POWVER

Technical Report 2019-12

Title: **A Modest Markov Automata Tutorial**

Authors: Arnd Hartmanns, Holger Hermanns

Report Number: 2019-12

ERC Project: Power to the People. Verified.

ERC Project ID: 695614

Funded Under: H2020-EU.1.1. – EXCELLENT SCIENCE

Host Institution: Universität des Saarlandes, Dependable Systems and Software
Saarland Informatics Campus

Published In: Reasoning Web 2019

A Modest Markov Automata Tutorial*

Arnd Hartmanns[1] and Holger Hermanns[2,3]

[1] University of Twente, Enschede, The Netherlands
[2] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[3] Institute of Intelligent Software, Guangzhou, China

Abstract. Distributed computing systems provide many important services. To explain and understand why and how well they work, it is common practice to build, maintain, and analyse models of the systems' behaviours. Markov models are frequently used to study operational phenomena of such systems. They are often represented with discrete state spaces, and come in various flavours, overarched by Markov automata. As such, Markov automata provide the ingredients that enable the study of a wide range of quantitative properties related to risk, cost, performance, and strategy. This tutorial paper gives an introduction to the formalism of Markov automata, to practical modelling of Markov automata in the Modest language, and to their analysis with the Modest Toolset. As case studies, we optimise an attack on Bitcoin, and evaluate the performance of a small but complex resource-sharing computing system.

1 Introduction

Distributed computing systems provide many important services, such as electronic banking, information and knowledge sharing, and social networking. They are enablers for innovation; for instance, blockchain technology is based on massively distributed computing. Since our societies increasingly depend on the services offered in this manner, it is important to ensure their performance, dependability, and correctness. The purpose of *performance evaluation* is to investigate and optimise the amount of useful work being accomplished. *Dependability evaluation* is concerned with assessing service continuity by means of measures such as reliability and availability. The evaluation of correctness—usually called *formal verification*—focusses on proving that the service delivered satisfies a formal specification of its behaviour. Usually, all of these techniques are based on a *model* of the system, which is an abstract representation of the system's behaviour.

Markov chains. In numerical performance and dependability evaluation, by far the most prominent models used to represent the temporal dynamics of a system are *Markov chains* [38]. In this model family, the system is supposed to occupy a state at any moment in time, with the set S of states (the state space) being

* Authors are listed alphabetically. This work has received financial support by DFG grant 389792660 as part of TRR 248 (see perspicuous-computing.science), by ERC Advanced Grant 69561 (POWVER), and by NWO VENI grant 639.021.754.

finite or countably infinite. Markov chains come in two flavours, dependent on whether the time domain $\mathbb{T}$ is considered to be discrete ($\mathbb{T} = \mathbb{N} = \{ 0, 1, \dots \}$) or continuous ($\mathbb{T} = \mathbb{R}_+ = [0, \infty)$). The dynamics of a *discrete-time Markov chain* (DTMC) is determined by a mapping from states to probability distributions over (successor) states. For instance, if state s is mapped to probability distribution μ, then the system once occupying state s is understood to jump to state s' with probability $\mu(s')$ in one time step. Notably, the probability is assumed to be independent of any further information (such as any past behaviour) apart from the state identity of s. This is known as the *Markov* (or *memoryless*) *property.* A *continuous-time Markov chain* (CTMC) adheres to this property, too, but it now needs to be interpreted in *stochastic time*, i.e. on a continuous time line where probability mass flows continuously between states. For CTMC, the Markov property implies that neither the past history nor the time already spent in the current state s influences the flow of probability into some state s'. Instead, it is governed by a time-independent *rate* λ, a positive real value (or zero if no flow exists). Thus the overall behaviour of a CTMC is determined by a mapping from state pairs to rates in $\mathbb{R}_+$. CTMC are arguably better fit to the nature of distributed computing [5], where it is difficult to assume a common discrete time base. The time spent in state s before jumping to another state s' is usually called the *residence time* (or *sojourn time*) in s. Residence times are geometrically distributed in DTMC, and exponentially distributed in CTMC.

Labelled transition systems. In formal verification, other models appear: State-transition diagrams, automata, and similar formalisms describe the dynamic behaviour of systems here. They often appear in the specific form of *labelled transition systems* (LTS). A transition system consists of a set of states S and a set of possible state changes. The latter is given as a binary relation on states, i.e. a subset of the cross product $S \times S$. Intuitively, a pair of states $\langle s, s' \rangle$ is in this relation if it is *possible* to jump from s to s' in a single step. In LTS, state changes are associated with occurrences of *actions*. A state change from s to s' then implies the occurrence of a specific action a, which labels the transition—thus we have an LTS. If multiple transitions are possible in a state, then the decision of which one to take is usually interpreted as being *nondeterministic*. Nondeterminism is especially useful to represent concurrency, a crucial aspect of distributed computing systems. If two systems run concurrently and independently, this is best represented as the nondeterministic interleaving of their individual steps. LTS can thus be endowed with *parallel composition* operators to model concurrency and interaction of component LTS [6, 40, 44]. With further operators, this is convenient for a *compositional modelling* style, where the behaviour of components is the result of compositions of smaller building blocks.

Model checking. Within the spectrum of techniques used in formal verification, model checking is an automated model-based technique to assess whether the possible system behaviours satisfy a property describing the desirable behaviour [3]. Typically, properties are expressed in temporal logics such as LTL or CTL. Model checking usually involves constructing an in-memory representation of the (part of the) state space (relevant to assess the property). It thus gives definitive answers, but faces the state space explosion problem. In the past decades, model checking has been extended to treat aspects such as discrete probabilities and stochastic time. It has become apparent that a joint consideration of performance, dependability and correctness is both possible and worthwhile [2].

This paper. The purpose of this tutorial paper is to provide a gentle introduction to working with a mathematical formalism integrating the modelling aspects discussed above. We focus especially on the specification and modelling of real systems. The formalism we introduce is called *Markov automata* (MA), and it can best be described as an orthogonal and compositional superposition of DTMC, CTMC, and LTS. MA have been coined in [22,23]. They are expressive enough to give a semantics to generalised stochastic Petri nets (GSPN) in their full generality [20]. The theoretical properties of MA are the subject of the Ph.D. thesis of Christian Eisentraut [19]; a process-algebraic perspective is covered in the Ph.D. thesis of Mark Timmer [50]. Various algorithmic analysis methods for Markov automata have been developed over the past decade [8,9,14–16,21,28,29,36,37,52].

