Counterexample Generation for Discrete-Time Markov Models: An Introductory Survey

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

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

Abstract. This paper is an introductory survey of available methods for the computation and representation of probabilistic counterexamples for discrete-time Markov chains and probabilistic automata. In contrast to traditional model checking, probabilistic counterexamples are sets of finite paths with a critical probability mass. Such counterexamples are not obtained as a by-product of model checking, but by dedicated algorithms. We define what probabilistic counterexamples are and present approaches how they can be generated. We discuss methods based on path enumeration, the computation of critical subsystems, and the generation of critical command sets, both, using explicit and symbolic techniques.

1 Introduction

The importance of counterexamples. One of the main strengths of model checking is its ability to automatically generate a *counterexample* in case a model refutes a given temporal logic formula [1]. Counterexamples are the most effective feature to convince system engineers about the value of formal verification [2]. First and foremost, counterexamples provide essential diagnostic information for *debugging* purposes. A counterexample-guided simulation of the model at hand typically gives good insight into the reason of refutation. The same applies when using counterexamples as witnesses showing the reason of fulfilling a property. Counterexamples are effectively used in *model-based testing* [3]. In this setting, models are used as blueprint for system implementations, i. e., the conformance of an implementation is checked against a high-level model. Here, counterexamples obtained by verifying the blueprint model act as test cases that, after an adaptation, can be issued on the system-under-test. Counterexamples are at the core of obtaining feasible *schedules* in planning applications. Here, the idea is to verify the negation of the property of interest—it is never possible to reach a given target state (typically the state in which all jobs have finished their execution) within k steps—and use the counterexample as an example schedule illustrating that all jobs can complete within k steps. This principle is exploited in e. g.,

task scheduling in timed model checking [4]. A more recent application is the synthesis of *attacks* from counterexamples for showing how the confidentiality of programs can be broken [5]. These so-called refinement attacks are important, tricky, and are notorious in practice. Automatically generated counterexamples act as attacks showing how multi-threaded programs under a given scheduler can leak information. Last but not least, counterexamples play an important role also in *counterexample-guided abstraction refinement* (*CEGAR*) [6], a successful technique in software verification. Spurious counterexamples resulting from verifying abstract models are exploited to refine the (too coarse) abstraction. This abstraction-refinement cycle is repeated until either a concrete counterexample is found or the property can be proven.

Counterexample generation. For these reasons, counterexamples have received considerable attention in the model checking community. Important issues have been (and to some extent still are) how counterexamples can be *generated* efficiently, preferably in an on-the-fly manner during model checking, how memory consumption can be kept small, and how counterexamples themselves can be kept succinct, and be *represented* at the model description level (rather than in terms of the model itself). The *shape* of a counterexample depends on the property specification language and the checked formula. The violation of *linear-time safety* properties is indicated by finite paths that end in a "bad" state. Therefore, for logics such as LTL, typically finite paths through the model suffice. Although LTL model checking is based on (nested) depth-first search, LTL model checkers such as Spin incorporate breadth-first search algorithms to generate *shortest* counterexamples, i. e., paths of minimal length. The violation of *liveness* properties, instead, require infinite paths ending in a cyclic behavior under which something "good" will never happen. These lassos are finitely represented by concatenating the path until reaching the cycle with a single cycle traversal. For *branching-time* logics such as CTL, such finite paths suffice as counterexamples for a subclass of universally quantified formulas. To cover a broader spectrum of formulas, though, more general shapes are necessary, such as tree-like counterexamples [7]. As model-checking suffers from the combinatorial growth of the number of states—the so-called state space explosion problem—various successful techniques have been developed to combat this. Most of these techniques, in particular *symbolic* model checking based on binary decision diagrams (BDDs, for short), have been extended with symbolic counterexample generation algorithms [8]. 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. A survey of practical and theoretical results on counterexample generation in model checking can be found in [2].

Probabilistic model checking. This paper surveys the state of the art in counterexample generation in the setting of *probabilistic* model checking [9–11]. Probabilistic model checking is a technique to verify system models in which transitions are equipped with random information. Popular models are discrete- and continuous-time Markov chains (DTMCs and CTMCs, respectively), and variants thereof which exhibit non-determinism such as probabilistic automata (PA). Efficient model-checking algorithms for these models have been developed, implemented in a variety of software tools, and applied to case studies from various application areas ranging from randomized distributed algorithms, computer systems and security protocols to biological systems and quantum computing. The crux of probabilistic model checking is to appropriately combine techniques from numerical mathematics and operations research with standard reachability analysis and model-checking techniques. In this way, properties such as "the (maximal) probability to reach a set of bad states is at most 0.1" can be automatically checked up to a user-defined precision. Markovian models comprising millions of states can be checked rather fast by dedicated tools such as PRISM [12] and MRMC [13]. These tools are currently being extended with counterexample generation facilities to enable the possibility to provide useful diagnostic feedback in case a property is violated. More details on probabilistic model checking can be found in, e. g., [9, 14, 15].

Counterexamples in a probabilistic setting. Let us consider a finite DTMC, i. e., a Kripke structure whose transitions are labeled with discrete probabilities. Assume that the property "the (maximal) probability to reach a set of bad states is at most 0.1" is violated. That means that the accumulated probability of all paths starting in the initial state s_0 and eventually reaching a bad state exceeds 10%. This can be witnessed by a set of finite paths all starting in s_0 and ending in a bad state whose total probability exceeds 0.1. Counterexamples are thus *sets of finite paths*, or viewed differently, a finite tree rooted at s_0 whose leafs are all bad. Evidently, one can take all such paths (i.e, the complete tree) as a counterexample, but typically succinct diagnostic information is called for. There are basically two approaches to accomplish this: *path enumeration* and *critical subsystems*. In contrast to standard model checking, these algorithmic approaches are employed after the model-checking phase in which the refutation of the property at hand has been established. Up to now, there is no algorithm to generate probabilistic counterexamples during model checking.

Path enumeration. For DTMCs, a counterexample can be obtained by explicitly enumerating the paths comprising a counterexample. A typical strategy is to start with the most probable paths and generate paths in order of descending probability. This procedure stops once the total probability of all generated paths exceeds the given bound, ensuring minimality in terms of number of paths. Algorithmically, this can be efficiently done by casting this problem as a *k shortest path* problem [16, 17] where k is not fixed a priori but determined on the fly during the computation. This method yields a *smallest* counterexample whose probability mass is maximal—and thus most discriminative—among all minimal

counterexamples. This approach can be extended to until properties, bounded versions thereof, ω-regular properties, and is applicable to non-strict upper and lower bounds on the admissible probability. Whereas [16, 17] exploit existing k shortest path algorithms with pseudo-polynomial complexity (in k), [18] uses *heuristic search* to obtain most probable paths. Path enumeration techniques have also been tackled with *symbolic* approaches like *bounded model checking* [19] extended with *satisfiability modulo theories* (*SMT*) techniques [20], and using *BDD-techniques* [21]. The work [22] proposes to compute and represent counterexamples in a succinct way by *regular expressions*. Inspired by [23], these regular expressions are computed using a state elimination approach from automata theory that is guided by a k shortest paths search. Another compaction of counterexamples is based on the abstraction of *strongly-connected components* (SCCs, for short) of a DTMC, resulting in an acyclic model in which counterexamples can be determined with reduced effort [24]. An approach to compute counterexamples for non-deterministic models was proposed in [25].

Critical subsystems. Alternatively to generating paths, here a fragment of the discrete-time Markov model at hand is determined such that in the resulting sub-model a bad state is reached with a likelihood exceeding the threshold. Such a fragment is called a *critical subsystem*, which is *minimal* if it is minimal in terms of number of states or transitions, and *smallest* if it is minimal and has a maximal probability to reach a bad state under all minimal critical subsystems. A critical subsystem induces a counterexample by the set of its paths. Determining smallest critical subsystems for probabilistic automata is an NP-complete problem [26], which can be solved using mixed integer linear programming techniques [27, 28]. Another option is to exploit *k shortest path* [29] and *heuristic search* [30] methods to obtain (not necessarily smallest or minimal) critical subsystems. *Symbolic* approaches towards finding small critical subsystems have been developed in [31, 32]. The approach [24] has been pursued further by doing SCC reduction in a hierarchical fashion yielding *hierarchical counterexamples* [29].

Modeling-language-based counterexamples. Typically, huge and complex Markov models are described using a *high-level modeling language*. Having a human-readable specification language, it seems natural that a user should be pointed to the part of the high-level model description which causes the error, instead of getting counterexamples at the state-space level. This has recently initiated finding *smallest critical command sets*, i. e., the minimal fragment of a model description such that the induced (not necessarily minimal) Markov model violates the property at hand, thereby maximizing the probability to reach bad states. For PRISM, models are described in a stochastic version of Alur and Henzinger's reactive modules [33]. In this setting, a probabilistic automaton is typically specified as a parallel composition of modules. The behavior of a single module is described using a set of probabilistic guarded commands. Computing a smallest critical command set amounts to determining a minimal set of guarded commands that together induce a critical subsystem, with maximal probability to reach bad states under all such minimal sets. This NP-complete problem has

been tackled using mixed integer linear programming [34]. This approach is not restricted to PRISM's input language, but it is also applicable to other modeling formalisms for probabilistic automata such as process algebras [35].

Tools and applications. DiPro [36] and Comics [37] are the only publicly available tools supporting counterexample generation for Markov models.[3] DiPro applies directed path search to discrete- and continuous-time Markov models to compute counterexamples for the violation of PCTL or CSL properties. Although the search works on explicit model representations, the relevant model parts are built on the fly, which makes DiPro very efficient and highly scalable. Comics computes hierarchically abstracted and refinable critical subsystems for discrete-time Markov models. Strongly connected components are the basis for the abstraction, whereas methods to compute k shortest paths are applied in different contexts to determine critical subsystems. Probabilistic counterexamples have been used in different applications. Path-based counterexamples have been applied to guide the refinement of too coarse abstractions in CEGAR-approaches for probabilistic programs [38]. Tree-based counterexamples have been used for a similar purpose in the setting of assume-guarantee reasoning on probabilistic automata [39]. Other applications include the identification of failures in FMEA analysis [40] and the safety analysis of an airbag system [41]. Using the notion of causality, [42, 43] have developed techniques to guide the user to the most responsible causes in a counterexample once a DTMC violates a probabilistic CTL formula, whereas [44] synthesizes fault trees from probabilistic counterexamples.

Organization of this paper. This paper surveys the existing techniques for generating and representing counterexamples for discrete-time Markov models. We cover both explicit as well as symbolic techniques, and also treat the recent development of generating counterexamples at the level of model descriptions, rather than for models themselves. The focus is on a tutorial-like presentation with various illustrative examples. For a full-fledged presentation of all technical aspects as well as formal proofs we refer to the literature. Section 2 provides the necessary background on discrete-time Markov models as well as their reachability analysis. Section 3 defines what counterexamples are. Section 4 is devoted to path-based counterexamples and their applications, whereas Section 5 deals with critical subsystems. Section 6 presents the generation of smallest critical command sets in terms of the model description language. A brief description and comparison of the available tools is given in Section 7. Finally, Section 8 concludes the survey.

2 Foundations

In this section we introduce discrete-time Markov models (Section 2.1) along with probabilistic reachability properties for them (Section 2.2). For further reading we refer to, e. g., [9, 14, 15, 45].

[3] DiPro is available from `http://www.inf.uni-konstanz.de/soft/dipro/` and Comics from `http://www-i2.informatik.rwth-aachen.de/i2/comics/`.

2.1 Models

When modeling real systems using formal modeling languages, due to the complexity of the real world, we usually need to abstract away certain details of the real system. For example, Kripke structures specify a set of model states representing the states of the real-world system, and transitions between the model states modeling the execution steps of the real system. However, the model states do not store any specific information about the real system state that they represent (e. g., concrete variable values in a program). To be able to specify and analyze properties that are dependent on information not included in the model, we can define a set of *atomic propositions* and label each model state with the set of those propositions that hold in the given state.

Example 1. Assume a program declaring two Boolean variables b_1 and b_2, both with initial value *false*, and executing $b_1 := \mathit{true}$ and $b_2 := \mathit{true}$ in parallel. We use $S = \{s_0, s_1, s_2, s_3\}$ as model state set with the following encoding:

Model state	Program variable values
s_0	$b_1 = \mathit{false}$ $b_2 = \mathit{false}$
s_1	$b_1 = \mathit{true}$ $b_2 = \mathit{false}$
s_2	$b_1 = \mathit{false}$ $b_2 = \mathit{true}$
s_3	$b_1 = \mathit{true}$ $b_2 = \mathit{true}$

We are interested in the equality of b_1 and b_2. We define an atomic proposition set $\mathrm{AP} = \{a\}$, where a encodes the equality of b_1 and b_2, and a state labeling function $L : \{s_0, s_1, s_2, s_3\} \to 2^{\{a\}}$ mapping the set $\{a\}$ to s_0 and s_3 and the empty set $\emptyset$ to the other two states. ■

In the following we fix a finite set AP of atomic propositions.

In systems that exhibit probabilistic behavior, the outcome of an executed action is determined probabilistically. When modeling such systems, the transitions must specify not only the successors but also the probabilities with which they are chosen, formalized by *probability distributions*.

Definition 1 (Sub-distribution, distribution, support). *A* sub-distribution *over a countable set S is a function $\mu : S \to [0,1]$ such that $\sum_{s \in S} \mu(s) \leq 1$; μ is a* (probability) distribution *if $\sum_{s \in S} \mu(s) = 1$. The set of all sub-distributions over S is denoted by* $\mathrm{SDistr}(S)$*, the set of probability distributions by* $\mathrm{Distr}(S)$*. By* $\mathrm{supp}(\mu) = \{s \in S \mid \mu(s) > 0\}$ *we denote the* support *of a (sub-)distribution μ.*

Example 2. Consider again the program from Example 1 and assume that in the initial state s_0 the statement $b_1 := \mathit{true}$ is executed with probability 0.6 and $b_2 := \mathit{true}$ with probability 0.4. This is reflected by the distribution $\mu_0 : \{s_0, s_1, s_2, s_3\} \to [0,1]$ with $\mu_0(s_1) = 0.6$, $\mu_0(s_2) = 0.4$, and $\mu_0(s_0) = \mu(s_3) = 0$. The support of the distribution is $\mathrm{supp}(\mu_0) = \{s_1, s_2\}$.

After executing $b_1 := \mathit{true}$, the system is in state s_1 and $b_2 := \mathit{true}$ will be executed with probability 1. The corresponding distribution is specified by

$\mu_1(s_3) = 1$ and $\mu_1(s_0) = \mu_1(s_1) = \mu_1(s_2) = 0$. Such a distribution, mapping the whole probability 1 to a single state, is called a *Dirac* distribution.

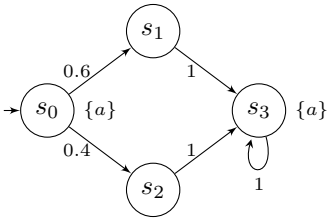
For state s_2, the distribution μ_2 equals μ_1. Finally for s_3, the Dirac distribution μ_3 defines a self-loop on s_3 with probability 1, modeling idling. ■

Discrete-time Markov chains Discrete-time Markov chains are a widely used formalism to model probabilistic behavior in a discrete-time model. State changes are modeled by discrete transitions whose probabilities are specified by (sub-)distributions as follows.

Definition 2 (Discrete-time Markov chain). *A* discrete-time Markov chain (DTMC) *over atomic propositions* AP *is a tuple* $\mathcal{D} = (S, s_{\text{init}}, P, L)$ *with* S *being a countable set of states,* $s_{\text{init}} \in S$ *the initial state,* $P : S \to \text{SDistr}(S)$ *the transition probability function, and* L *a labeling function with* $L : S \to 2^{\text{AP}}$.

We often see the transition probability function $P : S \to (S \to [0, 1])$ rather as being of type $P : (S \times S) \to [0, 1]$ and write $P(s, s')$ instead of $P(s)(s')$.

Example 3. The system from Example 2 can be modeled by the DTMC $\mathcal{D} = (S, s_0, P, L)$, where S and L are as in Example 1 and P assigns μ_i (defined in Example 2) to s_i for each $i \in \{0, \ldots, 3\}$. This DTMC model can be graphically depicted as follows:



■

Please note that in the above definition of DTMCs we generalize the standard definition and allow sub-distributions. Usually, $P(s)$ is required to be a probability distribution for all $s \in S$. We can transform a DTMC $\mathcal{D} = (S, s_{\text{init}}, P, L)$ with sub-distributions into a DTMC $\alpha_{s_\perp}(\mathcal{D}) = (S', s_{\text{init}}, P', L')$ with distributions using the transformation $\alpha_{s_\perp}$ with

- $S' = S \dot{\cup} \{s_\perp\}$ for a fresh sink state $s_\perp \notin S$,
- $P'(s, s') = \begin{cases} P(s, s'), & \text{for } s, s' \in S, \\ 1 - \sum_{s'' \in S} P(s, s''), & \text{for } s \in S \text{ and } s' = s_\perp, \\ 1, & \text{for } s = s' = s_\perp, \\ 0, & \text{otherwise (for } s = s_\perp \text{ and } s' \in S), \end{cases}$ and
- $L'(s) = L(s)$ for $s \in S$ and $L'(s_\perp) = \emptyset$.

According to the DTMC semantics below, the reachability probabilities in $\mathcal{D}$ and $\alpha_{s_\perp}(\mathcal{D})$ are equal for the states from S. The advantage of allowing sub-stochastic distributions is that a *subsystem* of a DTMC, determined by a subset of its states, is again a DTMC.

