High-level Counterexamples for Probabilistic Automata

Ralf Wimmer[1], Nils Jansen[2], Andreas Vorpahl[2], Erika Ábrahám[2], Joost-Pieter Katoen[2], and Bernd Becker[1]

[1] Albert-Ludwigs-University Freiburg, Germany
{wimmer | becker}@informatik.uni-freiburg.de
[2] RWTH Aachen University, Germany
{nils.jansen | abraham | katoen}@cs.rwth-aachen.de
andreas.vorpahl@rwth-aachen.de *

Abstract. Providing compact and understandable counterexamples for violated system properties is an essential task in model checking. Existing works on counterexamples for probabilistic systems so far computed either a large set of system runs or a subset of the system's states, both of which are of limited use in manual debugging. Many probabilistic systems are described in a guarded command language like the one used by the popular model checker PRISM. In this paper we describe how a minimal subset of the commands can be identified which together already make the system erroneous. We additionally show how the selected commands can be further simplified to obtain a well-understandable counterexample.

1 Introduction

The ability to provide *counterexamples* for violated properties is one of the most essential features of model checking [1]. Counterexamples make errors reproducible and are used to guide the designer of an erroneous system during the debugging process. Furthermore, they play an important role in counterexample-guided abstraction refinement (CEGAR) [2–5]. For linear-time properties of digital or hybrid systems, a single violating run suffices to refute the property. Thereby, this run—acquired during model checking—directly forms a counterexample.

Probabilistic formalisms like discrete-time Markov chains (DTMCs), Markov decision processes (MDPs) and probabilistic automata (PAs) are well-suited to model systems with uncertainties. Violating behavior in the probabilistic setting means that the probability that a certain property holds is outside of some required bounds. For probabilistic reachability properties, this can be reduced to the case where an upper bound on the probability is exceeded [6]. Thereby, a *probabilistic counterexample* is formed by a set of runs that all satisfy a given property while their probability mass is larger than the allowed upper bound.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center AVACS (SFB/TR 14), the DFG project CEBug (AB 461/1-1), and the EU-FP7 IRSES project MEALS.

Tools like PRISM [7] verify probabilistic systems by computing the solution of a linear equation system. While this technique is very efficient, the simultaneous generation of counterexamples is not supported.

During the last years, a number of approaches have been proposed to compute probabilistic counterexamples by enumerating certain paths of a system [8, 6, 9]. In general, such a set may be extremely large; for some systems it is at least double exponential in the number of system states [6]. Also different compact representations of counterexamples have been devised, e. g., counterexamples are described symbolically by regular expressions in [6], while in [10] and [11] the abstraction of strongly connected components yields loop-free systems.

A different representation is obtained by taking a preferably small subset of the state space, forming a *critical subsystem*. Inside this part of the original system the property is already violated, see [8] and [11]. Both approaches use heuristic path search algorithms to incrementally build such critical subsystems for probabilistic reachability properties. In [12–14], a different approach was suggested: not only a small subsystem, but a minimal one is computed for a large class of properties, namely probabilistic reachability and ω-regular properties for both DTMCs and MDPs. This is achieved using solver techniques such as mixed integer linear programming (MILP) [15].

An unanswered question for all these approaches is how they can actually be used for debugging. Most practical examples are built by the parallel composition of modules forming a flat state-space with millions of states. Although critical subsystems are often smaller by orders of magnitude than the original system, they may still be very large, rendering manual debugging practically impossible.

In this paper, we focus on the non-deterministic and fully compositional model of probabilistic automata (PA) [16, 17]. The specification of such models is generally done in a high-level language allowing the parallel composition of modules. The modules of the system are not specified by enumerating states and transitions but can be described using a *guarded command language* [18, 19] like the one used by PRISM. The communication between different modules takes place using synchronization on common actions and via shared variables. Having this human-readable specification language, it seems natural that a user should be pointed to the part of the system description which causes the error, instead of referring to the probabilistic automaton defined by the composition. To the best of our knowledge, no work on probabilistic counterexamples has considered this sort of *high-level counterexamples* yet.

We show how to identify a *smallest set of guarded commands* which induces a critical subsystem. In order to correct the system, at least one of the returned commands has to be changed. We additionally simplify the commands by removing branching choices which are not necessary to obtain a counterexample. We present this as a special case of a method where the number of different transition labels for a PA is minimized. This offers great flexibility in terms of human-readable counterexamples. The NP-hard computation of such a smallest critical label set is done by the established approach of mixed integer linear programming.

Structure of the paper. In Section 2 we review some foundations. Our approach to obtain smallest command sets is presented in Section 3. How the essential commands can be simplified is described in Section 4. After some experimental results in Section 5 we conclude the paper in Section 6.

2 Foundations

Let S be a countable set. A *sub-distribution* on S is a function $\mu : S \to [0,1]$ such that $0 \leq \sum_{s \in S} \mu(s) \leq 1$. We use the notation $\mu(S') = \sum_{s \in S'} \mu(s)$ for a subset $S' \subseteq S$. A sub-distribution with $\mu(S) = 1$ is called a *probability distribution*. We denote the set of all probability distributions on S by $\mathrm{Distr}(S)$ and analogously by $\mathrm{SubDistr}(S)$ for sub-distributions.

Probabilistic Automata

Definition 1 (Probabilistic automaton). *A* probabilistic automaton (PA) *is a tuple $\mathcal{M} = (S, s_{init}, \mathrm{Act}, P)$ such that S is a finite set of states, $s_{init} \in S$ is an initial state,* Act *is a finite set of actions, and $P : S \to 2^{\mathrm{Act} \times \mathrm{Distr}(S)}$ is a probabilistic transition relation such that $P(s)$ is finite for all $s \in S$.*

In the following we also use η to denote an action-distribution pair (α, μ). We further define $\mathrm{succ}(s, \alpha, \mu) = \{s' \in S \mid \mu(s') > 0\}$ for $(\alpha, \mu) \in P(s)$, $\mathrm{succ}(s) = \bigcup_{(\alpha,\mu) \in P(s)} \mathrm{succ}(s, \alpha, \mu)$, and $\mathrm{pred}(s) = \{s' \in S \mid \exists (\alpha, \mu) \in P(s') : \mu(s) > 0\}$.

The evolution of a probabilistic automaton is as follows: Starting in the initial state $s = s_{\mathrm{init}}$, first a transition $(\alpha, \mu) \in P(s)$ is chosen non-deterministically. Then the successor state $s' \in \mathrm{succ}(s, \alpha, \mu)$ is determined probabilistically according to the distribution μ. This process is repeated for the successor state s'. To prevent deadlocks we assume $P(s) \neq \emptyset$ for all $s \in S$.