Using the mathematical formalism of MA directly to build complex models is, however, cumbersome. We instead need a higher-level *modelling language*. Aside from parallel composition, such languages typically provide variables over finite domains that can be used in expressions to e.g. enable or disable transitions, allowing to compactly describe very large models. In this paper, we use Modest [30] to construct MA models. Rooted in process algebra, Modest provides various composition operators that allow large models to be assembled from smaller, easier-to-understand components. After a formal definition of MA, parallel composition, and various types of properties (that we may want to compute for a given MA model) in Sect. 2, we introduce the basics of Modest in a step-by-step fashion in Sect. 3. We compare it to alternative languages with respect to its succinctness, expressivity, and readability. We then guide the reader through the modelling and analysis of two very different applications: we optimise an *attack on Bitcoin* in Sect. 4, and we evaluate the performance of a small, but intricate resource-sharing *queueing system* in Sect. 5 with the Modest Toolset. Algorithmic aspects of the analysis of MA with the Modest Toolset are the subject of a companion paper [13].

Previous work. Our presentation of MA in Sect. 2 is adapted and extended from [13], as is the text in sects. 3.2 and 3.3. The Bitcoin models in Sect. 4 are inspired by [24], and the `bitcoin-attack.modest` model is part of the Quantitative Verification Benchmark Set [34]. The reentrant queueing system, of which we present a new Modest model in Sect. 5, was first described in [36].

2 Markov Automata

The mathematical formalism of Markov automata provides nondeterministic choices as in LTS, discrete probabilistic decisions as in DTMC, and stochastic time as in CTMC. The relationships between these and other formalisms are

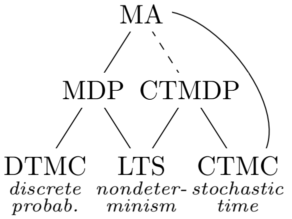


Fig. 1. The MA family tree

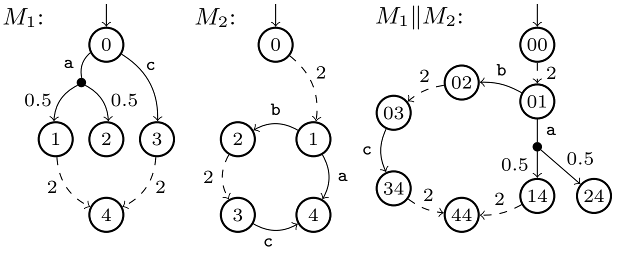


Fig. 2. Example Markov automata

visualised in Fig. 1. The combination of DTMC and LTS leads to the model family of (discrete-time) Markov decision processes [46] (MDP, or probabilistic automata [49]) 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 [46] (CTMDP), where action-labelled transitions are of the form $s \overset{a}{\dashrightarrow} e$ 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. 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 $\{ x \in \mathbb{R} \mid a < x < b \}$, 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 $\mathit{spt}(\mu) \overset{\text{def}}{=} \{ s \in S \mid \mu(s) > 0 \}$ is countable and $\sum_{s \in \mathit{spt}(\mu)} \mu(s) = 1$. $\mathit{Dist}(S)$ is 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.

Definition 1. *A* Markov automaton *(MA) 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, $P \colon S \to 2^{A \times \mathit{Dist}(S)}$ *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* $br \colon S \times \mathit{Tr}(M) \times S \to [0, \infty)$ *is the* branch reward *function.* $\mathit{Tr}(M) \overset{\text{def}}{=} \bigcup_{s \in S} P(s) \cup Q(s)$ *is the set of all* transitions*; it must be finite. We require that* $br(\langle s, tr, s' \rangle) \neq 0$ *implies* $tr \in P(s) \cup Q(s)$.

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 1. Fig. 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, 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 in a probabilistic (the choice of successor state of a probabilistic transition, the choice among Markovian transitions) or stochastic (the choice of residence time) way, 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. [35] 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. The co-existence of action-labelled probabilistic transitions of the form $s \xrightarrow{a} \mu$ and of Markovian transitions of the form $s \overset{\lambda}{\dashrightarrow} s'$ 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.

Definition 2. *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 $M_1 \parallel M_2 \overset{\text{def}}{=} \langle S_1 \times S_2, \langle s_{0_1}, s_{0_2} \rangle, A, P, Q, rr, br \rangle$ *where* P *is the smallest function that satisfies the inference rules*

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

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

Q *is the smallest function that satisfies the inference rules*

$$\frac{s_1 \overset{\lambda}{\dashrightarrow}_{Q_1} s_1'}{\langle s_1, s_2 \rangle \overset{\lambda}{\dashrightarrow}_Q \langle s_1', s_2 \rangle} \qquad \frac{s_2 \overset{\lambda}{\dashrightarrow}_{Q_2} s_2'}{\langle s_1, s_2 \rangle \overset{\lambda}{\dashrightarrow}_Q \langle s_1, s_2' \rangle},$$

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 (third inference rule), and otherwise preserves the original branch rewards.

The first two inference rules for P allow the individual MA to proceed independently of each other if allowed by *sync*; the third rule covers the case where both automata synchronise on a pair of actions as determined by *sync*. The rules for Q simply state that Markovian transitions are always performed independently. An element of *sync* is called 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 [40,44] to be encoded. Originally established by Cadp [26], it is today used for MA in the Jani format [12]. We refer to a general parallel composition of several MA as a *network* of MA.

Example 2. Fig. 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 $sync = \{ \langle \mathtt{a}, \mathtt{a} \rangle \mapsto \mathtt{a}, \langle \bot, \mathtt{b} \rangle \mapsto \mathtt{b}, \langle \mathtt{c}, \mathtt{c} \rangle \mapsto \mathtt{c} \}$.

We defined MA as *open* systems [10]: 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, 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 $P(s) = \varnothing$) and probabilistic states (where $Q(s) = \varnothing$). The behaviour of a closed, deadlock-free MA M is defined via its paths:

Definition 3. *Let M be a closed, deadlock-free MA M 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 $i \in \{0, \ldots\}$, $Q(s_i) = \varnothing$ implies $t_i = 0$, $tr_i \in P(s_i) \cup Q(s_i)$, $tr_i = \langle a, \mu \rangle \in P(s_i)$ 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 $\pi_{\leq j} \overset{\text{def}}{=} s_0\, t_0 \ldots s_j$. The* duration *$\mathrm{dur}(\pi_{fin})$ of a path prefix is the sum of its residence times t_i. A path's* reward *is*

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

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 4. *Let M be a closed, deadlock-free MA as above. A* scheduler *is a function $\sigma\colon \mathit{\Pi}_{\mathit{fin}}(M) \to \mathit{Tr}(M)$ s.t. $\forall s \in S\colon \sigma(s) = tr$ implies $tr \in P(s) \cup Q(s)$. We write $\mathfrak{S}(M)$ for the set of all schedulers of M. A* time-dependent *scheduler is in $S \times [0, \infty) \to \mathit{Tr}(M)$; a* memoryless *scheduler is in $S \to \mathit{Tr}(M)$. Given a time bound $b \in [0, \infty)$, every time-dependent scheduler σ_t defines a corresponding scheduler σ by $\sigma(\pi_{\mathit{fin}}) = \sigma_t(\langle \mathit{last}(\pi_{\mathit{fin}}), b - \mathrm{dur}(\pi_{\mathit{fin}})\rangle)$. Every memoryless scheduler σ_{ml} defines a corresponding scheduler σ by $\sigma(\pi_{\mathit{fin}}) = \sigma_{ml}(\mathit{last}(\pi_{\mathit{fin}}))$.*

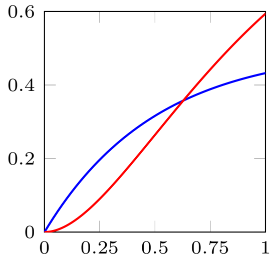
We define deterministic schedulers only since randomised schedulers are in practice only needed for multi-objective problems [47]. We note that CTMDP with early schedulers [48] can be encoded as closed MA. If we "apply" a scheduler to an MA, it removes all nondeterminism, and we are left with a fully 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. [35] 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.

