is the parallel composition of its *modules*, which uses CSP-style synchronisation like Modest's `par` construct. The Prism tool also provides finer-grained control over synchronisation via the `system-endsystem` construct, but support for this feature is generally limited in other tools. The Prism language requires the modeller to explicitly encode a state machine to structure control flow, and has limited support for code reuse (via module renaming for behaviour and via formulas/labels for expressions). Overall, it makes a rather different

tradeoff between the simplicity of the language and the ability to structure large models compared to Modest.

Process algebra with Scoop. Mapa [79] is a dedicated process algebra for MA. It is supported by Scoop [79], which can linearise, reduce, and finally export Mapa models to Imca for verification. We show the example of M_1 and M_2 in Mapa in Figure 4. As a classic concise process algebra, Mapa tends to be very succinct but also difficult to read. Mapa models can be much more flexibly composed than Prism models, yet there is less syntactic structure than in Modest—although the languages conceptually share many operators. Mapa notably has a predefined *queue* datatype, and users can specify custom non-recursive datatypes.

Jani for tool interoperation. The Jani model interchange format [18] is designed to ease tool development and interoperation. It is Json-based and thus human-debuggable, but not intended as human-writeable. It represents networks of automata with variables symbolically. Since both the Modest Toolset and Storm support Jani, it is possible to, e.g., build MA models in the Modest language, export them to Jani with moconv, and then verify them with Storm. Likewise in the other direction, we can, e.g., create a Petri net with GreatSPN, convert to Jani with Storm, and analyse it with mcsta or modes. In this way, the most appropriate modelling language can be combined with the best analysis method and tool for every specific scenario. The Json-based syntax however is too verbose to display the example in Jani format in this article.

4 CHECKING MARKOV AUTOMATA

While the values for some classes of properties can be computed by means of algorithms originally designed for MDP, others need dedicated MA-specific algorithms. We here give a brief overview of the set of relevant algorithms implemented in the mcsta, Storm, and Imca tools followed by an experimental performance comparison.

4.1 Model Checking Algorithms

To calculate the values of untimed and expected-time properties, we can directly reuse MDP model checking algorithms. Whenever a property refers to time, e.g., by means of a time bound, however, we need dedicated algorithms for MA.

4.1.1 Untimed and Expected-Reward Properties. Like for CTMC, properties that do not refer to time, or only to expected times, can be computed on an MDP embedded in the Markov automaton. These properties include unbounded as well as branch reward-bounded reachability probabilities and expected accumulated rewards. For simplicity, we refer to all of these as "unbounded properties." For a given closed, deadlock-free MA $M = \langle S, s_0, A, P, Q, rr, br\rangle$, the embedded MDP is

$$\langle S, s_0, A \uplus \{\tau\}, P', \{s \mapsto \varnothing \mid s \in S\}, \{s \mapsto 0 \mid s \in S\}, br'\rangle,$$

with

$$P'(s) = \begin{cases} \{\langle \tau, \{s' \mapsto \frac{\sum_{\langle \lambda, s'\rangle \in Q(s)} \lambda}{E(s)} \mid s' \in S\}\rangle\} & \text{if } P(s) = \varnothing, \\ P(s) & \text{otherwise,} \end{cases}$$

and

$$br'(\langle s, tr, s'\rangle) = \begin{cases} \frac{rr(s) + \sum_{\langle \lambda, s'\rangle \in Q(s)} \lambda \cdot br(\langle s, tr, s'\rangle)}{E(s)} & \text{if } P(s) = \varnothing, \\ br(\langle s, tr, s'\rangle) & \text{otherwise;} \end{cases}$$

i.e., in every Markovian state, we replace all outgoing transitions by a single probabilistic transition such that every branch corresponds to one of the original Markovian transitions with the rates

interpreted as probability weights. The branch rewards are calculated based on the expected time spent in each Markovian state as given by its exit rate.

The portfolio of algorithms for unbounded properties includes all the standard exhaustive model checking algorithms designed for MDP [70], in particular, using **linear programming (LP)**, policy iteration, value iteration, interval iteration [8, 45], sound [72], and optimistic [52] value iteration. Standard value iteration and many LP solver-based approaches are "unsound" in the sense that they do not provide guarantees (such as ϵ-closeness to the true probability or value) on their results, while interval iteration, sound value iteration, and optimistic value iteration do. To combat the state space explosion problem of the exhaustive methods, the ***bounded real-time dynamic programming*** **(BRTDP)** [68] approach can be used for probabilities [13]. It attempts to explore only a small part of the state space that is sufficient to provide a lower and an upper bound on the result that are close enough. Its efficiency both in terms of runtime and in terms of memory reduction highly depends on the structure of the model; we study its behaviour on MA in more detail in Section 5.

Tool support. mcsta implements value iteration, LP, interval iteration, sound value iteration, and optimistic value iteration for expected rewards and unbounded reachability probabilities. It also provides BRTDP as in Reference [3] where simulations with the uniform scheduler are used to explore a part of the state space. After every batch of simulation runs, interval iteration is used on the explored part of the state space to compute bounds. Storm implements value and policy iteration, LP, interval iteration, sound value iteration, and a variant of BRTDP. It also provides algorithms to compute exact (rational) solutions using exact arithmetic, but they are currently limited to small models. Imca supports value iteration only.

4.1.2 Time-Bounded Reachability. Time-bounded properties pose one of the most challenging problems in MA model checking. Several algorithms with rather different characteristics are currently available for approximating time-bounded reachability probabilities: The **discretisation** approach [43] discretises the time horizon into small intervals, such that the MA will likely perform at most one Markovian transition within each interval. **Unif+** was first presented for CTMDP [23] and later extended to MA [41]. It is based on an approximation of the optimal time-bounded reachability probability over timed schedulers by using untimed schedulers. The **switch-step** algorithm [21] attempts to compute *switching points*: the points at which the optimal scheduler changes the decision for at least one state, as illustrated in Example 2.7. Finally, the **BRTDP** idea for time-bounded reachability properties on CTMDP [3] can be extended to MA straightforwardly: the simulation phase performs CTMC-style simulation for Markovian states and MDP-style simulation over probabilistic states. Time progresses only over Markovian states and the simulation stops whenever the time bound expires or a target state is reached. Resolution of nondeterminism is performed via the uniform scheduler. The analysis phase can be performed by any of the other algorithms for time-bounded analysis on MA. We will see in Section 5 that BRTDP works very well for time-bounded properties on several models.

Tool support. mcsta implements Unif+ and switch-step while Storm supports Unif+ and the discretisation approach. Both provide sound implementations of these algorithms (i.e., they guarantee ϵ-correct results). mcsta also implements a variant of BRTDP for MA as in Reference [3] that is sound. Imca implements only discretisation and uses unsound techniques for certain subproblems.

4.1.3 Long-run Average Rewards. There exist two approaches for computing long-run average rewards: one based on a reduction to a linear program [44], and a value iteration-based algorithm [24] that approximates the reward up to a user-specified (and guaranteed) precision. In both cases, first the long-run average reward is determined for each maximal end component, then the

end components are collapsed, and the overall result is computed as an expected reward value on the collapsed state space.

Tool support. mcsta and Storm implement both of the algorithms while Imca implements only the linear programming-based approach.

4.1.4 Other Verification Problems. We briefly summarise other MA verification problems, name the corresponding available algorithms, and mention where they are implemented.

Time-bounded expected rewards. extend the time-bounded reachability problem to rewards. The property represents the expected accumulated reward until a time bound is reached. Algorithmic support for this property is limited to the discretisation-based approach of Reference [44], which is implemented in Imca.

Resource-bounded rewards. generalise both time-bounded reachability and time-bounded expected rewards. A resource-bounded reward property represents the expected accumulated reward within a finite resource budget. The resource is formally represented by a second type of (branch or rate) reward in the model. The only algorithm available to date is presented in Reference [55], with no tool support.

Discounted rewards. Expected discounted reward properties ask for the expected total reward where rewards collected at a certain time point are discounted with a value, depending on this time point. For example, when dealing with income, discounted rewards allow to take inflation into account. Iterative algorithms for computing and approximating the value exist, such as policy and value iteration [25]. There is however no tool support so far.

Multi-objective tradeoffs. Multi-objective MA model checking allows finding a scheduler that is optimal for several objectives, rather than only one. The only algorithm available to date and implemented in Storm is presented in Reference [71]. It does not support the full range of properties, in particular, excluding long-run average and discounted rewards. For the underlying time-bounded analysis, it resorts to discretisation, which tends to not scale well (see Section 4.2).

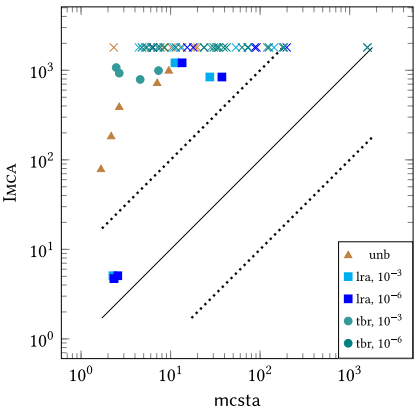
4.2 Model Checking Performance Comparison

The **Quantitative Verification Benchmark Set (QVBS)** [53] currently contains 18 MA models, specified in Modest, in Storm's extension of the Prism language for MA (cf. Section 3.2), as GreatSPN Petri nets, and as fault trees in the Galileo format [78]. For every model, there is also a Jani version. Most models have parameters (like B in our Modest example of Figure 3) that allow them to be scaled up from small to huge state spaces. We use most of these models to compare the performance and scalability of the model checking algorithms implemented in mcsta with Imca and Storm. We select parameters that make for challenging, but not impossible-to-check, state space sizes (up to a few millions of states). The models include variations of queueing systems, dependability models, scheduling problems, and security case studies. We excluded those models that only have spurious nondeterminism (i.e., those that are equivalent to a CTMC), and those that can be fully checked in just a few seconds for all of the parameter valuations given in the QVBS. Due to the absence of long-run average reward properties in most MA models of the benchmark set, we added sensible long-run average properties to most of the Modest models (which are easy to modify by hand, in contrast to Jani) to be able to do a meaningful performance comparison. They are mainly steady-state probabilities (i.e., the special case of a rate reward of 1 in some states and of 0 in all others) or properties describing the long-run average costs of running the modelled system.

All experiments were conducted on two servers with Intel Core i7-4790 processors and 16 resp. 32 GB of RAM running 64-bit Ubuntu Linux 18.04. We used mcsta version 3.0.145, Storm version

1.3.1, and IMCA version 1.6 beta. We keep the default values for all command line arguments of the tools unless we explicitly mention specific values. When we request a certain precision for results (with sound methods), we request absolute, not relative, precision. We show all results as scatter plots like the one below, with log-log axes. Every benchmark *instance*—a model, a valuation for its parameters, and a property to check—results in one point in these plots. A point $\langle x, y \rangle$ states that the runtime of the tool noted on the x-axis on one instance was x seconds while the runtime of the tool noted on the y-axis was y seconds. Thus points above the solid diagonal line indicate instances where the x-tool was faster; it was more than ten times faster (slower) on points above (below) the dotted line. We set the timeout to 30 min; a timeout is denoted by an "x" dot in the plots.