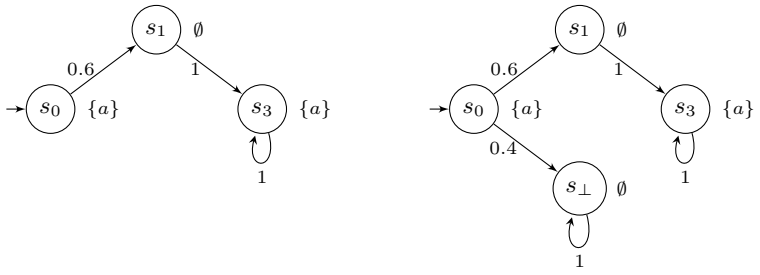


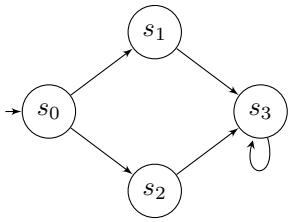
Fig. 1. Completing sub-distributions of a DTMC (cf. Example 4)

Example 4. Consider again the DTMC from Example 3. If we are only interested in the behavior for executing the statement $b_1 := \mathit{true}$ first, then the transition from s_0 to s_2 can be neglected. The DTMC model in this case has a sub-distribution assigned to s_0, as shown in Figure 1 on the left. We can transform this DTMC with a sub-distribution into a reachability-equivalent DTMC with distributions as shown in Figure 1 on the right. ■

Assume in the following a DTMC $\mathcal{D} = (S, s_{\text{init}}, P, L)$. We say that there is a *transition* (s, s') from the *source* $s \in S$ to the *successor* $s' \in S$ iff $s' \in \operatorname{supp}\big(P(s)\big)$. We say that the states in $\operatorname{supp}\big(P(s)\big)$ are the *successors* of s.

We sometimes refer to the *underlying graph* $\mathcal{G}_{\mathcal{D}} = (S, E_{\mathcal{D}})$ of $\mathcal{D}$, with nodes S and edges $E_{\mathcal{D}} = \big\{(s, s') \in S \times S \,\big|\, s' \in \operatorname{supp}\big(P(s)\big)\big\}$.

Example 5. The underlying graph of the DTMC from Example 3 on page 7 can be visualized as follows:



■

A *path* of $\mathcal{D}$ is a finite or infinite sequence $\pi = s_0 s_1 \ldots$ of states $s_i \in S$ such that $s_{i+1} \in \operatorname{supp}\big(P(s_i)\big)$ for all $i \geq 0$. We say that the transitions (s_i, s_{i+1}) are *contained* in the path π, written $(s_i, s_{i+1}) \in \pi$. Starting with $i = 0$, we write $\pi[i]$ for the $(i+1)^{\text{th}}$ state s_i on path π. The *length* $|\pi|$ of a finite path $\pi = s_0 \ldots s_n$ is the number n of its transitions. The last state of π is denoted by $\operatorname{last}(\pi) = s_n$.

By $\operatorname{Paths}^{\mathcal{D}}_{\text{inf}}(s)$ we denote the set of all infinite paths of $\mathcal{D}$ starting in $s \in S$. Similarly, $\operatorname{Paths}^{\mathcal{D}}_{\text{fin}}(s)$ contains all finite paths of $\mathcal{D}$ starting in $s \in S$, and $\operatorname{Paths}^{\mathcal{D}}_{\text{fin}}(s, t)$ those starting in $s \in S$ and ending in $t \in S$. For $T \subseteq S$ we also use the notation $\operatorname{Paths}^{\mathcal{D}}_{\text{fin}}(s, T)$ for $\bigcup_{t \in T} \operatorname{Paths}^{\mathcal{D}}_{\text{fin}}(s, t)$. A state $t \in S$ is *reachable* from another state $s \in S$ iff $\operatorname{Paths}^{\mathcal{D}}_{\text{fin}}(s, t) \neq \emptyset$.

Example 6. The DTMC model $\mathcal{D}$ from Example 3 on page 7 has two infinite paths starting in s_0, specified by $\mathrm{Paths}^{\mathcal{D}}_{\mathrm{inf}}(s_0) = \{s_0 s_1 s_3^{\omega},\, s_0 s_2 s_3^{\omega}\}$. The finite paths starting in s_0 are $\mathrm{Paths}^{\mathcal{D}}_{\mathrm{fin}}(s_0) = \{s_0,\, s_0 s_1,\, s_0 s_1 s_3^{+},\, s_0 s_2,\, s_0 s_2 s_3^{+}\}$. The finite paths starting in s_0 and ending in s_3 are $\mathrm{Paths}^{\mathcal{D}}_{\mathrm{fin}}(s_0, s_3) = \{s_0 s_1 s_3^{+},\, s_0 s_2 s_3^{+}\}$. ■

To be able to talk about the probabilities of certain behaviors (i. e., path sets), we follow the standard way [46] to define for each state $s \in S$ a probability space $(\Omega^{\mathcal{D}}_s, \mathcal{F}^{\mathcal{D}}_s, \Pr^{\mathcal{D}}_s)$ on the infinite paths of the DTMC $\mathcal{D}$ starting in s. The sample space $\Omega^{\mathcal{D}}_s$ is the set $\mathrm{Paths}^{\mathcal{D}}_{\mathrm{inf}}(s)$. The *cylinder set* of a finite path $\pi = s_0 \dots s_n$ of $\mathcal{D}$ is defined as $\mathrm{Cyl}(\pi) = \{\pi' \in \mathrm{Paths}^{\mathcal{D}}_{\mathrm{inf}}(s_0) \mid \pi \text{ is a prefix of } \pi'\}$. The set $\mathcal{F}^{\mathcal{D}}_s$ of events is the unique smallest σ-algebra that contains the cylinder sets of all finite paths in $\mathrm{Paths}^{\mathcal{D}}_{\mathrm{fin}}(s)$ and is closed under complement and countable union. The unique probability measure $\Pr^{\mathcal{D}}_s$ (or short Pr) on $\mathcal{F}^{\mathcal{D}}_s$ specifies the probabilities of the events recursively, for cylinder sets by

$$\Pr\big(\mathrm{Cyl}(s_0 \dots s_n)\big) = \prod_{i=0}^{n-1} P(s_i, s_{i+1})\,,$$

for the complement $\bar{\Pi}$ of a set $\Pi \in \mathcal{F}^{\mathcal{D}}_s$ by $\Pr^{\mathcal{D}}_s(\bar{\Pi}) = 1 - \Pr^{\mathcal{D}}_s(\Pi)$, and for the countable union $\Pi = \bigcup_{i=1}^{\infty} \Pi_i$ of pairwise disjoint sets $\Pi_i \in \mathcal{F}^{\mathcal{D}}_s$, $i \in \mathbb{N}$, by $\Pr^{\mathcal{D}}_s(\Pi) = \sum_{i=1}^{\infty} \Pr^{\mathcal{D}}_s(\Pi_i)$.

For finite paths π we set $\Pr_{\mathrm{fin}}(\pi) = \Pr\big(\mathrm{Cyl}(\pi)\big)$. For sets of finite paths $R \subseteq \mathrm{Paths}^{\mathcal{D}}_{\mathrm{fin}}(s)$ we define $\Pr_{\mathrm{fin}}(R) = \sum_{\pi \in R'} \Pr_{\mathrm{fin}}(\pi)$ with $R' = \{\pi \in R \mid \forall \pi' \in R.\ \pi' \text{ is not a proper prefix of } \pi\}$.

Example 7. Consider again the DTMC from Example 3 on page 7. For the initial state s_0, the probability space $(\Omega^{\mathcal{D}}_{s_0}, \mathcal{F}^{\mathcal{D}}_{s_0}, \Pr^{\mathcal{D}}_{s_0})$ is given by the following components:

- The sample space is $\Omega^{\mathcal{D}}_{s_0} = \mathrm{Paths}^{\mathcal{D}}_{\mathrm{inf}}(s_0) = \{s_0 s_1 s_3^{\omega},\, s_0 s_2 s_3^{\omega}\}$.
- The event set $\mathcal{F}^{\mathcal{D}}_{s_0}$ contains the cylinder sets of all finite paths starting in s_0 and the empty set, i. e.,

$$\begin{aligned}\mathcal{F}^{\mathcal{D}}_{s_0} = \{\ & \emptyset,\\ & \mathrm{Cyl}(s_0) = \mathrm{Paths}^{\mathcal{D}}_{\mathrm{inf}}(s_0) = \{s_0 s_1 s_3^{\omega},\, s_0 s_2 s_3^{\omega}\},\\ & \mathrm{Cyl}(s_0 s_1) = \mathrm{Cyl}(s_0 s_1 s_3^{+}) = \{s_0 s_1 s_3^{\omega}\},\\ & \mathrm{Cyl}(s_0 s_2) = \mathrm{Cyl}(s_0 s_2 s_3^{+}) = \{s_0 s_2 s_3^{\omega}\} \qquad \}\,.\end{aligned}$$

  The empty set is added as the complement of $\mathrm{Cyl}(s_0)$. The other cylinder set complements and all countable unions over these elements are cylinder sets themselves and therefore already included.
- The probability measure $\Pr^{\mathcal{D}}_{s_0}$ is defined by

$$\begin{aligned}&\Pr^{\mathcal{D}}_{s_0}(\emptyset) = 0, && \Pr^{\mathcal{D}}_{s_0}(\mathrm{Cyl}(s_0)) = 1,\\ &\Pr^{\mathcal{D}}_{s_0}(\mathrm{Cyl}(s_0 s_1)) = 0.6, && \Pr^{\mathcal{D}}_{s_0}(\mathrm{Cyl}(s_0 s_2)) = 0.4.\end{aligned}$$

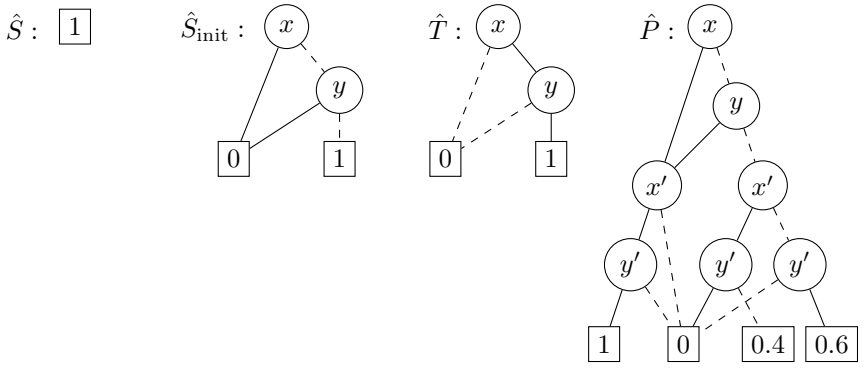
■

Besides using explicit model representations enumerating states and transitions, a DTMC can be represented *symbolically* using (ordered) binary decision diagrams (BDDs) and multi-terminal BDDs (MTBDDs). For an introduction to (MT)BDDs we refer to, e. g., [9]. In a symbolic representation, states are encoded using a set of Boolean variables such that each state is uniquely represented by an assignment to the Boolean variables. State sets, like the state space, the initial state or a set of states having a certain label of interest, are represented by some BDDs such that the variable evaluations along the paths leading to the leaf with label 1 encode those states that belong to the given set. Additionally, an MTBDD $\hat{P}$ stores the transition probabilities. This MTBDD uses two copies of the Boolean variables, one to encode the source states and one to encode the successor states of transitions. The evaluation along a path encodes the source and successor states, where the value of the leaf to which a path leads specifies the transition probability. Operations on (MT)BDDs can be used to compute, e. g., the successor set of a set of states or the probabilities to reach a certain set of states in a given number of steps.

Example 8. The four system states of the DTMC $\mathcal{D}$ from Example 3 on page 7 can be encoded by two Boolean variables x and y:

	s_0	s_1	s_2	s_3
x	0	0	1	1
y	0	1	0	1

The symbolic representation of $\mathcal{D}$ together with the state set $T = \{s_3\}$ of special interest would involve the following (MT)BDDs:



Though for this toy example the explicit representation seems to be more convenient, for large models the symbolic representation can be smaller by orders of magnitude. ■

Markov decision processes and probabilistic automata DTMCs behave deterministically, i. e., the choice of the next transition to be taken is purely probabilistic. Enriching DTMCs by nondeterminism leads to *Markov decision processes* and *probabilistic automata*.

Definition 3 (Probabilistic automaton [47]). *A* probabilistic automaton *(PA) is a tuple* $\mathcal{M} = (S, s_{\text{init}}, \text{Act}, \hat{P}, L)$ *where* S *is a finite set of states,* $s_{\text{init}} \in S$ *is the initial state,* Act *is a finite set of actions,* $\hat{P} : S \to (2^{\text{Act} \times \text{SDistr}(S)} \setminus \emptyset)$ *is a probabilistic transition relation such that* $\hat{P}(s)$ *is finite for all* $s \in S$*, and* $L : S \to 2^{\text{AP}}$ *is a labeling function.*

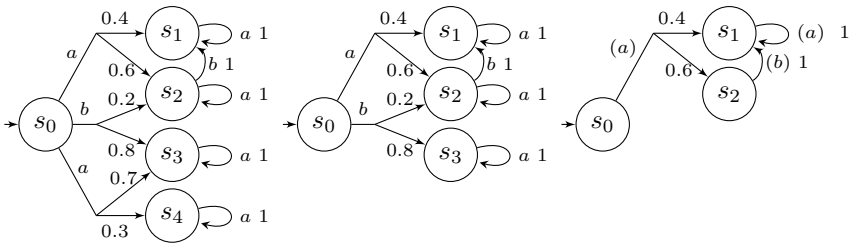
$\mathcal{M}$ *is a* Markov decision process *(MDP) if for all* $s \in S$ *and all* $\alpha \in \text{Act}$ $\big|\{\mu \in \text{SDistr}(S) \,|\, (\alpha, \mu) \in \hat{P}(s)\}\big| \leq 1$ *holds.*

Intuitively, the evolution of a probabilistic automaton is as follows. Starting in the initial state s_{init}, a pair $(\alpha, \mu) \in \hat{P}(s_{\text{init}})$ is chosen nondeterministically. Then, the successor state $s' \in S$ is determined probabilistically according to the distribution μ. A *deadlock* occurs in state s_{init} with probability $1 - \sum_{s' \in S} \mu(s')$. Repeating this process in s' yields the next state and so on.

The actions $\text{Act}_s = \big\{\alpha \in \text{Act} \mid \exists \mu \in \text{SDistr}(S).\, (\alpha, \mu) \in \hat{P}(s)\big\}$ are said to be *enabled* at state $s \in S$.

Note that DTMCs constitute a subclass of MDPs (apart from the fact that the actions are not relevant for DTMC and are therefore typically omitted) and MDPs build a subclass of PAs.

Example 9. To illustrate the difference between the different model classes, consider the following probabilistic models:



The involved distributions are

$$\mu_1(s) = \begin{cases} 0.4, & \text{if } s{=}s_1, \\ 0.6, & \text{if } s{=}s_2, \\ 0, & \text{else,} \end{cases} \quad \mu_2(s) = \begin{cases} 0.2, & \text{if } s{=}s_2, \\ 0.8, & \text{if } s{=}s_3, \\ 0, & \text{else,} \end{cases} \quad \mu_3(s) = \begin{cases} 0.7, & \text{if } s{=}s_3, \\ 0.3, & \text{if } s{=}s_4, \\ 0, & \text{else} \end{cases}$$

and the Dirac distributions d_i, $i = 1, 2, 3, 4$, assigning probability 1 to s_i and 0 to all other states.

The model on the left is a PA. In state s_0 there are two enabled actions a and b, where a appears in combination with two different distributions. Therefore, this model is not an MDP.

In contrast, the model in the middle is an MDP, since in each state and for each enabled action there is a single distribution available.

The model on the right is a DTMC, because a single distribution is mapped to each state. ■

An *infinite path* in a PA $\mathcal{M}$ is an infinite sequence $\pi = s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1)\ldots$ such that $(\alpha_i, \mu_i) \in \hat{P}(s_i)$ and $s_{i+1} \in \mathrm{supp}(\mu_i)$ for all $i \geq 0$. A *finite path* in $\mathcal{M}$ is a finite prefix $\pi = s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1)\ldots s_n$ of an infinite path in $\mathcal{M}$ with last state by $\mathrm{last}(\pi) = s_n$. Let $\pi[i]$ denote the $(i+1)^{\text{th}}$ state s_i on path π. The sets of all infinite and finite paths in $\mathcal{M}$ starting in $s \in S$ are denoted by $\mathrm{Paths}^{\mathcal{M}}_{\mathrm{inf}}(s)$ and $\mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s)$, respectively, whereas $\mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s,t)$ is the set of all finite paths starting in s and ending in t. For $T \subseteq S$ we also use the notation $\mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s,T)$ for $\bigcup_{t\in T} \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s,t)$.

Example 10. The sequence $s_0\,(a,\mu_1)\,s_1\,((a,d_1)\,s_1)^\omega$ is an infinite path in all three models from Example 9 on page 11. (To be precise, the path of the DTMC does not contain the action-distribution pairs.) ■

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

Definition 4 (Scheduler, deterministic, memoryless).