An *infinite path* of a PA $\mathcal{M}$ is an infinite sequence $s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1)\ldots$ with $s_i \in S$, $(\alpha_i, \mu_i) \in P(s_i)$ and $s_{i+1} \in \mathrm{succ}(s_i, \alpha_i, \mu_i)$ for all $i \geq 0$. A *finite path* π of $\mathcal{M}$ is a finite prefix $s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1)\ldots s_n$ of an infinite path of $\mathcal{M}$ with last state $\mathrm{last}(\pi) = s_n$. We denote the set of all finite paths of $\mathcal{M}$ by $\mathrm{Paths}^{\mathrm{fin}}_{\mathcal{M}}$.

A *sub-PA* is like a PA, but it allows sub-distributions instead of probability distributions in the definition of P.

Definition 2 (Subsystem). *A sub-PA $\mathcal{M}' = (S', s'_{init}, \mathrm{Act}', P')$ is a* subsystem *of a sub-PA $\mathcal{M} = (S, s_{init}, \mathrm{Act}, P)$, written $\mathcal{M}' \sqsubseteq \mathcal{M}$, iff $S' \subseteq S$, $s'_{init} = s_{init}$, $\mathrm{Act}' \subseteq \mathrm{Act}$ and for all $s \in S'$ there is an injective function $f : P'(s) \to P(s)$ such that for all $(\alpha', \mu') \in P'(s)$ with $f((\alpha', \mu')) = (\alpha, \mu)$ we have that $\alpha' = \alpha$ and for all $s' \in S'$ either $\mu'(s') = 0$ or $\mu'(s') = \mu(s')$.*

A sub-PA $\mathcal{M} = (S, s_{\mathrm{init}}, \mathrm{Act}, P)$ can be transformed into a PA as follows: We add a new state $s_\perp \notin S$, turn all sub-distributions into probability distributions by defining $\mu(s_\perp) := 1 - \mu(S)$ for each $s \in S$ and $(\alpha, \mu) \in P(s)$, and make $s_\perp$ absorbing by setting $P(s_\perp) := \{(\tau, \mu) \in \mathrm{Act} \times \mathrm{Distr}(S \cup \{s_\perp\}) \mid \mu(s_\perp) = 1\}$. This way all methods we formalize for PAs can also be applied to sub-PAs.

Before a probability measure on PAs can be defined, the nondeterminism has to be resolved. This is done by an entity called *scheduler*.

Definition 3 (Scheduler). *A* scheduler *for a PA* $\mathcal{M} = (S, s_{init}, \mathrm{Act}, P)$ *is a function* $\sigma : \mathrm{Paths}^{\mathrm{fin}}_{\mathcal{M}} \to \mathrm{SubDistr}(\mathrm{Act} \times \mathrm{Distr}(S))$ *such that* $\sigma(\pi)(\alpha, \mu) > 0$ *implies* $(\alpha, \mu) \in P(\mathrm{last}(\pi))$ *for all* $\pi \in \mathrm{Paths}^{\mathrm{fin}}_{\mathcal{M}}$ *and* $(\alpha, \mu) \in \mathrm{Act} \times \mathrm{Distr}(S)$. *We use* $\mathrm{Sched}_{\mathcal{M}}$ *to denote the set of all schedulers of* $\mathcal{M}$.

By resolving the nondeterminism, a scheduler turns a PA into a fully probabilistic model, for which a standard probability measure can be defined [20, Chapter 10.1]. In this paper we are interested in *probabilistic reachability properties*: Is the probability to reach a set $T \subseteq S$ of target states from s_{init} at most equal to a given bound $\lambda \in [0, 1] \subseteq \mathbb{R}$? Such a reachability property will be denoted with $\mathcal{P}_{\leq\lambda}(\Diamond\, T)$. Note that checking ω-regular properties can be reduced to checking reachability properties. For a fixed scheduler σ, this probability $\Pr^{\sigma}_{\mathcal{M}}(s_{\mathrm{init}}, \Diamond\, T)$ can be computed by solving a linear equation system. However, for a PA without a scheduler, this question is not well-posed. Instead we ask: Is the probability to reach a set $T \subseteq S$ of target states from s_{init} at most λ for all schedulers? That means, $\mathcal{P}_{\leq\lambda}(\Diamond\, T)$ has to hold for all schedulers. To check this, it suffices to compute the maximal probability over all schedulers that T is reached from s_{init}, which we denote with $\Pr^{+}_{\mathcal{M}}(s_{\mathrm{init}}, \Diamond\, T)$. One can show that for this kind of properties maximizing over a certain subclass of all schedulers suffices, namely the so-called memoryless deterministic schedulers, which can be seen as functions $\sigma : S \to \mathrm{Act} \times \mathrm{SubDistr}(S)$.

Definition 4 (Memoryless deterministic scheduler). *A scheduler* σ *of* $\mathcal{M} = (S, s_{init}, \mathrm{Act}, P)$ *is* memoryless *if* $\mathrm{last}(\pi) = \mathrm{last}(\pi')$ *implies* $\sigma(\pi) = \sigma(\pi')$ *for all* $\pi, \pi' \in \mathrm{Paths}^{\mathrm{fin}}_{\mathcal{M}}$. *The scheduler* σ *is* deterministic *if* $\sigma(\pi)(\eta) \in \{0, 1\}$ *for all* $\pi \in \mathrm{Paths}^{\mathrm{fin}}_{\mathcal{M}}$ *and* $\eta \in \mathrm{Act} \times \mathrm{Distr}(S)$.

The maximal probability $\Pr^{+}_{\mathcal{M}}(s, \Diamond\, T)$ to reach T from s is obtained as the unique solution of the following equation system: $\Pr^{+}_{\mathcal{M}}(s, \Diamond\, T) = 1$, if $s \in T$; $\Pr^{+}_{\mathcal{M}}(s, \Diamond\, T) = 0$, if T is unreachable from s under all schedulers, and $\Pr^{+}_{\mathcal{M}}(s, \Diamond\, T) = \max_{(\alpha,\mu)\in P(s)} \sum_{s'\in S} \mu(s') \cdot \Pr^{+}_{\mathcal{M}}(s', \Diamond\, T)$ otherwise. It can be solved by either rewriting it into a linear program, by applying a technique called value iteration, or by iterating over the possible schedulers (policy iteration) (see, e. g., [20, Chapter 10.6]). A memoryless deterministic scheduler is obtained from the solution by taking an arbitrary element of $P(s)$ in the first two cases and an element of $P(s)$ for which the maximum is obtained in the third case.

`PRISM`'s Guarded Command Language For a set Var of Boolean variables, let $\mathcal{A}_{\mathrm{Var}}$ denote the set of variable assignments, i. e., of functions $\nu : \mathrm{Var} \to \{0, 1\}$.

