Fast Debugging of PRISM Models*

Christian Dehnert[1], Nils Jansen[1], Ralf Wimmer[2], Erika Ábrahám[1],
Joost-Pieter Katoen[1]

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

Abstract. In addition to rigorously checking whether a system conforms to a specification, model checking can provide valuable feedback in the form of succinct and understandable counterexamples. In the context of probabilistic systems, path- and subsystem-based counterexamples at the state-space level can be of limited use in debugging. As many probabilistic systems are described in a guarded command language like the one used by the popular model checker Prism, a technique identifying a subset of critical commands has recently been proposed. Based on repeatedly solving MaxSat instances, our novel approach to computing a minimal critical command set achieves a speed-up of up to five orders of magnitude over the previously existing technique.

1 Introduction

Algorithmic counterexample generation is a key component of modern model checkers. Counterexamples are pivotal for debugging—experience has shown that counterexamples are the single most effective feature to convince system engineers about the value of formal verification [11]. They are an essential ingredient in counterexample-guided abstraction refinement [10] (CEGAR) and can be effectively used in model-based testing. Prominent model checkers such as Spin and NuSmv include powerful facilities to generate counterexamples in various formats. Such counterexamples are typically provided at the modeling level, like a diagram indicating how the change of model variables yields a property violation, or a message sequence chart illustrating the failing scenario. Substantial efforts have been made to generate succinct counterexamples, often at the price of an increased time complexity [14,17,26]. Despite the growing popularity of probabilistic model checkers, such facilities are absent in tools such as Prism [22] and Mrmc[21]. This paper presents an efficient scalable technique for computing minimal counterexamples for the Prism modeling language.

* This work is partly supported by the Excellence Initiative of the German federal and state governments, the EU-FP7 projects CARP and SENSATION, and by the German Research Council (DFG) as part of the Transregional Collaborative Research Center AVACS (SFB/TR 14).

Counterexample generation for probabilistic models is not easy. Showing that the reachability probability of a bad state b does not stay below a given threshold λ requires a set of finite paths leading to b, whose probability mass exceeds λ. Computing minimal sets is a k-shortest path problem [16] and can be done using heuristics [1] or bounded model checking [29] and may be enhanced by post-processing steps, such as building a fault-tree to better explain the causality in the model [23]. A viable alternative is to determine minimal critical subsystems [30,31], i. e., model fragments for which the likelihood of reaching b already exceeds λ. The drawback of most approaches in the literature is that they work at the state space level. As the size of a set of finite paths can be doubly exponential in the number of states [16], and minimal critical subsystems can have thousands of states [30], state-based diagnostic feedback is often incomprehensible and not effectively usable in CEGAR approaches for probabilistic systems [18,9]. Although symbolic approaches diminish this problem to some extent, the resulting counterexamples are still too large to handle [19].

We therefore take a radically different approach, and generate *counterexamples as* PRISM *probabilistic programs*. Our approach basically deletes commands of a PRISM probabilistic program yielding a *smallest* PRISM probabilistic program violating the reachability property at hand. PRISM uses a stochastic version of Alur and Henzinger's reactive modules [2] as modeling language. A module description consists of a set of guarded commands providing discrete probabilistic choices. The semantics of a module is a probabilistic automaton [27], a compositional variant of Markov decision processes. A PRISM probabilistic program consists of several modules that communicate by shared variables or using synchronization on common actions. (Remark that our approach is also applicable to other modeling formalisms for probabilistic automata such as PIOA [8], process algebra [20] and the graphical component-wise representation of systems as possible in UPPAAL [7].)

The problem considered is: determine a minimal set of guarded commands of a given PRISM probabilistic program (constituting a PRISM sub-program) that refutes the reachability property at hand. This problem is NP-hard [32]. We present an *incremental* approach for computing minimal critical command sets. The basic idea of our approach is to deduce necessary conditions for an optimal solution by a static analysis of the probabilistic program. We then use a MAXSAT solver to compute a smallest set of commands that is in accordance with these constraints. The resulting PRISM probabilistic program is model checked against the property at hand. If the reachability property is violated, the program constitutes the desired minimal critical command set. Otherwise, it is excluded from the search space and further conditions on the optimal solution are deduced. This paper presents the technical details of the approach and establishes its correctness. We report on a prototype implementation and show the practical applicability of our incremental MAXSAT approach on a number of PRISM benchmark case studies. The experimental results show that our approach scales to models with millions of states and achieves a speed-up of up to five orders of magnitude in comparison to a mixed-integer linear programming approach [32].

Whereas this paper focuses on reachability probabilities, our approach can be easily extended to properties φ that are monotonic in the sense that if $\mathcal{A}$ is a sub-PA of $\mathcal{A}'$ and $\mathcal{A} \not\models \varphi$, then also $\mathcal{A}' \not\models \varphi$.

2 Preliminaries

2.1 Probabilistic Automata

Let S be a countable set. A *probability distribution* over S is a function $\mu\colon S \to [0,1]$ such that $\sum_{s\in S} \mu(s) = 1$. We denote by $\mathit{Dist}(S)$ the set of all probability distributions over S. A distribution μ is called *Dirac* if there exists an element $s \in S$ with $\mu(s) = 1$ and $\mu(s') = 0$ for all $s' \in S$ with $s \neq s'$.

Definition 1 (Probabilistic Automaton [27]). *A* probabilistic automaton *(PA) is a tuple $\mathcal{A} = (S, s_{\mathrm{init}}, \mathit{Act}, \mathbf{P})$ where S is a finite set of states, $s_{\mathrm{init}} \in S$ is the initial state, Act is a finite set of actions, and $\mathbf{P} : S \to 2^{\mathit{Act}\times \mathit{Dist}(S)}$ is a probabilistic transition relation such that $\mathbf{P}(s)$ is finite for all $s \in S$.*

Intuitively, the evolution of a probabilistic automaton is as follows. Starting in the initial state s_{init}, a transition $(\alpha, \mu) \in \mathbf{P}(s_{\mathrm{init}})$ is chosen nondeterministically. Then, the successor state $s' \in S$ is determined probabilistically according to the probability distribution μ. Repeating this process in s' yields the next state and so on. To prevent deadlocks, we require $\mathbf{P}(s) \neq \emptyset$ for all $s \in S$.

Let $\mathrm{succ}_{\mathcal{A}}(s, \alpha, \mu) = \{s' \in S \mid \mu(s') > 0\}$ for $(\alpha, \mu) \in \mathbf{P}(s)$, $\mathrm{succ}_{\mathcal{A}}(s) = \bigcup_{(\alpha,\mu)\in\mathbf{P}(s)} \mathrm{succ}_{\mathcal{A}}(s, \alpha, \mu)$, and $\mathrm{pred}_{\mathcal{A}}(s) = \{s' \in S \mid \exists(\alpha, \mu) \in \mathbf{P}(s') : \mu(s) > 0\}$. We will omit the subscript $\mathcal{A}$ if the PA is clear from the context.