- *A* scheduler *for a PA* $\mathcal{M} = (S, s_{\mathrm{init}}, \mathrm{Act}, \hat{P}, L)$ *is a function*

$$\sigma\colon \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s_{\mathrm{init}}) \to \mathrm{Distr}(\mathrm{Act} \times \mathrm{SDistr}(S))$$

  such that $\mathrm{supp}(\sigma(\pi)) \subseteq \hat{P}\big(\mathrm{last}(\pi)\big)$ *for each* $\pi \in \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s_{\mathrm{init}})$. *The set of all schedulers for* $\mathcal{M}$ *is denoted by* $\mathrm{Sched}_{\mathcal{M}}$.
- *A scheduler* σ *for* $\mathcal{M}$ *is* memoryless *iff for all* $\pi, \pi' \in \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s_{\mathrm{init}})$ *with* $\mathrm{last}(\pi) = \mathrm{last}(\pi')$ *we have that* $\sigma(\pi) = \sigma(\pi')$.
- *A scheduler* σ *for* $\mathcal{M}$ *is* deterministic *iff for all* $\pi \in \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s_{\mathrm{init}})$ *and* $(\alpha,\mu) \in \mathrm{Act} \times \mathrm{SDistr}(S)$ *we have that* $\sigma(\pi)((\alpha,\mu)) \in \{0,1\}$.

Schedulers are also called *policies* or *adversaries*. Intuitively, a scheduler resolves the nondeterminism in a PA by assigning probabilities to the nondeterministic choices available in the last state of a finite path. It therefore reduces the nondeterministic model to a fully probabilistic one.

Example 11. Consider the PA depicted on the left-hand-side in Example 9 on page 11. We define a scheduler σ_0 by specifying for all $\pi \in \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s_{\mathrm{init}})$ and for all $(\alpha,\mu) \in \hat{P}\big(\mathrm{last}(\pi)\big)$

$$\sigma_0(\pi)(\alpha,\mu) = \begin{cases} 0.25, & \text{if } \mathrm{last}(\pi) = s_0 \text{ and } \alpha = a, \\ 0.5, & \text{if } \mathrm{last}(\pi) = s_0 \text{ and } \alpha = b, \\ 1, & \text{if } \mathrm{last}(\pi) \in \{s_1, s_3, s_4\}, \\ 0.9, & \text{if } \pi = \pi'\,(\alpha',\mu')\,s_2 \text{ and } \alpha' = \alpha, \\ 0.1, & \text{if } \pi = \pi'\,(\alpha',\mu')\,s_2 \text{ and } \alpha' \neq \alpha, \end{cases}$$

and $\sigma_0(\pi)(\alpha,\mu) = 0$ for all $\pi \in \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s_{\mathrm{init}})$ and $(\alpha,\mu) \in (\mathrm{Act} \times \mathrm{SDistr}(S)) \setminus \hat{P}(\mathrm{last}(\pi))$. The above scheduler σ_0 is not memoryless, since the schedule for paths with last state s_2 depends on the last action on the path. This scheduler is

also not deterministic, since it assigns also probabilities different from 0 and 1 to action-distribution pairs.

Let scheduler σ_1 be defined for all $\pi \in \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s_{\mathrm{init}})$ and for all $(\alpha, \mu) \in \hat{P}\big(\mathrm{last}(\pi)\big)$ by

$$\sigma_1(\pi)(\alpha,\mu) = \begin{cases} 0.25, & \text{if } \mathrm{last}(\pi) = s_0 \text{ and } \alpha = a, \\ 0.5, & \text{if } \mathrm{last}(\pi) = s_0 \text{ and } \alpha = b, \\ 1, & \text{if } \mathrm{last}(\pi) \in \{s_1, s_2, s_3, s_4\} \text{ and } \alpha = a, \\ 0, & \text{else (if } \mathrm{last}(\pi) = s_2 \text{ and } \alpha = b), \end{cases}$$

and $\sigma_1(\pi)(\alpha,\mu) = 0$ for all $\pi \in \mathrm{Paths}^{\mathcal{M}}_{\mathrm{fin}}(s_{\mathrm{init}})$ and $(\alpha,\mu) \in (\mathrm{Act} \times \mathrm{SDistr}(S)) \setminus \hat{P}(\mathrm{last}(\pi))$. The scheduler σ_1 is memoryless but not deterministic.

Finally, the following scheduler σ_2 is deterministic and memoryless:

$$\sigma_2(\pi)(\alpha,\mu) = \begin{cases} 1, & \text{if } \mathrm{last}(\pi) = s_0 \text{ and } (\alpha,\mu) = (a,\mu_1), \\ 1, & \text{if } \mathrm{last}(\pi) = s_2 \text{ and } (\alpha,\mu) = (b,d_1), \\ 1, & \text{if } \mathrm{last}(\pi) = s_i \text{ and } (\alpha,\mu) = (a,d_i) \text{ for } i \in \{1,3,4\}, \\ 0, & \text{else}. \end{cases}$$

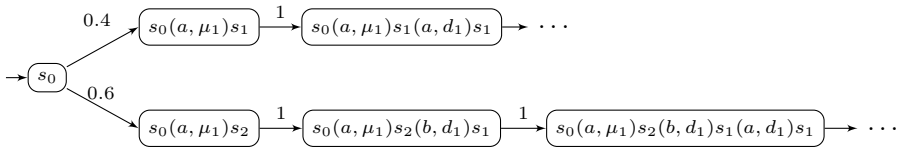
■

Definition 5 (Induced DTMC). *Let $\mathcal{M} = (S, s_{\mathrm{init}}, \mathrm{Act}, \hat{P}, L)$ be a PA and σ a scheduler for $\mathcal{M}$. We define the DTMC $\mathcal{M}^\sigma = (\mathrm{Paths}_{\mathrm{fin}}(s_{\mathrm{init}}), s_{\mathrm{init}}, P', L')$ with*

$$P'(\pi,\pi') = \begin{cases} \sigma(\pi)((\alpha,\mu)) \cdot \mu(s), & \text{if } \pi' = \pi\,(\alpha,\mu)\,s, \\ 0, & \text{otherwise}, \end{cases}$$

and $L'(\pi) = L(\mathrm{last}(\pi))$ for all $\pi, \pi' \in \mathrm{Paths}_{\mathrm{fin}}(s_{\mathrm{init}})$. We call $\mathcal{M}^\sigma$ the DTMC induced by $\mathcal{M}$ and σ.

Example 12. The scheduler σ_2 from Example 11 (on page 12) for the PA depicted on the left in Example 9 (on page 11) induces the following DTMC:



Since the scheduler σ_2 is memoryless, each pair of states π and π' with $\mathrm{last}(\pi) = \mathrm{last}(\pi')$ are equivalent (*bisimilar*) in the sense that the set of all label sequences (*traces*) along paths starting in those states are equal. (Note that the labeling is not depicted in the above picture.) Since the logics we consider can argue about the labelings only, such state pairs satisfy the same formulas. We say that the *observable behavior* of our models is given by the their trace sets.

Based on this observation, we can build an abstraction of the above induced DTMC by introducing abstract states $s \in S$ (the states of the inducing PA)

representing all states π with $\text{last}(\pi) = s$ of the induced DTMC. For the above example, the scheduler is not only memoryless but also deterministic. For those schedulers this abstraction defines a DTMC containing the states of the PA and all distributions selected by the scheduler. For σ_2 the result is the DTMC depicted on the right in Example 9 on page 11.

In the following, when talking about the DTMC induced by a PA and a *memoryless deterministic* scheduler, we mean this abstraction. ■

For the probability measure on paths of a PA $\mathcal{M}$ under a scheduler σ for $\mathcal{M}$, we use the standard probability measure on paths of the induced DTMC $\mathcal{M}^\sigma$, as described previously. We denote this probability measure by $\Pr^{\mathcal{M},\sigma}_{s_{\text{init}}}$ (or, briefly, $\Pr^{\mathcal{M},\sigma}$).

2.2 Reachability Properties

As specification for both DTMCs and PAs we consider so-called *reachability properties*. We are interested in a quantitative analysis such as:

"*What is the probability to reach a certain set of states T starting in state s?*"

Such a set of *target* states T might, e. g., model *bad* or *safety-critical* states, for which the probability to visit them should be kept below a certain upper bound. Formally, we identify target states by labeling them with some dedicated label $\texttt{target} \in \text{AP}$ such that $T = \left\{ s \in S \,\middle|\, \texttt{target} \in L(s) \right\}$. Instead of depicting target labels, in the following we illustrate target states in figures as double-framed gray-colored nodes.

We formulate reachability properties like $\mathbb{P}_{\bowtie\lambda}(\Diamond\texttt{target})$ for $\bowtie \in \{<, \leq, \geq, >\}$ and $\lambda \in [0,1] \cap \mathbb{Q}$. For simplicity, we will sometimes also write $\mathbb{P}_{\bowtie\lambda}(\Diamond T)$. Such a property holds in a state s of a DTMC $\mathcal{D}$ iff the probability to reach a state from T when starting in s in $\mathcal{D}$ satisfies the bound $\bowtie \lambda$. The DTMC satisfies the property iff it holds in its initial state. For a PA $\mathcal{M}$ we require the bound to be satisfied under all schedulers.

Example 13. For instance, $\mathbb{P}_{\leq 0.1}(\Diamond\texttt{target})$ states that the probability of reaching a state labeled with `target` is less or equal than 0.1, either for a DTMC or under all schedulers for a PA. If the probability is larger in a state, this property evaluates to `false` for this state. ■

In this paper we deal with reachability properties only. Deciding some other logics like, e. g., probabilistic computation tree logic (PCTL) or ω-regular properties can be reduced to the computation of reachability properties.

Furthermore, in the following we restrict ourselves to reachability properties of the form $\mathbb{P}_{\leq\lambda}(\Diamond\texttt{target})$. Formulas of the form $\mathbb{P}_{<\lambda}(\Diamond\texttt{target})$ can be handled similarly. The cases $\geq$ and $>$ can be reduced to $<$ and $\leq$, respectively, using negation, e. g., $\mathbb{P}_{>\lambda}(\Diamond\texttt{target})$ is equivalent to $\mathbb{P}_{\leq 1-\lambda}(\Diamond\neg\texttt{target})$.

At some places we will also mention *bounded* reachability properties of the form $\mathbb{P}_{\leq\lambda}(\Diamond^{\leq h} T)$ for a natural number h. The semantics of such formulas is similar to the unbounded case $\mathbb{P}_{\leq\lambda}(\Diamond T)$, however, here the probability to reach a state in T via paths of length at most h should satisfy the bound.

Reachability for DTMCs Assume a DTMC $\mathcal{D} = (S, s_{\text{init}}, P, L)$, a label $\texttt{target} \in \text{AP}$ and a target state set $T = \{t \in S \mid \texttt{target} \in L(t)\}$. We want to determine whether $\mathcal{D}$ satisfies the property $\mathbb{P}_{\leq\lambda}(\Diamond T)$, written $\mathcal{D} \models \mathbb{P}_{\leq\lambda}(\Diamond T)$. This is the case iff the property holds in the initial state of $\mathcal{D}$, denoted by $\mathcal{D}, s_{\text{init}} \models \mathbb{P}_{\leq\lambda}(\Diamond T)$.

Let $s \in S \setminus T$. The set of paths contributing to the probability of reaching T from s is given by

$$\Diamond T(s) = \{\pi \in \text{Paths}^{\mathcal{D}}_{\text{inf}}(s) \mid \exists i.\, \texttt{target} \in L(\pi[i])\}$$

where we overload $\Diamond T$ to both denote a set of paths and a property, and also write simply $\Diamond T$ if s is clear from the context.

The above set $\Diamond T(s)$ equals the union of the cylinder sets of all paths from $\text{Paths}^{\mathcal{D}}_{\text{fin}}(s, T)$:

$$\Diamond T(s) = \bigcup_{\pi \in \text{Paths}^{\mathcal{D}}_{\text{fin}}(s,T)} \text{Cyl}(\pi)\,.$$

Note that $\text{Paths}^{\mathcal{D}}_{\text{fin}}(s, T)$ contains in general also prefixes of other contained paths (if there are paths of length at least 1 from T to T). When computing the probability mass of $\Diamond T(s)$, such extensions are not considered. We can remove those extensions by restricting the finite paths to visit T only in their last state: $\Diamond T(s) = \bigcup_{\pi \in \Diamond T_{\text{fin}}(s)} \text{Cyl}(\pi)$ with

$$\Diamond T_{\text{fin}}(s) = \{\pi \in \text{Paths}^{\mathcal{D}}_{\text{fin}}(s, T) \mid \forall 0 \leq i < |\pi|.\, \pi[i] \notin T\}\,.$$

As no path in the set $\Diamond T_{\text{fin}}(s)$ is a prefix of another one, the probability of this set can be computed by the sum of the probabilities of its elements:

$$\begin{aligned}
\Pr^{\mathcal{D}}_s\big(\Diamond T(s)\big) &= \Pr^{\mathcal{D}}_s\Big(\bigcup_{\pi \in \Diamond T_{\text{fin}}(s)} \text{Cyl}(\pi)\Big) \\
&= \sum_{\pi \in \Diamond T_{\text{fin}}(s)} \Pr^{\mathcal{D}}_s\big(\text{Cyl}(\pi)\big) \\
&= \sum_{s' \in S \setminus T} P(s, s') \cdot \Pr^{\mathcal{D}}_{s'}\big(\Diamond T(s')\big) \;+\; \sum_{s' \in T} P(s, s')\,.
\end{aligned}$$

Therefore, we can compute for each state $s \in S$ the probability of reaching T from s by solving the equation system consisting of a constraint

$$p_s = \begin{cases} 1, & \text{if } s \in T, \\ 0, & \text{if } T \text{ is not reachable from } s, \\ \sum_{s' \in S} P(s, s') \cdot p_{s'}, & \text{otherwise} \end{cases}$$

for each $s \in S$. The unique solution $\nu : \{p_s \mid s \in S\} \to [0, 1]$ of this linear equation system assigns to p_s the probability of reaching T from s for each state $s \in S$. That means, $\mathcal{D} \models \mathbb{P}_{\leq\lambda}(\Diamond T)$ iff $\nu(p_{s_{\text{init}}}) \leq \lambda$.

We can simplify the above equation system if we first remove all states from the model from which T is not reachable.

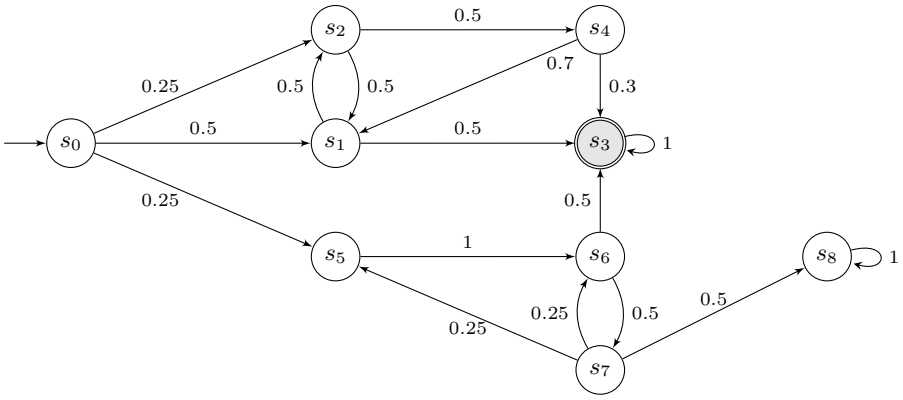


Fig. 2. An example DTMC (cf. Example 14)

Definition 6 (Relevant states of DTMCs). *Let*

$$S_{\mathcal{D}}^{\mathrm{rel}(T)} = \{s \in S \mid \mathrm{Paths}_{\mathrm{fin}}^{\mathcal{D}}(s, T) \neq \emptyset\}$$

and call its elements relevant *for T (or for* `target`*). States $s \notin S_{\mathcal{D}}^{\mathrm{rel}(T)}$ are called* irrelevant *for T (or for* `target`*).*

The set of relevant states can be computed in linear time by a backward reachability analysis on $\mathcal{D}$ [9, Algorithm 46].

If a model does not contain any irrelevant states, the above equation system reduces to the constraints

$$p_s = \begin{cases} 1, & \text{if } s \in T, \\ \sum_{s' \in S} P(s, s') \cdot p_{s'}, & \text{otherwise} \end{cases}$$

for each $s \in S$.

Example 14. Consider the DTMC illustrated in Figure 2 with target state set $T = \{s_3\}$. State s_8 is irrelevant for T and can be removed. The probabilities to reach s_3 can be computed by solving the following equation system:

$$\begin{aligned}
p_{s_0} &= 0.5 \cdot p_{s_1} + 0.25 \cdot p_{s_2} + 0.25 \cdot p_{s_5} & p_{s_1} &= 0.5 \cdot p_{s_2} + 0.5 \cdot p_{s_3} \\
p_{s_2} &= 0.5 \cdot p_{s_1} + 0.5 \cdot p_{s_4} & p_{s_3} &= 1 \\
p_{s_4} &= 0.7 \cdot p_{s_1} + 0.3 \cdot p_{s_3} & p_{s_5} &= 1 \cdot p_{s_6} \\
p_{s_6} &= 0.5 \cdot p_{s_3} + 0.5 \cdot p_{s_7} & p_{s_7} &= 0.25 \cdot p_{s_5} + 0.25 \cdot p_{s_6}
\end{aligned}$$

The unique solution ν defines $\nu(p_{s_0}) = 11/12$, $\nu(p_{s_1}) = \nu(p_{s_2}) = \nu(p_{s_3}) = \nu(p_{s_4}) = 1$, $\nu(p_{s_5}) = \nu(p_{s_6}) = 2/3$ and $\nu(p_{s_7}) = 1/3$. ■

Reachability for PAs Assume a PA $\mathcal{M} = (S, s_{\text{init}}, \text{Act}, \hat{P}, L)$, a label $\texttt{target} \in \text{AP}$ and a target state set $T = \big\{t \in S \,\big|\, \texttt{target} \in L(t)\big\}$. Intuitively, a reachability property holds for $\mathcal{M}$ if it holds under all possible schedulers. Formally, $\mathcal{M} \models \mathbb{P}_{\leq\lambda}(\Diamond T)$ if for all schedulers σ of $\mathcal{M}$ we have that $\mathcal{M}^\sigma \models \mathbb{P}_{\leq\lambda}(\Diamond T)$.