4.2.1 mcsta and IMCA. The plot on the right compares the runtime of mcsta and IMCA on time-bounded ("tbr"), long-run average ("lra"), and unbounded properties ("unb"). The input of IMCA is an explicit representation of a state space (cf. Section 3.2). Thus, before a model can be analysed with IMCA, the state space has to be fully explored, transformed into this format, and saved to disk. This takes additional time and memory. Models of a few kB in MODEST lead to IMCA files of several GB. We use mcsta to perform this transformation, which took up to 200 s on each of the benchmarks we selected for our experiments. The runtime presented for IMCA does not include the time to generate input models, but only the time it takes to load them into memory and analyse them. For mcsta, we include the time for state space exploration (from MODEST or JANI input). For all experiments, we chose the best runtime among all algorithms provided in each tool. For time-bounded properties, we set the precision to 10^{-3} and 10^{-6}. The same holds for long-run properties for mcsta but not for IMCA, since its command-line interface does not support setting the precision for these properties. For unbounded properties, we use the default parameters of both tools, including precision, since this again cannot be changed for IMCA.



We see that IMCA performs far worse than mcsta. This is despite the fact that the considered runtime does not include time for model generation and that its only algorithm for time-bounded properties is unsound (with unsound methods tending to be faster than sound ones [72]), while the one of mcsta is sound. The performance gap is likely due to IMCA only implementing the discretisation-based approach, which is known to be inefficient [21, 23], and not providing the most recent model checking algorithms for any of the property types.

4.2.2 mcsta and STORM. STORM, like mcsta, implements multiple and current algorithms. We thus present the results of this comparison in more detail. The runtimes for both tools include the time for state space exploration and for the numeric computations.[3]

[3]In principle, once it has explored a state space, a model checker can compute the values for multiple properties on this state space; mcsta indeed does so when asked to check multiple properties. The effort for state space exploration is then distributed over several properties, and the time needed for numeric computations may become more prominent. However not all tools support this, and due to the need to disable property-specific optimisations (such as cutting off all states that are only reachable via goal states when analysing a reachability property), performance comparisons in quantitative verification—such as those performed in the QComp competitions [19, 46]—typically keep to one property per run. We follow the same approach in this article and compute the value for one property only in each tool execution.

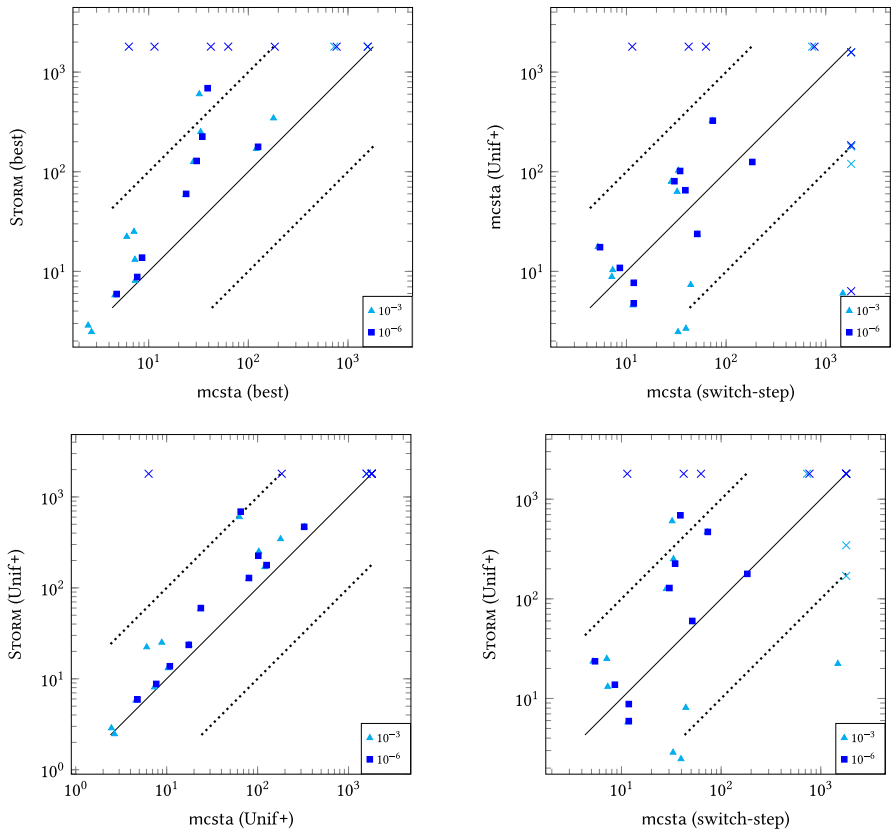


Fig. 7. Runtime of mcsta and Storm on time-bounded properties.

Time-bounded properties. Figure 7 summarises the comparison of time-bounded solvers in mcsta and Storm. We again run experiments with precision values 10^{-3} and 10^{-6} and configure the tools to produce sound results. In the top-left plot, we compare the best runtime for each tool among the algorithms that it implements; in the bottom-left plot, we compare mcsta's and Storm's implementations of Unif+. In both comparisons, mcsta achieves better runtimes than Storm. In particular, mcsta has no timeouts in the best-algorithm comparison. In the Unif+ comparison, mcsta and Storm both time out in some cases, yet whenever mcsta times out on a model, Storm does so, too (the "x" dot on the 45° line is actually a superposition of several such dots here). The two plots on the right compare the runtime of the switch-step implementation in mcsta with Unif+ in both tools. We do not compare to the discretisation algorithm for time-bounded properties implemented in Storm due to the consistent reports [21, 23] of its inefficiency (which we confirmed in Section 4.2.1 with Imca). We observe that neither Unif+ nor switch-step dominates the other, no matter which tool is used. This is because none of the two algorithms is strictly better than the other. Consider the top-right plot: it compares switch-step and Unif+ in mcsta and confirms the results presented in Reference [21] that the algorithms are good in complementary scenarios. There are cases where one of them times out while the other finishes quite quickly, and vice-versa. In particular, Unif+ performs somewhat better when a lower precision is required. Overall, the individual algorithms for time-bounded reachability in mcsta perform competitively, and especially when combined in a portfolio approach (i.e., using the best for each model, which could be done by running

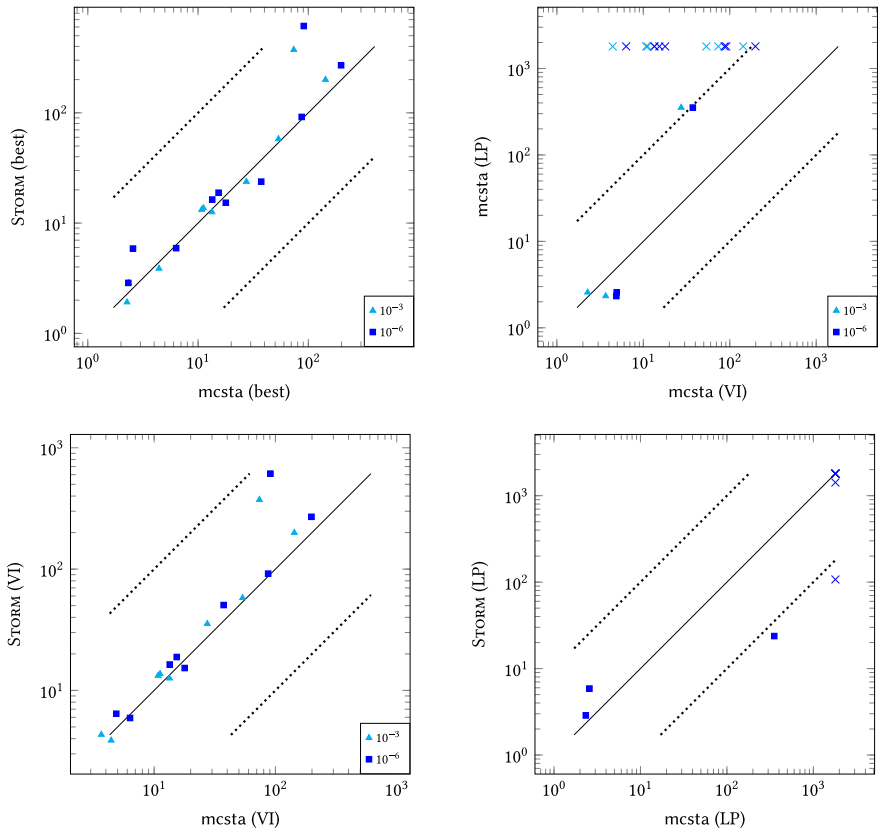
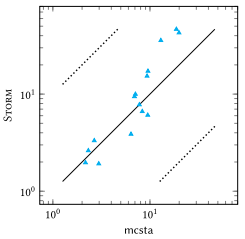


Fig. 8. Runtime of mcsta and Storm on long-run average reward properties.

both concurrently on a multi-core system), offer better performance and scalability than Storm overall.

Long-run average properties. Figure 8 summarises the comparison of algorithms for model checking long-run average properties in mcsta and Storm. For value iteration-based algorithms ("VI"), we run experiments on precision values 10^{-3} and 10^{-6}, and use only sound variations. For the linear programming-based approaches ("LP"), we set mcsta and Storm to use linear programming at *all* steps of the algorithm. The LP-based algorithms run with default parameters in both tools. For the top-left plot, we again chose the best runtime over the two algorithms for each tool. The LP-based approaches are not competitive: this can be seen from the three other plots. The bottom-right plot shows that they run out of time on most of the benchmarks in both mcsta and Storm. In contrast, the VI-based solutions in both tools terminate on the same benchmarks within the given time bound, as can be seen from the bottom-left and top-right plots. The exact reason for this is hard to extract. It may be possible that, when dealing with long-run properties, the LP-based approach itself is not as efficient as the one using VI, at least on existing benchmarks. Alternatively, it may be that the underlying LP algorithms or their implementations are not efficient. Overall, mcsta and Storm are roughly on par, albeit with mcsta having a few instances where it is significantly faster. The overall similarity is likely due to the set of implemented algorithms being exactly the same. We do notice, though, that specifically Storm's LP method appears to work better than mcsta's.

Unbounded properties. We finally add a small evaluation for unbounded properties. They can be checked via standard MDP algorithms and are thus not the focus of this article. An extensive evaluation of such properties for both mcsta and Storm was done for the QComp 2019 and 2020 tool competitions [19, 46]. The plot on the right confirms the QComp results of the two tools being competitive with no absolute winner.