An (infinite) path π in a PA $\mathcal{A}$ is an infinite sequence $s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1)\ldots$ such that $(\alpha_i, \mu_i) \in \mathbf{P}(s_i)$ and $s_{i+1} \in \mathrm{succ}(s_i, \alpha_i, \mu_i)$ for all $i \geq 0$. A finite path ρ in $\mathcal{A}$ is a finite prefix $s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1)\ldots s_n$ of an infinite path π in $\mathcal{A}$ and its last state is denoted $\mathit{last}(\rho) = s_n$. Let $\pi[i]$ denote the i^{th} state in path π. The sets of all infinite and finite paths in $\mathcal{A}$ starting in $s \in S$ are denoted by $\mathrm{Path}_{\mathcal{A}}(s)$ and $\mathrm{Path}^{\mathrm{fin}}_{\mathcal{A}}(s)$, respectively.

Example 1. Figure 2 on page 7 shows an example PA with five states. For instance, the state s_1 has a nondeterministic choice between the two transitions $(\mathrm{reset}, \mu_{s_{\mathrm{init}}})$ and $(\mathrm{proc}, \mu_{\mathrm{proc}})$ where $\mu_{s_{\mathrm{init}}}$ is the Dirac distribution at s_{init} and μ_{proc} is given by $\mu_{\mathrm{proc}}(s_3) = 0.99$ and $\mu_{\mathrm{proc}}(s_4) = 0.01$.

To define a suitable probability measure on PAs, the nondeterminism has to be resolved by *schedulers*.

Definition 2 (Scheduler). *A* scheduler *for a PA $\mathcal{A} = (S, s_{\mathrm{init}}, \mathit{Act}, \mathbf{P})$ is a function $\sigma\colon \mathrm{Path}^{\mathrm{fin}}_{\mathcal{A}}(s_{\mathrm{init}}) \to \mathit{Dist}(\mathit{Act} \times \mathit{Dist}(S))$ mapping each finite path $\rho \in \mathrm{Path}^{\mathrm{fin}}_{\mathcal{A}}(s_{\mathrm{init}})$ in $\mathcal{A}$ to a probability distribution over transitions such that $\sigma(\rho)(\alpha, \mu) > 0$ implies $(\alpha, \mu) \in \mathbf{P}\big(\mathit{last}(\rho)\big)$.*

Intuitively, a scheduler resolves the nondeterminism in a PA by assigning probabilities to the nondeterministic choices available in the last state of a given finite path. It therefore reduces the nondeterministic model to a fully probabilistic one. Given a PA $\mathcal{A}$ and a scheduler σ for $\mathcal{A}$, a standard probability measure on paths, which we denote by $\mathrm{Pr}^{\sigma}_{s_{\mathrm{init}},\mathcal{A}}$ (or, briefly, $\mathrm{Pr}^{\sigma}_{\mathcal{A}}$), can be defined [4].

In the context of 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 $\lambda \in [0,1]$? This property is denoted by $\mathcal{P}_{\leq\lambda}(\lozenge T)$. Note that checking arbitrary ω-regular properties can be reduced to checking reachability properties, see [4] for details. $\mathcal{A}$ satisfies a probabilistic reachability property $\mathcal{P}_{\leq\lambda}(\lozenge T)$, denoted $\mathcal{A} \models \mathcal{P}_{\leq\lambda}(\lozenge T)$, if $\mathrm{Pr}^{\sigma}_{\mathcal{A}}(\lozenge T) := \mathrm{Pr}^{\sigma}_{\mathcal{A}}(\{\pi \in \mathrm{Path}_{\mathcal{A}}(s_{\mathrm{init}}) \mid \exists i : \pi[i] \in T\}) \leq \lambda$ for all schedulers σ. Algorithmically, the maximal reachability probability $\mathrm{Pr}^{\max}_{\mathcal{A}}(\lozenge T) := \sup_{\sigma} \mathrm{Pr}^{\sigma}_{\mathcal{A}}(\lozenge T)$ is computed using standard techniques, such as value or policy iteration [5,25], and compared against the bound λ.

2.2 Prism's Probabilistic Guarded Command Language

For a set Var of Boolean variables, let $\mathcal{N}_{\mathit{Var}}$ denote the set of all variable *valuations*, i. e., the set of functions $\nu\colon \mathit{Var} \to \{0,1\}$.

Definition 3 (Probabilistic Program, Module, Command). *A* probabilistic program *is a tuple* $\mathfrak{P} = (\mathit{Var}, \nu_{\mathrm{init}}, \mathbf{M})$ *where* Var *is a finite set of Boolean variables*[3]*,* $\nu_{\mathrm{init}} \in \mathcal{N}_{\mathit{Var}}$ *is the initial variable valuation, and* $\mathbf{M} = \{M_1, \ldots, M_k\}$ *is a finite set of modules.*

A module *is a tuple* $M_i = (\mathit{Var}_i, \mathit{Act}_i, C_i)$ *where for* $1 \leq i, j \leq k$ $\mathit{Var}_i \subseteq \mathit{Var}$ *is a finite set of Boolean variables such that* $\mathit{Var}_i \cap \mathit{Var}_j = \emptyset$ *for* $i \neq j$*,* Act_i *is a finite set of synchronizing actions, and* C_i *is a finite set of commands. Additionally, to be consistent with the program, we require* $\mathit{Var} = \bigcup_{j=1}^{k} \mathit{Var}_j$*.*

Let $\tau \notin \bigcup_{i=1}^{k} \mathit{Act}_i$ *denote the internal non-synchronizing action. A* command $c \in C_i$ *is of the form* $c = [\alpha]\ g \ \ \to \ \ p_1 : f_1 + \ldots + p_n : f_n$*, where* $\alpha \in \mathit{Act}_i \,\dot\cup\, \{\tau\}$ *is the* action *of* c *that is referred to as* $\mathrm{act}(c)$*,* g *is a Boolean predicate over* Var *(called the* guard *of* c*), denoted by* $\mathrm{grd}(c)$*,* $p_j \in [0,1]$ *is a rational number such that* $\sum_{i=1}^{n} p_i = 1$*, and* $f_j\colon \mathcal{N}_{\mathit{Var}} \to \mathcal{N}_{\mathit{Var}_i}$ *is an update function that assigns to each variable of the module a new value based on the values of all variables in the program for all* $1 \leq j \leq n$*.*

Note that each variable $v \in \mathit{Var}_i$ may be written only by the module M_i, but the update may depend on variables of other modules. The restriction $(\mathit{Var}_i, \mathit{Act}_i, C_i \cap C)$ of module M_i to a set C of commands is denoted $M_i|_C$ and $\mathfrak{P}|_C = (\mathit{Var}, \nu_{\mathrm{init}}, \{M_1|_C, \ldots, M_k|_C\})$ is the restriction of the whole program to this set of commands.