Expected accumulated rewards: Compute the expected value of the random variable that assigns to π the value $\mathrm{rew}(\pi_{\mathit{fin}})$ with π_{fin} being the shortest prefix of π with a state in G. This is well-defined if the maximum (minimum) probability to reach G is 1; otherwise, we define the minimum (maximum) expected accumulated reward to be ∞. Memoryless schedulers suffice.

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



Example 3. Consider MA $M_1 \parallel M_2$ of Fig. 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.

We can extend MA with discrete *variables*: An MA with variables (MA$^{\mathrm{V}}$) is an MA like in Definition 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$^{\mathrm{V}}$ in terms of an MA: The MA$^{\mathrm{V}}$ 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$^{\mathrm{V}}$, parallel composition does not distribute over semantics; we need to compose the MA$^{\mathrm{V}}$ before converting them to MA.

3 Modelling with Markov Automata

Tools for the automated analysis of MA need a syntax in which the model and the properties of interest are specified. As noted in Sect. 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

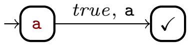
Modest [4, 30] is the modelling and description language for stochastic timed systems. At its core, it is a process algebra: it provides various operations such as parallel and sequential composition, parameterised process definitions, process calls, and guards to flexibly construct complex models out 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 Fig. 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, etc., that are available for use in the behaviour and inside nested processes. A process' behaviour defines an MA with those variables[4]. 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:

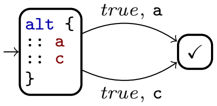
```
a
```



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:

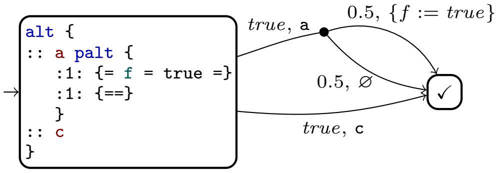
```
alt {
:: a
:: c
}
```



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
}
```



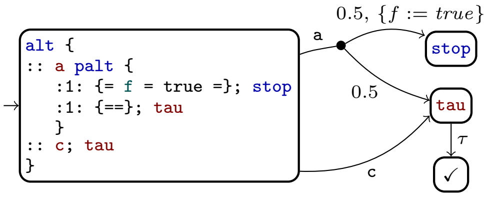
Probabilities are specified as *weights* between colons `:`, i.e. the actual probability in the semantics is calculated as the given weight divided by the sum of all weights in the `palt` construct. The assignments for every branch are specified in `{= =}` blocks, and they are executed *atomically*, 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

[4] Actually, the semantics of Modest [30] is defined in terms of stochastic hybrid automata (SHA), of which MA are a special case; we restrict to that case in this paper.

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.

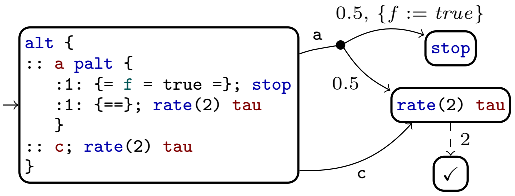
Continuing to model M_1 in Modest, we now add the Markovian transitions to state 4. We need two new constructs: for sequential composition, and for rates. First, the semantics of the sequential composition construct P; Q, for two behaviours P and Q, is to first behave like P, and upon successful termination 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 $\langle \bot, \tau \rangle \mapsto \tau$, but cannot occur in any other vectors). To turn the τ-labelled probabilistic edge into a Markovian one, we simply specify rates:

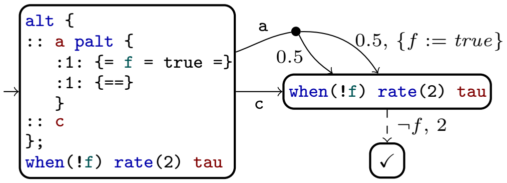
```
alt {
:: a palt {
   :1: {= f = true =}; stop
   :1: {==}; rate(2) tau
   }
:: c; rate(2) tau
}
```



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, 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^V. The result is:

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



The semantics of the MA^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 Fig. 3, we show the full Modest model of the parallel composition of MA M_1 and M_2 of Fig. 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 being assigned the values of the actual arguments, and new variable instances created 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), and as described above for τ. The model also declares two properties for verification, `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[5] with range $\{ 0, 1, 2 \}$—within time bound `B` akin to Example 3. `B` is an open parameter for which values can be specified at verification time.

At this point, we have covered most basic constructs of Modest. There are many features not used in this small model; we will introduce more constructs in sects. 4 and 5. The interested reader also finds additional Modest MA models in the Quantitative Verification Benchmark Set (QVBS, [34]) at qcomp.org.

3.2 The Modest Toolset

The creation and analysis of MA with Modest is supported by the Modest Toolset [32], a comprehensive suite of tools for quantitative modelling and verification. Aside from Modest, it also supports the Jani model interchange format [12] as an input language. MA are supported in the toolset's mosta, moconv[6], mcsta, and modes tools. mosta visualises the symbolic semantics of models (i.e. networks of MA^{V} before and after parallel composition as shown throughout Sect. 3.1) and is useful for model debugging. moconv transforms models between Modest and Jani, and performs syntactic rewriting and optimisations. mcsta is a fast explicit-state model checker that implements state-of-the-art MA-specific algorithms [13] and uses secondary storage to alleviate state space explosion [33]. modes [11] is a statistical model checker with automated rare event simulation capabilities. It implements lightweight scheduler sampling [43] for nondeterministic models, including MA [17]. The Modest Toolset is written in C#, works on Linux, Mac OS, and Windows, and is freely available at modestchecker.net. All its tools share a common infrastructure for parsing and syntactic transformations. mcsta and modes build on the same

[5] 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 the mcsta model checker (see Sect. 3.2) as long as only finitely many distinct values are ever assigned to the variables occurring in the model.

[6] moconv can also export CTMDP to Jani, but due to their lack of a natural parallel composition operator, the analysis of CTMDP is not supported in the other tools.