It can be shown that there always exists a *memoryless deterministic* scheduler that maximizes the reachability probability for $\Diamond T$ among all schedulers. Therefore, to check whether $\mathcal{M}^\sigma \models \mathbb{P}_{\leq\lambda}(\Diamond T)$ holds for all schedulers σ, it suffices to consider a memoryless deterministic scheduler σ^* which maximizes the reachability probability for $\Diamond T$ under all memoryless deterministic schedulers, and check the property for the induced DTMC $\mathcal{M}^{\sigma^*}$. For the computation of σ^* we need the notion of *relevant* states.

Definition 7 (Relevant states of PAs). *We define*

$$S_{\mathcal{M}}^{\text{rel}(T)} = \{s \in S \,|\, \exists \sigma \in \text{Sched}_{\mathcal{M}}.\, s \in S_{\mathcal{M}^\sigma}^{\text{rel}(T)}\}$$

and call its elements relevant *for T (or for* `target`*). States $s \notin S_{\mathcal{M}}^{\text{rel}(T)}$ are called* irrelevant *for T (or for* `target`*).*

Again, the set of relevant states can be computed in linear time by a backward reachability analysis on $\mathcal{M}$ [9, Algorithm 46].

The maximal probabilities $p_s = \Pr_s^{\mathcal{M}^{\sigma^*}}\big(\Diamond T(s)\big)$, $s \in S$, can be characterized by the following equation system:

$$p_s = \begin{cases} 1, & \text{if } s \in T, \\ 0, & \text{if } s \notin S_{\mathcal{M}}^{\text{rel}(T)}, \\ \max\Big\{\sum_{s' \in S} \mu(s,s') \cdot p_{s'} \,\Big|\, (\alpha,\mu) \in \hat{P}(s)\Big\}, & \text{otherwise} \end{cases}$$

for each $s \in S$. This equation system can be transformed into a linear optimization problem that yields the maximal reachability probability together with an optimal scheduler [9, Theorem 10.105].

Example 15. Consider the left-hand-side PA model from Example 9 on page 11. The probability to reach s_1 from s_0 is maximized by the deterministic memoryless scheduler σ_2 choosing (a, μ_1) in state s_0, (b, d_1) in state s_2, and (a, d_i) in all other states $s_i \in \{s_1, s_3, s_4\}$. ■

3 Counterexamples

When a DTMC $\mathcal{D}$ violates a reachability property $\mathbb{P}_{\leq\lambda}(\Diamond T)$ for some $T \subseteq S$ and $\lambda \in [0,1] \cap \mathbb{Q}$, an explanation for this violation can be given by a set of paths, each of them leading from the initial state to some target states, such that the probability mass of the path set is larger than λ. Such path sets are called *counterexamples*. For a PA $\mathcal{M}$, a counterexample specifies a deterministic memoryless scheduler σ and a counterexample for the induced DTMC $\mathcal{M}^\sigma$.

Counterexamples are valuable for different purposes, e. g., for the correction of systems or for counterexample-guided abstraction refinement. However, counterexamples may contain a very large or even infinite number of paths (note that for a DTMC $\mathcal{D}$ the whole set $\mathrm{Paths}^{\mathcal{D}}_{\mathrm{fin}}(s_{\mathrm{init}}, T)$ is the largest counterexample). Therefore, it can increase the practical usefulness if we aim at the computation of counterexamples satisfying certain properties. Some important aspects are:

- The *size of the counterexample*, i. e., the number of paths in it.
- The *probability mass* of the counterexample.
- The *computational costs*, i. e., the time and memory required to obtain a counterexample.
- Counterexamples can be given using *representations* at different language levels.
  - At the level of paths, besides path enumeration, a counterexample can be represented by, e. g., computation trees or regular expressions. Path-based representations will be discussed in Section 4.
  - At the model level, a part of the model can represent a counterexample by all paths leading inside the given model part from s_{init} to T. Such representations are the content of Section 5.
  - At a higher level, a fragment of a probabilistic program, for which a PA or a DTMC was generated as its semantics, can also represent a counterexample. We discuss such counterexamples in Section 6.

  Important in our considerations will be the *size of the representation*.

We first formalize counterexamples and measures regarding the first two points, and will discuss representation issues and computational costs in the following sections.

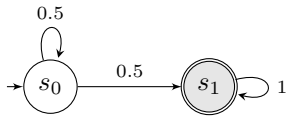
Definition 8 (DTMC evidence and counterexample, [17]). *Assume a DTMC $\mathcal{D} = (S, s_{\mathrm{init}}, P, L)$ violating a reachability property $\mathbb{P}_{\leq\lambda}(\Diamond T)$ with $T \subseteq S$ and $\lambda \in [0,1] \cap \mathbb{Q}$.*

An evidence *(for $\mathcal{D}$ and $\mathbb{P}_{\leq\lambda}(\Diamond T)$) is a finite path $\pi \in \mathrm{Paths}^{\mathcal{D}}_{\mathrm{fin}}(s_{\mathrm{init}}, T)$. A* counterexample *is a set C of evidences such that $\Pr^{\mathcal{D}}_{s_{\mathrm{init}}}(C) > \lambda$. A counterexample C is* minimal *if $|C| \leq |C'|$ for all counterexamples C'. It is a* smallest *counterexample if it is minimal and $\Pr^{\mathcal{D}}_{s_{\mathrm{init}}}(C) \geq \Pr^{\mathcal{D}}_{s_{\mathrm{init}}}(C')$ for all minimal counterexamples C'.*

Example 16. Consider the DTMC from Example 3 on page 7 and the reachability property $\mathbb{P}_{\leq 0.3}\big(\Diamond\{s_3\}\big)$. The path sets $\Pi_1 = \{s_0s_1s_3,\ s_0s_1s_3s_3\}$, $\Pi_2 = \{s_0s_1s_3,\ s_0s_2s_3\}$, $\Pi_3 = \{s_0s_1s_3\}$, and $\Pi_4 = \{s_0s_2s_3\}$ are all counterexamples (with probability mass 0.6, 1, 0.6, and 0.4, respectively). Only Π_3 and Π_4 are minimal, where only Π_3 is a smallest counterexample. ■

For reachability properties of the form $\mathbb{P}_{\leq\lambda}(\Diamond\texttt{target})$ with a non-strict upper bound on the admissible reachability property, a finite counterexample always exists, if the property is violated. For strict upper bounds $\mathbb{P}_{<\lambda}(\Diamond\texttt{target})$, however, an infinite number of paths can be required if the actual reachability probability equals λ [17].

Example 17. Consider the following DTMC:



The probability to reach s_1 is 1, i. e., the property $\mathbb{P}_{<1}\big(\Diamond\{s_1\}\big)$ is violated. However, a counterexample must contain all the infinite number of paths $s_0 s_1$, $s_0 s_0 s_1$, $s_0 s_0 s_0 s_1$ etc. ■

Even if the counterexample is finite, the number of required paths can be very large. Han *et al.* [17] determine for the case study of a probabilistic synchronous leader election protocol that the number of evidences is double exponential in the system parameters.

Definition 9 (PA counterexample). *Assume a PA* $\mathcal{M} = (S, s_{\text{init}}, \text{Act}, \hat{P}, L)$ *violating a reachability property* $\mathbb{P}_{\leq\lambda}(\Diamond T)$ *with* $T \subseteq S$ *and* $\lambda \in [0,1] \cap \mathbb{Q}$.

A counterexample *(for* $\mathcal{M}$ *and* $\mathbb{P}_{\leq\lambda}(\Diamond T)$*) is a pair* (σ, C) *such that* σ *is a scheduler for* $\mathcal{M}$ *and* C *is a counterexample for* $\mathcal{M}^\sigma$*. A counterexample* (σ, C) *is* minimal *if* $|C| \leq |C'|$ *for all counterexamples* (σ', C')*. It is a* smallest *counterexample if it is minimal and* $\Pr^{\mathcal{M}}_{s_{\text{init}}}(C) \geq \Pr^{\mathcal{M}}_{s_{\text{init}}}(C')$ *for all minimal counterexamples* (σ', C').

Example 18. Consider the left-hand-side PA model from Example 9 on page 11 and the reachability property $\mathbb{P}_{\leq 0.9}\big(\Diamond\{s_1\}\big)$. A smallest counterexample is $(\sigma_2, \{s_0 s_1,\ s_0 s_2 s_1\})$ with σ_2 as defined in Example 11 on page 12. ■

4 Path-Based Counterexamples

After having introduced discrete-time probabilistic models and counterexamples for reachability properties, in the following we discuss how we can *compute* such counterexamples for the different model classes in different representations. We start with methods that are based on the search for paths at the state-space level.

4.1 Path-Based Counterexamples for DTMCs

Smallest counterexamples For DTMCs, Han, Katoen and Damman show in [17] how the computation of a smallest counterexample can be reduced to the computation of k *shortest paths* in a directed weighted graph for a suitable $k \in \mathbb{N}$.

We need in the following the property that the DTMC $\mathcal{D} = (S, s_{\text{init}}, P, L)$ we consider has a *single absorbing target state*. If it is the case, we define $\mathcal{D}' = \mathcal{D}$. Otherwise, the DTMC $\mathcal{D}$ is first transformed by adding a new, absorbing target state $t \notin S$ and redirecting all transitions starting in former target states to lead

to the new one. This transformation yields the DTMC $\mathcal{D}' = (S', s_{\text{init}}, P', L')$ with $S' = S \dot\cup \{t\}$ and

$$P'(s,s') = \begin{cases} P(s,s'), & \text{if } s \in S \setminus T \text{ and } s' \in S, \\ 1, & \text{if } s \in T \text{ and } s' = t, \\ 1, & \text{if } s = s' = t, \\ 0, & \text{otherwise,} \end{cases} \qquad L'(s) = \begin{cases} \{\texttt{target}\}, & \text{if } s = t, \\ \emptyset, & \text{otherwise.} \end{cases}$$

Note that the probability to reach t from $s \in S$ in $\mathcal{D}'$ equals the probability to reach T from s in $\mathcal{D}$.

As the next step, a *directed weighted graph* $G_{\mathcal{D}} = (V, E, w)$ with nodes V, edges E and edge weights $w : E \to \mathbb{R}^{\geq 0}$ is obtained from $\mathcal{D}'$ as follows: $V = S'$, $(s,s') \in E$ iff $P'(s,s') > 0$, and $w(s,s') = -\log P'(s,s')$ (one could take any basis, we take the natural logarithm with basis e).

We define the weight $w(\pi)$ of a path π in $G_{\mathcal{D}}$ as the sum of the weights of the transitions in π. The relation between the weight of a finite path $\pi = s_0 \dots s_n$ in $G_{\mathcal{D}}$ and the probability of the same path in $\mathcal{D}$ is as follows:

$$\begin{aligned} w(\pi) = \quad & \textstyle\sum_{i=0}^{n-1} w(s_i, s_{i+1}) \quad = \quad \sum_{i=0}^{n-1} -\log P'(s_i, s_{i+1}) \\ & = -\textstyle\sum_{i=0}^{n-1} \log P'(s_i, s_{i+1}) = -\log \prod_{i=0}^{n-1} P'(s_i, s_{i+1}) \\ & = -\log \Pr_{s_0}^{\mathcal{D}'}(\pi)\,. \end{aligned}$$

Note that we can also compute the probabilities from the weights by $\Pr_{s_0}^{\mathcal{D}'}(\pi) = e^{-w(\pi)}$. Since the negative logarithm is monotonically decreasing in the interval $(0,1]$, more probable paths in $\mathcal{D}'$ have smaller weights in $G_{\mathcal{D}}$, i. e., $\Pr_s^{\mathcal{D}'}(\pi) \geq \Pr_s^{\mathcal{D}'}(\pi')$ iff $w(\pi) \leq w(\pi')$ for all states $s \in S$ and paths $\pi, \pi' \in \text{Paths}_{\text{fin}}^{\mathcal{D}}(s)$.

That means, the problem to find a sufficient number of most probable paths in $\mathcal{D}$ can be solved by finding a sufficient number of shortest paths in $G_{\mathcal{D}}$. The main advantage of this problem transformation, besides the lower complexity of the addition operation compared to multiplication, is that we can apply shortest path search algorithms without modification.

Definition 10 (k shortest path problem, [17]). *Given a directed weighted graph $G = (V, E, w)$, nodes $s, t \in V$, and $k \in \mathbb{N}$, the k* shortest path problem (KSP) *is to find k different paths $\pi_1, \dots, \pi_k$ from s to t in G (if they exist) such that for all $1 \leq i < j \leq k$, $w(\pi_i) \leq w(\pi_j)$ and for all paths π from s to t either $\pi \in \{\pi_1, \dots, \pi_k\}$ or $w(\pi) \geq w(\pi_k)$.*

Theorem 1 ([17]). *A smallest counterexample C for $\mathcal{D}$ contains $|C|$ shortest paths in $G_{\mathcal{D}}$ from s_{init} to t.*

Example 19. Consider the DTMC $\mathcal{D}$ from Example 14 on page 16, depicted in Figure 2, which already has a single absorbing target state. The corresponding directed weighted graph $G_{\mathcal{D}}$ is shown in Figure 3 (with rounded weights).

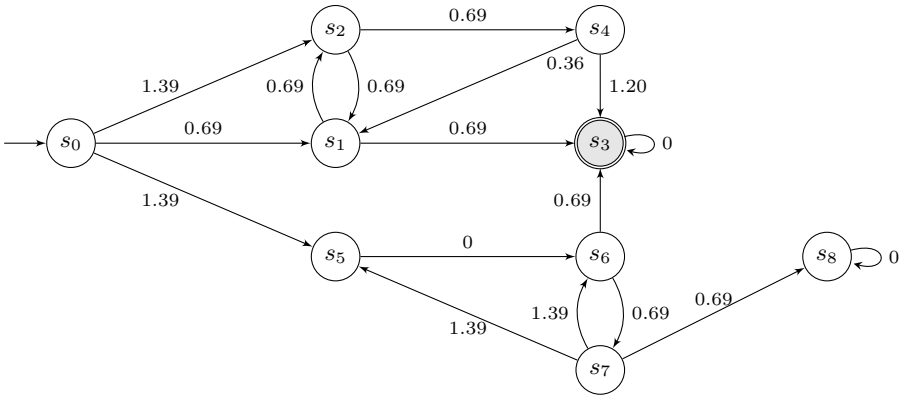


Fig. 3. The directed weighted graph for the DTMC from Figure 2 (cf. Example 19)

We would like to compute a counterexample for $\mathbb{P}_{\leq 0.4}\big(\Diamond\{s_3\}\big)$. Thus we search for k shortest paths $\pi_1, \ldots, \pi_k$ in $G_{\mathcal{D}}$ for an appropriate k such that $\sum_{i=1}^{k} e^{-w(\pi_i)} > 0.4$. The four shortest paths in $G_{\mathcal{D}}$, with their (rounded) weights in $G_{\mathcal{D}}$ and probabilities in $\mathcal{D}$ are as follows:

Path	Weight (rounded)	Probability
$\pi_1 = s_0 s_1 s_3$	1.39	$1/4$
$\pi_2 = s_0 s_5 s_6 s_3$	2.08	$1/8$
$\pi_3 = s_0 s_2 s_1 s_3$	2.77	$1/16$
$\pi_4 = s_0 s_1 s_2 s_1 s_3$	2.77	$1/16$

Since $\sum_{i\in\{1,2,3\}} e^{-w(\pi_i)} = \sum_{i\in\{1,2,4\}} e^{-w(\pi_i)} = 1/4 + 1/8 + 1/16 = 0.4375 > 0.4$, both path sets $\{\pi_1, \pi_2, \pi_3\}$ and $\{\pi_1, \pi_2, \pi_4\}$ are smallest counterexamples. ■

As the size of a smallest counterexample is not known in advance, we need k shortest paths computation algorithms that can determine the value of k *on the fly*. Examples of such algorithms are Eppstein's algorithm [48], the algorithm by Jiménez and Marzal [49], and the K* algorithm [50] by Aljazzar and Leue. While the former two methods require the whole graph to be placed in memory in advance, the *K* algorithm* (see also Section 4.2) expands the state space on the fly and generates only those parts of the graph that are needed. Additionally, it can apply directed search, i. e., it exploits heuristic estimates of the distance of the current node to a target node in order to speed up the search. The heuristic has thereby to be admissible, i. e., it must never over-estimate the distance.

For *bounded* reachability properties $\mathbb{P}_{\leq\lambda}(\Diamond^{\leq h} T)$, a *hop-constrained k shortest paths problem* (*HKSP*) can be used to determine a smallest counterexample. In this case the additional constraint that each evidence may contain at most h transitions must be imposed. In [17] an adaption of Jiménez and Marzal's algorithm to the HKSP problem is presented.

Heuristic approaches Besides the above methods to compute smallest counterexamples, heuristic approaches can be used to compute not necessarily smallest or even minimal ones. *Bounded model checking* (*BMC*) [51] is applied by Wimmer *et al.* in [19, 20] to generate evidences until the bound λ is exceeded. The basic idea of BMC is to formulate the existence of an evidence of length k (or $\leq k$) for some natural number k as a satisfiability problem. In [19] purely propositional formulas are used, which does not allow to take the actual probability of an evidence into account; in [20] this was extended to SMT formulas over linear real arithmetic, which allows to enforce a minimal probability of evidences. Using strategies like binary search, evidences with high probability (but still bounded length) can be found first.