5 VERIFICATION BY PARTIAL EXPLORATION

Whereas the algorithms evaluated in the previous section rely on a complete in-memory representation of the MA corresponding to a model, i.e., on an exhaustive state space exploration, various techniques rooted in learning and probabilistic planning attempt to compute our values of interest based on partially-explored state spaces only. Probabilistic planning, in particular, is very similar to probabilistic model checking at its core. Its focus is however on finding (good or optimal) schedulers in MDP instead of actually computing values, and probabilistic planning algorithms crucially employ heuristics to try to avoid exploring the entire state space. Since we aim for sound results, the heuristics-based BRTDP [68] technique is of particular interest: It computes both a lower and an upper bound on the value—in the same way as interval iteration—while considering a sampled subset of all states only. The underlying idea was recently transferred to the probabilistic model checking setting [13], and adapted for use with time-bounded properties on CTMDP [3]. In particular, BRTDP as proposed originally samples a random path through the model and then updates values for the states visited on the path ("asynchronous backpropagation"). Transferring the asynchronous approach to time-bounded properties on MA, however, would usually negate the memory savings of partial exploration [3]. The adaptation for CTMDP thus still samples random paths, but simply stores the (ever-growing) sub-CTMDP visited by all sampled paths. It then periodically calls one of the algorithms of Section 4.1.2 on an overapproximating sub-CTMDP, where all "boundary states"—the *successors* of states visited on paths that have not themselves been part of a path yet—are treated like goal states, and on an underapproximating one, where the boundary states are treated like non-goal deadlock states. The result is an interval bounding the actual value, and the algorithm can terminate once this interval is small enough. We refer the interested reader to Reference [3] for all technical details of this approach.

5.1 BRTDP for MA

We implement a straightforward adaptation of the BRTDP-based technique of Reference [3] for MA in mcsta: to sample paths, we perform CTMC-style simulation for Markovian states, keeping track of the time spent, and MDP-style simulation using the uniform scheduler to resolve nondeterminism for probabilistic states. The periodic analysis phase currently uses Unif+ with interval iteration, but can easily be changed to use switch-step or other sound value iteration techniques instead. We show the corresponding pseudocode as Algorithm 1. We use a **pseudo-random number generator (PRNG)** to simulate probabilistic choices and stochastic delays. We denote by $\mathcal{U}(\mu)$ the pseudo-random selection of a value from $spt(\mu)$ according to a value sampled using PRNG $\mathcal{U}$ and the probabilities in $\mu \in Dist(S)$ for some given set S. We use the same notation to denote the pseudo-random selection of a real number from a continuous probability distribution when $\mu : \mathbb{R} \to [0, 1]$ characterises its cumulative distribution function. Our pseudocode is slightly simplified compared to the actual implementation in mcsta; for example, we assume the MA to be closed and deadlock-free as usual but also non-Zeno (so that the time bound or a goal state will eventually be reached), and we assume $s_0 \notin G$. For simplicity of presentation, we give a flat MA

ALGORITHM 1: BRTDP adapted for time-bounded probabilities in MA as implemented in mcsta

Input: MA $\langle S, s_0, A, P, Q, rr, br\rangle$, goal set $G \subseteq S$, time bound $b > 0$, error $\epsilon > 0$, PRNG $\mathcal{U}_{\text{pr}}$

1 $S_{done} := \emptyset, S_? := \{s_0\}, S_G := \emptyset$ *// fully explored, seen unknown, and seen goal states*
2 $[l, u] = [0, 1]$ *// current interval bounding the result*
3 **while** $u - l > 2 \cdot \epsilon$ **do** *// continue until error bound is met*
// Simulation phase
4 **for** $i \in \{1, \ldots, 100\}$ **do** *// sample 100 random paths*
5 $s := s_0, t := 0$ *// start a new path from the initial state*
6 **while** $s \notin G \wedge t < b$ **do** *// end path on goal state or time bound*
7 $S_{done} := S_{done} \cup \{s\}$ *// successors of s will be fully explored below*
8 **if** $P(s) = \emptyset$ **then** *// state with Markovian transitions only:*
9 **if** $\forall\, s \xrightarrow{\lambda} s' \colon s = s'$ **then break** *// terminate if in self-loop state*
10 $S' := \bigcup_s \xrightarrow{\lambda} s'\{s'\}, S_? := S_? \cup S' \setminus G, S_G := S_G \cup S' \cap G$ *// store successor states*
11 $t := t + \mathcal{U}_{\text{pr}}(\{x \mapsto 1 - \mathrm{e}^{-E(s)\cdot x} \mid x \in [0, \infty)\})$ *// delay according to exit rate*
12 $s := \mathcal{U}_{\text{pr}}(\{s' \mapsto \frac{\lambda}{E(s)} \mid s \xrightarrow{\lambda} s'\})$ *// random next state according to the rates*
13 **else** *// state with probabilistic transitions:*
14 **if** $\forall\, s \xrightarrow{a} \mu \colon \mu = \{s \mapsto 1\}$ **then break** *// terminate if in self-loop state*
15 $S' := \bigcup_s \xrightarrow{a}_{\mu} spt(\mu), S_? := S_? \cup S' \setminus G, S_G := S_G \cup S' \cap G$ *// store successor states*
16 $\langle a, \mu\rangle := \mathcal{U}_{\text{pr}}(\{tr \mapsto \frac{1}{|P(s)|} \mid tr \in P(s)\})$ *// pick transition uniformly at random*
17 $s := \mathcal{U}_{\text{pr}}(\mu)$ *// select random next state according to μ*
// Analysis phase
18 $S_* := S_? \cup S_G, P_* := P|_{S_{done}}, Q_* := Q|_{S_{done}} \cup \{s \xrightarrow{1} s \mid s \in S_* \setminus S_{done}\}$ *// stored sub-MA only*
19 $[l, u_l] := \text{Unif+}(\langle S_*, s_0, A, P_*, Q_*, \emptyset, \emptyset\rangle, S_G, b, \epsilon)$ *// lower bound: use known goal states*
20 $[l_u, u] := \text{Unif+}(\langle S_*, s_0, A, P_*, Q_*, \emptyset, \emptyset\rangle, S_G \cup (S_? \setminus S_{done}), b, \epsilon)$ *// upper bound: add unknowns*
21 **return** $[l, u]$

$\langle S, s_0, A, P, Q, rr, br\rangle$ as input to the algorithm; however, an implementation will use a compact executable representation of a network of MA with variables instead—else the desired memory savings could obviously not be realised. In each simulation phase, we perform 100 runs (line 4); this criterion may be replaced by more effective checks that ensure some amount of new states being found. The approach works in the same way to approximate unbounded reachability probabilities if we drop the time bound check and replace Unif+ by standard embedded-MDP analysis algorithms for unbounded reachability. Note that it cannot work for expected-reward properties: To exclude the infinity cases, we cannot have states whose successors are not yet fully explored, since they might give rise to a path that avoids the goal with positive probability.

5.2 Performance and Scalability Comparison

BRTDP is useful in cases where the property under consideration does not require the full state space to be explored to achieve results with specified precision. In fact, the explored state space might be only a few percent of the full state space—but it may also be nearly the entire full state space. The contribution of BRTDP in combating state space explosion as well as in improving performance compared to exhaustive model checking as in Section 4.2 is thus highly dependent on the structure of the model and the property at hand. We thus performed an extensive investigation of its behaviour using all MA instances—combinations of a model, a valuation for its parameters (cf. Section 4.2), and a property to check—from the QVBS that satisfy the following criteria:

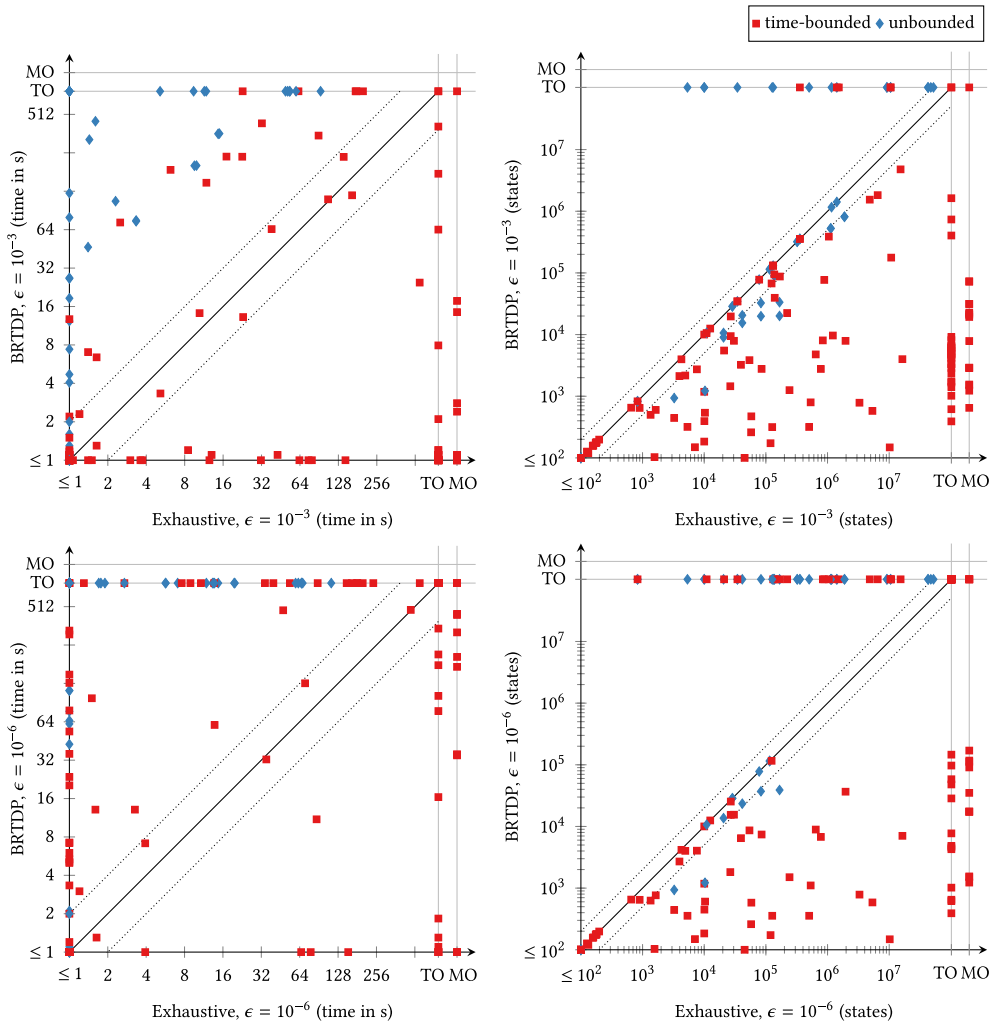


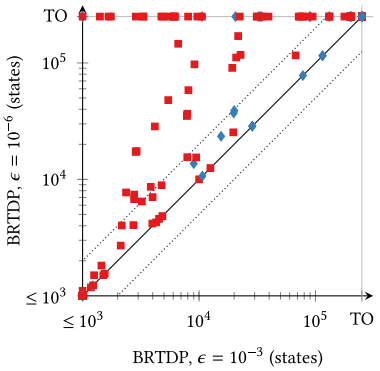
Fig. 9. Runtime and number of states stored in memory for BRTDP vs. exhaustive exploration.