A model with $k > 1$ modules is equivalent to a model with a single module resulting from the *parallel composition* $M_1 \parallel \cdots \parallel M_k$ of all modules. Intuitively, the parallel composition of two modules corresponds to a new module that enables all non-synchronizing behavior of the two modules as well as the composition of all command-pairs that need to synchronize because of a common action

[3] Note that for Prism, the variables do not have to be Boolean. However, as finite variable domains are required, every program can be transformed into one only having Boolean variables.

```
module coin
  $f$: bool init $0$; $c$: bool init $0$;
  [flip] $\neg f \rightarrow 0.5 : (f' = 1)\&(c' = 1) + 0.5 : (f' = 1)\&(c' = 0)$;     ($c_1$)
  [reset] $f \wedge \neg c \rightarrow 1 : (f' = 0)$;                                   ($c_2$)
  [proc] $f \rightarrow 0.99 : (f' = 1) + 0.01 : (c' = 1)$;                              ($c_3$)
endmodule
module processor
  $p$: bool init $0$;
  [proc] $\neg p \rightarrow 1 : (p' = 1)$;                                              ($c_4$)
  [loop] $p \rightarrow 1 : (p' = 1)$;                                                   ($c_5$)
  [reset] $true \rightarrow 1 : (p' = 0)$;                                               ($c_6$)
endmodule
```

Fig. 1. A probabilistic program $\mathfrak{P}_{Ex}$ in PRISM's input language.

name. Formally, the binary composition $M_i \parallel M_j = (Var_i \cup Var_j, Act_i \cup Act_j, C)$ of two modules M_i and M_j with $i \neq j$ has the set of commands

$$\begin{aligned} C = {} & \{c \mid c \in C_i \cup C_j \wedge \mathrm{act}(c) \in (\{\tau\} \cup Act_i \ominus Act_j)\} \\ & \cup \{c \otimes c' \mid c \in C_1 \wedge c' \in C_2 \wedge \mathrm{act}(c) = \mathrm{act}(c') \in Act_i \cap Act_j\} \end{aligned}$$

where $A \ominus B$ is the symmetric difference of the sets A and B. The composition $c \otimes c'$ of two commands $c = [\alpha]\ g \rightarrow p_1 : f_1 + \ldots + p_n : f_n$ and $c' = [\alpha]\ g' \rightarrow p'_1 : f'_1 + \ldots + p'_m : f'_m$ with the same action α is defined as

$$c \otimes c' = [\alpha]\ g \wedge g' \ \ \rightarrow \ \ \sum_{i=1}^{n} \sum_{j=1}^{m} p_i \cdot p'_j : f_i \oplus f'_j.$$

Here, the composition $f_r \oplus f_s \colon \mathcal{N}_{Var} \rightarrow \mathcal{N}_{Var_i \cup Var_j}$ of two update functions $f_r \colon \mathcal{N}_{Var} \rightarrow \mathcal{N}_{Var_i}$ and $f_s \colon \mathcal{N}_{Var} \rightarrow \mathcal{N}_{Var_j}$ is defined by

$$(f_r \oplus f_s)(\nu)(v) = \begin{cases} f_r(\nu)(v), & \text{if } v \in Var_i, \\ f_s(\nu)(v), & \text{otherwise.} \end{cases}$$

Example 2. Figure 1 shows a probabilistic program $\mathfrak{P}_{\mathrm{Ex}}$ with two modules `coin` and `processor`. It models a system that first does a coin flip and then processes some data. While doing so, it may erroneously modify the coin. Depending on the outcome of the coin flip, the system may reset to the initial configuration. The program uses three variables $Var_{\mathrm{Ex}} = \{f, c, p\}$ that indicate whether a coin has been flipped (f), the coin shows tails ($c = 0$) or heads ($c = 1$) and whether some data was processed (p). Initially the module `coin` can do a coin flip (command c_1). Then, both modules can process some data by synchronizing on the `proc` action (c_3 and c_4). However, the processing step can by mistake set the coin to show heads with probability 0.01 (c_3). Additionally, if the coin showed tails, the coin flip can be undone by a reset (c_2 and c_6). Finally, if data has been processed the system may loop forever (c_5).

The semantics of a probabilistic program $\mathfrak{P} = (\mathit{Var}, \nu_{\text{init}}, \{M\})$ with only one module $M = (\mathit{Var}, \mathit{Act}, C)$ is defined in terms of a PA $\mathcal{A} = [\![\mathfrak{P}]\!] = (S, s_{\text{init}}, \mathit{Act}, \mathbf{P})$. $S = \mathcal{N}_{\mathit{Var}}$ is the set of all valuations of the program variables[4]. Hence, each state $s \in S$ can be seen as a bit vector $(x_1, \ldots, x_m)$ with x_i being the value of the variable $v_i \in \mathit{Var} = \{v_1, \ldots, v_m\}$. The initial state s_{init} of the PA corresponds to the initial valuation ν_{init} of variables in the program. A guard g defines a subset $S_g \subseteq S$ of states in which the guard evaluates to true. Now, a command $c = [\alpha]\ g \rightarrow p_1 : f_1 + \ldots + p_n : f_n$ induces a probability distribution $\mu_{s,c} \in \mathit{Dist}(S)$ for all states $s \in S_g$ by setting

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

for each $s' \in S$. The transition relation $\mathbf{P}$ is then defined for all $s \in S$ by

$$\mathbf{P}(s) = \big\{(\alpha, \mu_{s,c}) \,\big|\, \exists c \in C : \text{act}(c) = \alpha \wedge s \in S_{\text{grd}(c)}\big\} .$$

We say that the transition $(\alpha, \mu_{s,c})$ is *generated* by the command c. In case c resulted from the parallel composition of a set of commands C from a probabilistic program with more than one module, we say that the commands in C (jointly) generate the transition. From now on we assume a labeling function $L \colon S \times \mathit{Act} \times \mathit{Dist}(S) \rightarrow 2^{\mathit{Lab}}$ that labels each transition $(\alpha, \mu) \in \mathbf{P}(s)$ with a set of labels $L(s, \alpha, \mu) \subseteq \mathit{Lab} = \{\ell_c \mid \exists i \in \{1, \ldots, k\} : c \in C_i\}$ to indicate which commands generated the transition. Note that in case of synchronization the labeling of a transition is a set with more than one element. In order to distinguish the transitions generated by different commands later on, we create different copies of the transition and label them appropriately, if a particular transition is generated by different commands or command sets. We will abbreviate the set of states $\{s \in S \mid \exists (\alpha, \mu) \in \mathbf{P}(s) : c \in L(s, \alpha, \mu)\}$ that have an outgoing transition generated by $c \in C$ by $\text{src}(c)$. Analogously, we let $\text{dst}(c)$ be the set of states that have an incoming transition (α, μ) from some state s' with $c \in L(s', \alpha, \mu)$. If a state s has no command enabled, i.e. $s \notin S_{\text{grd}(c)}$ for any $c \in C$, the state is equipped with a self-loop transition (α_s, μ_s) where $\alpha_s \notin \mathit{Act}$ is a new action and μ_s is the Dirac distribution on s. For all transitions added this way, we let $L(s, \alpha_s, \mu_s) = \emptyset$ to reflect that they were not generated by any command, but were added to avoid deadlock states.

Example 3. $\mathcal{A} = [\![\mathfrak{P}_{\text{Ex}}]\!]$ is depicted in Figure 2 where all unreachable states are omitted. The states of the automaton are given by the valuations of the variables in the form $\langle f, c, p \rangle$ and the arrows between the states define the transition relation $\mathbf{P}$, where the highlighting of arrows only becomes relevant in a following example and can be ignored for now. Assume that the probabilistic reachability property $\varphi = \mathcal{P}_{\leq 0.5}(\Diamond\{s_4\})$ is given. Clearly, $\mathcal{A} \not\models \varphi$, because, for example,

[4] Actually, Prism programs also allow to specify discrete-time and continuous Markov chains (DTMCs and CTMCs, respectively) and probabilistic timed-automata (PTA). While this paper focuses on PAs, our technique can be readily applied to DTMCs and PTA and also on CTMCs if the guards of commands are non-overlapping.