In both cases, the starting point is a symbolic representation of the DTMC at hand as an MTBDD $\hat{P}$ for the transition probability matrix. For generating propositional formulas, this MTBDD is abstracted into a BDD $\hat{P}_{\text{BDD}}$ by mapping each leaf labeled with a positive probability to 1. Hence, the BDD $\hat{P}_{\text{BDD}}$ stores the edges of the underlying graph. The generation of propositional formulas is done by applying Tseitin's transformation [52] to this BDD, resulting in a predicate trans such that $\mathsf{trans}(v, v')$ is satisfied for an assignment of the variables v and v' if and only if the assignment corresponds to a transition with positive probability in the DTMC. The same is done for the initial state, resulting in a predicate init such that $\mathsf{init}(v)$ is satisfied if the assignment of v corresponds to the initial state of the DTMC and a predicate $\mathsf{target}(v)$ for the set of target states. With these predicates at hand, the BMC-formula is given as follows:

$$\mathsf{BMC}(k) = \mathsf{init}(v_0) \wedge \bigwedge_{i=0}^{k-1} \mathsf{trans}(v_i, v_{i+1}) \wedge \mathsf{target}(v_k)\,. \tag{1}$$

This formula is satisfied by an assignment ν iff $\nu(v_i)$ corresponds to a state s_i for $i = 1, \ldots, k$ such that $s_0 s_1 \ldots s_k$ is an evidence for the considered reachability property.

Starting at $k = 0$, evidences are collected and excluded from further search by adding new clauses to the current formula, until either the set of collected paths forms a counterexample or the current formula becomes unsatisfiable. In the latter case we increase k and continue the search.

During the BMC search, loops on found paths can be identified. A found path containing a loop can be added to the collection of evidences with arbitrary unrollings of the loop. However, since loop unrollings lead to longer paths, attention must be payed to exclude those paths when k reaches the length of previously added paths with unrolled loops.

The propositional BMC approach yields a counterexample consisting of evidences with a minimal number of transitions, but the drawback is that the actual probabilities of the evidences are ignored. This issue can be solved by using a SAT-modulo-theories formula instead of a purely propositional formula [20]. Thereby the transition predicate trans is modified to take the probabilities into account: $\mathsf{trans}(v_i, p_i, v_{i+1})$ is satisfied by an assignment ν iff $\nu(v_i)$ corresponds

to state s_i, $\nu(v_{i+1})$ to state s_{i+1}, $P(s_i, s_{i+1}) > 0$, and $\nu(p_i) = \log P(s_i, s_{i+1})$. By adding the constraint $\sum_{i=0}^{k-1} p_i \geq \log \delta$ for some constant $\delta \in (0, 1]$, we can enforce that only paths with probability at least δ are found.

Additionally, using an SMT formulation allows us to take *rewards* into account: we can extend the DTMC by a function $\rho : S \times S \to \mathbb{R}$, which specifies the reward of a transition. Rewards can—depending on the context—either represent *costs* (e. g., energy consumption, computation time, etc.) or *benefits* (number of packets transmitted, money earned, etc). Similar to constraints on the probability of an evidence, we can enforce that the accumulated reward along an evidence satisfies a linear constraint [20, 53].

Symbolic methods For a DTMC $\mathcal{D} = (S, s_{\text{init}}, P, L)$ together with a set of target states T that are represented *symbolically* in the form of BDDs $\hat{I}$ and $\hat{T}$ for the initial state and the target states, respectively, and an MTBDD $\hat{P}$ for the transition probability matrix, Günther, Schuster and Siegle [21] propose a BDD-based algorithm for computing the k most probable paths of a DTMC. They use an adaption of Dijkstra's shortest path algorithm [54], called *flooding Dijkstra*, to determine the most probable path. Then they transform the DTMC such that the most probable path of the transformed system corresponds to the second-most-probable path in the original DTMC. For this they create two copies of the DTMC: The new initial state is the initial state of the first copy, the new target states are the target states in the second copy. The transitions of the second copy remain unchanged. In the first copy, all transitions on the already found most probable path also remain unchanged. All other transitions lead from the first copy to the corresponding state in the second copy. Thus, to reach a target state from the initial state, at least one transition has to be taken which is not contained in the most probable path. The corresponding function has as input BDDs the symbolic representation of the DTMC as well as a BDD *SP* representing the current most probable path. Returned is a new symbolic DTMC:

$$(\hat{P}, \hat{I}, \hat{T}) := \texttt{Change}(\hat{P}, \hat{I}, \hat{T}, SP)$$

We illustrate this process using an example.

Example 20. Consider again the DTMC in Figure 2 on page 16. The first application of Dijkstra's algorithm yields the most probable path $s_0 s_1 s_3$ with probability $1/2 \cdot 1/2 = 1/4$ from the initial state s_0 to the target state s_3. To obtain the second-most-probable path, the DTMC in Figure 4 is constructed. In the modified DTMC, the initial state is s_0^0, the target state is s_3^1. The most probable path from s_0^0 to s_3^1 is $s_0^0 s_5^1 s_6^1 s_3^1$ with probability $1/4 \cdot 1 \cdot 1/2 = 1/8$. This path corresponds to $s_0 s_5 s_6 s_3$ in the original DTMC, which is the second-most probable path there.■

To obtain the next path, the same transformation is applied again. After k paths the underlying graph has increased exponentially in k. Each transformation step requires to introduce two new BDD-variables and typically increases the

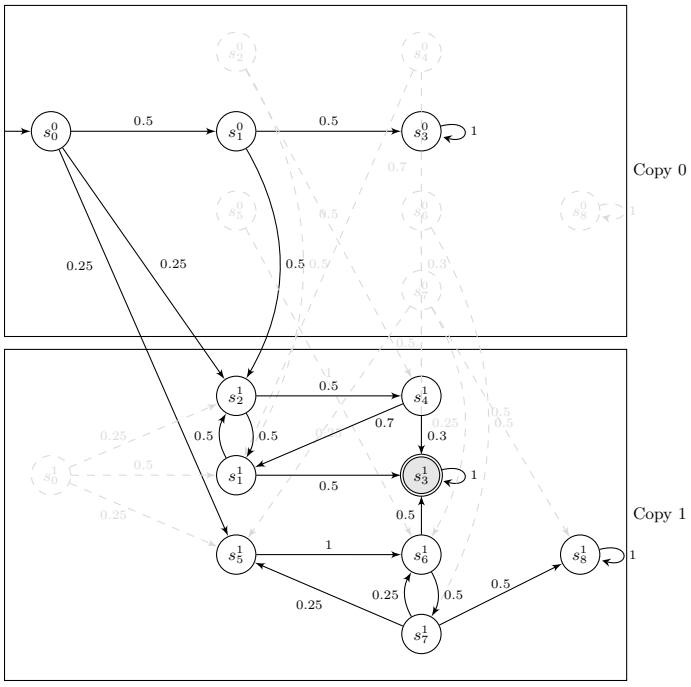


Fig. 4. Exclusion of the most probable path (the states and transitions which are not reachable from the initial state s_0^0 have been colored grey to improve readability)

size of the symbolic representation. Therefore this methods scales well to large state spaces, but not for large values of k.

Compact representations Alone the mere number of evidences in a counterexample can render the counterexample unusable for debugging purposes. Therefore a number of approaches have been proposed to obtain smaller, better understandable *representations* of counterexamples. Typically they exploit the fact that many paths in a counterexample differ only in the number and order of unrollings of loops.

Building upon ideas by Daws [55] for model checking parametric DTMCs, Han, Katoen and Damman [17, 22] proposed the representation of counterexamples as regular expressions: First the DTMC is turned into a deterministic finite automaton (DFA), whose transitions are labeled with (state, probability) pairs: Essentially, a transition from s to s' with probability $p = P(s, s') > 0$ in the DTMC is turned into the transition $s \xrightarrow{(s',p)} s'$ of the DFA. State elimination is used to turn the DFA into a regular expression. The state elimination removes states iteratively, and for each removed state it connects its predecessors with its successors by direct transitions. These new transitions are labeled with regular expressions describing the inputs read on the possible path from a predecessor

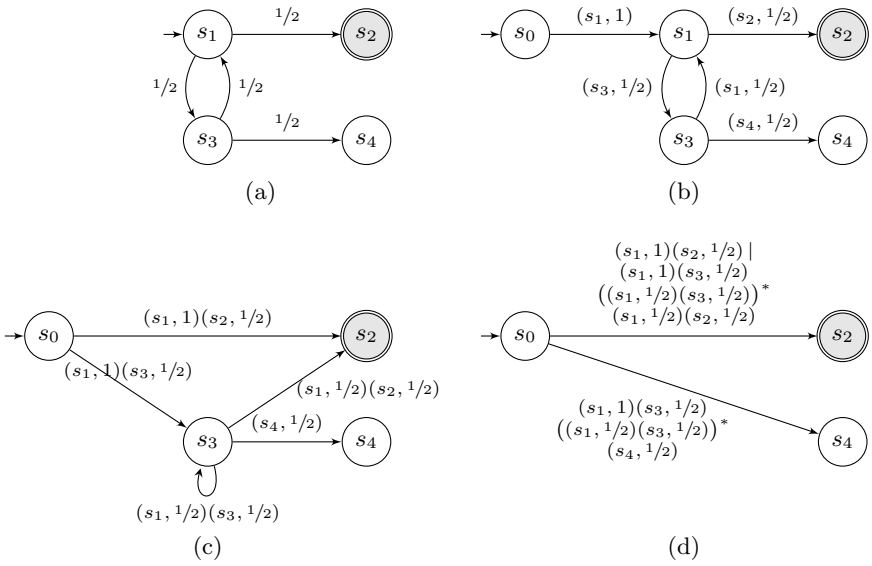


Fig. 5. Representing counterexamples as regular expressions (cf. Example 21)

via the removed state to a successor. In order to obtain a small regular expression for a counterexample, the authors proposed to iterate the following steps:

1. Find a most probable path in the remaining automaton using Dijkstra's shortest path algorithm.
2. Eliminate all states (except the first and last one) on this path; the order of elimination is determined according to a heuristics like [56], well known from the literature on automata theory. This gives a regular expression describing the considered most probable path.
3. Evaluate the set of regular expressions generated so far and check whether the joint probability mass of the represented paths is already beyond the given bound λ. If this is the case, terminate and return the regular expressions. Otherwise start a new iteration of the elimination loop.

Example 21. Consider the DTMC in Figure 5 (a) with target state s_2. Its DFA is depicted under (b). The first most probable path is $s_0 s_1 s_2$, i. e., we eliminate s_1, resulting in the DFA (c). The probability value of the regular expression generated for the found path is

$$\mathrm{val}\big((s_1, 1)(s_2, 1/2)\big) = \mathrm{val}\big((s_1, 1)\big) \cdot \mathrm{val}\big((s_2, 1/2)\big) = 1 \cdot 1/2 = 1/2\,.$$

If this mass is not yet sufficient to violate the bound, we search for the most probable path in (c), which is $s_0 s_3 s_2$. We eliminate s_3 resulting in the DFA (d).

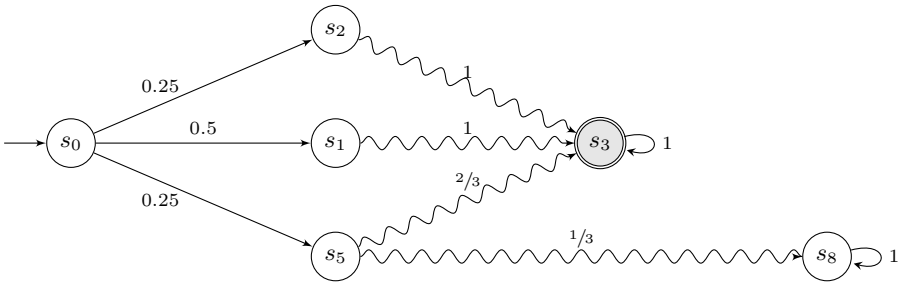


Fig. 6. The result of SCC abstraction applied to the DTMC in Figure 2 (cf. Example 22)

The probability value of the regular expression for the second found path is

$$\begin{aligned}
&\mathrm{val}\big((s_1,1)(s_3,1/2) \quad ((s_1,1/2)(s_3,1/2))^* \quad (s_1,1/2)(s_2,1/2)\big) &&= \\
&\mathrm{val}\big((s_1,1)(s_3,1/2)\big) \cdot \frac{1}{1-\mathrm{val}\big((s_1,1/2)(s_3,1/2)\big)} \cdot \mathrm{val}\big((s_1,1/2)(s_2,1/2)\big) &&= \\
&\quad 1 \cdot 1/2 \quad \cdot \quad 4/3 \quad \cdot \quad 1/2 \cdot 1/2 &&= \\
&\quad 1/6\,.
\end{aligned}$$

Since there are no more paths from the initial state s_0 to s_2, the total probability to reach s_2 from s_0 is the value $1/2 + 1/6 = 2/3$ of the regular expression $(s_1,1)(s_2,1/2) \,|\, (s_1,1)(s_3,1/2)((s_1,1/2)(s_3,1/2))^*(s_1,1/2)(s_2,1/2)$. ■

The same can also be applied for *bounded* reachability properties $\mathbb{P}_{\leq\lambda}(\Diamond^{\leq h} T)$. The only changes are the usage of a hop-constraint shortest path algorithm and a different method for determining the probability of the represented path, such that only the probability of those paths represented by the regular expressions is counted whose length is at most h.

A different compaction of counterexamples is described by Andrés, D'Argenio and van Rossum in [24]. As many paths only differ in the number and order of unrollings of *loops* in the system, the non-trivial *strongly connected components*[4] (*SCCs*) of the DTMC under consideration, i. e., those SCCs which contain more than one state, are abstracted into direct edges from the input to the output states of the SCC. Input states are states in the SCC which have an incoming edge from outside the SCC, and output states are outside of the SCC, but have an incoming edge from inside the SCC. The probability of these edges is determined using model checking as the probabilities to reach the output states from the input states. After this abstraction, counterexamples as sets of paths can be easily determined in the resulting acyclic model.

Example 22. In the DTMC from Figure 2 on page 16, there are two non-trivial SCCs consisting of the states (i) $\{s_5, s_6, s_7\}$ with input state s_5 and output states

[4] A strongly connected component (SCC) is a maximal set of states such that for all s and s' in the SCC, s' can be reached from s inside the SCC.

s_3 and s_8, and (ii) $\{s_1, s_2, s_4\}$ with input states s_1 and s_2 and output state s_3. Eliminating these SCCs results in the DTMC shown in Figure 6. The wave-like edges represent paths through SCCs that have been abstracted. ■

4.2 Path-Based Counterexamples for PA

The simplest way to generate path-based counterexamples for a PA $\mathcal{M}$ [25, 38] is to first generate a *memoryless deterministic scheduler* σ^* which *maximizes* the reachability probability. Such a scheduler can be obtained as a by-product from model checking. This scheduler σ^* induces a DTMC $\mathcal{M}^{\sigma^*}$, such that $\Pr_{s_{\text{init}}}^{\mathcal{M}^{\sigma^*}}(\Diamond T) = \max_{\sigma \in \text{Sched}_{\mathcal{M}}} \Pr_{s_{\text{init}}}^{\mathcal{M}^{\sigma}}(\Diamond T) > \lambda$. In a second step, the methods for counterexample generation described above are applied to $\mathcal{M}^{\sigma^*}$, resulting in a counterexample C for $\mathcal{M}^{\sigma^*}$. Then (σ^*, C) is a counterexample for $\mathcal{M}$.

However, as the computation of a maximizing scheduler requires to have the whole state space of $\mathcal{M}$ residing in memory, the advantage of using an algorithm like K* [50] which expands the state space on the fly when necessary, is lost. Therefore, Aljazzar and Leue [25] proposed a method which allows to not only compute the paths but also the scheduler on the fly as follows.

The problem when applying K* to a PA is that the generated paths are in general not compatible to the same scheduler. Therefore all paths are kept and clustered according to the scheduler choice made in each state. To do so an AND/OR-tree is maintained, which is initially empty. The OR-nodes correspond to the state nodes, in which the scheduler makes a decision. The AND-nodes correspond to the probabilistic decisions after an action-distribution pair has been chosen by the scheduler. Applying the K* algorithm to the PA $\mathcal{M}$, the next most probable path is determined. The new path π is inserted into the tree by first determining the longest prefix which is already contained in the tree. The remainder of the path becomes a new sub-tree, rooted at the node where the longest prefix ends. By a bottom-up traversal, a counterexample and a (partial) scheduler can be determined from the AND/OR-tree.