— The property is an unbounded or time-bounded probabilistic reachability property; otherwise, BRTDP cannot be applied.
— The instance can be solved using at least one of exhaustive Unif+ or BRTDP with $\epsilon = 10^{-3}$ within 5 min on our benchmark system.

This results in a list of 18 different models for a total of 188 instances. We then executed mcsta version 3.1.75 on these instances on an Intel Core i7-4790 workstation (3.6–4.0 GHz, 4 cores) with 8 GB of memory running 64-bit Ubuntu Linux 18.04, using a timeout of 10 min, in four configurations: with exhaustive model checking and with BRTDP, in both cases using Unif+ and interval iteration, in combination with $\epsilon = 10^{-3}$ and $\epsilon = 10^{-6}$ (absolute error, as in Algorithm 1). We repeated the experiment three times for every instance and report the average of the results.

In Figure 9, we show scatter plots summarising the results of our experiments. Points below the solid diagonal lines correspond to instances where BRTDP was faster or explored fewer states than exhaustive model checking; the "TO" lines indicate timeouts and the "MO" lines indicate a tool

running out of memory. Points on the x- or y-axis of the two plots on the left-hand side result from instances where exhaustive model checking or BRTDP terminated in less than one second whereas the other took longer to complete, respectively. In terms of runtime, we clearly see that BRTDP is consistently slower to check unbounded reachability properties on these MA models compared to performing an exhaustive exploration followed by the same Unif+ analysis in one go. This is because BRTDP needs to explore relatively many states to obtain the desired level of precision on these instances (as shown by the plots for the numbers of explored states on the right in Figure 9). For time-bounded reachability properties with $\epsilon = 10^{-3}$, the situation is rather different: while both methods encounter timeouts on distinct sets of instances, BRTDP solves more instances, and also terminates much faster for many (with many pointy lying on the x-axis of the runtime plot and on the timeout line for exhaustive exploration). Looking at the number of states that need to be stored in memory, BRTDP offers very considerable savings in many cases. Once we increase the required precision to $\epsilon = 10^{-6}$, however, BRTDP becomes much less competitive in terms of runtime overall, though it still solves several instances where the exhaustive approach ran out of time or memory. In the plot on the right, we compare the number of states that BRTDP explores with $\epsilon = 10^{-3}$ and $\epsilon = 10^{-6}$. We see three classes of instances here: the ones where the number of explored states does not increase significantly (on or near the solid diagonal line), which are mainly the instances where BRTDP needs to explore (nearly) all states anyway and manages to do so; the ones where BRTDP with $\epsilon = 10^{-6}$ now times out (on the horizontal "TO" line); and most interestingly the set of instances that are still solved, but need moderately to significantly more states. We looked at the output of mcsta as BRTDP runs on some of the cases that turned difficult or impossible to solve with $\epsilon = 10^{-6}$, and observed that its simulation-based heuristics sometimes takes a very long time to explore a sufficient set of states; with this higher precision, states that are more unlikely to be reached suddenly become relevant.



While the scatter plots make for an easy visual comparison, they do not show how BRTDP behaves very differently on different models. In particular, some models contribute many instances (e.g., the dpm model with 36 instances resulting from 7 different parametrisations together with 5 properties) and others few (such as bitcoin-attack with a single instance); the former may then dominate the scatter plots. We thus provide in Table 1 a different view of our experimental results, now grouped by model. Columns "P" are for instances with unbounded and columns "P_t" with time-bounded reachability properties. We now see that, with $\epsilon = 10^{-3}$, BRTDP works extremely well on a small set of particular models with time-bounded properties: **cabinets**, **ftpp**, **hecs**, **mcs**, and **vgs**. These are five of the seven models in our benchmark set that represent dynamic fault trees [35] (the other two being **sf** and **sms**). Exhaustive exploration performs much worse, and in many cases fails entirely due to running out of time or memory, on these specific models. As expected, once we increase precision to $\epsilon = 10^{-6}$, BRTDP no longer works as well overall.

Overall, we find that BRTDP works very well for certain models, where it offers drastic reductions in memory usage and analysis runtime. By selecting *all* instances from the QVBS, we, in particular, include several very small state spaces where exhaustive model checking "wins" just by avoiding the overhead of having to perform simulations to obtain all the states. Nevertheless, we scaled up parameters (as provided in the QVBS) as far as *either* method allowed, and thus expect that we found most of the cases where BRTDP offers better scalability. We, in particular,

Table 1. Number of Instances Solved Successfully and Solved Faster for BRTDP vs. Exhaustive Exploration

	$\epsilon = 10^{-3}$								$\epsilon = 10^{-6}$							
	exhaustive				BRTDP				exhaustive				BRTDP			
	solved		faster		solved		faster		solved		faster		solved		faster	
model	P	P_t	P	P_t	P	P_t	P	P_t	P	P_t	P	P_t	P	P_t	P	P_t
bitcoin-attack		1		**1**		1		0		1		**1**		1		0
breakdown-q.	8		**8**		8		0		**8**		**8**		4		0	
cabinets		13		7		**18**		**11**		13		**12**		**14**		4
dpm	**24**	**11**	**24**	**11**	12	7	0	0	**24**	**4**	**24**	**4**	4	0	0	0
erlang	**3**	4	**3**	**3**	1	4	0	1	**3**	4	**3**	**3**	1	4	0	1
flexible-man.		6		**4**		6		2		**6**		**6**		4		0
ftpp		8		4		**16**		**12**		8		3		**14**		**10**
ftwc	2	2	**2**	**2**	2	2	0	0	2	2	**2**	**2**	2	2	0	0
hecs		3		1		**28**		**27**		3		2		**17**		**15**
jobs		**3**		**3**		2		0		**3**		**3**		2		0
mcs		5		2		**8**		**6**		**5**		**5**		2		0
polling-sys.	1	1	**1**	**1**	1	1	0	0	1	**1**	**1**	**1**	1	0	0	0
readers-wr.	8	**2**	**8**	**2**	8	2	0	0	**8**	**2**	**8**	**2**	0	0	0	0
reentrant-q.	1	1	**1**	**1**	1	1	0	0	1	0	**1**	0	1	0	0	0
sf		9		**8**		9		1		**9**		**9**		4		0
sms		18		**18**		18		0		18		**18**		18		0
stream	**4**	4	**4**	3	1	4	0	1	**4**	4	**4**	**4**	1	4	0	0
vgs		0		0		**2**		**2**		0		0		**1**		**1**
sum	***51***	*91*	***51***	***71***	*34*	***129***	*0*	*63*	***51***	*83*	***51***	***75***	*14*	***87***	*0*	*31*

notice that it performs much better for time-bounded properties overall; such properties offer an extra opportunity to truncate the state space early, since it usually takes more time to reach states at greater transition distances from the initial state. When the time bound is small, BRTDP can perform extremely well.

5.3 Other Partial-Exploration Approaches in Verification

Many partial-exploration heuristics for MDP have been developed in the probabilistic planning and machine learning communities. In particular, ***labelled real-time dynamic programming*** **(LRTDP)** [11] and ***Monte Carlo tree search*** **(MCTS)** [15] have recently seen applications and tool implementations in the area of verification.

Modest FRET-π LRTDP **(MFPL)** [59] is an implementation of LRTDP that uses the input language and model compilation infrastructure of the Modest Toolset. It supports the computation of unbounded minimum and maximum reachability probabilities and expected accumulated rewards on MDP. To make LRTDP work for the general MDP problems considered in model checking (where MDPs may, in particular, contain nontrivial end components), MFPL wraps it in FRET iterations [60], using the FRET-π variant [77]. However, LRTDP only converges to the value of interest from below, like standard value iteration; thus MFPL currently does not produce sound results. The tool has achieved promising results in the recent QComp competitions [19, 46].

MCTS combines a more systematic incremental unfolding of the state space into a search tree with sampling of paths. The search tree construction makes MCTS consider less probable paths earlier than BRTDP. It uses sampling to estimate the values of newly added states, and to guide the search. MCTS has been used very successfully with MDP; it can thus naturally be applied to untimed reachability in MA. We are not currently aware of a tool implementation supporting MA in this setting, though. Ashok et al. [2] recently showed that MCTS can be combined with the ideas of BRTDP in various ways to obtain different "hybrid" algorithms that provide sound results and perform better than BRTDP alone. MCTS does not apply directly to timed MA settings like checking time-bounded reachability properties due to the need to consider non-memoryless schedulers. MCTS has been extended to continuous-state settings with transitions that can sample from continuous probability distributions [26]. In such a framework, the remaining time t to the bound b could be encoded as a state variable. Whether approaches for very general continuous settings work well for the very specific sub-case of MA has not yet been investigated.

We also mention Uppaal Stratego [32], which explicitly synthesises a "good" scheduler before using it for a standard Monte Carlo simulation (or *statistical model checking*, see the next section) analysis. While it makes some use of symbolic data structures from timed automata model checking to reduce memory usage, it needs to fully explore the symbolic state space and ultimately its worst-case memory usage is linear in the number of states. Finally, the many classical memory-efficient sampling approaches (e.g., [58]) address discounted models only.

6 SIMULATING MARKOV AUTOMATA

In contrast to partial exploration techniques, **statistical model checking (SMC)** [56, 65, 80] aims to analyse formal models without storing any more than a constant number of states in memory. It is, in essence, Monte Carlo simulation of formal models: using pseudo-random number generators, we sample a large number n of *simulation runs* according to the probability distributions in the model and use them to statistically estimate the value of a given property. Simulation only ever needs two states in memory: the current state that we have reached on the path prefix sampled so far and the next state that we compute based on the current one when we take a transition. For illustration, let us consider a time-bounded reachability property with goal state set G and time bound $b \in [0, \infty)$ on an MA M without nondeterminism, i.e., where $|P(s)| \leq 1$ for all states s. We again assume M to be deadlock-free and closed, and also require it to be *non-Zeno*, i.e., for every cycle of only probabilistic transitions in the graph corresponding to M, the probability to remain in that cycle forever must be zero. The runs we generate for such a property are then path prefixes $\pi_{fin_1}, \ldots, \pi_{fin_n}$ of two kinds: for each $i \in \{1, \ldots, n\}$,

— either $last(\pi_{fin_i}) \in G$ and $\mathrm{dur}(\pi_{fin_i}) \leq b$, i.e., we stop simulation when we reach a goal state: then the run is "successful," and we define $\phi(\pi_{fin_i}) = 1$,

— or $\mathrm{dur}(\pi_{fin_i}) > b$, i.e., we stop simulation when we exceed the time bound: then $\phi(\pi_{fin_i}) = 0$.