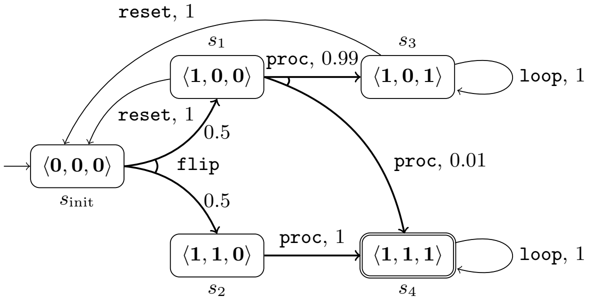
reset, 1
s_1
s_3
$\langle \mathbf{1},\mathbf{0},\mathbf{0}\rangle$
proc, 0.99
$\langle \mathbf{1},\mathbf{0},\mathbf{1}\rangle$
loop, 1
reset, 1
0.5
$\langle \mathbf{0},\mathbf{0},\mathbf{0}\rangle$
flip
proc, 0.01
s_{init}
0.5
proc, 1
$\langle \mathbf{1},\mathbf{1},\mathbf{0}\rangle$
$\langle \mathbf{1},\mathbf{1},\mathbf{1}\rangle$
loop, 1
s_2
s_4

Fig. 2. The reachable fragment of the probabilistic automaton $[\![\mathfrak{P}_{\text{Ex}}]\!]$.

$\Pr^{\sigma}_{\mathcal{A}}(\Diamond\{s_4\}) = 0.505$ for the scheduler σ that chooses the `proc` action in both s_1 and s_2 and loops in s_3 and s_4.

Critical command sets. Consider a probabilistic program $\mathfrak{P} = (\mathit{Var}, \nu_{\text{init}}, \{M = (\mathit{Var}, \mathit{Act}, C)\})$, its associated PA $\mathcal{A} = [\![\mathfrak{P}]\!] = (S, s_{\text{init}}, \mathit{Act}, \mathbf{P})$, and the reachability property $\varphi = \mathcal{P}_{\leq\lambda}(\Diamond T)$ for a set of target states $T \subseteq S$. We assume φ to be violated by $[\![\mathfrak{P}]\!]$, i. e., $\mathcal{A} \not\models \varphi$. We aim at identifying a set $C' \subseteq C$ of commands such that the program restricted to these commands still violates the property φ, i. e., $[\![\mathfrak{P}|_{C'}]\!] \not\models \varphi$. We call these subsets of commands *critical command sets*, as they induce a critical fragment of the probabilistic automaton that already proves the violation of the property.

Example 4. Reconsider the probabilistic automaton $[\![\mathfrak{P}_{\text{Ex}}]\!]$ and the probabilistic reachability property $\varphi = \mathcal{P}_{\leq 0.5}(\Diamond\{s_4\})$ given in Example 3. While the program $\mathfrak{P}_{\text{Ex}}$ has 5 commands, the commands $C^*_{\text{Ex}} = \{c_1, c_3, c_4\}$ are already critical, because $[\![\mathfrak{P}_{\text{Ex}}|_{C^*_{\text{Ex}}}]\!] \not\models \varphi$. The transitions of the restricted model are drawn as bold arrows in Figure 2.

2.3 MaxSat

Given two finite sets Φ, Ψ of propositional formulae over variables Var such that Ψ is satisfiable, the goal is to determine an assignment $\nu \in \mathcal{N}_{\mathit{Var}}$ which satisfies all formulae in Ψ and a maximal number of formulae in Φ, i. e., $\textsc{MaxSat}(\Phi, \Psi) = \nu$ such that $\nu \models \Theta \cup \Psi$ where $\Theta \in \operatorname{argmax}_{\Phi' \subseteq \Phi} \big\{|\Phi'| \,\big|\, \Phi' \cup \Psi \text{ is satisfiable}\big\}$. Note that by negating each formula in Φ, i. e., letting $\overline{\Phi} = \{\neg\varphi \mid \varphi \in \Phi\}$, $\textsc{MaxSat}(\overline{\Phi}, \Psi)$ yields an assignment that satisfies a minimal number of formulae of Φ while still satisfying all constraints in Ψ. Consequently, we let $\textsc{MinSat}(\Phi, \Psi) := \textsc{MaxSat}(\overline{\Phi}, \Psi)$. There are different techniques to solve the MaxSat problem for a given instance, but we focus on a counter-based technique that is particularly suited if an instance needs to be solved repeatedly after adding additional constraints. For further details, we refer to [13].

3 Computing Minimal Critical Command Sets

In this section we present our novel approach to compute a critical command set as introduced in Section 2.2. For the remainder of this section, let $\mathfrak{P} =$

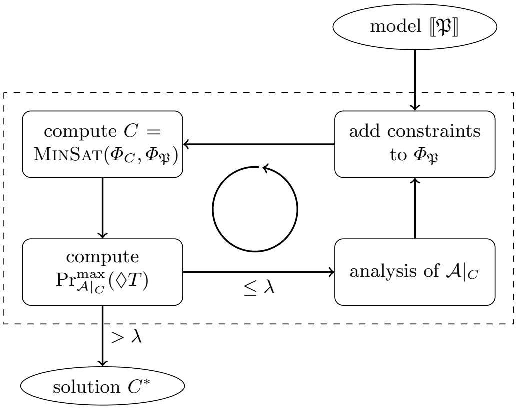
model $\llbracket\mathfrak{P}\rrbracket$
add constraints to $\Phi_{\mathfrak{P}}$
compute $C = $ MinSat$(\Phi_C, \Phi_{\mathfrak{P}})$
compute $\Pr^{\max}_{\mathcal{A}|_C}(\lozenge T)$
$\leq \lambda$
analysis of $\mathcal{A}|_C$
$> \lambda$
solution C^*

Fig. 3. A schematic overview of our MaxSat-based approach.

$(\mathit{Var}, \nu_{\text{init}}, \mathbf{M})$ with $\mathbf{M} = \{M_1, \ldots, M_k\}$ be a probabilistic program with modules $M_i = (\mathit{Var}_i, \mathit{Act}_i, C_i)$ for $1 \leq i \leq k$. Let $\mathcal{A} = \llbracket\mathfrak{P}\rrbracket = (S, s_{\text{init}}, \mathit{Act}, \mathbf{P})$ and $\mathcal{A}|_C = \llbracket\mathfrak{P}|_C\rrbracket$ for a given set C of commands. We assume that the labeling L of transitions with command labels according to Section 2.2 is given. Furthermore, let $T \subseteq S$ be a set of target states and $\lambda \in [0, 1]$ such that $\mathcal{A} \not\models \mathcal{P}_{\leq\lambda}(\lozenge T)$ in order to guarantee the existence of a critical command set. The task is to compute a *minimal* critical command set, i. e., a smallest set C^* of commands such that $\mathcal{A}|_{C^*} \not\models \mathcal{P}_{\leq\lambda}(\lozenge T)$ or, equivalently, $\Pr^{\max}_{\mathcal{A}|_{C^*}}(\lozenge T) > \lambda$. The fact that this problem is NP-hard in the size of the probabilistic program can be shown by a reduction from exact 3-cover (X3C) very similar to the one in [32]. Note that a solution C^* of this problem is not unique as there may be more than one set of commands of size $|C^*|$ that suffices to violate the reachability property.