Example 23. Assume the MDP in Figure 7 left, which violates the reachability property $\mathbb{P}_{\leq 0.75}(\Diamond\{s_4\})$. Assume furthermore that the path search gives us the following paths in this order:

Path	Path probability
$\pi_1 = s_0 s_1 s_4$	0.5
$\pi_2 = s_0 s_2 s_4$	0.4
$\pi_3 = s_0 s_2 s_0 s_1 s_4$	0.25
$\pi_4 = s_0 s_1 s_0 s_1 s_4$	0.2
$\pi_5 = s_0 s_2 s_0 s_2 s_4$	0.2
$\pi_6 = s_0 s_1 s_0 s_2 s_4$	0.16
$\pi_7 = s_0 s_2 s_0 s_2 s_0 s_1 s_4$	0.125
$\pi_8 = s_0 s_1 s_0 s_2 s_0 s_1 s_4$	0.1

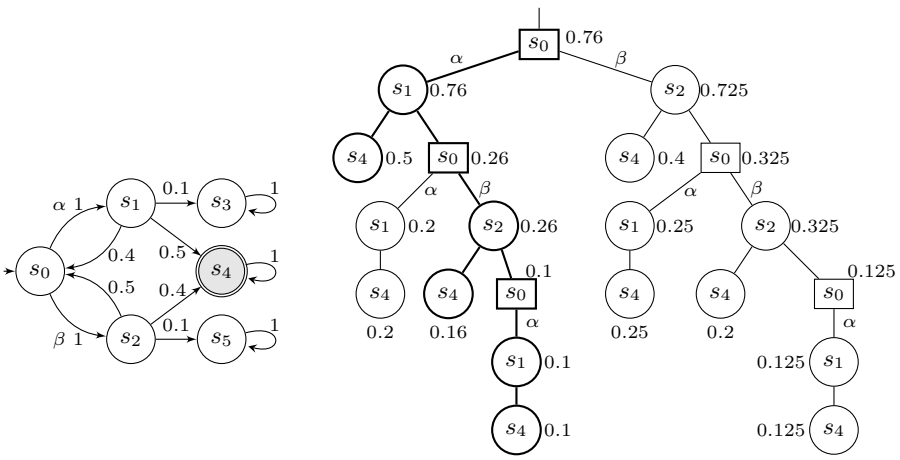


Fig. 7. Example MDP (cf. Example 23)

The generated path tree is depicted in Figure 7 on the right-hand side. The rectangular nodes are OR-nodes, the circles are AND-nodes. The value attached to a leaf is the probability of the path from the root to the leaf. The value attached to an inner AND-node is the sum of the values of its children, whereas the value of an OR-node is the maximum of all children values. Thus the value of the root specifies the maximal probability of found compatible paths, which are possible under a common scheduler.

After having added the last path, the probability of the root is above 0.75; the boldface subtree specifies a suitable scheduler to build a counterexample with the path set $\{\pi_1, \pi_6, \pi_8\}$. Note that this scheduler is deterministic but not memoryless. ■

4.3 Applications of Path-Based Counterexamples

Path-based counterexamples are mostly used in two main areas: Firstly, for extracting the actual causes why a system fails. This information can be used for *debugging* an erroneous system [42–44]. Secondly, for *counterexample-guided abstraction refinement* of probabilistic automata [38]. We briefly sketch the main ideas of these works.

The extraction of reasons why a system fails is based on the notion of *causality* [57]. The idea behind that is that an event A is *critical* for event B, if A had not happened, then B would not have happened either. However, this simple notion of criticality is sometimes too coarse to be applicable. Therefore Halpern and Pearl [57] have refined it to take a *side-condition* into account: Essentially, if the events in some set E did not have happened, then A would be critical for the occurrence of B. In this case A is a *cause* of B.

Example 24 (taken from [57]). Assume Suzy and Billy are both throwing stones at a bottle, and both throw perfectly, so each stone would shatter the bottle. But Suzy throws a little harder such that her stone reaches the bottle first.

Clearly we would say that the cause of the shattering of the bottle is Suzy throwing a stone. However, Suzy throwing is not critical, since if she did not throw, the bottle would be shattered anyway (by Billy's stone). But under the side-condition that Billy does not throw, Suzy's throw becomes critical.[5] ■

For details on this notion of causality, its formal definition, and a series of examples we refer the reader to [57].

Chockler and Halpern [58] use a quantitative notion regarding causes, given by the *degree of responsibility* $\mathrm{dR}(A)$ of a cause A: Essentially $\mathrm{dR}(A) = \frac{1}{1+k}$ where k is the size of the smallest side-condition needed to make A critical.

Debbi and Bourahla [42, 43] consider *constrained reachability properties* of the form $\mathbb{P}_{\leq\lambda}(\varphi_1 \,\mathbb{U}\, \varphi_2)$ where φ_1 and φ_2 are arbitrary Boolean combinations of atomic propositions from the set AP, and $\mathbb{U}$ is the temporal until operator. As potential causes for the violation of the property they consider propositions of certain states, i. e., pairs $\langle s, a\rangle$ for $s \in S$ and $a \in \mathrm{AP}$: If the value of such a proposition is switched (under some side-condition), some paths in the considered counterexample no longer satisfy the formula $\varphi_1 \,\mathbb{U}\, \varphi_2$, and the probability mass of the remaining paths is no longer above the bound λ. They assign weights to the causes as follows: The probability $\Pr(s, a)$ of a cause (s, a) is the sum of the probabilities of all paths π in the counterexample which contain state s. The weight $w(s, a)$ of a cause (s, a) is given by $w(s, a) = \Pr(s, a) \cdot \mathrm{dR}(s, a)$. The causes are presented to the user with decreasing weight.

A different approach, also based on the notion of causality of [57], is described by Leitner-Fischer and Leue in [44, 59]. The authors proposed to extract *fault trees* from path-based probabilistic counterexamples. For this they do not consider just evidences of the underlying DTMC, but they rather keep track of the events which caused the transitions along an evidence. Since the order of events along the evidences can be crucial for the failure, they extend the notion of causality to also take the event order into account. Hence, a cause is a sequence of events together with restrictions on the order of the events. Additionally, the joint probability of the evidences which correspond to such a cause is computed. A fault tree is generated from the causes by using the undesired behavior as the root, which has one subtree per cause. Each cause is turned into a tree by using an AND gate over those events whose order does not matter, and an ordered-AND gate if the order does matter. Additionally the subtree corresponding to a cause is annotated by the probability of the corresponding evidences.

An interactive *visualization* technique is proposed by Aljazzar and Leue in [60] to support the user-guided identification of causal factors in large counterexamples. The authors apply this visualization technique to debug an embedded control system and a workstation cluster.

[5] The precise formal definition encompasses more constraints in order to avoid Billy throwing being a cause.

Failure mode and effects analysis (FMEA) allows to analyze potential system hazards resulting from system (component) failures. An extension of the original FMEA method can also handle probabilistic systems. In this context, path-based probabilistic counterexamples were used by Aljazzar *et al.* in [41] to facilitate the redesign of a potentially unsafe airbag system.

A different application of path-based counterexamples is described by Hermanns, Wachter and Zhang in [38] for *counterexample-guided abstraction refinement (CEGAR)*: The starting point is an abstraction of a PA, over-approximating the behavior of a concrete PA model. If this abstraction is too coarse, it might violate a property even if the concrete system satisfies it. In this case counterexamples are used to refine the abstraction.

A PA is abstracted by defining a finite partitioning of its state space and representing each block of the partition by an abstract state; all transitions targeting a concrete state are redirected to its abstract state, and similarly all outgoing transitions of a concrete state start in the abstract state to which it belongs.

Starting with an initial abstraction, model checking is performed to check whether the property at hand is satisfied. If this is the case, one can conclude that it is also satisfied in the concrete model. However, if the property is violated by the abstraction, the optimal scheduler, obtained from the model checking process, is used to compute the induced DTMC. Therein a path-based counterexample is determined. Now two cases are possible: Either the counterexample of the abstract system corresponds to a counterexample in the concrete model, in which case the property is also violated by the concrete model. Or the counterexample is *spurious*, i. e., it exists only in the abstraction due to the over-approximating behavior, in which case the abstraction needs to be refined. This is done by *predicate abstraction*, splitting the abstract states according to a predicate P into a subset satisfying P and one violating it. The predicate P is obtained from the counterexample evidences via interpolation.

Experimental results show that in some cases a definite statement about the satisfaction of the property at hand can be made on a very coarse approximation. This speeds up the model checking process and allows to handle much larger systems than with conventional methods.

5 Critical Subsystems

Path-based representations of counterexamples, as discussed in the previous Section 4, have some major drawbacks: The number of paths needed might be very large (or even infinite), leading to high memory requirements. As a consequence, the number of search iterations in terms of path-searches is equally high, leading to high computational costs. Finally, a counterexample consisting of a high number of potentially long paths is hard to understand and analyze, therefore its usefulness is restricted.

An alternative is to use *critical subsystems*, which are fractions of DTMC, MDP or PA models violating a property, such that the behavior of the models restricted

to the critical subsystems already violates the property. It is often possible to generate critical subsystems whose size is smaller by orders of magnitude in comparison to the input system. Thereby, the *critical part* of the original system leading to the violation is highlighted.

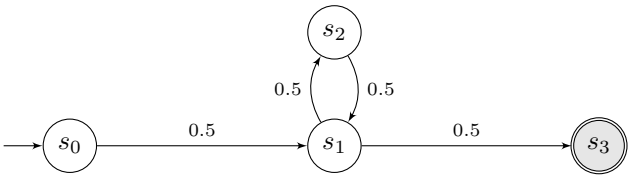
Definition 11 (Critical subsystems of DTMCs). *Assume a DTMC* $\mathcal{D} = (S, s_{\text{init}}, P, L)$*, a target state set* $T \subseteq S$ *and some* $\lambda \in [0,1] \cap \mathbb{Q}$ *such that* $\mathcal{D} \not\models \mathbb{P}_{\leq\lambda}(\Diamond T)$.

A subsystem $\mathcal{D}'$ *of* $\mathcal{D}$*, written* $\mathcal{D}' \sqsubseteq \mathcal{D}$*, is a DTMC* $\mathcal{D}' = (S', s_{\text{init}}, P', L')$ *such that* $S' \subseteq S$*,* $s_{\text{init}} \in S'$*,* $P'(s,s') > 0$ *implies* $P'(s,s') = P(s,s')$ *for all* $s, s' \in S'$*, and* $L'(s) = L(s)$ *for all* $s \in S'$.

Given $S' \subseteq S$ *with* $s_{\text{init}} \in S'$*, the subsystem* $\mathcal{D}_{S'} = (S', s_{\text{init}}, P', L')$ *of* $\mathcal{D}$ *with* $P'(s,s') = P(s,s')$ *and* $L'(s) = L(s)$ *for all* $s, s' \in S'$ *is called the* subsystem of $\mathcal{D}$ induced by S'.

A subsystem $\mathcal{D}'$ *of* $\mathcal{D}$ *is* critical *for* $\mathbb{P}_{\leq\lambda}(\Diamond T)$ *if* $T \cap S' \neq \emptyset$ *and* $\mathcal{D}' \not\models \mathbb{P}_{\leq\lambda}\big(\Diamond(T \cap S')\big)$.

Example 25. For the DTMC in Figure 2 on page 16 and the reachability property $\mathbb{P}_{\leq 0.3}\big(\Diamond\{s_3\}\big)$, the following DTMC is a critical subsystem, since the probability to reach s_3 from s_0 is $\frac{1}{2} \cdot \frac{1}{1-\frac{1}{2}\cdot\frac{1}{2}} \cdot \frac{1}{2} = \frac{1}{3} > 0.3$:



■

The above definition of critical subsystems of DTMCs is a special case of the following definition generalized for PAs:

Definition 12 (Critical subsystems for PAs). *Assume a PA* $\mathcal{M} = (S, s_{\text{init}}, \text{Act}, \hat{P}, L)$*, a target state set* $T \subseteq S$ *and some* $\lambda \in [0,1] \cap \mathbb{Q}$ *such that* $\mathcal{M} \not\models \mathbb{P}_{\leq\lambda}(\Diamond T)$.

A subsystem $\mathcal{M}'$ *of* $\mathcal{M}$*, written* $\mathcal{M}' \sqsubseteq \mathcal{M}$*, is a PA* $\mathcal{M}' = (S', s_{\text{init}}, \text{Act}, \hat{P}', L')$ *such that* $S' \subseteq S$*,* $s_{\text{init}} \in S'$*,* $L'(s) = L(s)$ *for all* $s \in S'$*, and for each* $s \in S'$ *there is an injective function* $f : \hat{P}'(s) \to \hat{P}(s)$ *such that for all* $(\alpha', \mu') \in \hat{P}'(s)$ *with* $f\big((\alpha', \mu')\big) = (\alpha, \mu)$ *if it holds that* $\alpha' = \alpha$ *and* $\mu'(s') = \mu(s')$ *for all* $s' \in \text{supp}(\mu')$.

A subsystem $\mathcal{M}'$ *of* $\mathcal{M}$ *is* critical *for* $\mathbb{P}_{\leq\lambda}(\Diamond T)$ *if* $T \cap S' \neq \emptyset$ *and* $\mathcal{M}' \not\models \mathbb{P}_{\leq\lambda}\big(\Diamond(T \cap S')\big)$.

To have well-understandable explanations for the property violation, for PAs we are interested in their critical subsystems induced by deterministic memoryless schedulers. Therefore, in the context of counterexamples in the following we consider only DTMCs (as deterministic PAs) as critical subsystems.

The set of those paths of a critical subsystem $\mathcal{D}'$ which are evidences for a reachability property form a counterexample in the classical sense as in Definition 8, i. e.,

$$C := \text{Paths}^{\mathcal{D}'}_{\text{fin}}(s_{\text{init}}, T)$$

is a counterexample. Therefore, a critical subsystem can be seen as a *symbolic representation* of a counterexample.

We define minimality of critical subsystems in terms of their state space size: A critical subsystem is *minimal* if it has a minimal set of states under all critical subsystems. Analogously to counterexamples, we can also define a *smallest* critical subsystem to be a minimal critical subsystem in which the probability to reach a target state is maximal under all minimal critical subsystems. Note that even if a critical subsystem is smallest or minimal, this does not induce a smallest or minimal counterexample in the sense of [17].

Critical subsystems can be generated in various ways. In this section, we first discuss the generation of critical subsystems for DTMCs: We start by describing how solver technologies can be used to compute *smallest critical subsystems* of DTMCs. This powerful method is also applicable to arbitrary ω-regular properties [27, 28, 61]. Afterward we describe heuristic algorithms which determine a (small) critical subsystem by means of *graph algorithms* as presented by Aljazzar and Leue in [30] and by Jansen *et al.* in [29]. We also give the intuition of an *extension to symbolic graph representations* [32]. The second part of this section is devoted to the computation of smallest critical subsystems for MDPs and PAs.

5.1 Critical Subsystems for DTMCs

Smallest Critical Subsystems In [27, 28, 61] an approach to compute *smallest critical subsystems* is proposed. The idea is to encode the problem of finding a smallest critical subsystem as a *mixed integer linear programming* (*MILP*) problem (see, e. g., [62]). It is also possible to give an SMT-formulation over linear real arithmetic, but the experiments in [27] clearly show that the MILP formulation is much more efficiently solvable. We therefore restrict our presentation here to the MILP formulation.

Definition 13 (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* $x \in \mathbb{R}^n,\ y \in \mathbb{Z}^k$.

In the following let $\mathcal{D} = (S, s_{\text{init}}, P, L)$ be a DTMC and $\mathbb{P}_{\leq\lambda}(\Diamond T)$ a reachability property that is violated by $\mathcal{D}$. We assume that $\mathcal{D}$ does not contain any state that is irrelevant for reaching T from s_{init}.

We want to determine a minimal set $S' \subseteq S$ of states such that $\mathcal{D}_{S'}$ is a critical subsystem. To do so, we introduce for each state $s \in S$ a decision variable $x_s \in \{0, 1\} \subseteq \mathbb{Z}$, which should have the value 1 iff s is contained in the selected subsystem, i. e., if $s \in S'$. Additionally we need for each $s \in S$ a variable $p_s \in [0, 1] \cap \mathbb{Q}$ which stores the probability to reach T from s within the selected subsystem $\mathcal{D}_{S'}$. The following MILP then yields a smallest critical subsystem of $\mathcal{D}$ and $\mathbb{P}_{\leq\lambda}(\Diamond T)$:

$$\text{minimize} \quad -\frac{1}{2} \cdot p_{s_{\text{init}}} + \sum_{s\in S} x_s \tag{2a}$$

such that

$$\forall s \in T: \quad p_s = x_s \tag{2b}$$

$$\forall s \in S \setminus T: \quad p_s \leq x_s \tag{2c}$$

$$\forall s \in S \setminus T: \quad p_s \leq \sum_{s' \in \mathrm{supp}\left(P(s)\right)} P(s,s') \cdot p_{s'} \tag{2d}$$

$$p_{s_{\mathrm{init}}} > \lambda \; . \tag{2e}$$

If ν is a satisfying assignment of this MILP, then $\mathcal{D}_{S'}$ with $S' = \{s \in S \,|\, \nu(x_s) = 1\}$ is a smallest critical subsystem. Constraint (2b) states that the probability of a target state is 1 if it is contained in the subsystem, and 0 otherwise. Constraint (2c) ensures that the probability contribution of states not contained in the subsystem is 0. Constraint (2d) bounds the probability contribution of each non-target state by the sum of the probabilities to go to a successor state times the probability contribution of the successor state. Finally, (2e) encodes that the subsystem is critical.

The objective function (2a) ensures (i) that the subsystem is minimal by minimizing the number of x_s-variables with value 1 and (ii) that the subsystem is smallest by minimizing $-1/2 \cdot p_{s_{\mathrm{init}}}$.