In this setting, SMC terminates with probability 1, and $\hat{p}_n = \frac{1}{n} \sum_{i=1}^{n} \phi(\pi_{fin_i})$ is an unbiased estimator of the actual time-bounded reachability probability p. Similar schemes can be set up for the other types of properties. For unbounded probabilities and expected rewards, due to the absence of a time bound, we need a different criterion to stop unsuccessful simulation runs. There are various methods to achieve this, ranging from requiring that every run eventually reaches a goal state or a state whose only outgoing transitions are self-loops (which at first sight may be a strong requirement—but it is satisfied by many formal models made for model checking), to statistically detecting whether a run entered a bottom strongly connected component [27]. The modes simulator currently implements the former, and can be configured to also detect longer

deterministic loops. The choice of n depends on the desired statistical properties of $\hat{p}$. For instance, n can be determined such that a confidence interval constructed around $\hat{p}$ with confidence δ will have half-width w. For a detailed description of statistical methods and especially hypothesis tests for SMC, we refer the reader to Reference [74]. An overview of the methods implemented in modes is given in Reference [17, Table 1].

6.1 SMC Challenges

SMC is attractive as an alternative to model checking, because it entirely avoids the state space explosion problem: it needs to keep at most two states—the current and next states on the simulation run being generated—in memory at any time. However, it faces two challenges: rare events and nondeterminism. A rare event is a behaviour of very low probability, e.g., a time-bounded reachability probability on the order of 10^{-9}—which is rather common in models of highly reliable and safety-critical systems. In such a case, a meaningful estimate needs to have a small relative error: for a probability on the order of 10^{-9}, the error—e.g., the width of the confidence interval—should reasonably be on the order of 10^{-10}. In a standard Monte Carlo approach, this would require infeasibly many simulation runs, with the number of runs required increasing roughly quadratically as the desired error decreases. The field of *rare event simulation* [75] deals with this challenge, and modes implements an automated variant of the *importance splitting* rare event simulation approach [16]. We do not focus further on rare events in this article.

Nondeterminism, however, is a core feature of MA. As a simulation-based approach, SMC is fundamentally incompatible with nondeterministic choices: they give rise to an *optimisation* problem, whereas SMC solves *estimation* problems. If we try to simulate a nondeterministic MA, then we eventually reach a state s where $s \xrightarrow{a_1} s'$ and $s \xrightarrow{a_2} s''$ with $s' \neq s''$. At that point, simulation cannot continue, because we have no way to resolve the choice between the two transitions; we would need a scheduler—in particular, an optimal scheduler that gives rise to the minimum or maximum value that we are interested in. Many SMC tools appear to support nondeterministic models, e.g., Prism [63] and Uppaal smc [33], but use a single implicit scheduler that makes all choices randomly. Their results thus lie *somewhere* between the minimum and maximum. Such implicit resolutions are known to undermine the trustworthiness of simulation studies [9, 62]. While the partial exploration techniques investigated in Section 5 treat nondeterminism properly and also use simulation (to guide the partial exploration), their worst-case memory usage is always linear in the number of states; as we have seen with BRTDP in Section 5.2, they often still need to explore nearly the full state space.

6.2 Lightweight Scheduler Sampling

First implemented in Plasma for MDP, **lightweight scheduler sampling (LSS)** [66] is currently the only technique for SMC on nondeterministic models with undiscounted properties, as typically considered in formal verification, which preserves the constant memory usage feature that makes SMC so useful. It approximates the optimal schedulers, i.e., those that realise the maximum or minimum value for a property, in constant memory relative to the size of the state space by identifying a scheduler with a single (32-bit) integer. The basic idea of LSS is as follows:

(1) Randomly select m 32-bit integers. Each of them is a *scheduler identifier* σ.
(2) For each σ, perform standard SMC under the scheduler identified by σ.
(3) Return the maximum (or minimum) result and the corresponding σ.

During the simulation runs within step 2, when there is a choice between k transitions from state s, LSS concatenates the bit-vector representations of s and σ into $s.\sigma$, hashes the result into a single (32-bit) number $h = \mathcal{H}(s.\sigma)$, and picks the h mod k-th transition. The hash function $\mathcal{H}$ must be

ALGORITHM 2: Simulation with lightweight scheduler sampling for MA

Input: MA $\langle S, s_0, A, P, Q, rr, br\rangle$, goal set $G \subseteq S$, $\sigma \in \mathbb{Z}_{32}$, $\mathcal{H}$ uniform deterministic, PRNG $\mathcal{U}_{\text{pr}}$

1 $s := s_0$
2 **while** $s \notin G$ **do** *// break on goal state*
3 **if** $P(s) = \emptyset$ **then** *// state with Markovian transitions only:*
4 **if** $\forall\, s \xrightarrow{\lambda} s' \colon s = s'$ **then break** *// terminate if we are in a self-loop state*
5 $s := \mathcal{U}_{\text{pr}}(\{s' \mapsto \frac{\lambda}{E(s)} \mid s \xrightarrow{\lambda} s'\})$ *// select random next state according to the rates*
6 **else** *// state with probabilistic transitions:*
7 **if** $\forall\, s \xrightarrow{a} \mu \colon \mu = \{s \mapsto 1\}$ **then break** *// terminate if we are in a self-loop state*
8 $\langle a, \mu\rangle := (\mathcal{H}(\sigma.s) \bmod |P(s)|)$-th element of $P(s)$ *// deterministically select a transition*
9 $s := \mathcal{U}_{\text{pr}}(\mu)$ *// select random next state according to μ*
10 **return** $s \in G$

deterministic; then each σ defines a fixed memoryless scheduler. If $\mathcal{H}$ is also uniform (w.r.t. all bits of $s.\sigma$), then LSS uniformly samples among memoryless schedulers. We show as Algorithm 2 the pseudocode implementing the generation of a single simulation run as called n times in step 2, for MA and unbounded probabilistic reachability properties, using the simplest stopping criterion for termination of detecting self-loops. Note the similarities to the simulation part of Algorithm 1; the crucial differences lies in the transition selection in line 8. We use a single PRNG $\mathcal{U}_{\text{pr}}$ that we assume to have been initialised with some user-specified or randomly obtained seed (usually based on the current time).

Bounds, error accumulation, and efficiency. The results of LSS are lower bounds for maximum and upper bounds for minimum property values up to a specified statistical error. They can thus be used to, e.g., *disprove* the safety of a safety-critical system or *prove* schedulability of tasks under real-time requirements, but not the opposite. The accumulation of statistical error introduced by the repeated simulation experiments over m schedulers must also be accounted for, using, e.g., Šidák correction or the modified tests described in Reference [28].

The efficiency of LSS—how large an m we need to get a good approximation—depends on the probability of sampling a near-optimal scheduler. Since we do not know *a priori* what makes a scheduler optimal, we want to sample "uniformly" from the space of all schedulers. This at least avoids actively biasing against "good" schedulers. More precisely, a uniformly random choice of σ will result in a uniformly chosen (but fixed) resolution of all nondeterministic choices. Algorithm 2 achieves this naturally for MA and memoryless schedulers.

Two-phase and smart sampling. If, for fixed statistical parameters, SMC needs n runs on a DTMC or CTMC, then LSS needs significantly more than $m \cdot n$ runs on an MA to avoid error accumulation. The *two-phase* and *smart* sampling approaches implemented in modes can reduce this overhead. The former's first phase consists of performing n simulation runs for each of the m schedulers. The scheduler that resulted in the maximum (or minimum) value is selected, and independently evaluated once more with n runs to produce the final estimate. The first phase is a heuristic to find a near-optimal scheduler before the second phase estimates the value under this scheduler according to the required statistical parameters. Smart sampling [28] generalises this principle to multiple phases: it is parameterised by the number of initial schedulers $m = m_0$ and the per-phase budget of simulation runs n_0. In phase i, smart sampling performs $\lceil \frac{n_0}{m_i} \rceil$ simulation runs for each of the m_i schedulers, discards the "worst" half of the schedulers according to their current estimate,

and moves to phase $i+1$ with $m_{i+1} = \lfloor \frac{m_i}{2} \rfloor$. It can thus cover a large number of schedulers with only $\approx \log_2(m) \cdot n_0 + n$ simulation runs in total. The two-phase approach, in contrast, always needs $(m+1) \cdot n$ runs. We use smart sampling for all experiments reported in Section 6.3.

LSS beyond unbounded properties. Where memoryless schedulers suffice, LSS can straightforwardly be applied to MA as discussed above. For time-bounded properties, however, schedulers achieve optimality only if taking into account the amount of time remaining until the time bound is reached. A naive extension of LSS to such properties would be to input $s.\sigma.t$, where t is the total time elapsed during the run so far, to $\mathcal{H}$ in line 8 of Algorithm 2. Since t has been obtained by (a series of) sampling from exponential distributions, the probability to have the same value for t when entering a state s after some Markovian transitions in different runs is zero. This can make (near-)optimal schedulers, which need to make the same decision in s over intervals of time (cf. Example 2.7), infeasibly rare. The underlying problem is that the number of critical decisions is infinite, such that optimal schedulers have measure zero; this is the same problem that hindered the application of LSS to **probabilistic timed automata (PTA)** [64] as previously summarised in Reference [31]. To be effective, LSS needs the number of critical decisions to be finite.

For PTA, the problem is solved by simulating the equivalent zone or region graphs [30, 54]. For MA, we could similarly adapt the original discretisation approach from model checking (cf. Section 4.1.2). However, for the error to be small, a fine discretisation is needed. For example, to achieve an absolute error ≤ 0.01 for time bound $b = 0.5$ on an MA with maximum exit rate 3 requires a discretisation step of $\delta = 0.0025$ and thus the model to be "unfolded" 200 times. Then schedulers face every nondeterministic choice up to 200 times. Even with a single binary choice in one state, the probability of sampling an optimal scheduler (i.e., one that always makes the optimal choice) is thus 0.5^{200}—so again optimal schedulers are exceedingly rare. Adapting a uniformisation-based technique like Unif+ equally faces several difficulties. For example, it does not provide an *a priori* error bound. When used for model checking, the error is bounded by simultaneously computing an over- and underapproximation of the (maximum) probability. However LSS intrinsically underapproximates and introduces a statistical error. Further research into methods for effective LSS with time-bounded properties on MA is thus needed, in particular, to investigate whether the new switch-step algorithm could be suitably adapted. In the remainder of this article, we thus restrict our LSS experiments to unbounded probabilities and expected rewards.