Definition 5 (Model, module, command). *A* model *is a tuple* $(\mathrm{Var}, s_{init}, M)$ *where* Var *is a finite set of Boolean variables,* $s_{init} : \mathrm{Var} \to \{0, 1\}$ *the initial state, and* $M = \{M_1, \ldots, M_k\}$ *a finite set of modules.*

A module *is a tuple* $M_i = (\mathrm{Var}_i, \mathrm{Act}_i, C_i)$ *with* $\mathrm{Var}_i \subseteq \mathrm{Var}$ *a set of variables such that* $\mathrm{Var}_i \cap \mathrm{Var}_j = \emptyset$ *for* $i \neq j$, Act_i *a finite set of synchronizing actions, and* C_i *a finite set of commands. The action* τ *with* $\tau \notin \bigcup_{i=1}^{k} \mathrm{Act}_i$ *denotes the internal non-synchronizing action. A* command $c \in C_i$ *has the form*

$$c \;=\; [\alpha]\; g \rightarrow\; p_1 : f_1 + \ldots + p_n : f_n$$

with $\alpha \in \mathrm{Act}_i \,\dot\cup\, \{\tau\}$, g *a Boolean predicate ("guard") over the variables in* Var, $p_i \in [0,1]$ *a rational number with* $\sum_{i=1}^{n} p_i = 1$, *and* $f_i : \mathcal{A}_{\mathrm{Var}} \rightarrow \mathcal{A}_{\mathrm{Var}_i}$ *being a variable update function. We refer to the action* α *of* c *by* $act(c)$.

Note that each variable may be written by only one module, but the update may depend on variables of other modules. Each model with several modules is equivalent to a model with a single module which is obtained by computing the parallel composition of these modules. We give a short intuition on how this composition is built. For more details we refer to the documentation of `PRISM`. Assume two modules $M_1 = (\mathrm{Var}_1, \mathrm{Act}_1, C_1)$ and $M_2 = (\mathrm{Var}_2, \mathrm{Act}_2, C_2)$ with $\mathrm{Var}_1 \cap \mathrm{Var}_2 = \emptyset$. The *parallel composition* $M = M_1 || M_2 = (\mathrm{Var}, \mathrm{Act}, C)$ is given by $\mathrm{Var} = \mathrm{Var}_1 \cup \mathrm{Var}_2$, $\mathrm{Act} = \mathrm{Act}_1 \cup \mathrm{Act}_2$ and

$$\begin{aligned} C = \{\; c \quad & \mid c \in C_1 \cup C_2 \wedge act(c) \in \{\tau\} \cup (\mathrm{Act}_1 \setminus \mathrm{Act}_2) \cup (\mathrm{Act}_2 \setminus \mathrm{Act}_1) \;\} \cup \\ \{\; c \otimes c' & \mid c \in C_1 \wedge c' \in C_2 \wedge act(c) = act(c') \in \mathrm{Act}_1 \cap \mathrm{Act}_2 \;\}\,, \end{aligned}$$

where $c \otimes c'$ for $c = [\alpha]\; g \rightarrow\; p_1 : f_1 + \ldots + p_n : f_n \in C_1$ and $c' = [\alpha]\; g' \rightarrow\; p'_1 : f'_1 + \ldots + p'_m : f'_m \in C_2$ is defined as

$$\begin{aligned} c \otimes c' \;=\; [\alpha]\; g \wedge g' \rightarrow \quad & p_1 \cdot p'_1 : f_1 \otimes f'_1 + \ldots + p_n \cdot p'_1 : f_n \otimes f'_1 \\ & \ldots \\ & + p_1 \cdot p'_m : f_1 \otimes f'_n + \ldots + p_n \cdot p'_n : f_n \otimes f'_m. \end{aligned}$$

Here, for $f_i : \mathcal{A}_{\mathrm{Var}} \rightarrow \mathcal{A}_{\mathrm{Var}_1}$ and $f'_j : \mathcal{A}_{\mathrm{Var}} \rightarrow \mathcal{A}_{\mathrm{Var}_2}$ we define $f_i \otimes f'_j : \mathcal{A}_{\mathrm{Var}} \rightarrow \mathcal{A}_{\mathrm{Var}_1 \cup \mathrm{Var}_2}$ such that for all $\nu \in \mathcal{A}_{\mathrm{Var}}$ we have that $(f_i \otimes f'_j)(\nu)(x)$ equals $f_i(\nu)(x)$ for each $x \in \mathrm{Var}_1$ and $f'_j(\nu)(x)$ for each $x \in \mathrm{Var}_2$.

Intuitively, commands labeled with non-synchronizing actions are executed on their own, while for synchronizing actions a command from each synchronizing module is executed simultaneously. Note that if a module has an action in its synchronizing action set but no commands labeled with this action, this module will block the execution of commands with this action in the composition. This is considered to be a modeling error and the corresponding commands are ignored.

The PA-semantics of a model is as follows. Assume a model $(\mathrm{Var}, s_{\mathrm{init}}, M)$ with a single module $M = (\mathrm{Var}, \mathrm{Act}, C)$ which will not be subject to parallel composition any more. The *state space* S of the corresponding PA $\mathcal{M} = (S, s_{\mathrm{init}}, \mathrm{Act}, P)$ is given by the set of all possible variable assignments $\mathcal{A}_{\mathrm{Var}}$, i. e., a state s is a vector $(x_1, \ldots, x_m)$ with x_i being a value of the variable $v_i \in \mathrm{Var} = \{v_1, \ldots, v_m\}$. To construct the transitions, we observe that the guard g of each command

$$c \;=\; [\alpha]\; g \rightarrow\; p_1 : f_1 + \ldots + p_n : f_n \;\in C$$

defines a subset of the state space $S_c \subseteq \mathcal{A}_{\text{Var}}$ with $s \in S_c$ iff s satisfies g. Each update $f_i : \mathcal{A}_{\text{Var}} \to \mathcal{A}_{\text{Var}}$ maps a state $s' \in S$ to each $s \in S_c$. Together with the associated values p_i, we define a probability distribution $\mu_{c,s} : S \to [0,1]$ with

$$\mu_{c,s}(s') = \sum_{\{i \mid 1 \leq i \leq n \wedge f_i(s) = s'\}} p_i$$

for each $s' \in \mathcal{A}_{\text{Var}}$. The probabilistic transition relation $P : \mathcal{A}_{\text{Var}} \to 2^{\text{Act} \times \text{Distr}(\mathcal{A}_{\text{Var}})}$ is given by $P(s) = \{(\alpha, \mu_{c,s}) \mid c \in C \wedge act(c) = \alpha \wedge s \in S_c\}$ for all $s \in \mathcal{A}_{\text{Var}}$.

Mixed Integer Programming A *mixed integer linear program* optimizes a linear objective function under a condition specified by a conjunction of linear inequalities. A subset of the variables in the inequalities is restricted to take only integer values, which makes solving MILPs NP-hard [21, Problem MP1].

Definition 6 (Mixed integer linear program). *Let $A \in \mathbb{Q}^{m \times n}$, $B \in \mathbb{Q}^{m \times k}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$, and $d \in \mathbb{Q}^k$. A* mixed integer linear program *(MILP) consists in computing* $\min c^T x + d^T y$ *such that* $Ax + By \leq b$ *and* $x \in \mathbb{R}^n,\ y \in \mathbb{Z}^k$.