Example 26. Consider again the DTMC $\mathcal{D}$ in Figure 2 on page 16 and the violated reachability property $\mathbb{P}_{\leq 0.3}(\Diamond\{s_3\})$. Note that s_8 is irrelevant and can therefore be ignored together with all its incident transitions. The constraints to compute a smallest critical subsystem are as follows:

$$\begin{aligned}
&\text{minimize } -1/2 \cdot p_{s_0} + x_{s_0} + x_{s_1} + x_{s_2} + x_{s_3} + x_{s_4} + x_{s_5} + x_{s_6} + x_{s_7} \\
&\text{such that} \\
&\quad p_{s_3} = x_{s_3} \\
&\quad p_{s_0} \leq x_{s_0} \qquad p_{s_0} \leq 0.5p_{s_1} + 0.25p_{s_2} + 0.25p_{s_5} \\
&\quad p_{s_1} \leq x_{s_1} \qquad p_{s_1} \leq 0.5p_{s_2} + 0.5p_{s_3} \\
&\quad p_{s_2} \leq x_{s_2} \qquad p_{s_2} \leq 0.5p_{s_1} + 0.5p_{s_4} \\
&\quad p_{s_4} \leq x_{s_4} \qquad p_{s_4} \leq 0.7p_{s_1} + 0.3p_{s_3} \\
&\quad p_{s_5} \leq x_{s_5} \qquad p_{s_5} \leq 1.0p_{s_6} \\
&\quad p_{s_6} \leq x_{s_6} \qquad p_{s_6} \leq 0.5p_{s_3} + 0.5p_{s_7} \\
&\quad p_{s_7} \leq x_{s_7} \qquad p_{s_7} \leq 0.25p_{s_5} + 0.25p_{s_6} \\
&\quad p_{s_0} > 0.3
\end{aligned}$$

Solving this MILP yields the following assignment:

Variable	x_{s_0}	p_{s_0}	x_{s_1}	p_{s_1}	x_{s_2}	p_{s_2}	x_{s_3}	p_{s_3}	x_{s_4}	p_{s_4}	x_{s_5}	p_{s_5}	x_{s_6}	p_{s_6}	x_{s_7}	p_{s_7}
Value	1	$5/12$	1	$2/3$	1	$1/3$	1	1	0	0	0	0	0	0	0	0

This solution corresponds to the DTMC $\mathcal{D}_{S'}$ with $S' = \{s_0, s_1, s_2, s_3\}$, shown in Figure 11(b) on page 38. ■

The solution of this MILP is rather costly (solving MILPs in general is NP-complete). However, the solution process can be accelerated by adding redundant

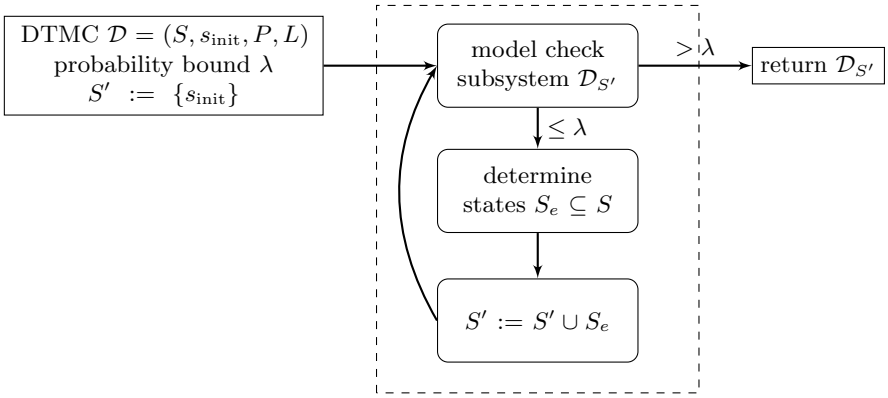


Fig. 8. Incremental generation of critical subsystems

constraints which exclude non-optimal solutions from the search space [27, 61]. For example, one can require that each state $s \notin T$ contained in the subsystem has a successor state which is also contained in the subsystem:

$$\forall s \in S \setminus T : \quad x_s \leq \sum_{s' \in \text{supp}\left(P(s)\right)} x_{s'} \ .$$

The described approach has been generalized to arbitrary ω-regular properties [28, 61].

Heuristic Approaches An alternative approach to determine critical subsystems is to use the classical path search algorithms as presented in Section 4 to search for evidences and use the states or transitions of these evidences to incrementally build a subsystem until it becomes critical. Here we focus on building critical subsystems using the states in evidences. Analogously, we could also use the transitions to build a subsystem with a similar approach.

Assume in the following a DTMC $\mathcal{D} = (S, s_{\text{init}}, P, L)$, a set $T \subseteq S$ of target states and an upper probability bound $\lambda \in [0, 1] \cap \mathbb{Q}$ of reaching target states from T. We assume this probability to be exceeded in $\mathcal{D}$.

The process of computing a critical subsystem is depicted in Figure 8. We start with the smallest possible subsystem containing just the initial state (see Definition 11 for the definition of $\mathcal{D}_{S'}$). As long as the subsystem is not yet critical, we iteratively determine a new state set and extend the previous subsystem with these states. Thereby the method that determines the state sets must assure progress, i. e., that new states are added to the subsystem after a finite number of iterations. Under this condition, the finiteness of the state space guarantees termination.

Calling a model checker in every iteration is quite costly. Therefore, all approaches based on this framework use some heuristics to avoid this. For

instance, one might think of performing model checking only after a certain number of iterations or only start to check the system after a certain size of the subsystem is reached.

We will now shortly discuss what approaches have been proposed to determine the state sets to incrementally extend subsystems.

Extended best-first search The first method to compute critical subsystems using graph-search algorithms was given in [30] by Aljazzar and Leue. The authors extend the *best-first* (*BF*) search method [63] to what they call *eXtended Best-First* (*XBF*) search, implemented in [36]. Below we describe the XBF search, without highlighting the differences to the BF search, which are discussed in [30].

For the XBF search, the system does not need to be given explicitly in the beginning but is explored *on the fly*, which is a great advantage for very large systems where a counterexample might be reasonably small. Instead, a symbolic model representation can be used.

Starting from the initial state, new states are discovered by visiting the successors of already discovered states. Two state lists `open` and `closed` store the states discovered so far. The ordered list `open` contains discovered states whose successors have not been expanded yet. In each step, one (with respect to the ordering maximal) state s from `open` is chosen, its not yet discovered successors are added to `open`, and s is moved from `open` to `closed`. To have all relevant information about the explored part of the model, for all states in the above two lists we also store all incoming transitions through which the state was visited.

The list `open` is ordered with respect to an evaluation function $f : S \to \mathbb{Q}$ which estimates for each discovered state s the probability of the most probable path from the initial state to a target state through s. The estimation

$$f(s) = g(s) \cdot h(s)$$

is composed by two factors: Firstly, $g(s)$ estimates the probability of the most probable path from s_{init} to s by the probability of the most probable such path found so far. Secondly, $h(s)$ uses further knowledge about the system at hand (if available) to estimate the probability of the most probable path from s to T. If the latter function is not constant, the search is called *informed search*.

Initially, $g(s_{\text{init}}) = 1$. When expanding the successor s' of a state s, we define $g(s')$ to be $g(s) \cdot P(s, s')$ if s' is encountered the first time, and the maximum of $g(s) \cdot P(s, s')$ and the old $g(s')$ value else. When in the latter case $g(s')$ is updated to a larger value, if s' was already in the `closed` set, it is moved back to the `open` set to propagate the improvement.

The algorithm maintains an initially empty subsystem $\mathcal{D}'$ of the already discovered model part. Each time a state s is visited, such that s is either a target state or it is included in $\mathcal{D}'$, the subsystem $\mathcal{D}'$ gets extended with the fragment of the currently known model part that is backward reachable from s. The algorithm terminates if this subsystem becomes critical ([30] calls it a *diagnostic subgraph*).

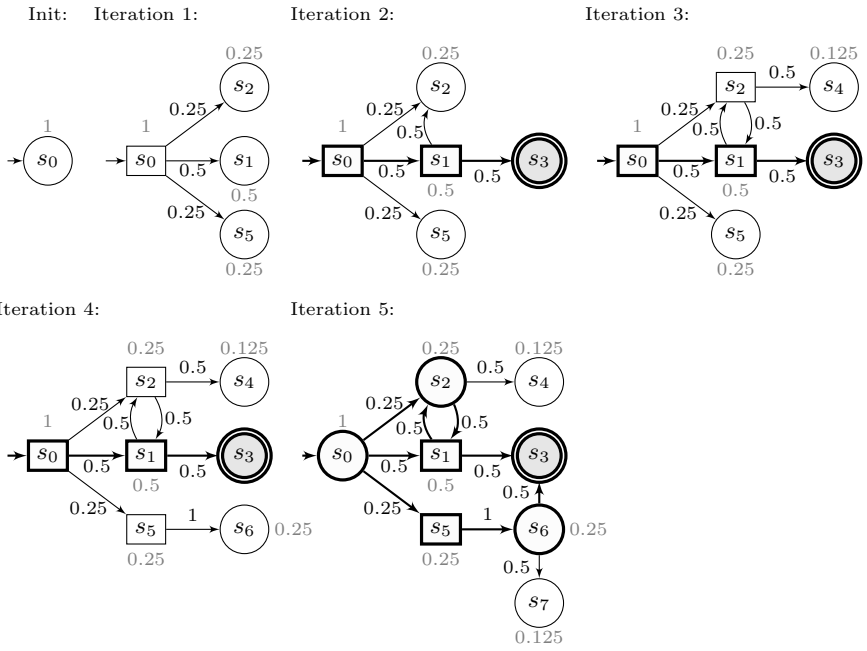


Fig. 9. Illustration of the XBF search (cf. Example 27)

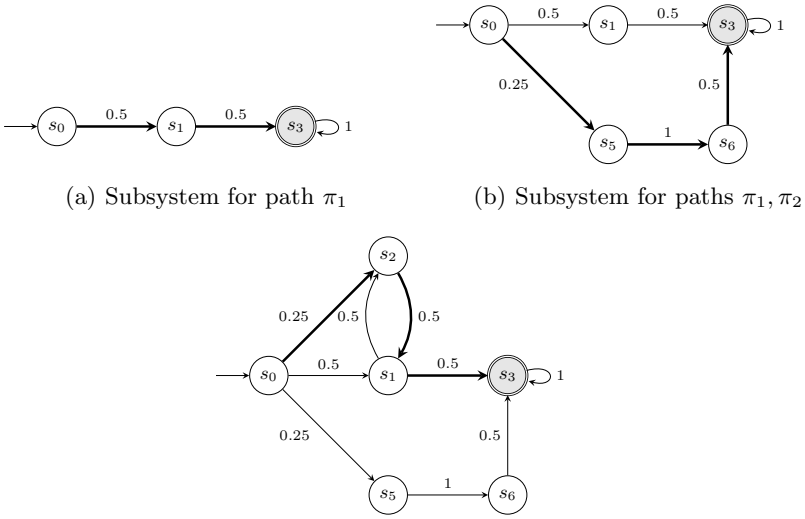
Example 27. For the DTMC in Figure 2 on page 16 and the reachability property $\mathbb{P}_{\leq 0.5}\big(\Diamond\{s_3\}\big)$, the computation of the XBF search is illustrated in Figure 9. Rectangular nodes are stored in the `closed` list, circles in the `open` list. For simplicity we assume $h(s) = 1$ for all states $s \in S$. Thus the current estimate values $f(s) = g(s) \cdot 1$ (shown beside the states in gray color) equal the highest known path probability from s_0 to s. The boldface fraction of the discovered model part is the current subsystem, which is critical after the fifth iteration (with probability $^{13}/_{24}$ to reach s_3 from s_0). ■

Search based on k shortest paths In [29] two different graph search algorithms are utilized. We distinguish the *global search* and *local search* approach.

The *global search* is an adaption of the k shortest paths search as described in Section 4. However, paths are collected not until a counterexample as a list of paths is formed, but until the subsystem $\mathcal{D}_{S'}$ induced by the states S' on found paths has enough probability mass, i. e., until it becomes critical.

Example 28. For the DTMC $\mathcal{D}$ in Figure 2 on page 16 and the violated property $\mathbb{P}_{\leq 0.4}\big(\Diamond\{s_3\}\big)$, three most probable paths are:

Path	Probability
$\pi_1 = s_0 s_1 s_3$	0.25
$\pi_2 = s_0 s_5 s_6 s_3$	0.125
$\pi_3 = s_0 s_2 s_1 s_3$	0.0625



(a) Subsystem for path π_1

(b) Subsystem for paths π_1, π_2

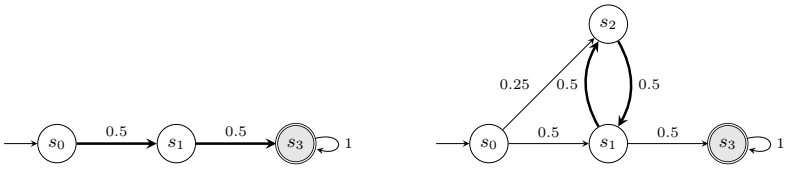
(c) Subsystem for paths π_1, π_2, π_3

Fig. 10. Illustration of the global search approach (cf. Example 28)

Now, we subsequently add these paths to an initially empty subsystem, until inside this system the probability to reach the state s_3 exceeds 0.4. We highlight the latest paths by thick edges in the subsystem. Starting with π_1, the initial subsystem consists of the states of this path, see Figure 10(a), with the reachability probability $0.25 < 0.4$. In the next iteration, the subsystem is extended by the states of path π_2, see Figure 10(b). The probability is now 0.375 which is still not high enough. Adding path π_3 in the next iteration effectively extends the subsystem by state s_2 as the other states are already part of the subsystem. Note that we add to the subsystem not only the states and transitions along found paths, but all transitions connecting them in the full model. The model checking result is now $13/24 \approx 0.542$, so the subsystem depicted in Figure 10(c) is critical and the search terminates. ■

The *local search* also searches for most probable paths to form a subsystem, however, not the most probable paths from the initial to target states, but the most probable paths connecting fragments of already found paths. Intuitively, every new path to be found has to be the most probable one that both starts and ends in states that are already contained in the current subsystem while the states in between are new.

Example 29. Reconsider the DTMC in Figure 2 on page 16 and the violated property $\mathbb{P}_{\leq 0.4}(\Diamond s_3)$ as in Example 28. Initially, we search for the most probable path that connects the initial state and target states, i. e., again path $\pi_1 = s_0 s_1 s_3$ is found and added to the subsystem, depicted in Figure 11(a). The subsystem



(a) Subsystem for path π_1 (b) Subsystem for paths π_1, π_2

Fig. 11. Illustration of the local search approach (cf. Example 29)

has probability 0.25 of reaching s_3. Now, we search for the most probable path that both starts and ends in one of the states s_0, s_1, or s_3, and find $\pi_2' = s_1 s_2 s_1$ with probability 0.25. As adding state s_2 induces also the transition from s_0 to s_2 this already gives enough probability $5/12 \approx 0.416$ for the subsystem depicted in Figure 11(b) to be critical. ■

Symbolic methods In order to enable the generation of counterexamples for very large input DTMCs, the computation of critical subsystems was adapted for *symbolic graph representations*, in particular BDDs and MTBDDs, see Section 2.

The framework for the symbolic method is the same as depicted in Figure 8, while special attention is required regarding certain properties of BDDs. As methods to find new states to extend a subsystem, symbolic versions of the *global search* and the *local search* were devised. This was done for both *bounded model checking* and *symbolic graph search algorithms*; for an introduction to the underlying concepts see Section 4. The adaptions were first proposed in [31] and improved and extended in [32].

Recall, that a DTMC $\mathcal{D} = (S, s_{\text{init}}, P, L)$ together with a set of target states $T \subseteq S$ is symbolically represented by a BDD $\hat{I}$ representing the initial state s_{init}, a BDD $\hat{T}$ representing the target states T and an MTBDD $\hat{P}$ representing the transition probability matrix P. In the symbolic algorithms, an MTBDD *SubSys* is maintained which stands for the current subsystem. The goal of all methods given in the following is to compute a set of states that is used to extend the current subsystem, saved in a BDD *NewStates*. The subsystem is verified by a symbolic version of the standard DTMC model checking procedure, see [64, 65]. This is also used in PRISM [12].

Bounded Model Checking Using the bounded model checking approach from [19] for DTMCs in combination with the incremental generation of subsystems, this directly yields a global search approach for symbolic graph structures. Recall Formula 1 from Section 4.1, where from the (MT)BDDs $\hat{I}$, $\hat{T}$ and $\hat{P}$ predicates init, target and trans are created. In every iteration, the SAT solver computes a path of the DTMC starting at $\hat{I}$ and ending in a state of $\hat{T}$ using transitions of $\hat{P}$. This is achieved by satisfaction of the corresponding predicates. *NewStates* is assigned the states of this path and *SubSys* is extended accordingly. This goes on until model checking reports that the subsystem has enough probability mass.

In contrast to adapting the global search, for the *local search*, also referred to as *fragment search*, we need predicates that are changed *dynamically*. This is due to the fact that in each iteration a path starting at any state of the current subsystem and ending in such a state is to be searched for. As *SubSys* is changed all throughout the process, we need a predicate K that captures this changing set of states. This is technically achieved by utilizing the *assumption* functionality of the SAT solver in the sense that in every iteration the predicate K is satisfied if the SAT solver assigns its variables such that a state of the current subsystem corresponds to these variable values. The goal is now to find paths of arbitrary but bounded length n by assigning the variable sets $v_0, \ldots, v_n$ such that they correspond to such a path. The formula reads as follows:

$$\mathsf{K}(v_0) \ \wedge \ \mathsf{trans}(v_0, v_1) \ \wedge \neg\mathsf{K}(v_1) \ \wedge \ \bigvee_{i=2}^{n} \mathsf{K}(v_i)$$
$$\wedge \bigwedge_{j=1}^{n-1} [(\neg\mathsf{K}(v_j) \rightarrow \mathsf{trans}(v_j, v_{j+1})) \ \wedge \ (\mathsf{K}(v_j) \rightarrow v_j = v_{j+1})]$$

Intuitively, every path starts in a state of K ($\mathsf{K}(v_0)$). From this state, a transition ($\mathsf{trans}(v_0, v_1)$) has to be taken to a state that is *not* part of K ($\neg\mathsf{K}(v_1)$). One of the following states has to be part of K again ($\bigvee_{i=2}^{n} \mathsf{K}(v_i)$). For all states it has to hold that as long as a state is not part of K, a transition is taken to another state ($\neg\mathsf{K}(v_j) \rightarrow \mathsf{trans}(v_j, v_{j+1})$). As soon as a state is inside K, all following variables are assigned the same values creating an implicit self-loop on this state ($\mathsf{K}(v_{j-1}) \rightarrow v_j = v_{j+1}$). For more technical details such as the handling of the initial path starting in the initial state s_{init} and ending in a target state or how to actually form the set K, we refer to the original publications. In addition, a heuristic was given guiding the SAT solver to assign variables such that more probable paths are found.