6.3 Scheduler Sampling Efficiency Evaluation

Experimental evaluations of SMC with LSS for MA have so far been few and severely limited: In Reference [17], it was applied to four different parametrisations of a single model only, and merely $m = 20$ schedulers were sampled for each. In Reference [31], two non-parametrised models were studied, using $m \in \{100, 1000\}$ for each. In this section, we thus present the—to the best of our knowledge—first extensive experimental evaluation of the efficiency of LSS on MA. We use all MA instances—combinations of a model, a valuation for its parameters (cf. Section 4.2), and a property to check—from the QVBS that satisfy the following criteria:

— The property is an unbounded probabilistic reachability or expected accumulated reward property; otherwise, LSS as currently implemented in modes cannot be applied.
— The instance can be model-checked with mcsta using Unif+ within 30 min on a machine with 8 GB of RAM, to produce a reference result to compare with the values delivered by LSS.
— For every property that asks for a maximum (minimum) value, we add the corresponding property asking for the minimum (maximum) to the model if it was not already included. The

Table 2. Lightweight Scheduler Sampling Results on MA Models

model	params	property	value	uniform	lss-100	lss-1000	lss-10000	lss-100000
bitcoin-attack	20-6	T_MWin_{min}	3736.591	28486.08 (0)	8525.58 (0)	5943.85 (0)	5162.98 (0)	5106.35 (0)
		T_MWin_{max}	234360.002	27565.80 (0)	234194.00 (5)	233168.97 (4)	228440.27 (4)	— T/O —
breakdown-q	8	Min	0.028	0.10 (0)	0.08 (0)	0.05 (0)	0.04 (0)	0.03 (4)
		Max	0.232	(0)	0.12 (0)	0.14 (0)	0.16 (0)	0.17 (0)
jobs	5-2	$avgtime_{min}$	0.759	0.83 (0)	0.81 (2)	0.79 (4)	0.75 (5)	0.77 (4)
		$avgtime_{max}$	0.900	(1)	0.86 (3)	0.90 (5)	0.91 (4)	0.90 (5)
	10-3	$avgtime_{min}$	0.991	1.10 (0)	1.09 (1)	1.07 (1)	1.06 (2)	1.05 (3)
		$avgtime_{max}$	1.286	(0)	1.13 (0)	1.20 (0)	1.19 (0)	1.21 (1)
	15-3	$avgtime_{min}$	1.348	1.55 (0)	1.55 (0)	1.52 (0)	1.51 (0)	— T/O —
		$avgtime_{max}$	1.952	(0)	1.61 (0)	1.65 (0)	1.71 (0)	— T/O —
polling-sys	3-3-5	$T\_BothFull_{min}$	10.959	477.52 (0)	20.79 (0)	16.51 (0)	14.93 (0)	13.21 (0)
		$T\_BothFull_{max}$	6297835.466	(0)	164.91 (0)	826.10 (0)	2492.17 (0)	— T/O —
reentrant-q	3-3-3-5	$T\_BothFull_{min}$	5.937	11.63 (0)	11.24 (0)	10.65 (0)	10.14 (0)	9.58 (0)
		$T\_BothFull_{max}$	27.326	(0)	12.43 (0)	14.08 (0)	15.33 (0)	15.71 (0)
stream	10	$exp\_buffertime_{min}$	0.881	1.08 (0)	0.97 (1)	0.89 (5)	0.90 (5)	0.88 (5)
		$exp\_buffertime_{max}$	2.426	(0)	1.89 (2)	2.42 (5)	2.44 (5)	2.41 (5)
		$exp\_restarts_{min}$	0.024	1.27 (0)	0.22 (0)	0.05 (0)	0.04 (0)	0.04 (0)
		$exp\_restarts_{max}$	2.524	(0)	2.03 (0)	2.36 (2)	2.48 (5)	2.50 (5)
		$pr\_underrun_{min}$	0.025	0.65 (0)	0.12 (0)	0.10 (0)	0.03 (4)	0.03 (5)
		$pr\_underrun_{max}$	0.815	(0)	0.81 (5)	0.82 (5)	0.82 (5)	0.82 (5)
	100	$exp\_buffertime_{min}$	2.817	3.02 (1)	3.03 (0)	3.06 (0)	3.01 (1)	2.94 (3)
		$exp\_buffertime_{max}$	17.002	(0)	3.04 (0)	3.47 (0)	3.82 (0)	4.11 (0)
		$exp\_restarts_{min}$	0.057	5.13 (0)	3.62 (0)	2.07 (0)	1.61 (0)	1.48 (0)
		$exp\_restarts_{max}$	10.270	(0)	6.21 (0)	7.09 (0)	7.40 (0)	7.69 (0)
		$pr\_underrun_{min}$	0.095	0.89 (0)	0.70 (0)	0.45 (0)	0.39 (0)	0.37 (0)
		$pr\_underrun_{max}$	0.943	(1)	0.95 (5)	0.96 (5)	0.94 (5)	0.96 (5)
	500	$exp\_buffertime_{min}$	6.306	6.59 (5)	6.50 (4)	6.53 (4)	6.57 (3)	6.49 (4)
		$exp\_buffertime_{max}$	30.171	(0)	6.60 (0)	6.50 (0)	6.65 (0)	6.64 (0)
		$exp\_restarts_{min}$	0.029	12.07 (0)	11.31 (0)	8.54 (0)	8.10 (0)	6.93 (0)
		$exp\_restarts_{max}$	24.225	(0)	12.85 (0)	13.90 (0)	14.36 (0)	14.77 (0)
		$pr\_underrun_{min}$	0.203	0.95 (0)	0.92 (0)	0.74 (0)	0.70 (0)	0.69 (0)
		$pr\_underrun_{max}$	0.975	(5)	0.98 (5)	0.99 (5)	0.98 (5)	0.98 (5)
	1000	$exp\_buffertime_{min}$	8.920	9.04 (4)	9.24 (4)	9.17 (5)	9.22 (3)	9.07 (5)
		$exp\_buffertime_{max}$	33.200	(0)	9.13 (0)	9.18 (0)	9.28 (0)	9.22 (0)
		$exp\_restarts_{min}$	0.019	17.16 (0)	15.71 (0)	13.75 (0)	12.65 (0)	12.35 (0)
		$exp\_restarts_{max}$	34.678	(0)	18.24 (0)	19.23 (0)	19.57 (0)	19.82 (0)
		$pr\_underrun_{min}$	0.271	0.97 (0)	0.90 (0)	0.81 (0)	0.79 (0)	0.78 (0)
		$pr\_underrun_{max}$	0.982	(5)	0.98 (5)	0.98 (5)	0.98 (5)	0.98 (5)

relative difference between the maximum and the minimum must be greater than 10^{-1}. This ensures that we can separate the results when using a relative statistical error of $5 \cdot 10^{-2}$ in our experiments.

— The model must be such that, for every scheduler, the probability to eventually reach a goal state or a non-goal self-loop state is 1. This is needed to guarantee termination with a simple stopping criterion as in Algorithm 2.

The restriction to models that can be solved via model checking of course runs contrary to the purpose of SMC, which is to analyse models where model checking fails due to state space explosion. However, we want to evaluate the efficiency of LSS, so we need the optimal values for reference. All in all, the above criteria leave us with six different models for a total of 38 instances as listed in the first three columns of Table 2. The values obtained by mcsta are given in column "value."

We then performed SMC to obtain an estimate $\hat{v}$ of the true value v, configuring the statistical evaluation such that $\hat{v} \in [2 \cdot 10^{-2} \cdot v, (1 + 2 \cdot 10^{-2}) \cdot v]$ in 95 % of all experiments (i.e., we request a relative error of $2 \cdot 10^{-2}$ with 95 % "confidence"), for each model in five configurations. First, we use the uniform randomised scheduler, which simply makes a uniform random selection among the available transitions whenever it encounters a nondeterministic choice. This is similar to what Prism and Uppaal smc do, and provides a baseline as to the behaviour of the "average" scheduler. We then use LSS with smart sampling and $n_0 \in \{100, 1,000, 10,000, 100,000\}$. For expected rewards, modes then chooses $m_0 = n_0$; for reachability probabilities, it uses the scheme described in Reference [28] to select an appropriate m_0 depending on a first rough estimate of the probability. Varying n_0 and thus m allows us to observe how the values improve; ideally, as n_0 grows, they would approach the true values obtained by mcsta. All of these experiments were performed on an Intel Core i7-4790 workstation (3.6–4.0 GHz, 4 cores) running 64-bit Ubuntu Linux 18.04 and mcsta version 3.1.39. We used a timeout of 15 min. Runtimes scaled mostly linearly with n_0. They remained below one minute for $n_0 = 10,000$ for all instances except for those that were aborted due to a timeout (indicated as "T/O") for $n_0 = 100,000$. We do not focus on runtimes further, since we are interested in LSS' efficiency in terms of *being able to* find near-optimal schedulers with a given m; in particular, we use instances that can be solved by exhaustive model checking, which due to its error guarantees will in practice be preferable to LSS unless the model is too large to be exhaustively (or sufficiently partially) explored.

Each experiment was repeated five times. We report the average results of these runs in Table 2. Column "uniform" shows the result obtained by the uniform randomised scheduler. Since this scheduler is the same for maximum and minimum values, we report the average of all 10 executions under this scheduler, i.e., one value per pair of properties. The remaining columns "lss-n_0" report the results for LSS with per-phase budget n_0. In parentheses, we note the number of experiments (out of five) that delivered a result within the $\pm 2 \cdot 10^{-2}$ relative error bound w.r.t. the true value. For the **bitcoin-attack** model, we see that LSS significantly improves upon the values obtained by the uniform scheduler, actually reaching the true maximum value (up to the statistical error, i.e., the estimate needs to be in approx. [229672.8, 239047.2]) already with $n_0 = 100$ despite the uniform result being far away. For the minimum value, the results noticeably improve up to $n_0 = 10,000$, but schedulers closer to the optimum appear to be too rare to be found by LSS. T_MWin is an expected-time property, and notably simulation runs that achieve long durations as needed for $\text{T\_MWin}_{\max}$ take longer to generate, explaining the timeout for $n_0 = 100,000$. On the **breakdown-queues** model, the distribution of schedulers appears to be the other way—we manage to sample schedulers close to the minimum value, but remain far from the maximum—and near-optimal schedulers appear rarer as we need $n_0 = 100,000$ to achieve some "successes" at least for property Min. For the **jobs** model, we were able to use three different parameter valuations, with state space sizes ranging from $|S| = 117$ for valuation 5-2 to 1.9 million for 15-3. On the latter, model checking with mcsta took around 4 min. Here, we see that the number of critical nondeterministic choices grows with the state space size, making it increasingly unlikely for LSS to find good schedulers as the MA grows. It still manages to find, even for parameter valuation 15-3, schedulers whose values are statistically significantly different, thus delivering a clear improvement over using the uniform scheduler only. For the **polling-system** and **reentrant-queues** models, the results are similar. For the **stream** model, the state space sizes range from 176 to 1.5 million across the four parameter valuations. Again, LSS manages to deliver nontrivial intervals from minimum to maximum across all instances, but does not find near-optimal schedulers on the larger instances except where the uniform scheduler already achieves near-optimal results. In particular, the result of the uniform scheduler approaches the minimum (maximum) value for the $\text{exp\_buffertime}_{\min}$ ($\text{pr\_underrun}_{\max}$) property as the model grows.