MILPs are typically solved by a combination of a branch-and-bound algorithm and the generation of so-called cutting planes. These algorithms heavily rely on the fact that relaxations of MILPs which result by removing the integrality constraints can be efficiently solved. MILPs are widely used in operations research, hardware-software co-design, and numerous other applications. Efficient open source as well as commercial implementations are available like `Scip` [22], `Cplex` [23], or `Gurobi` [24]. We refer to, e. g., [15] for more information on solving MILPs.

3 Computing Counterexamples

In this section we show how to compute smallest critical command sets. For this, we introduce a generalization of this problem, namely smallest critical labelings, state the complexity of the problem, and specify an MILP formulation which yields a smallest critical labeling.

Let $\mathcal{M} = (S, s_{\text{init}}, \text{Act}, P)$ be a PA, $T \subseteq S$, Lab a finite set of labels, and $L : S \times \text{Act} \times \text{Distr}(S) \nrightarrow 2^{\text{Lab}}$ a partial labeling function such that $L(s,\eta)$ is defined iff $\eta \in P(s)$. Let $\text{Lab}' \subseteq \text{Lab}$ be a subset of the labels. The *PA induced by* Lab' is $\mathcal{M}_{|\text{Lab}'} = (S, s_{\text{init}}, \text{Act}, P')$ such that for all $s \in S$ we have $P'(s) = \{\eta \in P(s) \mid L(s,\eta) \subseteq \text{Lab}'\}$.

Definition 7 (Smallest critical labeling problem). *Let $\mathcal{M}$, T,* Lab *and L be defined as above and $\mathcal{P}_{\leq\lambda}(\Diamond T)$ be a reachability property that is violated by s_{init} in $\mathcal{M}$. A subset* $\text{Lab}' \subseteq \text{Lab}$ *is* critical *if* $\Pr^{+}_{\mathcal{M}_{|\text{Lab}'}}(s_{init}, \Diamond T) > \lambda$.

Given a weight function $w : \text{Lab} \to \mathbb{R}^{\geq 0}$, the smallest critical labeling problem *is to determine a critical subset* $\text{Lab}' \subseteq \text{Lab}$ *such that* $w(\text{Lab}') := \sum_{\ell \in \text{Lab}'} w(\ell)$ *is minimal among all critical subsets of* Lab.

Theorem 1. *To decide whether there is a critical labeling* $\mathrm{Lab}' \subseteq \mathrm{Lab}$ *with* $w(\mathrm{Lab}') \leq k$ *for a given integer* $k \geq 0$ *is NP-complete.*

A proof of this theorem, which is based on the reduction of exact 3-cover [21, Problem SP2] is given in the extended version [25] of this paper.

The concept of smallest critical labelings gives us a flexible description of counterexamples being minimal with respect to different quantities.

Commands In order to minimize the number of commands that together induce an erroneous system, let $\mathcal{M} = (S, s_{\mathrm{init}}, \mathrm{Act}, P)$ be a PA generated by modules $M_i = (\mathrm{Var}_i, \mathrm{Act}_i, C_i)$, $i = 1, \ldots, k$. For each module M_i and each command $c \in C_i$ we introduce a unique label[3] $\ell_{c,i}$ with weight 1 and define the labeling function $L : S \times \mathrm{Act} \times \mathrm{Distr}(S) \to 2^{\mathrm{Lab}}$ such that each transition is labeled with the set of commands which together generate this transition[4]. Note that in case of synchronization several commands together create a certain transition. A smallest critical labeling corresponds to a *smallest critical command set*, being a smallest set of commands which together generate an erroneous system.

Modules We can also minimize the number of modules involved in a counterexample by using the same label for all commands in a module. Often systems consist of a number of copies of the same module, containing the same commands, only with the variables renamed, plus a few extra modules. Consider for example a wireless network: n nodes want to transmit messages using a protocol for medium access control [26]. All nodes run the same protocol. Additionally there may be a module describing the channel. When fixing an erroneous system, one wants to preserve the identical structure of the nodes. Therefore the selected commands should contain the same subset of commands from all identical modules. This can be obtained by assigning the same label to all corresponding commands from the symmetric modules and using the number of symmetric modules as its weight.

States The state-minimal subsystems as introduced in [12] can be obtained as special case of smallest critical labelings: For each state $s \in S$ introduce a label ℓ_s and set $L(s, \eta) = \{\ell_s\}$ for all $\eta \in P(s)$. $\mathrm{Lab}' \subseteq \mathrm{Lab} = \{\ell_s \mid s \in S\}$ is a smallest critical labeling iff $S' = \{s \in S \mid \ell_s \in \mathrm{Lab}'\}$ induces a minimal critical subsystem.

We will now explain how these smallest critical labelings are computed. First, the notions of *relevant* and *problematic* states are considered. Intuitively, a state s is relevant, if there exists a scheduler such that a target state is reachable from s. A state s is problematic, if there additionally exists a deadlock-free scheduler under that no target state is reachable from s.

Definition 8 (Relevant and problematic states). *Let* $\mathcal{M}$*,* T*, and* L *be as above. The* relevant states of $\mathcal{M}$ for T *are given by* $S_T^{\mathrm{rel}} = \{s \in S \mid \exists \sigma \in \mathrm{Sched}_{\mathcal{M}} : \Pr^{\sigma}_{\mathcal{M}}(s, \Diamond T) > 0\}$*. A label* ℓ *is* relevant *for* T *if there is* $s \in S_T^{\mathrm{rel}}$ *and* $\eta \in P(s)$ *such that* $S_T^{\mathrm{rel}} \cap \mathrm{succ}(s, \eta) \neq \emptyset$ *and* $\ell \in L(s, \eta)$*.*

[3] In the following we write short l_c instead of $l_{c,i}$ if the index i is clear from the context.

[4] If several command sets generate the same transition, we make copies of the transition.

Let $\mathrm{Sched}^{+}_{\mathcal{M}}$ *be the set of all schedulers* σ *with* $\{\eta \mid \sigma(\pi)(\eta) > 0\} \neq \emptyset$ *for all* π*. The states in* $S^{\mathrm{prob}}_T = \{s \in S^{\mathrm{rel}}_T \,|\, \exists \sigma \in \mathrm{Sched}^{+}_{\mathcal{M}} : \Pr^{\sigma}_{\mathcal{M}}(s, \Diamond T) = 0\}$ *are* problematic states *and the set* $P^{\mathrm{prob}}_T = \left\{(s,\eta) \in S^{\mathrm{prob}}_T \times \mathrm{Act} \times \mathrm{Distr}(S) \,\middle|\, \eta \in P(s) \wedge \mathrm{succ}(s,\eta) \subseteq S^{\mathrm{prob}}_T\right\}$ *are* problematic transitions *regarding* T*.*

Both relevant states and problematic states and actions can be computed in linear time using graph algorithms [27].