```
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 $M_1 \parallel M_2$

```
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

```
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

```
#INITIALS
s00
#GOALS
s44
#TRANSITIONS
s00 !
* s01 2
s01 a
* s02 1
s01 b
* 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

state space exploration engine that compiles models to bytecode at runtime for memory efficiency and performance.

3.3 Alternative Modelling Languages

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

State space files for Imca. The first MA-specific algorithms were implemented in the Imca tool [27]. Its only input language is a text-based explicit state space format as illustrated for our example of $M_1 \parallel M_2$ in Fig. 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 [18] 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 [20] in GreatSPN format [1], and through an extension of the Prism guarded command language. We show our example in the latter in Fig. 5. This is a very simple and small language that is easy to learn, however it completely lacks higher-level constructs to structure and compose models aside from the implicit parallel composition of its *modules*.

Process algebra with Scoop. Mapa [51] is a dedicated process algebra for MA. It is supported by Scoop [51], 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 Fig. 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 [12] is a model interchange format designed to ease tool development and interoperation. It is Json-based and thus human-debuggable, but not intended as human-writable. 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 paper.

4 Optimising Attacks on Bitcoin

Bitcoin [45] is currently the most popular cryptocurrency. It is built on blockchain technology using the proof-of-work approach. Every block in the blockchain contains a *nonce* (a randomly chosen number), a set of (monetary) *transactions*, and a hash of the predecessor block in the chain. In this way, no past block can be changed without invalidating (the hashes in) all its successors. A block is valid if the hash of the block's contents falls below a *target value*. To create a valid block, a node in the Bitcoin network repeatedly selects a new nonce until it finds one that makes the block valid. Creating new blocks is called *mining*, and overall constitutes the *proof-of-work* approach since the repeated hashing is computationally (and thus environmentally) expensive. As the computational power used for mining (the *hash rate*) changes, the Bitcoin network periodically adjusts the target value such that the average time to find a new block (the *confirmation time*) is 10 minutes. In practice, the actual confirmation time varies; it was about

12 minutes in 2017 [24]. Every node in the network stores its own copy of the entire blockchain. Once a new node finds a new valid block, it broadcasts the block to the network. Due to network delays, multiple new blocks may propagate at the same time. Nodes add the first block they receive to their local chain. Thus multiple *forks* of the blockchain may exist on different nodes. Each node always considers the longest chain known to it as valid, and miners extend the longest chain. A transaction is *n-confirmed* with *confirmation depth* $n = 0$ if it is not part of any valid block and otherwise with $n > 1$ if there are $n - 1$ blocks in the chain beyond the block b that the transaction is part of. The amount of work to invalidate a fork that starts with b increases with n. Many services only accept Bitcoin payments once they are at least 6-confirmed [7].

In this section, we use Modest and the Modest Toolset to study two variants of a secret-fork attack on Bitcoin, inspired by the Andresen attack proposal and a study performed with Uppaal smc in [24]. The attackers secretly create a fork, keep mining on it until it reaches a certain length greater than that of the publicly known blockchain, and then publish it all at once. This would invalidate the public fork, with the private one becoming the valid blockchain. The original aim of the attack was to undermine the trust in Bitcoin; if it succeeds on the first attempted fork, it can equally be used for double spending by invalidating a specific transaction. For the attack to be feasible, the malicious attacker must control a significant fraction m of the hash rate.

4.1 Modelling and Evaluating the Double-Spending Attack

If the goal of the attacker is double spending, then it creates a transaction that spends some Bitcoin funds and announces it to the network for inclusion in the next block. At the same time, it starts mining on its own secret fork. Let cd be the confirmation depth after which a transaction is accepted by the receiver of the funds. If the attacker manages for its secret fork to become longer than the public fork, and longer than cd, then it can publish this fork immediately after the public one reaches length cd. At that point, the receiver of the funds has just accepted the transaction (and presumably fulfilled its part of the contract). The secret fork however invalidates the public one since it is longer, and thus invalidates the transaction. The attacker is now free to spend the same funds again. Due to the proof-of-work system, such an attack is possible, but—as long as the attacker controls less than 50 % of the hash rate—has a low success probability and an immense computational cost.

Modelling the attack in Modest. We build an abstract model of mining in Modest, reduced to the aspects relevant to the attack. The observation that a new block is mined every 12 minutes on average fits well with MA: we model block creation via Markovian transitions with a total rate of $\frac{1}{12}$. We abstract from network delays, i.e. blocks propagate instantaneously. We consider a single attacker, assuming that the rest of the world's miners behave in the normal "honest" manner and publish all mined blocks immediately.

Honest mining model. To start, we define a process `HonestPool` representing the pool of honest miners, which control $(1 - m) \cdot 100\,\%$ of the global computational resources used for mining, with m realised as model parameter `M`:

```
const real M; // fraction of hash rate controlled by malicious mining pool
action sln;   // indicates that the honest pool mined a new block

process HonestPool()
{
   rate(1/12 * (1 - M)) tau; // wait 12 / (1 − M) minutes on average
   sln;                      // signal that a new block was found
   HonestPool()              // repeat
}
```

Action `sln` models the propagation of a new block through the network, which can also be observed by the attacker. Due to the separation of timing and interaction in MA, we need two separate edges for mining delay and communication.

Attacker model. We keep track of the length of the attacker's fork, and of the difference in length to the public fork. To make the MA finite, we identify all fork lengths greater than *cd* with the value $cd + 1$ (since we only need to know *whether* the fork is longer than *cd*, but not *how much* longer), and we assume that the attacker gives up on its fork once it is *db* blocks shorter than the public one. The attacker process is then as follows:

```
const int CD;                 // confirmation depth required by victim
const int DB = CD;            // attacker gives up when this far behind
action cnt;                   // indicates that the attacker continues
int(0..CD+1) m_len;           // length of the secret fork
int(-DB..CD+1) m_diff = 0;    // length of secret fork minus honest fork
bool gup;                     // indicates whether the attacker gave up

process DoubleSpendingAttacker()
{
   do {
   :: rate(1/12 * M) {= m_len = min(CD+1, m_len + 1), m_diff++ =}
   :: sln {= m_diff-- =}; // public fork extended
      if(m_diff <= -DB) { tau {= gup = true =}; stop } // give up
      else { cnt }                                      // continue
   }
}
```

For illustration, we use the `do` construct to implement a loop here instead of the recursive process call used in `HonestPool`. A `do` loop is in essence a looping `alt`: There is an initial nondeterministic choice between the child behaviours; once the chosen behaviour successfully terminates, control loops back to the nondeterministic choice. `do` loops can be exited via the predefined `break` action. We also use the `if` shorthand: `if`(*e*) `{` *P* `}` `else` `{` *Q* `}` is syntactic sugar for `alt{ :: when`(*e*) *P* `:: when(!`*e*`)` *Q* `}`. Thus the behaviour of the attacker process is as follows: it waits until it either mines a new block itself (first child behaviour of the `do` loop), or until it observes a new block in the public fork. In both cases, it appropriately updates `m_len` and `m_diff`. In the second case, it then either gives up if it has fallen too far behind, or otherwise continues the attack.

Composition and nondeterminism. The overall behaviour of our model is the parallel composition of the two processes, with synchronisation on `sln`:

```
par {
:: HonestPool()
:: DoubleSpendingAttacker()
}
```

Observe that the behaviour of neither of the two processes contains an actual nondeterministic choice: `HonestPool` is entirely sequential, and the choices in the attacker process are between a Markovian and a probabilistic edge (in `do`), i.e. the probability for both to be available at the same time is 0, and between two edges with disjoint guards (in `if`). Since the only probabilistic edge in `HonestPool` synchronises with the attacker, and is immediately followed by a Markovian edge, the parallel composition cannot introduce nondeterminism due to interleaving probabilistic transitions, either. Thus the entire model takes the form of an MA, but is in fact equivalent to a CTMC. MA that are equivalent to CTMC are a class of models that occurs frequently in practice. Several of the MA models in the QVBS belong to this class.

Evaluating the attack. We are interested in the probability that the attacker eventually wins, and that it eventually gives up without winning. We expect it to eventually either win or give up, thus—due to the absence of nondeterminism—the probabilities should sum to 1. We declare the two properties in Modest:

```
function bool win() = m_len > CD && m_diff > 0; // winning condition
property P_Win = Pmin(<> win());                // attacker wins
property P_GiveUp = Pmin(!win() U gup);         // attacker gives up
```

To avoid repeating the expression that characterises the winning condition, we encapsulate it in the user-defined function `win()`. Functions in Modest can also take parameters, and they can be (mutually) recursive. The body of a function is an expression; since expressions in Modest are free of side effects, functions provide for pure functional programming inside Modest models. Combined with user-defined recursive datatypes (not shown in this paper), they make Modest Turing-complete. Property `P_Win` is straightforward: we ask for the (minimum) probability to eventually (`<>`) enter a state that satisfies the winning condition. Since there is no nondeterminism, there is no difference between `Pmin` and `Pmax` for this model. Property `P_GiveUp` uses the until (`U`) operator to ask for the probability of those paths on which no state satisfies `win()` until a state where `gup` is *true* is reached. If we invoke mcsta on this model by executing

```
./modest mcsta bitcoin-ds.modest -E "M = 0.2, CD = 6"
```

we obtain probability ≈ 0.0087 for `P_Win` and ≈ 0.9913 for `P_GiveUp`: the attack is unlikely to succeed if the attacker controls only 20 % of the hash rate. However, at $m = 0.4$, we get `P_Win` ≈ 0.343, and at $m = 0.5$, it is ≈ 0.719. It is not 1 here because the attacker gives up when falling behind too much. If we modify the model such that the attacker never gives up, it becomes an infinite-state MA since m_diff is no longer bounded from below. We cannot model-check this model, but due to the absence of nondeterminism, we can easily perform statistical model checking with modes by running

```
./modest modes bitcoin-ds-inf.modest -E "M=0.2, CD=6" --max-run-length 0
```

The output confirms our expectation that the probability is now 1, although we only know this with the statistical confidence provided by modes.

4.2 Optimising the Attack on Trust in Bitcoin

If the goal of the attack is to undermine the trust in the Bitcoin system by invalidating a large amount of work performed by the honest miners, the attacker gains some freedom in choices: Instead of having to give up when it gets too far behind, it can simply restart its attack from the then-current public fork. We thus keep the *cd* parameter, which now indicates the minimum desired length of the secret fork for it to be published. The winning condition becomes the length of the secret fork being greater than *or equal to cd*. Instead of only giving up (which now means resetting the secret fork) when *db* blocks behind, the attacker can additionally choose to continue the attack or reset its fork every time that the honest mining pool publishes a new block.

Modelling the attack. Our new attacker process, which replaces the DoubleSpendingAttacker process presented previously, is thus as follows:

```
action rst; // indicates that the attacker restarts from the public fork
process TrustAttacker()
{
   do {
   :: rate((1/12) * M) {= m_len = min(CD, m_len + 1), m_diff++ =}
   :: sln {= m_diff-- =}; // public fork extended
      alt { // strategy choice: restart or continue malicious fork
      :: rst {= m_len = 0, m_diff = 0 =} // can always restart
      :: when(m_diff > -DB) cnt // can continue if not too far behind
      }
   }
}
```

This model is nondeterministic due to the choice between rst and cnt in the attacker process. We use actions rst and cnt to indicate the choice made; they have no synchronisation partner, but will help understand the optimal scheduler.

Evaluation. The probability for the attacker to eventually win as expressed by an adjusted version of P_Win is now 1 since it can retry indefinitely. It is thus more interesting to investigate the expected time until it wins:

```
property T_WinMin = Xmin(T, m_len >= CD && m_diff > 0);
```

We ask for the *minimum* time here, i.e. for the attacker to make its choices such that the time to success is minimised, which arguably is its best strategy. mcsta reports that the value is ≈ 3735.94 minutes for $m = 0.2$, i.e. a little over two and a half days. Let us thus compute the probability to succeed in just two days:

```
property P_WinMax2 = Pmax(<>[T<=2880](m_len >= CD && m_diff > 0));
```

We now ask for the *maximum* probability, since this again corresponds to an optimal attack. The result that mcsta gives is ≈ 0.535. As originally discovered in [24], we thus have a more than 50 % chance to undermine the trust in Bitcoin if we control only 20 % of the hash rate and invest only two days of mining. According to blockchain.com/pools, on July 8, 2019, the BTC.com pool in fact controlled 21.6 % of the global hash rate; it could thus perform the attack.

Optimising the attack strategy. While the above numbers tell us the time and probability for the attack to succeed, they do not give any information about the attack strategy: What are the points, in terms of the length of the secret and public forks, where we should restart in order to obtain these optimal times and probabilities? Probabilistic model checking as implemented by mcsta, however, implicitly computes the optimal choice for every state of the MA underlying the model it checks, and it can be instructed to write this scheduler to a file:

```
./modest mcsta bitcoin-attack.modest -E "M=0.2,CD=6" --scheduler sched.txt
```

The result is a text file `sched.txt` with entries of the form