Overall, LSS consistently manages to improve upon the naïve approach of using the uniform scheduler only, but struggles to attain near-optimal probabilities on these models. In general, it should work well if there are few critical decisions to make; the QVBS, however, appears to mostly contain MA models where the number of critical decisions grows as the model is scaled up—aside from several MA models where all nondeterminism was spurious, which are not interesting for LSS in the first place.

7 CONCLUSION

We have presented a fully integrated toolchain to create and verify Markov automata models based on the high-level compositional modelling language Modest in combination with the mcsta model checker and the modes simulator of the Modest Toolset. Other tools of the Modest Toolset complement the approach, such as the moconv tool that can export Modest models to Jani. We have compared the performance of the dedicated MA model checking algorithms in mcsta with Imca and Storm. We found mcsta to significantly outperform Imca, and to be faster than Storm in many cases. The Jani support in both the Modest Toolset and Storm allows the user to choose the most appropriate tool in every instance, thus mcsta and Storm ought to be seen as complementary tools for a common goal. Partial state space exploration, implemented in mcsta in a BRTDP-based manner, can significantly reduce the time and memory needed to analyse an MA; however, its effectiveness varies significantly with the structure of the MA and the property of interest that we consider. We found that it works particularly well to approximate time-bounded reachability probabilities. The scheduler sampling implementation in modes further complements the abilities of mcsta where the latter fails due to state space explosion. It provides significantly more useful results than other tools that have to rely on a single scheduler such as the uniform randomised one, and can in this way, e.g., disprove safety or provide implementable strategies [29] where other tools cannot. Still, it is important to keep its limitations, in particular, the inability to quantify the optimality of the sampled scheduler and thus, e.g., *prove* safety, in mind. Overall, Markov automata now have a user-friendly modelling language and efficient verification support in complementary tools that are actively maintained.

DATA AVAILABILITY

The data generated in our experimental evaluation as well as instructions to replicate the experiments are archived and available at DOI 10.4121/uuid:98d571be-cdd4-4e5a-a589-7c5b1320e569 [20] for Section 4.2 and at DOI 10.4121/14182523 [48] for Sections 5.2 and 6.3.

REFERENCES

[1] Elvio Gilberto Amparore, Gianfranco Balbo, Marco Beccuti, Susanna Donatelli, and Giuliana Franceschinis. 2016. 30 years of GreatSPN. In *Principles of Performance and Reliability Modeling and Evaluation*. Springer, 227–254. https://doi.org/10.1007/978-3-319-30599-8_9

[2] Pranav Ashok, Tomás Brázdil, Jan Kretínský, and Ondrej Slámecka. 2018. Monte Carlo tree search for verifying reachability in Markov decision processes. In *Proceedings of ISoLA (Lecture Notes in Computer Science)*, Vol. 11245. Springer, 322–335. https://doi.org/10.1007/978-3-030-03421-4_21

[3] Pranav Ashok, Yuliya Butkova, Holger Hermanns, and Jan Kretínský. 2018. Continuous-time Markov decisions based on partial exploration. In *Proceedings of ATVA (Lecture Notes in Computer Science)*, Vol. 11138. Springer, 317–334. https://doi.org/10.1007/978-3-030-01090-4_19

[4] Carlos Azevedo, Bruno Lacerda, Nick Hawes, and Pedro U. Lima. 2020. Long-run multi-robot planning with uncertain task durations. In *Proceedings of AAMAS*. International Foundation for Autonomous Agents and Multiagent Systems, 1750–1752.

[5] Christel Baier, Luca de Alfaro, Vojtech Forejt, and Marta Kwiatkowska. 2018. Model checking probabilistic systems. In *Handbook of Model Checking*. Springer, 963–999. https://doi.org/10.1007/978-3-319-10575-8_28

[6] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. 2010. Performance evaluation and model checking join forces. *Commun. ACM* 53, 9 (2010), 76–85. https://doi.org/10.1145/1810891.1810912
[7] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.
[8] Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. 2017. Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In *Proceedings of CAV (Lecture Notes in Computer Science)*, Vol. 10426. Springer, 160–180. https://doi.org/10.1007/978-3-319-63387-9_8
[9] Dimitri Bohlender, Harold Bruintjes, Sebastian Junges, Jens Katelaan, Viet Yen Nguyen, and Thomas Noll. 2014. A review of statistical model checking pitfalls on real-time stochastic models. In *Proceedings of ISoLA (Lecture Notes in Computer Science)*, Vol. 8803. Springer, 177–192.
[10] Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. 2006. MoDeST: A compositional modeling formalism for hard and softly timed systems. *IEEE Trans. Software Eng.* 32, 10 (2006), 812–830. https://doi.org/10.1109/TSE.2006.104
[11] Blai Bonet and Hector Geffner. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of ICAPS*. AAAI Press, 12–21.
[12] Hichem Boudali, Pepijn Crouzen, and Mariëlle Stoelinga. 2010. A rigorous, compositional, and extensible framework for dynamic fault tree analysis. *IEEE Trans. Dependable Sec. Comput.* 7, 2 (2010), 128–143. https://doi.org/10.1109/TDSC.2009.45
[13] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Kretínský, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. 2014. Verification of Markov decision processes using learning algorithms. In *Proceedings of ATVA (Lecture Notes in Computer Science)*, Vol. 8837. Springer, 98–114. https://doi.org/10.1007/978-3-319-11936-6_8
[14] Tomás Brázdil, Holger Hermanns, Jan Krcál, Jan Kretínský, and Vojtech Rehák. 2012. Verification of open interactive Markov chains. In *Proceedings of FSTTCS (LIPIcs)*, Vol. 18. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 474–485. https://doi.org/10.4230/LIPIcs.FSTTCS.2012.474
[15] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games* 4, 1 (2012), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810
[16] Carlos E. Budde, Pedro R. D'Argenio, and Arnd Hartmanns. 2019. Automated compositional importance splitting. *Sci. Comput. Program.* 174 (2019), 90–108. https://doi.org/10.1016/j.scico.2019.01.006
[17] Carlos E. Budde, Pedro R. D'Argenio, Arnd Hartmanns, and Sean Sedwards. 2020. An efficient statistical model checker for nondeterminism and rare events. *Proceedings of STTT* 22, 6 (2020), 759–780. https://doi.org/10.1007/s10009-020-00563-2
[18] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. 2017. JANI: Quantitative model and tool interaction. In *Proceedings of TACAS (Lecture Notes in Computer Science)*, Vol. 10206. 151–168. https://doi.org/10.1007/978-3-662-54580-5_9
[19] Carlos E. Budde, Arnd Hartmanns, Michaela Klauck, Jan Kretínský, David Parker, Tim Quatmann, Andrea Turrini, and Zhen Zhang. 2021. On correctness, precision, and performance in quantitative verification (QComp 2020 competition report). In *Proceedings of ISoLA (Lecture Notes in Computer Science)*. Springer. To appear.
[20] Yuliya Butkova. 2019. A Modest Approach to Modelling and Checking Markov Automata (Artifact). 4TU.ResearchData. https://doi.org/10.4121/uuid:98d571be-cdd4-4e5a-a589-7c5b1320e569
[21] Yuliya Butkova and Gereon Fox. 2019. Optimal time-bounded reachability analysis for concurrent systems. In *Proceedings of TACAS (Lecture Notes in Computer Science)*, Vol. 11428. Springer, 191–208. https://doi.org/10.1007/978-3-030-17465-1_11
[22] Yuliya Butkova, Arnd Hartmanns, and Holger Hermanns. 2019. A modest approach to modelling and checking Markov automata. In *Proceedings of QEST (Lecture Notes in Computer Science)*, Vol. 11785. Springer, 52–69. https://doi.org/10.1007/978-3-030-30281-8_4
[23] Yuliya Butkova, Hassan Hatefi, Holger Hermanns, and Jan Krcál. 2015. Optimal continuous time Markov decisions. In *Proceedings of ATVA (Lecture Notes in Computer Science)*, Vol. 9364. Springer, 166–182. https://doi.org/10.1007/978-3-319-24953-7_12
[24] Yuliya Butkova, Ralf Wimmer, and Holger Hermanns. 2017. Long-run rewards for Markov automata. In *Proceedings of TACAS (Lecture Notes in Computer Science)*, Vol. 10206. 188–203. https://doi.org/10.1007/978-3-662-54580-5_11
[25] Yuliya Butkova, Ralf Wimmer, and Holger Hermanns. 2018. Markov automata on discount! In *Proceedings of MMB (Lecture Notes in Computer Science)*, Vol. 10740. Springer, 19–34. https://doi.org/10.1007/978-3-319-74947-1_2
[26] Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. 2011. Continuous upper confidence trees. In *Proceedings of LION (Lecture Notes in Computer Science)*, Vol. 6683. Springer, 433–445. https://doi.org/10.1007/978-3-642-25566-3_32
[27] Przemyslaw Daca, Thomas A. Henzinger, Jan Kretínský, and Tatjana Petrov. 2017. Faster statistical model checking for unbounded temporal properties. *ACM Trans. Comput. Log.* 18, 2 (2017), 12:1–12:25. https://doi.org/10.1145/3060139

[28] Pedro D'Argenio, Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. 2015. Smart sampling for lightweight verification of Markov decision processes. *STTT* 17, 4 (2015), 469–484. https://doi.org/10.1007/s10009-015-0383-0

[29] Pedro R. D'Argenio, Juan A. Fraire, and Arnd Hartmanns. 2020. Sampling distributed schedulers for resilient space communication. In *Proceedings of NFM (Lecture Notes in Computer Science)*, Vol. 12229. Springer, 291–310. https://doi.org/10.1007/978-3-030-55754-6_17

[30] Pedro R. D'Argenio, Arnd Hartmanns, Axel Legay, and Sean Sedwards. 2016. Statistical approximation of optimal schedulers for probabilistic timed automata. In *Proceedings of iFM (Lecture Notes in Computer Science)*, Vol. 9681. Springer, 99–114. https://doi.org/10.1007/978-3-319-33693-0_7

[31] Pedro R. D'Argenio, Arnd Hartmanns, and Sean Sedwards. 2018. Lightweight statistical model checking in nondeterministic continuous time. In *Proceedings of ISoLA (Lecture Notes in Computer Science)*, Vol. 11245. Springer, 336–353. https://doi.org/10.1007/978-3-030-03421-4_22

[32] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikucionis, and Jakob Haahr Taankvist. 2015. Uppaal Stratego. In *Proceedings of TACAS (Lecture Notes in Computer Science)*, Vol. 9035. Springer, 206–211. https://doi.org/10.1007/978-3-662-46681-0_16

[33] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Zheng Wang. 2011. Time for statistical model checking of real-time systems. In *Proceedings of CAV (Lecture Notes in Computer Science)*, Vol. 6806. Springer, 349–355. https://doi.org/10.1007/978-3-642-22110-1_27

[34] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A Storm is coming: A modern probabilistic model checker. In *Proceedings of CAV (Lecture Notes in Computer Science)*, Vol. 10427. Springer, 592–600. https://doi.org/10.1007/978-3-319-63390-9_31

[35] J. B. Dugan, S. J. Bavuso, and M. A. Boyd. 1992. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Trans. Reliabil.* 41, 3 (1992), 363–377.