States that are not relevant can be removed from the PA together with all their incident edges without changing the probability of reaching T from s_{init}. Additionally, all labels that do not occur in the relevant part of the PA can be deleted. We therefore assume that the (sub-)PA under consideration contains only states and labels that are relevant for T.

In our computation, we need to ensure that from each problematic state an unproblematic state is reachable under the selected scheduler, otherwise the probability of the problematic states is not well defined by the constraints [14]. We solve this problem by attaching a value r_s to each problematic state $s \in S^{\mathrm{prob}}_T$ and encoding that a distribution of s is selected only if it has at least one successor state s' with a value $r_{s'} > r_s$ attached to it. This requirement assures by induction that there is an *increasing path* from s to an unproblematic state, along which the values attached to the states are strictly increasing.

To encode the selection of smallest critical command sets as an MILP, we need the following variables:

- for each $\ell \in \mathrm{Lab}$ a variable $x_\ell \in \{0,1\}$ which is 1 iff ℓ is part of the critical labeling,
- for each state $s \in S \setminus T$ and each transition $\eta \in P(s)$ a variable $\sigma_{s,\eta} \in \{0,1\}$ which is 1 iff η is chosen in s by the scheduler; the scheduler is free not to choose any transition,
- for each state $s \in S$ a variable $p_s \in [0,1]$ which stores the probability to reach a target state from s under the selected scheduler within the subsystem defined by the selected labeling,
- for each state $s \in S$ being either a problematic state or a successor of a problematic state a variable $r_s \in [0,1] \subseteq \mathbb{R}$ for the encoding of increasing paths, and
- for each problematic state $s \in S^{\mathrm{prob}}_T$ and each successor state $s' \in \mathrm{succ}(s)$ a variable $t_{s,s'} \in \{0,1\}$, where $t_{s,s'} = 1$ implies that the values attached to the states increase along the edge (s,s'), i.e., $r_s < r_{s'}$.

Let $w_{\min} := \min\{w(\ell) \,|\, \ell \in \mathrm{Lab} \wedge w(\ell) > 0\}$ be the smallest positive weight that is assigned to any label. The MILP for the smallest critical labeling problem is then as follows:

$$\text{minimize} \quad -\frac{1}{2} w_{\min} \cdot p_{s_{\mathrm{init}}} + \sum_{\ell \in \mathrm{Lab}} w(\ell) \cdot x_\ell \tag{1a}$$

such that

$$p_{s_{\mathrm{init}}} > \lambda \tag{1b}$$

$$\forall s \in S \setminus T: \quad \sum_{\eta \in P(s)} \sigma_{s,\eta} \leq 1 \tag{1c}$$

$$\forall s \in S\ \forall \eta \in P(s)\ \forall \ell \in L(s,\eta): \quad x_\ell \geq \sigma_{s,\eta} \tag{1d}$$

$$\forall s \in T: \quad p_s = 1 \tag{1e}$$

$$\forall s \in S \setminus T: \quad p_s \leq \sum_{\eta \in P(s)} \sigma_{s,\eta} \tag{1f}$$

$$\forall s \in S \setminus T\ \forall \eta \in P(s): \quad p_s \leq \sum_{s' \in \mathrm{succ}(s,\eta)} \mu(s') \cdot p_{s'} + (1 - \sigma_{s,\eta}) \tag{1g}$$

$$\forall (s,\eta) \in P_T^{\mathrm{prob}}: \quad \sigma_{s,\eta} \leq \sum_{s' \in \mathrm{succ}(s,\eta)} t_{ss'} \tag{1h}$$

$$\forall s \in S_T^{\mathrm{prob}}\ \forall s' \in \mathrm{succ}(s): \quad r_s < r_{s'} + (1 - t_{ss'})\ . \tag{1i}$$

The number of variables in this MILP is in $O(l+n+m)$ and the number of constraints in $O(n+l \cdot m)$ where l is the number of labels, n the number of states, and m the number of transitions of $\mathcal{M}$, i. e., $m = \big|\{(s,\eta,s') \mid s' \in \mathrm{succ}(s,\eta)\}\big|$.

We first explain the constraints in lines (1b)–(1i) of the MILP, which describe a critical labeling. First, we ensure that the probability of the initial state is greater than the probability bound λ (1b). For reachability properties, we can restrict ourselves to memoryless deterministic schedulers. So for each state $s \in S \setminus T$ at most one scheduler variable $\sigma_{s,\eta} \in P(s)$ can be set to 1 (1c). Note, that there may be states where no transition is chosen. For target states we do not need any restriction. If the scheduler selects a transition $\eta \in P(s)$, all labels $\ell \in L(s,\eta)$ have to be chosen (1d). For all target states $s \in T$ the probability p_s is set to 1 (1e), while for all non-target states without chosen transition ($\sigma_{s,\eta} = 0$ for all $\eta \in P(s)$), the probability is set to zero (1f); if $\sigma_{s,\eta} = 1$ for some $\eta \in P(s)$, this constraint is no restriction to probability p_s. However, in this case constraint (1g) is responsible for assigning a valid probability to p_s. The constraint is trivially satisfied if $\sigma_{s,\eta} = 0$. If transition η is selected, the probability p_s is bounded from above by the probability to go to one of the successor states of η and to reach the target states from there.

The reachability of at least one unproblematic state is ensured by (1h) and (1i). First, for every state s with transition η that is problematic regarding T, at least one transition variable must be activated. Second, for a path according to these transition variables, an increasing order is enforced for the problematic states. Because of this order, no problematic states can be revisited on an increasing path which enforces the final reachability of a non-problematic state.

These constraints enforce that each satisfying assignment of the label variables x_ℓ corresponds to a critical labeling. By minimizing the weight of the selected labels we obtain a smallest critical labeling. By the additional term $-\frac{1}{2} w_{\min} \cdot p_{s_{\mathrm{init}}}$ we obtain not only a smallest critical labeling but one with maximal probability. The coefficient $-\frac{1}{2} w_{\min}$ is needed to ensure that the benefit from maximizing the probability is smaller than the loss by adding an additional label. Please note, that any coefficient c with $0 < c < w_{\min}$ could be used.

Theorem 2. *The MILP given in* (1a)–(1i) *yields a smallest critical labeling.*

A proof of this theorem can be found in the extended version [25] of this paper.

Optimizations The constraints of the MILP describe *critical labelings*, whereas *minimality* is enforced by the objective function. In this section we describe how some additional constraints can be imposed, which explicitly exclude variable assignments that are either not optimal or encode labelings that are also encoded by other assignments. Adding such redundant constraints to the MILP often speeds up the search.