3.1 Algorithm

Basic idea. Clearly, for realistic problems, an enumeration of all possible command sets is infeasible. Hence, it is crucial to obtain additional information from the model to rapidly guide the search. For example, if an optimal solution C^* contains a synchronizing command c of module M_i, it must also contain at least one command of each module M_j that needs to synchronize with c, i. e., $\text{act}(c) \in \mathit{Act}_j$. Likewise, if a command c does not lead to a target state in T directly, adding c to the set C^* implies that it must also contain at least one command (or command combination) that may directly follow c in $\mathcal{A}$, which in turn may trigger other implications. We therefore strive to encode as much information as possible from the program $\mathfrak{P}$ in the form of a set $\Phi_{\mathfrak{P}}$ of logical formulae. Primarily, the formulae are built over variables $\Phi_C := \{x_c \mid \exists i \in \{1, \ldots, k\} : c \in C_i\}$ whose truth values indicate whether a certain command is included in the current hypothesis or not. We then use a MaxSat solver to compute the smallest set C of commands that is in accordance with all constraints in $\Phi_{\mathfrak{P}}$ by solving $C = \textsc{MinSat}(\Phi_C, \Phi_{\mathfrak{P}})$.[5] Finally, a model checker is invoked to determine $p = \mathrm{Pr}^{\max}_{\mathcal{A}|_C}(\Diamond T)$. If p exceeds λ, we do not only know that the current set C suffices to exceed λ but also that C is among the sets of minimal size for which this holds, because smaller candidate sets are enumerated first by the solver. If, on the other hand, $p \leq \lambda$, we analyze why C was insufficient, add appropriate implications to $\Phi_{\mathfrak{P}}$ and iterate the algorithm until a sufficient set C was found. A schematic overview of this procedure is depicted in Figure 3.

3.2 Building the Initial Constraint System $\Phi_{\mathfrak{P}}$

As previously mentioned, the first step of our algorithm consists of statically deriving information about the model $\mathcal{A} = [\![\mathfrak{P}]\!]$ to guide the search for a minimal critical command set C^*. Note that it suffices to consider the reachable state space of $\mathcal{A}$ for all the constraints we derive.

Guaranteed commands. For typical models, some commands need to be taken along all paths from the initial state to a target state. It is thus beneficial to determine this set in a preprocessing step to the actual search and thereby possibly prune large parts of the search space. We therefore compute the set of *guaranteed commands* using a standard fixed point analysis [24] on $\mathcal{A}$.

Example 5. All paths that lead to the target state in $[\![\mathfrak{P}_{\mathrm{Ex}}]\!]$ must go along transitions generated by the commands c_1, c_3 and c_4, so it is a priori known that a solution must contain all of them.

Synchronization implications. By the semantics of the program $\mathfrak{P}$, it is required that a synchronizing command c in module M_i can only generate a transition together with synchronizing commands c_j of all modules M_j, $i \neq j$, with $\mathrm{act}(c) \in Act_j$. Consequently, we can conclude that any optimal solution C^* with $c \in C^*$ must also contain at least one command c_j of each synchronizing module such that the commands c and c_j are simultaneously enabled. Formally, we assert

$$x_c \rightarrow \bigvee_{s \in \mathrm{src}(c)} \bigvee_{\substack{(\alpha,\mu)\in \mathbf{P}(s)\\ \ell_c \in L(s,\alpha,\mu)}} \bigwedge_{\substack{\ell_{c'} \in L(s,\alpha,\mu)\\ c \neq c'}} x_{c'} \quad \text{for all } M_i \in \mathbf{M} \text{ and } c \in C_i. \tag{1}$$

Example 6. In $\mathfrak{P}_{\mathrm{Ex}}$, the command c_3 in the first module and c_4 in the second module need to synchronize in order to generate a transition. The synchronization implications $x_{c_3} \rightarrow x_{c_4}$ and $x_{c_4} \rightarrow x_{c_3}$ ensure that candidate sets must either contain both or none of the two commands.

[5] Formally, this is not entirely correct, since MinSat returns a satisfying assignment. More formally, we let $C = \{c \mid \nu(x_c) = 1\}$ where $\nu = \textsc{MinSat}(\Phi_C, \Phi_{\mathfrak{P}})$.

Successor and predecessor implications. Observe that a candidate set C is surely sub-optimal if $c \in C$ only participates in generating transitions in $\mathcal{A}|_C$ that lead to non-target states without outgoing transitions. In this case, no path from the initial state to a target state can visit a transition that was generated by c and, hence, c can be dropped from C without affecting the reachability probability. Thus, we can assert that each $c \in C$ either possibly leads to a target state directly or leads to some state that has a non-empty transition set. Hence, we add for all $M_i \in \mathbf{M}$ and $c \in C_i$ with $\mathrm{dst}(c) \cap T = \emptyset$ the constraint

$$x_c \rightarrow \bigvee_{s' \in \mathrm{dst}(c)} \; \bigvee_{(\alpha,\mu) \in \mathbf{P}(s')} \; \bigwedge_{\substack{\ell_{c'} \in L(s',\alpha,\mu) \\ c \neq c'}} x_{c'} \tag{2}$$

to $\varPhi_{\mathfrak{P}}$. Analogously, for each command c' that is not enabled in the initial state, i. e., $s_{\text{init}} \notin \mathrm{src}(c')$, we select a combination of commands that leads to some state $s \in \mathrm{src}(c')$ by enforcing

$$x_{c'} \rightarrow \bigvee_{s \in \mathrm{src}(c')} \; \bigvee_{s' \in \mathrm{pred}(s)} \; \bigvee_{\substack{(\alpha,\mu) \in \mathbf{P}(s') \\ s \in \mathrm{succ}(s',\alpha,\mu)}} \; \bigwedge_{\substack{\ell_c \in L(s',\alpha,\mu) \\ c \neq c'}} x_c \tag{3}$$

As slight variations of these implications, we can encode that at least one of the transitions of the initial state and at least one transition that has a target state as a direct successor are generated by (a subset of) C.

Example 7. In our running example $\mathfrak{P}_{\text{Ex}}$, the command c_1 must be used in order to reach states that have c_3 enabled. Consequently, we can add the predecessor implication $x_{c_3} \rightarrow x_{c_1}$. Likewise, all transitions generated by c_2 must be preceded by either a transition generated by c_1 or by the synchronization of c_3 and c_4, so $x_{c_2} \rightarrow x_{c_1} \vee (x_{c_3} \wedge x_{c_4})$ can be added to $\varPhi_{\mathfrak{P}_{\text{Ex}}}$. Finally, since the initial state must have an outgoing transition, we can assert x_{c_1}. Note that these are only a few of the constraints that can be constructed for $\mathfrak{P}_{\text{Ex}}$.

Extended backward implications. Reconsider the probabilistic program $\mathfrak{P}_{\text{Ex}}$. Our previously presented backward implications assert that if a candidate set C contains command c_2, it also contains either c_1 or both c_3 and c_4, because both command combinations may directly precede c_2. However, it is obvious that c_1 must always be executed before c_2, because otherwise the guard of c_2 never becomes true. Put differently, only command c_1 "enables" c_2 and should therefore be implied by the choice of c_2.