[36] Christian Eisentraut. 2017. *Principles of Markov Automata*. Ph.D. Dissertation. Saarland University, Germany. publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/27085/1/thesis-Pflichtexemplar.pdf.

[37] Christian Eisentraut, Holger Hermanns, Joost-Pieter Katoen, and Lijun Zhang. 2013. A semantics for every GSPN. In *Proceedings of Petri Nets (Lecture Notes in Computer Science)*, Vol. 7927. Springer, 90–109. https://doi.org/10.1007/978-3-642-38697-8_6

[38] Christian Eisentraut, Holger Hermanns, and Lijun Zhang. 2010. On probabilistic automata in continuous time. In *Proceedings of LICS*. IEEE Computer Society, 342–351. https://doi.org/10.1109/LICS.2010.41

[39] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. 2013. CADP 2011: A toolbox for the construction and analysis of distributed processes. *STTT* 15, 2 (2013), 89–107.

[40] Marcus Gerhold, Arnd Hartmanns, and Mariëlle Stoelinga. 2019. Model-based testing of stochastically timed systems. *Innov. Syst. Softw. Eng.* 15, 3-4 (2019), 207–233. https://doi.org/10.1007/s11334-019-00349-z

[41] Timo P. Gros. 2018. *Markov Automata Taken by Storm*. Master's thesis. Saarland University, Germany.

[42] Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer. 2012. Quantitative timed analysis of interactive Markov chains. In *Proceedings of NFM (Lecture Notes in Computer Science)*, Vol. 7226. Springer, 8–23. https://doi.org/10.1007/978-3-642-28891-3_4

[43] Dennis Guck, Hassan Hatefi, Holger Hermanns, Joost-Pieter Katoen, and Mark Timmer. 2014. Analysis of timed and long-run objectives for Markov automata. *Logic. Methods Comput. Sci.* 10, 3 (2014). https://doi.org/10.2168/LMCS-10(3:17)2014

[44] Dennis Guck, Mark Timmer, Hassan Hatefi, Enno Ruijters, and Mariëlle Stoelinga. 2014. Modelling and analysis of Markov reward automata. In *Proceedings of ATVA (Lecture Notes in Computer Science)*, Vol. 8837. Springer, 168–184. https://doi.org/10.1007/978-3-319-11936-6_13

[45] Serge Haddad and Benjamin Monmege. 2018. Interval iteration algorithm for MDPs and IMDPs. *Theor. Comput. Sci.* 735 (2018), 111–131. https://doi.org/10.1016/j.tcs.2016.12.003

[46] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Kretínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. 2019. The 2019 comparison of tools for the analysis of quantitative formal models. In *Proceedings of 25 Years of TACAS: TOOLympics (Lecture Notes in Computer Science)*, Vol. 11429. Springer, 69–92. https://doi.org/10.1007/978-3-030-17502-3_5

[47] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. 2013. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods Syst. Design* 43, 2 (2013), 191–232. https://doi.org/10.1007/s10703-012-0167-z

[48] Arnd Hartmanns. 2021. A Modest Approach to Markov Automata (Artifact). 4TU.ResearchData. https://doi.org/10.4121/14182523

[49] Arnd Hartmanns and Holger Hermanns. 2014. The modest toolset: An integrated environment for quantitative modelling and verification. In *Proceedings of TACAS (Lecture Notes in Computer Science)*, Vol. 8413. Springer, 593–598. https://doi.org/10.1007/978-3-642-54862-8_51

[50] Arnd Hartmanns and Holger Hermanns. 2015. Explicit model checking of very large MDP using partitioning and secondary storage. In *Proceedings of ATVA (Lecture Notes in Computer Science)*, Vol. 9364. Springer, 131–147. https://doi.org/10.1007/978-3-319-24953-7_10

[51] Arnd Hartmanns and Holger Hermanns. 2019. A modest Markov automata tutorial. In *Proceedings of 15th International Reasoning Web Summer School (Lecture Notes in Computer Science)*, Vol. 11810. Springer, 250–276. https://doi.org/10.1007/978-3-030-31423-1_8

[52] Arnd Hartmanns and Benjamin Lucien Kaminski. 2020. Optimistic value iteration. In *Proceedings of CAV (Lecture Notes in Computer Science)*, Vol. 12225. Springer, 488–511. https://doi.org/10.1007/978-3-030-53291-8_26

[53] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. 2019. The quantitative verification benchmark set. In *Proceedings of TACAS (Lecture Notes in Computer Science)*, Vol. 11427. Springer, 344–350. https://doi.org/10.1007/978-3-030-17462-0_20

[54] Arnd Hartmanns, Sean Sedwards, and Pedro R. D'Argenio. 2017. Efficient simulation-based verification of probabilistic timed automata. In *Proceedings of Winter Simulation Conference*. IEEE, 1419–1430. https://doi.org/10.1109/WSC.2017.8247885

[55] Hassan Hatefi. 2017. *Finite Horizon Analysis of Markov Automata*. Ph.D. Dissertation. Saarland University, Germany. scidok.sulb.uni-saarland.de/volltexte/2017/6743/.

[56] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. 2004. Approximate probabilistic model checking. In *Proceedings of VMCAI (Lecture Notes in Computer Science)*, Vol. 2937. Springer, 73–84. https://doi.org/10.1007/978-3-540-24622-0_8

[57] C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall.

[58] Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. 2002. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Mach. Learn.* 49, 2-3 (2002), 193–208.

[59] Michaela Klauck. 2020. Modest Fret-pi LRTDP. Retrieved from https://dgit.cs.uni-saarland.de/Michaela/modest-fret-pi-lrtdp.

[60] Andrey Kolobov, Mausam, Daniel S. Weld, and Hector Geffner. 2011. Heuristic search for generalized stochastic shortest path MDPs. In *Proceedings of ICAPS*. AAAI Press.

[61] Jan Krcál and Pavel Krcál. 2015. Scalable analysis of fault trees with dynamic features. In *Proceedings of DSN*. IEEE Computer Society, 89–100. https://doi.org/10.1109/DSN.2015.29

[62] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. 2005. MANET simulation studies: The incredibles. *Mobile Comput. Commun. Rev.* 9, 4 (2005), 50–61. https://doi.org/10.1145/1096166.1096174

[63] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of CAV (Lecture Notes in Computer Science)*, Vol. 6806. Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

[64] Marta Z. Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. 2002. Automatic verification of real-time systems with discrete probability distributions. *Theor. Comput. Sci.* 282, 1 (2002), 101–150. https://doi.org/10.1016/S0304-3975(01)00046-9

[65] Kim Guldstrand Larsen and Axel Legay. 2018. Statistical model checking the 2018 edition! In *Proceedings of ISoLA (Lecture Notes in Computer Science)*, Vol. 11245. Springer, 261–270. https://doi.org/10.1007/978-3-030-03421-4_17

[66] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. 2014. Scalable verification of Markov decision processes. In *Proceedings of WS-FMDS at SEFM (Lecture Notes in Computer Science)*, Vol. 8938. Springer, 350–362. https://doi.org/10.1007/978-3-319-15201-1_23

[67] Masoumeh Mansouri, Bruno Lacerda, Nick Hawes, and Federico Pecora. 2019. Multi-robot planning under uncertain travel times and safety constraints. In *Proceedings of IJCAI*. ijcai.org, 478–484. https://doi.org/10.24963/ijcai.2019/68

[68] H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. 2005. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of ICML (ACM International Conference Proceeding Series)*, Vol. 119. ACM, 569–576. https://doi.org/10.1145/1102351.1102423

[69] Robin Milner. 1989. *Communication and Concurrency*. Prentice-Hall.

[70] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc.

[71] Tim Quatmann, Sebastian Junges, and Joost-Pieter Katoen. 2017. Markov automata with multiple objectives. In *Proceedings of CAV (Lecture Notes in Computer Science)*, Vol. 10426. Springer, 140–159. https://doi.org/10.1007/978-3-319-63387-9_7

[72] Tim Quatmann and Joost-Pieter Katoen. 2018. Sound value iteration. In *Proceedings of CAV (Lecture Notes in Computer Science)*, Vol. 10981. Springer, 643–661. https://doi.org/10.1007/978-3-319-96145-3_37

[73] Markus N. Rabe and Sven Schewe. 2011. Finite optimal control for time-bounded reachability in CTMDPs and continuous-time Markov games. *Acta Info.* 48, 5-6 (2011), 291–315. https://doi.org/10.1007/s00236-011-0140-0

[74] Daniël Reijsbergen, Pieter-Tjerk de Boer, Werner R. W. Scheinhardt, and Boudewijn R. Haverkort. 2015. On hypothesis testing for statistical model checking. *Proceedings of STTT* 17, 4 (2015), 377–395. https://doi.org/10.1007/s10009-014-0350-1

[75] Gerardo Rubino and Bruno Tuffin (Eds.). 2009. *Rare Event Simulation Using Monte Carlo Methods*. Wiley.

[76] Roberto Segala. 1995. *Modeling and Verification of Randomized Distributed Real-time Systems*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. hdl.handle.net/1721.1/36560.

[77] Marcel Steinmetz, Jörg Hoffmann, and Olivier Buffet. 2016. Goal probability analysis in probabilistic planning: Exploring and enhancing the state of the art. *J. Artif. Intell. Res.* 57 (2016), 229–271. https://doi.org/10.1613/jair.5153

[78] Kevin J. Sullivan, Joanne Bechta Dugan, and David Coppit. 1999. The Galileo fault tree analysis tool. In *Proceedings of FTCS*. IEEE Computer Society, 232–235. https://doi.org/10.1109/FTCS.1999.781056

[79] Mark Timmer, Joost-Pieter Katoen, Jaco van de Pol, and Mariëlle Stoelinga. 2012. Efficient modelling and generation of Markov automata. In *Proceedings of CONCUR (Lecture Notes in Computer Science)*, Vol. 7454. Springer, 364–379. https://doi.org/10.1007/978-3-642-32940-1_26

[80] Håkan L. S. Younes and Reid G. Simmons. 2002. Probabilistic verification of discrete event systems using acceptance sampling. In *Proceedings of CAV (Lecture Notes in Computer Science)*, Vol. 2404. Springer, 223–235. https://doi.org/10.1007/3-540-45657-0_17

Received April 2020; revised September 2020; accepted February 2021