Scheduler cuts We want to exclude solutions of the constraint set for which a state $s \in S$ has a selected action-distribution pair $\eta \in P(s)$ with $\sigma_{s,\eta} = 1$ but all successors of s under η are non-target states without any selected action-distribution pairs. Note that such solutions would define $p_s = 0$. We add for all $s \in S \setminus T$ and all $\eta \in P(s)$ with $\mathrm{succ}(s,\eta) \cap T = \emptyset$ the constraint

$$\sigma_{s,\eta} \;\leq \sum_{s' \in \mathrm{succ}(s,\eta) \setminus \{s\}} \; \sum_{\eta' \in P(s')} \sigma_{s',\eta'} \;. \tag{2}$$

Analogously, we require for each non-initial state s with a selected action-distribution pair $\eta \in P(s)$ that there is a selected action-distribution pair leading to s. Thus, we add for all states $s \in S \setminus \{s_{\mathrm{init}}\}$ the constraint

$$\sum_{\eta \in P(s)} \sigma_{s,\eta} \leq \sum_{s' \in \mathrm{pred}(s) \setminus \{s\}} \; \sum_{\{\eta' \in P(s') \,\mid\, s' \in \mathrm{succ}(s,\eta)\}} \sigma_{s',\eta'} \;. \tag{3}$$

As special cases of these cuts, we can encode that the initial state has at least one activated outgoing transition and that at least one of the target states has an selected incoming transition. These special cuts come with very few additional constraints and often have a great impact on the solving times.

Label cuts In order to guide the solver to select the correct combinations of labels and scheduler variables, we want to enforce that for every selected label ℓ there is at least one scheduler variable $\sigma_{s,\eta}$ activated such that $\ell \in L(s,\eta)$:

$$x_\ell \leq \sum_{s \in S} \; \sum_{\{\eta \in P(s) \,\mid\, \ell \in L(s,\eta)\}} \sigma_{s,\eta} \;. \tag{4}$$

Synchronization cuts While scheduler and label cuts are applicable to the general smallest critical labeling problem, synchronization cuts take the proper synchronization of commands into account. They are therefore only applicable for the computation of smallest critical command sets.

Let M_i, M_j $(i \neq j)$ be two modules which synchronize on action α, c a command of M_i with action α, and $C_{j,\alpha}$ the set of commands with action α in module M_j. The following constraint ensures that if command c is selected by activating the variable x_{l_c}, then at least one command $d \in C_{j,\alpha}$ is selected, too.

$$x_{\ell_c} \;\leq \sum_{d \in C_{j,\alpha}} x_{\ell_d} \;. \tag{5}$$

4 Simplification of Counterexamples

Even though we can obtain a smallest set of commands which together induce an erroneous system by the techniques described in the previous section, further simplifications may be possible. For this we identify branching choices of each command in the counterexample which can be removed, still yielding an erroneous system. To accomplish this, we specify an MILP formulation which identifies a smallest set of branching choices that need to be preserved for the critical command set, such that the induced sub-PA still violates the property under consideration.

For this we need a more detailed labeling of the commands. Given a command c_i of the form $[\alpha]\ g \to p_1 : f_1 + p_2 : f_2 + \cdots + p_n : f_n$, we assign to each branching choice $p_j : f_j$ a unique label $b_{i,j}$. Let Lab_b be the set of all such labels.

When composing the modules, we compute the union of the labeling of the branching choices being executed together. When computing the corresponding PA $\mathcal{M}$, we transfer this labeling to the branching choices of the transition relation of $\mathcal{M}$. We define the partial function $L_b : S \times \mathrm{Act} \times \mathrm{Distr}(S) \times S \nrightarrow 2^{\mathrm{Lab}_b}$ such that $L_b(s, \nu, s')$ is defined iff $\nu \in P(s)$ and $s' \in \mathrm{succ}(s, \nu)$. In this case, $L_b(s, \nu, s')$ contains the labels of the branching choices of all commands that are involved in generating the transition from s to s' via the transition ν.

The following MILP identifies a largest number of branching choices which can be removed. The program is similar to the MILP for command selection, but instead of selecting commands it uses decision variables x_b to select branching choices in the commands. Additionally to the probability p_s of the composed states $s \in S$, we use variables $p_{s,\nu,s'} \in [0,1] \subseteq \mathbb{R}$ for $s \in S$, $\nu \in P(s)$ and $s' \in \mathrm{succ}(s, \nu)$, which are forced to be zero if not all branching choices which are needed to generate the transition from s to s' in ν are available (6g). For the definition of p_s in (6h), the expression $\mu(s') \cdot p_{s'}$ of (1g) is replaced by $p_{s,\eta,s'}$. The remaining constraints are unchanged.

$$\text{minimize} \quad -\frac{1}{2} p_{s_{\text{init}}} + \sum_{b \in \mathrm{Lab}_b} x_b \tag{6a}$$

such that

$$p_{s_{\text{init}}} > \lambda \tag{6b}$$

$$\forall s \in S \setminus T: \quad \sum_{\eta \in P(s)} \sigma_{s,\eta} \leq 1 \tag{6c}$$

$$\forall s \in T: \quad p_s = 1 \tag{6d}$$

$$\forall s \in S \setminus T\ \forall \eta \in P(s)\ \forall s' \in \mathrm{succ}(s, \eta):$$

$$p_{s,\eta,s'} \leq \mu(s') \cdot p_{s'} \tag{6e}$$

$$p_{s,\eta,s'} \leq \sigma_{s,\eta} \tag{6f}$$

$$\forall b \in L_b(s, \eta, s') : p_{s,\eta,s'} \leq x_b \tag{6g}$$

$$\forall s \in S \setminus T\ \forall \eta \in P(s): \quad p_s \leq \sum_{s' \in \mathrm{succ}(s,\eta)} p_{s,\eta,s'} + (1 - \sigma_{s,\eta}) \tag{6h}$$

$$\forall s \in S \setminus T: \quad p_s \leq \sum_{\eta \in P(s)} \sigma_{s,\eta} \tag{6i}$$

$$\forall (s,\eta) \in P_T^{\text{prob}}: \quad \sigma_{s,\eta} \leq \sum_{s' \in \text{succ}(s,\eta)} t_{s,s'} \tag{6j}$$

$$\forall s \in S_T^{\text{prob}} \forall s' \in \text{succ}(s): \quad r_s < r_{s'} + (1 - t_{s,s'}) \tag{6k}$$

5 Experiments

We have implemented the described techniques in C++ using the MILP solver `Gurobi` [24]. The experiments were performed on an Intel® Xeon® CPU E5-2450 with 2.10 GHz clock frequency and 32 GB of main memory, running Ubuntu 12.04 Linux in 64 bit mode. We focus on the minimization of the number of commands needed to obtain a counterexample and simplify them by deleting a maximum number of branchings. We do not consider symmetries in the models. We ran our tool with two threads in parallel and aborted any experiment which did not finish within 10 min (1200 CPU seconds). We conducted a number of experiments that are publicly available on the web page of `PRISM` [28].

▶ `coin`-N-K models the shared coin protocol of a randomized consensus algorithm [29]. The protocol returns a preference between two choices with a certain probability, whenever requested by a process at some point in the execution of the consensus algorithm. The shared coin protocol is parameterized by the number N of involved processes and a constant $K > 1$. Internally, the protocol is based on flipping a coin to come to a decision. We consider the property $\mathcal{P}_{\leq\lambda}\big(\Diamond\,(\text{finished} \wedge \text{all\_coins\_equal})\big)$, which is satisfied if the probability to finish the protocol with all coins equal is at most λ.