More formally, we say that a set of commands C' *enables* a non-synchronizing command c if there is at least one state s such that (i) $s \notin \mathrm{src}(c)$, (ii) there is an $(\alpha, \mu) \in \mathbf{P}(s)$ with $L(s, \alpha, \mu) = C'$ and a successor state $s' \in \mathrm{succ}(s, \alpha, \mu)$ such that $s' \in \mathrm{src}(c)$, (iii) c is not enabled in the initial state, i. e., $s_{\text{init}} \notin \mathrm{src}(c)$. Let $\mathrm{enab}(c)$ denote the set of all command sets that enable c. We can then assert

$$x_c \rightarrow \bigvee_{C' \in \mathrm{enab}(c)} \; \bigwedge_{c' \in C'} x_{c'} \tag{4}$$

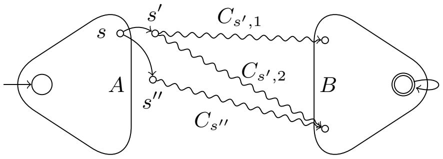

Fig. 4. A restricted model with only unreachable target states.

for all commands c with $s_{\text{init}} \notin \text{src}(c)$ without ruling out optimal solutions. A similar, yet more involved implication can also be asserted for synchronizing commands, but is omitted for the sake of simplicity.

Enforce reachability of a target state. Using a similar construction as the one in [32], reachability of a target state can be encoded in the constraints if a MaxSmt solver is used. For further details we refer to Appendix A.

3.3 Analysis of Insufficient Command Sets

After the initial constraint set was constructed, a MinSat problem is solved to obtain a smallest command set C that adheres to these constraints. The restricted model $\mathcal{A}|_C$ is then dispatched to a model checker to verify or refute $\mathcal{P}_{\leq\lambda}(\Diamond T)$. If the reachability probability in $\mathcal{A}|_C$ exceeds λ, a solution for the minimal critical command set problem has been found, because the set C is, by construction, the smallest candidate set. However, in the more likely event of not exceeding λ, we aim to derive additional constraints from the constrained model that guide the solver towards a solution with a higher reachability probability. While it is easily possible to rule out just the current (insufficient) candidate set C by adding a formula to $\Phi_{\mathfrak{P}}$, we strive to rule out more insufficient candidate sets to guide the search. We illustrate this procedure for the case where the reachability probability is zero, i. e., the target states are unreachable altogether (which can, of course, only happen if the constraints to enforce reachability of a target state are not used). A similar reasoning can be applied in case the probability is non-zero. Assume that the current candidate C induces a restricted model $\mathcal{A}|_C$ in which the target states are unreachable. Figure 4 sketches the shape of $\mathcal{A}|_C$ in this scenario where A is the set of states reachable from the initial state and B is formed by all states that can reach a target state. In order to increase the probability of reaching T, any future candidate set $C' \supsetneq C$ must generate a path from A to B in $\mathcal{A}|_{C'}$ in order to reach a target state. More concretely, we do not need to consider all states in A but rather those states that are on the border $\mathit{border}_C(A, B)$ of A, meaning that they possess a transition in the unrestricted model $\mathcal{A}$ that is (i) not present in $\mathcal{A}|_C$, (ii) leaves the set A and (iii) is the first transition of a finite path that ends in B. The states in question can be obtained using efficient graph searches that are essentially breadth-first searches. Having identified these states, we perform a usually cheap analysis that for a given state s determines the set of commands $C_{s \rightsquigarrow B}$ that need to be taken along all paths from s to some state $s' \in B$. We could then assert

$$\bigwedge_{c\in C} x_c \rightarrow \bigvee_{s\in \mathit{border}_C(A,B)} \bigwedge_{c\in C_{s\rightsquigarrow B}} x_c \tag{5}$$

to express that any future candidate $C' \supsetneq C$ must contain all commands necessary for reaching B from some state $s \in \mathit{border}_C(A,B)$. However, as the sets $C_{s\rightsquigarrow B}$ are possibly empty, the above constraint does not necessarily eliminate any candidate (not even C). Hence, to guarantee elimination of C as a candidate, we take an intermediate step from a border state s to some state $s' \notin A$ to ensure some transition leaving A is generated. Formally, this leads to the constraint

$$\bigwedge_{c\in C} x_c \rightarrow \bigvee_{s\in \mathit{border}_C(A,B)} \; \bigvee_{(\alpha,\mu)\in \mathbf{P}(s)} \; \bigvee_{\substack{s'\in \mathrm{succ}(s,\alpha,\mu)\\ s'\notin A}} \Big(\underbrace{\bigwedge_{\ell_c\in L(s,\alpha,\mu)} x_c}_{s\rightarrow s'} \wedge \underbrace{\bigwedge_{c'\in C_{s'\rightsquigarrow B}} x_{c'}}_{s'\rightsquigarrow B} \Big).$$

Example 8. Assume that the solver found the candidate set $C = \{c_1\}$ while searching for a critical command set in $\mathfrak{P}_{\mathrm{Ex}}$ for φ. This would not be possible if all previously mentioned constraints are added, but is assumed for the sake of simplicity. Then, $\mathcal{A}|_C$ comprises the three reachable states $A = \{s_{\mathrm{init}}, s_1, s_2\}$, the latter two of which are on the $\mathit{border}_C(A, \{s_4\})$. Since all transitions leaving the two states s_1 and s_2 towards s_4 are jointly generated by the commands c_3 and c_4, we add the constraint $x_{c_1} \rightarrow x_{c_3} \wedge x_{c_4}$ to rule out C.

3.4 Correctness and Completeness

The *correctness* of our approach basically depends on the fact, that all possible sets of commands are incrementally enumerated until one set fits the requirement given by the violated property. If no additional constraints are used, the MaxSat method starts with the minimal possible subset of commands and increases this size until the model checker reports the violation of the property for the then optimal set of commands C^*. *Completeness* of the algorithm holds, as all candidates are enumerated at most once and there are finitely many candidate command sets.

What remains to be argued is that all constraints in the set $\Phi_{\mathfrak{P}}$ are correct in the sense that each optimal solution C^* of the critical command set problem necessarily satisfies all of these constraints. Put differently, every constraint $\varphi \in \Phi_{\mathfrak{P}}$ only restricts the solution space such that no optimal solution, i. e., no *minimal critical command set*, is ruled out. Due to the page limit, we abstain from giving a formal proof, but refer to Sections 3.2 and 3.3 where the correctness of all constraints is explained.

4 Evaluation

Implementation. We implemented our technique in roughly 1000 lines of `C++` code. The prototype was developed in the context of a model checker under development. We employ the counter-based MaxSat procedure described in Section 2.3 using Z3 4.3 [12] as the underlying Sat/Smt solver. To provide a fair comparison, we additionally implemented the MILP-based approach [32] using the commercial solver Gurobi 5.6 [15]. We also added the detection of guaranteed commands (see Section 3.2) as an optimization to the MILP approach and added the resulting information to the problem encoding. As proposed in [32], we added the so-called *scheduler cuts*, an additional set of constraints to rule out suboptimal solutions, to the MILP encoding, because they strongly tend to improve the performance of the solver. According to [32], all other cuts may have a mixed influence on the performance of the solver and were thus omitted.

Case studies. For the evaluation of the prototype we used four benchmarks that were all previously considered in [32]. They can be found on Prism's website.