```
+ State: (HonestPool.location = loc_1, TrustAttacker.location = loc_10,
          m_len = 1, m_diff = -2)
  Choice: rst
```

for every state; here, in a state where the secret fork's length is 1, and it is two blocks shorter than the public one, the attacker restarts. We processed the file by projecting to `m_len` and `m_diff` and then eliminating all subsequent duplicate entries to find that the optimal strategy is to restart the attack if

- the honest pool announces a block, but the secret fork is still empty,
- the secret fork has one block and the public fork adds a third block, or
- the secret fork has ≥ 2 blocks and gets 3 blocks shorter than the public one,

and to continue the attack in all other cases.

Summary. Throughout this section, we first built an MA model that was equivalent to a CTMC, and then a truly nondeterministic MA. However, even that model does not use all features of the MA formalism: it lacks discrete probabilistic branching. As such, it falls into the interactive Markov chain (IMC, [39]) subset of MA. In the next section, we will introduce a model that is a true MA.

5 Evaluating a Reentrant Queueing System

In the previous section, we considered quantitative aspects of attacks on a stochastic timed system. We now turn our attention to a prominent use of

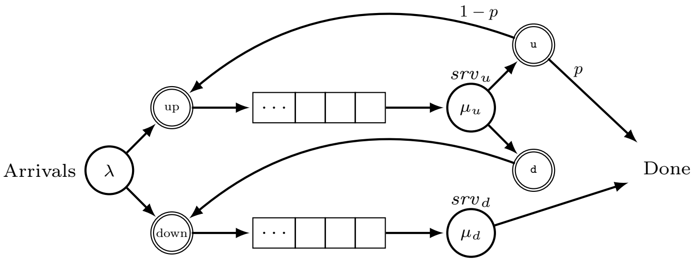


Fig. 7. A queuing system with postprocessing needs [36]

continuous-time Markov models: performance and dependability evaluation. A classic application is resource-sharing *queueing systems*, using various CTMC-based formalisms like (Jackson) queueing networks [41], with analytical or simulation-based techniques for the analysis. Yet these approaches are restricted, both in modelling and in analysis, to fully stochastic systems. MA as a model, and our analysis tools in the Modest Toolset, sit right at the edge between performance evaluation and model checking [2]. In particular, they add the concept of nondeterminism, which is at the core of classic qualitative model checking, to modelling formalisms and analysis algorithms that directly apply to performance evaluation scenarios. We now study a queueing system with stochastic timing, discrete probabilistic choices, and nondeterministic decisions—its model is thus an MA that does not fall into any of the existing subsets.

We consider the system with two queues depicted in Fig. 7, originally presented in [36]. Both queues have the same capacity c. Jobs arrive with rate λ and enter one of the queues according to the standard join-the-shortest-queue strategy. This strategy is implicitly nondeterministic if both queues are equally filled. For each queue, jobs are processed by a dedicated server, serving jobs with rates μ_u and μ_d, respectively. Jobs leaving the lower server leave the system, while jobs once processed by the upper server are subject to an additional check. Dependent on the (nondeterministic) outcome thereof, they are either sent into the lower queue again (action d), or (action u) they may either leave the queue (with probability p) or reenter the upper queue (with probability $1 - p$).

A Modest model. As usual, we start our Modest model by declaring all relevant constants, including the model parameters without specified values:

```
const int C;            // queue capacity
const int LAMBDA = 5;   // job arrival rate
const real MU_UP = 10;  // service rate of up server
const int MU_DOWN = 4;  // service rate of down server
const real P = 0.3;     // probability to be done after up server
```

In this model, we will use two *transient variables* to track when jobs are done, and when a job is dropped because both queues are full on arrival, or the queue in which it is due to re-enter after being processed by the up server is full:

```
transient int(0..1) done = 0; // 1 when a job is done, otherwise 0
transient int(0..1) loss = 0; // 1 when a job is dropped, otherwise 0
```

Unlike regular variables in MA^{V}, transient variables do not become part of the states. They can be used in assignments, but the assigned values are lost once the successor state is entered. However, the assigned value is visible to properties when the branch is taken, and we will make use of this later to define rewards.

We structure our model along the components shown in Fig. 7, defining a Modest process for each of them. The arrivals process and the down server have the simplest behaviours:

```
action put, get;
process Arrivals()                process ServerDown()
{                                 {
   rate(LAMBDA) tau;                 get;
   put;                              rate(MU_DOWN) tau {= done = 1 =};
   Arrivals()                        ServerDown()
}                                 }
```

Both processes synchronise with the input queues: Arrivals uses action put to enqueue a job that just arrived, and ServerDown uses action get to obtain a job to work on when idle, as soon as one is available. We will use synchronisation vectors to ensure that the synchronisation on put happens between Arrivals and exactly one of the two queues. Both queues use the same process definition:

```
int(0..C)[] q = [0, 0]; // array: number of jobs in the two queues
function bool isShortest(int id) = q[id] <= q[(id + 1) % 2];
process Queue(int id)
{
   do {
   :: when(isShortest(id)) // enqueue
      put {= q[id] = min(q[id]+1, C), loss = q[id] == C ? 1 : 0 =}
   :: when(q[id] > 0) get {= q[id]-- =} // dequeue
   }
}
```

To distinguish the two queues, we use a process parameter id, and a two-element array storing the lengths of the queues that the processes index with their id. Function isShortest indicates whether the queue with the given id is no longer than the other one. A queue only accepts new jobs when isShortest(id) is *true*; if the queue is full in that case, the job is dropped, and loss is (temporarily) set to 1. The get action removes a job from a non-empty queue.

Finally, the up server has the most complicated structure, since it manages the reentry of jobs that it has finished serving into the two queues:

```
action u, d, rup, rdn;
process ServerUp()
{
   get;                        // get job from queue
   rate(MU_UP) tau;            // serve job
   alt {                       // nondeterministic choice between u and d:
   :: u palt {                 // action u: probabilistic choice to either
      :1-P: {==}; rup          // reenter the up queue with probability 1−P,
      :  P: {= done = 1 =} // or leave the system with probability P
      }
   :: d; rdn               // action d: reenter the down queue
   };
   ServerUp()
}
```

The nondeterministic choice between u and d is a choice between (d) making the job surely leave the system within a certain expected time, at the cost of processing by the slower down server, and (u) taking the chance for the job to leave the system immediately, at the risk of it reentering the up queue. The optimal choice will likely depend on the current lengths of both queues.

Now that we have specified all the necessary processes, we can put them into a parallel composition. We have rather different synchronisation requirements: put shall use a binary synchronisation between Arrivals and one of the two queues, with a nondeterministic choice if both have the same lengths; get in a queue shall synchronise only with the one server for that queue; and rup and rdn shall look like a put to the respective queues. We could declare put as a binary action, and cleverly use the relabel construct to rename the other actions in a way that makes Modest create the correct synchronisation vectors internally. However, we can also just specify the desired vectors explicitly in a par:

```
par { put, put, get, get, put, put, u, d } {
    : put, put,  - ,  - ,  - ,  - , -, - : Arrivals()
    : put,  - , get,  - , put,  - , -, - : Queue(0)
    :  - ,  - , get,  - , rup, rdn, u, d : ServerUp()
    :  - , put,  - , get,  - , put, -, - : Queue(1)
    :  - ,  - ,  - , get,  - ,  - , -, - : ServerDown()
}
```

If we read the “columns” in the above specification from bottom to top, we read the synchronisation vectors, with the topmost entry being the action that labels the synchronising edge in the composed MA^{V}, and - corresponding to $\perp$.

Performance evaluation. We first add properties to investigate the probability and time until the queues are full, which is an undesirable condition that affects the dependability of the system by making it likely for jobs to be lost:

```
property ProbFullIsOne = Pmin(<> (q[0] == C && q[1] == C)) == 1;
property TminFull = Xmin(T, q[0] == C && q[1] == C);
property TmaxFull = Xmax(T, q[0] == C && q[1] == C);
property PminFull10 = Pmin(<>[T<=10] (q[0] == C && q[1] == C));
property PmaxFull10 = Pmax(<>[T<=10] (q[0] == C && q[1] == C));
```

We thus assert that the minimum probability for both queues to eventually be full is 1, which is a sanity check for the model; then we ask for the minimum and maximum of the expected time for both queues to be full, and of the probability for this to happen within 10 time units. By repeating the bottom two properties for different values of the time bound, we can obtain an approximation of the underlying cumulative distribution function over time. If we run mcsta with

```
./modest mcsta reentrant-q.modest -E "C=5" -O results.txt Minimal
```

we get an easy-to-parse file `results.txt` with the results:

```
"ProbFullIsOne": True
"TminFull": 7.165959461963808
"TmaxFull": 54.167593727326874
"PminFull10": 0.1338675853224175
"PmaxFull10": 0.7958342318163893
```

We see that the nondeterministic choices have a significant influence on the behaviour of the system; between the worst and best choices, the time to and probability for the undesirable event differs by a factor of 6 to 7. Since the standard probabilistic model checking algorithms implemented in mcsta are iterative numeric algorithms using double-precision floating-point numbers, every result is only an approximation of the true value despite the high number of decimal digits included in the output. The precision of mcsta is configurable.

Assume that we are designing a system of which our reentrant queueing system is an abstract model, and we have one parameter for which we must decide on a concrete value: the queue capacity c. We expect a higher capacity to improve throughput, utilisation, and reduce the number of lost jobs; however, it is also more costly to implement. We would thus like to find a good tradeoff between c and these quantities. We first specify properties that query for them:

```
property Throughput = Smax(S(done));
property Loss = Smax(S(loss));
property IdleOne = Smin(T(q[0] == 0 || q[1] == 0 ? 1 : 0));
property IdleBoth = Smin(T(q[0] == 0 && q[1] == 0 ? 1 : 0));
```

These queries are for long-run average rewards. The rewards are described by *accumulation expressions*: `S(done)` attaches to every branch (i.e. to every discrete step) the value of `done` after the branch's assignments have been executed (but before transient variables lose their values) as a branch reward. Expression `T(q[0] == 0 || q[1] == 0 ? 1 : 0)` sets the rate reward (accumulated over time) in every state to 1 if both queues are empty, and to 0 otherwise. We chose maximisation/minimisation as appropriate to correspond to the best possible strategy. We can ask mcsta to compute these quantities for many different values of c by specifying multiple experiments via the `-E` parameter:

```
./modest mcsta reentrant-q.modest -E "C=1" -E "C=2" -E "C=3" -E "C=4" \
    -E "C=5" -E "C=6" -E "C=7" -E "C=8" -E "C=9" -E "C=10" -E "C=11" \
    -E "C=12" -E "C=13" -E "C=14" -E "C=15" -E "C=16" -O perf.txt Minimal
```

We visualise the results in Fig. 8. The two lines converging to zero plot `IdleOne` (red, upper line) and `IdleBoth` (orange, lower). The other two lines plot `Throughput` (blue, upper) and `Loss` (purple, lower). We see that the fraction of time that the servers spent idle drops quickly with increasing c, whereas throughput and loss do not improve so much. Looking at this plot, we might choose c around 5 to 8.

Summary. In this section, we built a model for a queueing system that utilises all the features of the MA formalism. mcsta offers algorithms to calculate a

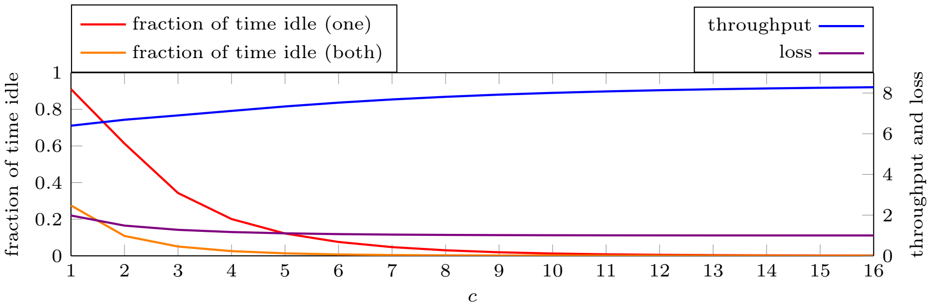


Fig. 8. Long-run average performance values for the reentrant queueing system

variety of quantities (cf. Sect. 2), and we fully utilised them to evaluate the system from several perspectives.

6 Conclusion

This tutorial paper has discussed how Modest can be used as a convenient modelling language for Markov automata, together with some hints on what analysis is possible for such models. Markov automata can be considered as a central model family for studying the performance, dependability, and correctness of randomised and distributed systems.

We introduced all the basic and several advanced constructs of the Modest language for MA. Among the features that we did not cover are exception handling (using the `throw` and `try-catch` constructs), the specification of values for transient variables in locations (using the `with` construct), dynamic array constructors, user-defined recursive datatypes (which allow the specification of, for example, unbounded list types), recursive functions, and `binary` and `broadcast` actions (which automatically generate appropriate synchronisation vectors, just like "normal" actions do for multi-way synchronisation). Going beyond MA, Modest also supports the formalisms of probabilistic timed automata [42] (which add a `clock` type and time progress conditions via the `constrain` construct), stochastic timed automata [4] (which allow sampling values from continuous probability distributions in assignments; they are a generalisation of MA), and stochastic hybrid automata [25] (which add continuous variables of type `var` whose behaviour over time is specified via differential equations and inclusions using the `der` operator for derivatives). Further Modest models are included in the Modest Toolset download, available at modestchecker.net, and in the Quantitative Verification Benchmark Set at qcomp.org.

Data availability. The models, example command lines, and results presented in this paper are archived and available at DOI 10.4121/uuid:5a73169e-b494-411b-b3a8-051e62efba9e [31].

Acknowledgments. The authors thank Michaela Klauck (Saarland University) for preparing an initial version of the Modest model appearing in Sect. 5 and for helpful comments on a draft of this paper.

References

1. Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 years of GreatSPN. In: Principles of Performance and Reliability Modeling and Evaluation. pp. 227–254. Springer (2016)
2. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Commun. ACM 53(9), 76–85 (2010)
3. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
4. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. 32(10), 812–830 (2006)
5. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing networks and Markov chains - modeling and performance evaluation with computer science applications (2nd edition). Wiley (2006)
6. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks 14, 25–59 (1987)
7. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: Sok: Research perspectives and challenges for Bitcoin and cryptocurrencies. In: SP. pp. 104–121. IEEE Computer Society (2015)
8. Braitling, B., Fioriti, L.M.F., Hatefi, H., Wimmer, R., Becker, B., Hermanns, H.: MeGARA: Menu-based game abstraction and abstraction refinement of Markov automata. In: QAPL. EPTCS, vol. 154, pp. 48–63 (2014)
9. Braitling, B., Fioriti, L.M.F., Hatefi, H., Wimmer, R., Becker, B., Hermanns, H.: Abstraction-based computation of reward measures for Markov automata. In: VMCAI. LNCS, vol. 8931, pp. 172–189. Springer (2015)
10. Brázdil, T., Hermanns, H., Krcál, J., Kretínský, J., Rehák, V.: Verification of open interactive Markov chains. In: FSTTCS. LIPIcs, vol. 18, pp. 474–485. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik (2012)
11. Budde, C.E., D'Argenio, P.R., Hartmanns, A., Sedwards, S.: A statistical model checker for nondeterminism and rare events. In: TACAS. LNCS, vol. 10806, pp. 340–358. Springer (2018)
12. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative model and tool interaction. In: TACAS. LNCS, vol. 10206, pp. 151–168 (2017)
13. Butkova, Y., Hartmanns, A., Hermanns, H.: A Modest approach to modelling and checking Markov automata. In: QEST. LNCS, Springer (2019), to appear.
14. Butkova, Y., Hatefi, H., Hermanns, H., Krcál, J.: Optimal continuous time Markov decisions. In: ATVA. LNCS, vol. 9364, pp. 166–182. Springer (2015)
15. Butkova, Y., Wimmer, R., Hermanns, H.: Long-run rewards for Markov automata. In: TACAS. LNCS, vol. 10206, pp. 188–203 (2017)
16. Butkova, Y., Wimmer, R., Hermanns, H.: Markov automata on discount! In: MMB. LNCS, vol. 10740, pp. 19–34. Springer (2018)
17. D'Argenio, P.R., Hartmanns, A., Sedwards, S.: Lightweight statistical model checking in nondeterministic continuous time. In: ISoLA. LNCS, vol. 11245, pp. 336–353. Springer (2018)

18. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is coming: A modern probabilistic model checker. In: CAV. LNCS, vol. 10427, pp. 592–600. Springer (2017)
19. Eisentraut, C.: Principles of Markov automata. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2017)
20. Eisentraut, C., Hermanns, H., Katoen, J.P., Zhang, L.: A semantics for every GSPN. In: Petri Nets. LNCS, vol. 7927, pp. 90–109. Springer (2013)
21. Eisentraut, C., Hermanns, H., Schuster, J., Turrini, A., Zhang, L.: The quest for minimal quotients for probabilistic and Markov automata. Inf. Comput. 262(Part), 162–186 (2018)
22. Eisentraut, C., Hermanns, H., Zhang, L.: Concurrency and composition in a stochastic world. In: CONCUR. LNCS, vol. 6269, pp. 21–39. Springer (2010)
23. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS. pp. 342–351. IEEE Computer Society (2010)
24. Fehnker, A., Chaudhary, K.: Twenty percent and a few days - optimising a Bitcoin majority attack. In: NFM. LNCS, vol. 10811, pp. 157–163. Springer (2018)
25. Fränzle, M., Hahn, E.M., Hermanns, H., Wolovick, N., Zhang, L.: Measurability and safety verification for stochastic hybrid systems. In: HSCC. pp. 43–52. ACM (2011)
26. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT 15(2), 89–107 (2013)
27. Guck, D., Han, T., Katoen, J.P., Neuhäußer, M.R.: Quantitative timed analysis of interactive Markov chains. In: NFM. LNCS, vol. 7226, pp. 8–23. Springer (2012)
28. Guck, D., Hatefi, H., Hermanns, H., Katoen, J.P., Timmer, M.: Analysis of timed and long-run objectives for Markov automata. Logical Methods in Computer Science 10(3) (2014)
29. Guck, D., Timmer, M., Hatefi, H., Ruijters, E., Stoelinga, M.: Modelling and analysis of Markov reward automata. In: ATVA. LNCS, vol. 8837, pp. 168–184. Springer (2014)
30. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods in System Design 43(2), 191–232 (2013)
31. Hartmanns, A.: A Modest Markov automata tutorial (artifact). 4TU.Centre for Research Data (2019), doi.org/10.4121/uuid:5a73169e-b494-411b-b3a8-051e62efba9e
32. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: TACAS. LNCS, vol. 8413, pp. 593–598. Springer (2014)
33. Hartmanns, A., Hermanns, H.: Explicit model checking of very large MDP using partitioning and secondary storage. In: ATVA. vol. 9364, pp. 131–147. Springer (2015)
34. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS. LNCS, vol. 11427, pp. 344–350. Springer (2019)
35. Hatefi, H.: Finite horizon analysis of Markov automata. Ph.D. thesis, Saarland University, Germany (2017), scidok.sulb.uni-saarland.de/volltexte/2017/6743/
36. Hatefi, H., Hermanns, H.: Model checking algorithms for Markov automata. Electronic Communications of the EASST 53 (2012)
37. Hatefi, H., Wimmer, R., Braitling, B., Fioriti, L.M.F., Becker, B., Hermanns, H.: Cost vs. time in stochastic games and Markov automata. Formal Asp. Comput. 29(4), 629–649 (2017)

38. Haverkort, B.R.: Performance of computer communication systems - a model-based approach. Wiley (1998)
39. Hermanns, H.: Interactive Markov Chains: The Quest for Quantified Quality, LNCS, vol. 2428. Springer (2002)
40. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
41. Jackson, J.R.: Jobshop-like queueing systems. Management Science 10(1), 131–142 (1963)
42. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. 282(1), 101–150 (2002)
43. Legay, A., Sedwards, S., Traonouez, L.M.: Scalable verification of Markov decision processes. In: WS-FMDS at SEFM. LNCS, vol. 8938, pp. 350–362. Springer (2014)
44. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
45. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009), bitcoin.org
46. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994)
47. Quatmann, T., Junges, S., Katoen, J.P.: Markov automata with multiple objectives. In: CAV. LNCS, vol. 10426, pp. 140–159. Springer (2017)
48. Rabe, M.N., Schewe, S.: Finite optimal control for time-bounded reachability in CTMDPs and continuous-time Markov games. Acta Inf. 48(5-6), 291–315 (2011)
49. Segala, R.: Modeling and verification of randomized distributed real-time systems. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (1995)
50. Timmer, M.: Efficient modelling, generation and analysis of Markov automata. Ph.D. thesis, University of Twente, Enschede, Netherlands (2013)
51. Timmer, M., Katoen, J.P., van de Pol, J., Stoelinga, M.: Efficient modelling and generation of Markov automata. In: CONCUR. LNCS, vol. 7454, pp. 364–379. Springer (2012)
52. Timmer, M., Katoen, J.P., van de Pol, J., Stoelinga, M.: Confluence reduction for Markov automata. Theor. Comput. Sci. 655, 193–219 (2016)