▶ `wlan`-B-C models the two-way handshake mechanism of the IEEE 802.11 Wireless LAN protocol. Two stations try to send data, but run into a collision. Therefore they enter the randomized exponential backoff scheme. The parameter B denotes the maximal allowed value of the backoff counter. We check the property $\mathcal{P}_{\leq\lambda}\big(\Diamond\,(\text{num\_collisions} = C)\big)$ putting an upper bound on the probability that a maximal allowed number C of collisions occur.

▶ `csma`-N-C concerns the IEEE 802.3 CSMA/CD network protocol. N is the number of processes that want to access a common channel, C is the maximal value of the backoff counter. We check $\mathcal{P}_{\leq\lambda}(\neg\text{collision\_max\_backoff}\ U\ \text{delivered})$ expressing that the probability that all stations successfully send their messages before a collision with maximal backoff occurs is at most λ.

▶ `fw`-N models the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus (called "FireWire") [30]. It is a leader election protocol which is executed each time a node enters or leaves the network. The parameter N denotes the delay of the wire as multiples of 10 ns. We check $\mathcal{P}_{\leq\lambda}(\Diamond\,\text{leader\_elected})$, i.e., that the probability of finally electing a leader is at most λ.

Some statistics of the models for different parameter values are shown in Table 1. The columns contain the name of the model, its number of states,

Table 1. Model statistics

Model	#states	#trans.	#mod.	#comm.	$\Pr^+(s_{\text{init}}, \Diamond T)$	λ	MCS
coin-2-1	144	252	2	14 (12)	0.6	0.4	13
coin-2-2	272	492	2	14 (12)	0.5556	0.4	25*
coin-2-4	528	972	2	14 (12)	0.529 40	0.4	55*
coin-2-5	656	1212	2	14 (12)	0.523 79	0.4	67*
coin-2-6	784	1452	2	14 (12)	0.519 98	0.4	83*
coin-4-1	12416	40672	4	28 (20)	0.636 26	0.4	171*
coin-4-2	22656	75232	4	28 (20)	0.578 94	0.4	244*
csma-2-2	1038	1282	3	34 (34)	0.875	0.5	540
csma-2-4	7958	10594	3	38 (38)	0.999 02	0.5	1769*
fw-1	1743	2197	4	68 (64)	1.0	0.5	412
fw-4	5452	7724	4	68 (64)	1.0	0.5	412*
fw-10	17190	29364	4	68 (64)	1.0	0.5	412*
fw-15	33425	63379	4	68 (64)	1.0	0.5	412*
wlan-0-2	6063	10619	3	70 (42)	0.183 59	0.1	121
wlan-0-5	14883	26138	3	70 (42)	0.001 14	0.001	952*
wlan-2-1	28597	57331	3	76 (14)	1.0	0.5	7
wlan-2-2	28598	57332	3	76 (42)	0.182 60	0.1	121*
wlan-2-3	35197	70216	3	76 (42)	0.017 93	0.01	514*
wlan-3-1	96419	204743	3	78 (14)	1.0	0.5	7
wlan-3-2	96420	204744	3	78 (42)	0.183 59	0.1	121*

transitions, modules, and commands. The value in braces is the number of relevant commands. Column 6 contains the reachability probability and column 7 the bound λ. The last column shows the number of states in the minimal critical subsystem, i. e., the smallest subsystem of the PA such that the probability to reach a target state inside the subsystem is still above the bound. Entries which are marked with a star, correspond to the smallest critical subsystem we could find within the time bound of 10 min using our tool `LTLSubsys` [12], but they are not necessarily optimal.

The results of our experiments are displayed in Table 2. The first column contains the name of the model. The following three blocks contain the results of runs without any cuts, with all cuts, and with the best combination of cuts: If there were cut combinations with which the MILP could be solved within the time limit, we report the one with the shortest solving time. If all combinations timed out, we report the one that yielded the largest lower bound.

For each block we give the computation time in seconds ("Time"), the memory consumption in MB ("Mem."), the number of commands in the critical command set ("n") and, in case the time limit was exceeded, a lower bound on the size of the smallest critical command set ("lb"), which the solver obtains by solving a linear programming relaxation of the MILP. An entry "??" for the number of commands means that the solver was not able to find a non-trivial critical command set within the time limit. For the run without cuts we additionally give the number of variables ("Var.") and constraints ("Constr.") of the MILP.

Table 2. Experimental results (time limit = 600 seconds)

	no cuts						all cuts				best cut combination				branches	
Model	Var.	Constr.	Time	Mem.	n	lb	Time	Mem.	n	lb	Time	Mem.	n	lb	simp.	$\|S'\|$
coin-2-1	277	491	TO	773	9	8	298.56	146	9	opt	145.76	95	9	opt	1/12	28
coin-2-2	533	1004	TO	864	9	6	TO	676	9	7	TO	562	9	7	1/12	72
coin-2-4	1045	2028	TO	511	9	6	TO	162	9	6	TO	426	9	7	1/12	105
coin-2-5	1301	2540	TO	485	9	5	TO	121	9	6	TO	408	9	6	1/12	165
coin-2-6	1557	3052	TO	550	9	5	TO	159	9	6	TO	495	9	6	1/12	103
coin-4-1	26767	50079	TO	642	??	3	TO	627	20	3	TO	703	20	5	2/24	391
coin-4-2	47759	92063	TO	947	??	3	TO	993	??	3	TO	961	??	4	??	??
csma-2-2	2123	5990	2.49	24	32	opt	17.88	50	32	opt	2.11	24	32	opt	3/42	879
csma-2-4	15977	46882	195.39	208	36	opt	263.89	397	36	opt	184.05	208	36	opt	20/90	4522
fw-1	3974	13121	TO	205	28	27	184.49	119	28	opt	44.21	135	28	opt	38/68	419
fw-4	13144	43836	TO	268	28	21	TO	367	28	21	107.71	328	28	opt	38/68	424
fw-10	46282	153764	TO	790	28	13	TO	1141	28	18	545.68	993	28	opt	38/68	428
fw-15	96222	318579	TO	1496	28	9	TO	958	31	14	TO	1789	28	18	33/68	416
wlan-0-2	7072	6602	TO	324	33	15	TO	209	33	30	TO	174	33	32	23/72	3178
wlan-0-5	19012	25808	TO	570	??	10	TO	351	??	30	TO	357	??	30	??	??
wlan-2-1	28538	192	0.04	43	8	opt	0.07	44	8	opt	0.04	43	8	opt	6/14	7
wlan-2-2	29607	15768	TO	413	33	14	TO	188	33	30	TO	180	33	30	23/72	25708
wlan-2-3	36351	18922	TO	600	38	14	TO	315	37	32	TO	275	38	32	31/72	25173
wlan-3-1	96360	192	0.09	137	8	opt	0.13	137	8	opt	0.08	137	8	opt	6/14	7
wlan-3-2	97429	6602	TO	450	33	15	TO	292	33	30	TO	260	33	31	23/72	93639

In the last block we give information about the number of branching choices which could be removed from the critical command set ("simp."). In case the different runs did not compute the same set, we used the one obtained with all cuts. An entry k/m means that we could remove k out of m relevant branching choices. We omit the running times of the simplification since in all cases it was faster than the command selection due to the reduced state space. The last column ("$|S'|$") contains the number of states in the PA that is induced by the minimized command set.