▶ Consensus Protocol. The probabilistic program coin(N, K) models the shared coin protocol of a randomized consensus algorithm [3]. It is used to determine a preference between two choices, each of which appears with a certain probability. The shared coin protocol is parametrized in 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 $\mathcal{P}_{<\lambda}(\Diamond(\text{finished} \land \text{all\_coins\_equal\_1}))$ that is satisfied if the probability to finish the protocol with all coins showing the value 1 is below λ.

▶ Wireless LAN. The case study wlan(B, D) concerns 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. Parameter B denotes the maximally allowed value of the backoff counter. We check the property $\mathcal{P}_{<\lambda}(\Diamond(\text{num\_collisions} = D))$, putting an upper bound on the probability that the maximal number of collisions D occurs.

▶ CSMA. The csma(N, B) model concerns the IEEE 802.3 CSMA/CD network protocol with N the number of processes wanting to access a common channel and B is the maximal value of the backoff counter. We check that the probability of all stations successfully sending their messages before a collision with maximal backoff occurs is less than λ, i. e., $\mathcal{P}_{<\lambda}(\neg\text{collision } U \text{ delivered})$.

▶ Firewire. Finally, fw(N) models the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus (called "FireWire") [28]. It is a leader election protocol that is executed each time a node enters or leaves the network. The parameter N denotes the delay of the wire. We check $\mathcal{P}_{<\lambda}(\Diamond\text{leader\_elected})$, i. e., that the probability of finally electing a leader is below λ.

Experimental results. All experiments were conducted on an Intel Core i7 920 quadcore processor clocked at 2.66 GHz with 12 GB RAM running Mac OS 10.9. We set a timeout of 12 hours for each individual (single-threaded) experiment. Table 1 summarizes the results of our experiments. Next to some model statistics about the particular model, the considered probability bound λ and the maximal reachability probability p^* of the unrestricted model are shown. Furthermore, we give the number of relevant commands of the probabilistic program

						MILP[32]		MaxSat		
model	states	trans.	λ/p^*	comm.	$\lvert C^*\rvert$	Time	Mem.	Time	Mem.	enum.
coin(2, 2)	272	492	0.4 / 0.56	10 (4)	9	TO	> 0.04	**0.08**	**0.02**	54%
coin(4, 4)	43136	144352	0.4 / 0.54	20 (8)	17	TO	> 2.60	**1876**	**0.07**	50%
coin(4, 6)	63616	213472	0.4 / 0.53	20 (8)	17	TO	> 6.70	**6231**	**0.09**	50%
coin(6, 2)	1258240	6236736	0.4 / 0.59	30 (12)	–	TO	> 8.36	TO	> 1.54	–
csma(2, 4)	7958	10594	0.5 / 0.999	38 (21)	36	31.4	0.07	**2.26**	**0.04**	0.09%
csma(4, 2)	761962	1327068	0.4 / 0.78	68 (22)	53	TO	> 9.60	**18272**	**0.92**	3.9E-9%
fw(1)	1743	2199	0.5 / 1	64 (6)	24	207.25	0.16	**16.14**	**0.05**	1.4E-10%
fw(10)	17190	29366	0.5 / 1	64 (6)	24	9196	0.84	**90.47**	**0.07**	1.4E-10%
fw(36)	212268	481792	0.5 / 1	64 (6)	24	TO	> 3.20	**1542**	**0.34**	1.4E-10%
wlan(0, 2)	6063	10619	0.1 / 0.184	42 (22)	33	TO	> 1.99	**1.6**	**0.03**	0.02%
wlan(2, 4)	59416	119957	4E-4 / 7.9E-4	48 (26)	39	TO	> 4.03	**50.27**	**0.07**	0.01%
wlan(6, 6)	5007670	11475920	1E-7 / 2.2E-7	52 (30)	43	ERR	–	**5035**	**3.86**	0.01%

Table 1. The results of the experiments.

and how many of them are guaranteed commands (see Section 3.2). Here, relevant means that they appear on at least one path from the initial to a target state. The size of an optimal solution C^* as well as the runtimes and memory consumption (in seconds and gigabytes, respectively) of both the MILP- and the MaxSat-based approach are listed in the following five columns, where TO indicates a timeout. For the MILP approach [32], we performed experiments with and without using the scheduler cuts and report on the best of these results. Encoding reachability of a target state (see Section 3.2) tended to be rather expensive for Z3: in almost all cases it slowed down the overall computation and thus we list the times obtained without adding these constraints. For all considered models the MaxSat approach significantly outperforms the MILP-based technique. While for the fw and csma models the speed-up is about one to two orders of magnitude, for the coin and wlan case studies it goes as high as five orders of magnitude. Enabling the multi-threading capabilities of Gurobi (8 threads on our machine) did not change the order of improvement we obtained. Furthermore it can be seen that the MaxSat approach consistently uses one order of magnitude less memory. For the largest wlan example, Gurobi reported a wrong result ($|C^*| = 38$). Performing model checking on the restricted model revealed that the computed command set does not suffice to violate the property. After careful inspection of our implementation and considering that all other results coincide, we believe this is due to numerical instabilities in the solving technique that could not be eliminated by setting its tolerances to the lowest possible value. Finally, to indicate to what extent the constraints in our new approach guide the search as opposed to an unguided enumeration of candidate sets, Table 1 also shows the fraction $\frac{\ell}{\sum_{i=0}^{k}\binom{n}{i}}$ (column enum.) where ℓ is the number of candidate sets enumerated, k is the number of commands in C^* minus the guaranteed commands, and n is the number of all relevant but not guaranteed commands. It represents the ratio of candidate sets that were tested to all candidate sets with at most $|C^*|$ commands that contain all guaranteed commands. Hence, it indicates which fragment of the search space could be pruned. For all case studies except the coin models, the constraints avoided huge parts of the search space. Interestingly, despite exploring more than half of the search space, the MaxSat approach is still much faster on the `coin` examples, which is due to the efficient underlying testing procedure for candidate sets.

Further, it is noteworthy that, depending on the model, the analysis of insufficient command sets (see Section 3.3) consumes the largest fraction of the runtime: for the `csma` and `fw` examples more than 80 % of the runtime is spent on it, whereas for the other examples it only contributes a very small fraction to the overall runtime, which is then clearly dominated by model checking.

Since it is a known characteristic of MILP solvers that good solutions are found quickly, but the solver is unable to prove their optimality in reasonable time, we also examined the solution progress of `Gurobi` on the models for which it times out. For the smaller to medium-sized models, for example `wlan`(0,2) and `fw`(10), the MILP solver finds a solution of optimal size in 4 and 388 seconds, respectively, but then fails at proving optimality. However, for the largest models of each case study that could be solved within time by the MaxSat approach, `Gurobi` is unable to find any solution until the time limit is reached.

5 Conclusion

We have presented a novel technique for computing counterexamples at the modeling level of probabilistic programs, which we believe to complement existing counterexample techniques in the probabilistic setting. In contrast to the previous approach tackling the minimal command set problem, our new technique substantially improves computation time and memory consumption and scales to systems with millions of states. Furthermore, it can be readily applied to the wider range of monotonic properties by introducing problem-specific constraints. However, the performance of the technique can still be improved. It is easily parallelizable and could therefore benefit from accelerators like GPUs. More sophisticated analysis techniques of candidate sets that failed to exceed the probability threshold could be both more efficient to compute and more beneficial with respect to the number of suboptimal sets that could be pruned. Moreover, the computed counterexamples can be further reduced in size by applying branch minimization [32]. Future work also includes possible applications in techniques that are guided by counterexamples, such as CEGAR [18,9] or assume-guarantee reasoning [6].