Although we ran into timeouts for many instances, in particular without any cuts, in almost all cases (with the exception of `coin4-2` and `wlan0-5`) a solution could be found within the time limit. We suppose that also the solutions of the aborted instances are optimal or close to optimal. It seems that the MILP solver is able to quickly find good (or even optimal) solutions due to sophisticated heuristics, but proving their optimality is hard. A solution is proven optimal as soon as the objective value of the best solution and the lower bound coincide. The additional cuts strengthen this lower bound considerably. Further experiments have shown that the scheduler cuts of Eq.(2) have the strongest effect on the lower bound. Choosing good cuts consequently enables the solver to obtain optimal solutions for more benchmarks.

Our method provides the user not only with a smallest set of simplified commands which induce an erroneous system, but also with a critical subsystem of the state space. Comparing its size with the size of the minimal critical subsystem (cf. Table 1) we can observe that for some models it is is close to optimal (e. g., the `coin`-instances), for others it is much larger (e. g., the `wlan`-

instances). In all cases, however, we are able to reduce the number of commands and to simplify the commands, in some cases considerably.

6 Conclusion

We have presented a new type of counterexamples for probabilistic automata which are described using a guarded command language: We computed a smallest subset of the commands which alone induces an erroneous system. This requires the solution of a mixed integer linear program whose size is linear in the size of the state space of the PA. State-of-the-art MILP solvers apply sophisticated techniques to find small command sets quickly, but they are often unable to prove the optimality of their solution.

For the MILP formulation of the smallest critical labeling problem we both need decision variables for the labels and for the scheduler inducing the maximal reachability probabilities of the subsystem. On the other hand, model checking can be executed without any decision variables. Therefore we plan to develop a dedicated branch & bound algorithm which only branches on the decision variables for the labels. We expect a considerable speedup by using this method. Furthermore, we will investigate heuristic methods based on graph algorithms.

References

1. Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking – History, Achievements, Perspectives. Vol. 5000 of LNCS. Springer (2008) 1–26
2. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. of CAV. Vol. 1855 of LNCS, Springer (2000) 154–169
3. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Proc. of CAV. Vol. 5123 of LNCS, Springer (2008) 162–175
4. Bobaru, M.G., Pasareanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Proc. of CAV. Vol. 5123 of LNCS, Springer (2008) 135–148
5. Komuravelli, A., Pasareanu, C.S., Clarke, E.M.: Assume-guarantee abstraction refinement for probabilistic systems. In: Proc. of CAV. Vol. 7358 of LNCS, Springer (2012) 310–326
6. Han, T., Katoen, J.P., Damman, B.: Counterexample generation in probabilistic model checking. IEEE Trans. on Software Engineering **35**(2) (2009) 241–257
7. Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: Proc. of CAV. Vol. 6806 of LNCS, Springer (2011) 585–591
8. Aljazzar, H., Leue, S.: Directed explicit state-space search in the generation of counterexamples for stochastic model checking. IEEE Trans. on Software Engineering **36**(1) (2010) 37–60
9. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Proc. of VMCAI. Vol. 5403 of LNCS, Springer (2009) 366–380
10. Andrés, M.E., D'Argenio, P., van Rossum, P.: Significant diagnostic counterexamples in probabilistic model checking. In: Proc. of HVC. Vol. 5394 of LNCS, Springer (2008) 129–148

11. Jansen, N., Ábrahám, E., Katelaan, J., Wimmer, R., Katoen, J.P., Becker, B.: Hierarchical counterexamples for discrete-time Markov chains. In: Proc. of ATVA. Vol. 6996 of LNCS, Springer (2011) 443–452
12. Wimmer, R., Becker, B., Jansen, N., Ábrahám, E., Katoen, J.P.: Minimal critical subsystems for discrete-time Markov models. In: Proc. of TACAS. Vol. 7214 of LNCS, Springer (2012) 299–314
13. Wimmer, R., Becker, B., Jansen, N., Ábrahám, E., Katoen, J.P.: Minimal critical subsystems as counterexamples for ω-regular DTMC properties. In: Proc. of MBMV, Verlag Dr. Kovač (2012) 169–180
14. Wimmer, R., Jansen, N., Ábrahám, E., Katoen, J.P., Becker, B.: Minimal counterexamples for refuting ω-regular properties of Markov decision processes. Reports of SFB/TR 14 AVACS 88 (2012) ISSN: 1860-9821, `http://www.avacs.org`.
15. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
16. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, Massachusetts Institute of Technology (1995) available as Technical Report MIT/LCS/TR-676.
17. Segala, R.: A compositional trace-based semantics for probabilistic automata. In: Proc. of CONCUR. Vol. 962 of LNCS, Springer (1995) 234–248
18. Dijkstra, E.W.: Guarded commands, non-determinacy and formal derivation of programs. Communications of the ACM **18**(8) (1975) 453–457
19. He, J., Seidel, K., McIver, A.: Probabilistic models for the guarded command language. Science of Computer Programming **28**(2–3) (1997) 171–192
20. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
21. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co Ltd (1979)
22. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation **1**(1) (2009) 1–41
23. IBM: CPLEX optimization studio, version 12.5. `http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/` (2012)
24. Gurobi Optimization, Inc.: Gurobi optimizer reference manual. `http://www.gurobi.com` (2012)
25. Wimmer, R., Jansen, N., Vorpahl, A., Ábrahám, E., Katoen, J.P., Becker, B.: High-level counterexamples for probabilistic automata (extended version). Technical Report arXiv:1305.5055 (2013) available at `http://arxiv.org/abs/1305.5055`.
26. Kwiatkowska, M., Norman, G., Sproston, J.: Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In: Proc. of PAPM/PROBMIV. Vol. 2399 of LNCS, Springer (2002) 169–187
27. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Proc. of FSTTCS. Vol. 1026 of LNCS, Springer (1995) 499–513
28. Kwiatkowska, M., Norman, G., Parker, D.: The PRISM benchmark suite. In: Proc. of QEST, IEEE CS Press (2012) 203–204
29. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. Journal of Algorithms **15**(1) (1990) 441–460
30. Stoelinga, M.: Fun with FireWire: A comparative study of formal verification methods applied to the IEEE 1394 Root Contention Protocol. Formal Aspects of Computing **14**(3) (2003) 328–337