References

1. H. Aljazzar and S. Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Trans. on Software Engineering*, 36(1):37–60, 2010.
2. R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
3. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990.

4. C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.
5. R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1st edition, 1957.
6. M. G. Bobaru, C. S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 135–148. Springer, 2008.
7. P. E. Bulychev, A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen. Checking and distributing statistical model checking. In *Proc. of NFM*, volume 7226 of *LNCS*, pages 449–463. Springer, 2012.
8. R. Canetti, L. Cheung, D. K. Kaynar, M. Liskov, N. A. Lynch, O. Pereira, and R. Segala. Analyzing security protocols using time-bounded task-PIOAs. *Discrete Event Dynamic Systems*, 18(1):111–159, 2008.
9. K. Chatterjee, M. Chmelík, and P. Daca. CEGAR for qualitative analysis of probabilistic systems. In *Proc. of CAV*, volume 8559 of *LNCS*, pages 473–490. Springer, 2014.
10. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
11. E. M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 208–224. Springer, 2003.
12. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
13. Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *Proc. of SAT*, volume 4121 of *LNCS*, pages 252–265. Springer, 2006.
14. P. Gastin and P. Moro. Minimal counterexample generation for SPIN. In *Proc. of SPIN*, volume 4595 of *LNCS*, pages 24–38. Springer, 2007.
15. Gurobi optimization, inc.: `Gurobi` optimizer reference manual version 5.6 (2014). `http://www.gurobi.com/resources/documentation`.
16. T. Han, J.-P. Katoen, and B. Damman. Counterexample generation in probabilistic model checking. *IEEE Trans. on Software Engineering*, 35(2):241–257, 2009.
17. H. Hansen and J. Geldenhuys. Cheap and small counterexamples. In *Proc. of SEFM*, pages 53–62. IEEE Computer Society, 2008.
18. H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 162–175. Springer, 2008.
19. N. Jansen, R. Wimmer, E. Ábrahám, B. Zajzon, J.-P. Katoen, and B. Becker. Symbolic counterexample generation for large discrete-time Markov chains. *Science of Computer Programming*, 91(A):90–114, 2014.
20. J.-P. Katoen, J. van de Pol, M. Stoelinga, and M. Timmer. A linear process-algebraic format with data for probabilistic automata. *Theoretical Computer Science*, 413(1):36–57, 2012.
21. J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(2):90–104, 2011.
22. M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
23. F. Leitner-Fischer and S. Leue. Probabilistic fault tree synthesis using causality computation. *IJCCBS*, 4(2):119–143, 2013.
24. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis (2. corr. print).* Springer, 2005.

25. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.
26. V. Schuppan and A. Biere. Shortest counterexamples for symbolic model checking of LTL with past. In *Proc. of TACAS*, volume 3440 of *LNCS*, pages 493–509. Springer, 2005.
27. R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.
28. M. Stoelinga. Fun with firewire: A comparative study of formal verification methods applied to the IEEE 1394 root contention protocol. *Formal Aspects of Computing*, 14(3):328–337, 2003.
29. R. Wimmer, B. Braitling, and B. Becker. Counterexample generation for discrete-time Markov chains using bounded model checking. In *Proc. of VMCAI*, volume 5403 of *LNCS*, pages 366–380. Springer, 2009.
30. R. Wimmer, N. Jansen, E. Ábrahám, J.-P. Katoen, and B. Becker. Minimal critical subsystems for discrete-time Markov models. In *Proc. of TACAS*, volume 7214 of *LNCS*, pages 299–314. Springer, 2012.
31. R. Wimmer, N. Jansen, E. Ábrahám, J.-P. Katoen, and B. Becker. Minimal counterexamples for linear-time probabilistic verification. *Theoretical Computer Science*, 2014. DOI: 10.1016/j.tcs.2014.06.020 (accepted for publication).
32. R. Wimmer, N. Jansen, A. Vorpahl, E. Ábrahám, J.-P. Katoen, and B. Becker. High-level counterexamples for probabilistic automata. In *Proc. of QEST*, volume 8054 of *LNCS*, pages 18–33. Springer, 2013.

A Encoding Reachability of Target States

Depending on the model, our guided enumeration technique may produce many hypotheses with unreachable target states (leading to reachability probability 0). However, it is possible to completely avoid such hypotheses at the expense of adding additional SMT constraints and using a MaxSmt solver instead of a MaxSat solver. To encode reachability of target states, we use additional real-valued variables r_s for each $s \in S$ and Boolean variables $t_{s,s'}$ for all pairs $(s, s') \in S \times S$ of states such that $s' \in \mathrm{succ}_{\mathcal{A}}(s)$. The r_s variables are used to associate each state with a real number to induce a partial ordering of states. A variable $t_{s,s'}$ may only be set to 1 if the value r_s is strictly smaller than the value of $r_{s'}$ and s' is a successor of s in the restricted model. We require that the initial state s_{init} has a successor state s' with $t_{s_{\mathrm{init}},s'} = 1$ and that each non-target state s' having a predecessor state s with $t_{s,s'} = 1$ also has a successor state s'' with $t_{s',s''} = 1$. This induces a path starting at the initial state and ending at a target state. This path is loop-free since the r_s-values strictly increase along the path. This construction is similar to the one in [32] and comprises the following constraints:

$$\bigvee_{s' \in \mathrm{succ}(s_{\mathrm{init}})} t_{s_{\mathrm{init}},s'} \tag{6a}$$

$$\forall s \in S \setminus T : \bigvee_{s' \in \mathrm{pred}(s)} t_{s',s} \rightarrow \bigvee_{s'' \in \mathrm{succ}(s)} t_{s,s''} \tag{6b}$$

$$\forall s, s' \in S : t_{s,s'} \rightarrow \bigvee_{\substack{(\alpha,\mu) \in \mathbf{P}(s),\\ s' \in \mathrm{succ}(s,\alpha,\mu)}} \bigwedge_{\ell_c \in L(s,\alpha,\mu)} x_c \tag{6c}$$

$$\forall s \in S, s' \in S \setminus T : t_{s,s'} \rightarrow r_s < r_{s'} \tag{6d}$$

Let $\mathcal{T} = \{(s, s') \mid t_{s,s'} = 1\}$ be the relation induced by the $t_{s,s'}$ variables. Firstly, the constraints (6c) make sure that $\mathcal{T}$ only relates states that are also related by a generated transition. Secondly, (6a) and (6b) ensure that the initial state is related to some other state and that $(s, s') \in \mathcal{T}$ with $s' \notin T$ implies $(s', s'') \in \mathcal{T}$ for some state s''. Finally, (6d) guarantees a strictly ascending order between related states. Roughly speaking, these constraints encode the reachability of a target state, because the requirement of a strict ordering on the states along the chosen transitions is only released when a target state is reached. While this set of constraints possibly rules out a lot of unnecessary candidate sets C, it is obvious that it might be costly regarding the runtimes of the MaxSmt solver.