Symbolic Graph Search In Section 4.1 we described how the k shortest path search was implemented symbolically [21]. The key ingredients were a symbolic version of Dijkstra's shortest path, called *flooding Dijkstra*, and the method `Change(`$\hat{P}, \hat{I}, \hat{T}, SP$`)` which transformed the input system given by $\hat{P}, \hat{I}$ and $\hat{T}$ with respect to the current shortest path SP such that SP is not found any more by the flooding Dijkstra. Instead, the second most probable path in the context of the original system is returned in this modified system. This process is iterated until the sufficient number of paths is achieved. For the shortest path algorithm we write

`ShortestPath(`$\hat{P}, \hat{I}, \hat{T}$`)`

for paths that have transitions out of $\hat{P}$, start in $\hat{I}$ and end in $\hat{T}$.

Although this is conceptually working, the transition MTBDD $\hat{P}$ is basically doubled in each step by the graph transformation. This causes an exponential blow-up of the MTBDD-size which renders this approach not applicable for

relevant benchmarks. Therefore, a straightforward adaption to the generation of critical subsystems is not feasible.

In modification called the *adaptive global search*, the method for changing the graph was used in a different way. Instead of incrementally changing the system according to the current shortest path, in each step the *original system* $(\hat{P}, \hat{I}, \hat{T})$ is transformed such that the new shortest path will have only states that are not already part of *SubSys*:

$$(\hat{P}, \hat{I}, \hat{T}) := \texttt{Change}(\hat{P}, \hat{I}, \hat{T}, \mathit{SubSys})$$

Thereby, the size of the system only increases linearly in each step. Additionally, the flooding Dijkstra computes not only one shortest path but actually the set of all shortest paths that have the same probability and length. All of these paths are added to the current subsystem at once. If the probability mass is exceeded extensively, the *adaptive* algorithm performs a backtracking.

An adaption of the fragment search to symbolic graph algorithms was also done. Consider a BDD *SubSysStates* which represents the states of the current subsystem. Then, in every iteration the shortest path starting and ending in states of the subsystem via transitions from the original system without the subsystem is computed:

$$\texttt{ShortestPath}(\hat{P} \setminus \mathit{SubSys}, \mathit{SubSysStates}, \mathit{SubSysStates})$$

These symbolic graph algorithms enabled the generation of counterexamples for input DTMCs with billions of states in their explicit representation.

Compact representations Based on a hierarchical SCC abstraction presented in [66], the authors of [29] proposed a method for generating *hierarchical counterexamples* for DTMCs. The starting point is an abstract model, for which a critical subsystem is computed (in [29] the local and global search from Section 5.1 are used, but any other approach could be also applied). In order to explore the system in more detail, important parts of the critical subsystem can be concretized and, to reduce its size, in this concretized system again a critical subsystem can be determined. This allows to search for counterexamples on very large input graphs, as the abstract input systems are both very small and simply structured. As concretization up to the original system can be done only in certain parts of interest, no information is lost while only a fraction of the whole system has to be explored.

We first describe the basic idea of the *SCC abstraction* from [66], which can also be used for model checking. The underlying graph of the DTMC gets hierarchically decomposed first into its SCCs, then the SCCs into sub-SCCs not containing the input states and so on, until at the inner-most levels no further non-trivial sub-SCCs exist.

Example 30. The hierarchical SCC decomposition of the DTMC in Figure 2 on page 16 is illustrated in Figure 12, where the SCCs and sub-SCCs are indicated

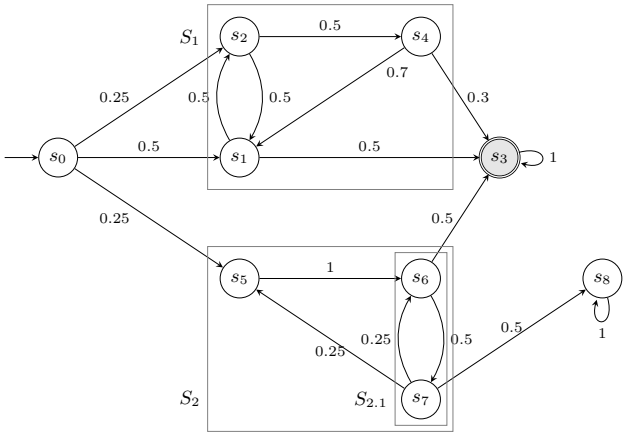


Fig. 12. The SCC decomposition of a DTMC $\mathcal{D}$ (cf. Example 30)

by rectangles. When neglecting its input states, the SCC $S_1 = \{s_1, s_2, s_4\}$ does not have any sub-SCCs. The SCC $S_2 = \{s_5, s_6, s_7\}$ contains a sub-SCC $S_{2.1} = \{s_6, s_7\} \subseteq S_2$. ■

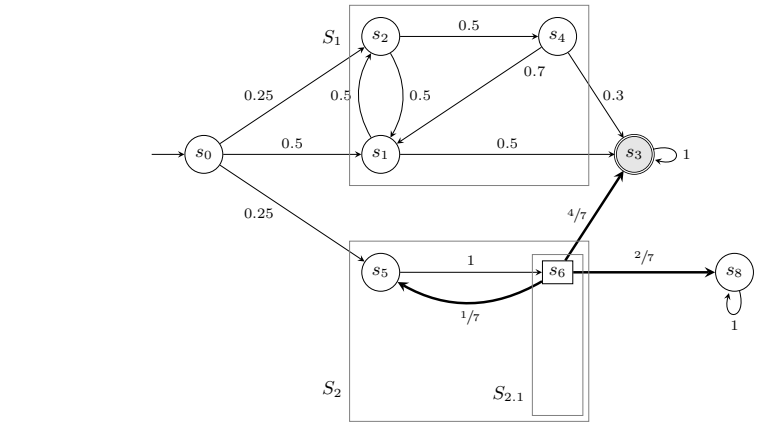
In a bottom-up traversal starting at the inner-most sub-SCCs, the reachability probabilities $p_{\text{abs}}(s, s')$ from each input state s to each output state s' inside the given sub-SCC are computed by utilizing certain properties of DTMCs. This computation was inspired by the work of Andrés, D'Argenio and van Rossum [24]. All non-input nodes and all transitions inside the sub-SCC are removed and abstract transitions are added from each input state s to each output state s' carrying the whole probability mass $p_{\text{abs}}(s, s')$. Note that the probability to reach target states from the initial state in the resulting DTMC equals the probability in the DTMC before the transformation.

Example 31. Consider the hierarchically decomposed DTMC $\mathcal{D}$ in Figure 12. In Figure 13(a), the abstraction of the sub-SCC $S_{2.1}$ is shown. Basically, the sub-SCC is abstracted by a single abstract state s_6. We denote such abstract states by a rectangular shape, and abstract transitions by thick lines. The abstract probabilities are:

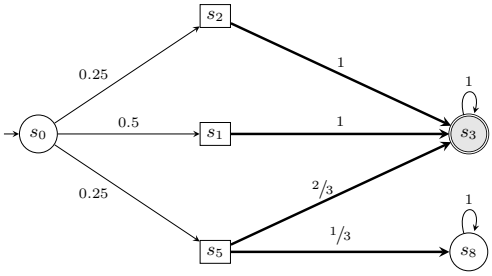
$$p_{\text{abs}}(s_6, s_3) = 4/7 \quad p_{\text{abs}}(s_6, s_5) = 1/7 \quad p_{\text{abs}}(s_6, s_8) = 2/7 \,.$$

At the next outer level, now the SCC S_2 can be abstracted. After abstracting also SCC S_1, an acyclic graph remains which is depicted in Figure 13(b). Note that SCC S_1 results in two abstract states as it has two input states, i. e., states that have an incoming transitions from outside the SCC.

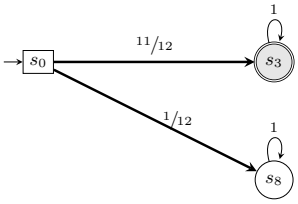
It is easy to compute the abstract probabilities therein, e. g., by solving the simple linear equation system as explained in Section 2. The resulting abstract graph is depicted in Figure 13(c). Here, only transitions from the initial state



(a) Abstraction of SCC $S_{2.1}$



(b) Acyclic graph after abstracting all SCCs



(c) Model checking result

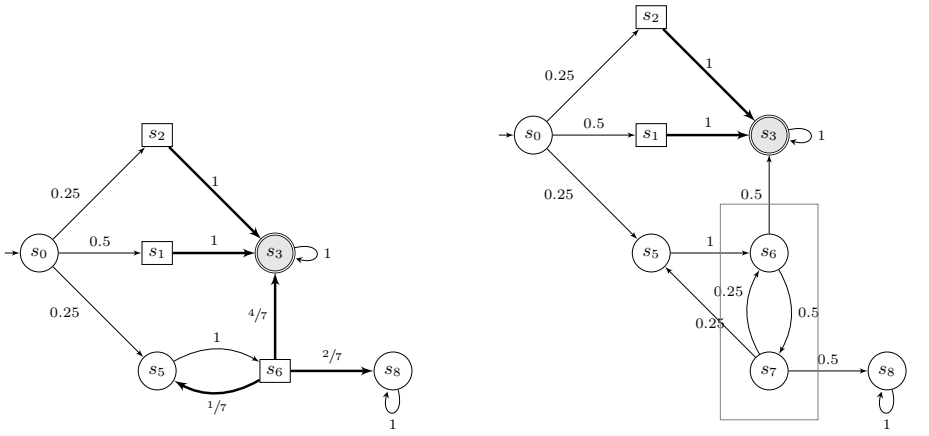
Fig. 13. SCC-based model checking (cf. Example 31)

s_0 to all absorbing states including the target state s_3 are contained. As this corresponds to the probability of reaching state s_3 in the original DTMC $\mathcal{D}$, we have the model checking result for the reachability property:

$$\Pr^{\mathcal{D}}\big(\Diamond\{s_3\}\big) = p_{\text{abs}}(s_1, s_3) = {}^{11}/_{12} \approx 0.9167\,.$$

■

The key idea of the hierarchical counterexample generation is now to start the search on the abstract graph. Following the general procedure as depicted in Figure 8, states are collected using path search algorithms until the subsystem has enough probability mass to be critical. If the resulting critical subsystem gives enough debugging information, the process terminates, otherwise certain abstract states, i. e., abstracted SCCs or sub-SCCs, inside the abstract critical subsystem can be *concretized* with respect to the former SCC abstraction and a new search can be started on this more detailed system. The choice of one or more states to be concretized can either be done interactively by user input or guided by certain heuristics.



(a) Abstract DTMC $\mathcal{D}$ (b) DTMC $\mathcal{D}$ after concretization of s_6

Fig. 14. Concretizing state s_6 (cf. Example 32)

Example 32. To explain the procedure of concretizing states, consider a partially abstracted version of the DTMC from Figure 12 depicted in Figure 14(a), where states s_0 and s_5 are concretized, but s_1, s_2 and s_6 are still abstract. Assume now, s_6 is chosen to be concretized. All abstract transitions leaving s_6 are removed and the SCC which was abstracted by them is inserted. The result of the concretization step is depicted in Figure 14(b). ■

5.2 Critical Subsystems for PAs

Let $\mathcal{M} = (S, s_{\text{init}}, \text{Act}, \hat{P}, L)$ be a PA and $\mathbb{P}_{\leq\lambda}(\Diamond T)$ be a reachability property which is violated by $\mathcal{M}$. We want to compute a smallest critical subsystem for PA, which is an NP-hard problem. The approach below [27] is formalized for reachability properties, but it can be extended to ω-regular properties.

The main difference to DTMCs is that the MILP has to be enriched by the computation of an appropriate scheduler. Please note that a scheduler that maximizes the reachability probability does not necessarily induce a DTMC having a critical subsystem which is minimal among all critical subsystems of the PA. Hence, we cannot compute a scheduler beforehand, but have to integrate the scheduler computation into the MILP.

Doing so we have to take into account that, for some state $s \in S$, under some schedulers the target states T can be unreachable from s, but reachable for other schedulers. Such states are called *problematic*. Let

$$S_p = \{s \in S \mid \exists \sigma \in \text{Sched}_{\mathcal{M}} : \Pr_s^{\mathcal{M}^\sigma}(\Diamond T) = 0\}$$

be the set of all problematic states and let S_p^+ the set of all problematic states and their successors. If $s \notin S_p$ then s is called *unproblematic* for T. A transition

$(\alpha, \mu) \in \hat{P}(s)$ for some $s \in S$ is problematic if all its successor states are problematic:

$$\hat{P}_p = \left\{ (s, \alpha, \mu) \,|\, (\alpha, \mu) \in \hat{P}(s) \wedge \operatorname{supp}(\mu) \subseteq S_p \right\} .$$

Problematic states and transitions can be determined in linear time in the size of the PA using standard graph algorithms [9].

As before, we need variables $x_s \in \{0,1\} \subseteq \mathbb{Z}$ and $p_s \in [0,1] \cap \mathbb{Q}$ for each state $s \in S$. Additionally we need variables $\sigma_{s,\alpha,\mu} \in \{0,1\} \subseteq \mathbb{Z}$ for $s \in S \setminus T$ and $(\alpha, \mu) \in \hat{P}(s)$ to encode the chosen scheduler. We have to add constraints which ensure that from each selected problematic state $s \in S_p$ a target state is reachable within the selected subsystem. For this we need further variables: $r_s \in [0,1] \cap \mathbb{Q}$ for problematic states and their successors $s \in S_p^+$, and $t_{s,s'} \in \{0,1\} \subseteq \mathbb{Z}$ for each pair (s, s') such that there is $(s, \alpha, \mu) \in \hat{P}_p$ with $s' \in \operatorname{supp}(\mu)$.

With these variables the MILP is given as follows:

$$\text{minimize} \quad -\frac{1}{2}\, p_{s_{\text{init}}} + \sum_{s \in S} x_s \tag{3a}$$

such that

$$p_{s_{\text{init}}} > \lambda \tag{3b}$$

$$\forall s \in T: \quad p_s = x_s \tag{3c}$$

$$\forall s \in S \setminus T: \quad p_s \leq x_s \tag{3d}$$

$$\forall s \in S \setminus T: \quad \sum_{(\alpha,\mu) \in \hat{P}(s)} \sigma_{s,\alpha,\mu} = x_s \tag{3e}$$

$$\forall s \in S \setminus T \; \forall (\alpha, \mu) \in \hat{P}(s): \quad p_s \leq (1 - \sigma_{s,\alpha,\mu}) + \sum_{s' \in \operatorname{supp}(\mu)} \mu(s') \cdot p_{s'} \tag{3f}$$

$$\forall (s, \alpha, \mu) \in \hat{P}_p \; \forall s' \in \operatorname{supp}(\mu): \quad t_{s,s'} \leq x_{s'} \tag{3g}$$

$$\forall (s, \alpha, \mu) \in \hat{P}_p \; \forall s' \in \operatorname{supp}(\mu): \quad r_s < r_{s'} + (1 - t_{s,s'}) \tag{3h}$$

$$\forall (s, \alpha, \mu) \in \hat{P}_p: \quad (1 - \sigma_{s,\alpha,\mu}) + \sum_{s' \in \operatorname{supp}(\mu)} t_{s,s'} \geq x_s \; . \tag{3i}$$

The objective function (3a) and the constraints (3b), (3c), and (3d) are the same as for DTMCs. Constraint (3e) takes care that the scheduler selects exactly one pair $(\alpha, \mu) \in P(s)$ for each state $s \in S$ that is contained in the subsystem, and none for states not in the subsystem. Constraint (3f) is the pendant to constraint (2d) of the MILP for DTMCs. The difference is the term $(1 - \sigma_{s,\alpha,\mu})$. It ensures that the constraint is trivially satisfied for all transitions that are not selected by the scheduler. The remaining constraints (3g)–(3i) take care that from each problematic state a non-problematic (and therefore a target) state is reachable within the selected subsystem. For details on these reachability constraints we refer the reader to [61].

Example 33. Consider the same MDP in Figure 7 (page 28) and the reachability property $\mathbb{P}_{\leq 0.75}(\Diamond\{s_4\})$ as in Example 23 (page 27). To compute a smallest

critical subsystem, we first remove the irrelevant states s_3 and s_5 from the model. Note that the resulting MDP has no problematic states. Since the considered model is an MDP, for readability we write $\sigma_{s,\alpha}$ instead of $\sigma_{s,\alpha,\mu}$ for the scheduler choices in the following MILP formulation:

$$\text{minimize} \quad -\tfrac{1}{2}p_{s_0} + (x_{s_0} + x_{s_1} + x_{s_2} + x_{s_4}) \text{ such that}$$

$$\begin{array}{ll} & p_{s_4} = x_{s_4} \\ & p_{s_0} \leq x_{s_0} \\ p_{s_0} > 0.75 & p_{s_1} \leq x_{s_1} \\ & p_{s_2} \leq x_{s_2} \\ & p_{s_0} \leq (1 - \sigma_{s_0,\alpha}) + p_{s_1} \\ \sigma_{s_0,\alpha} + \sigma_{s_0,\beta} = x_{s_0} & p_{s_0} \leq (1 - \sigma_{s_0,\beta}) + p_{s_2} \\ \sigma_{s_1,\tau} = x_{s_1} & p_{s_1} \leq (1 - \sigma_{s_1,\tau}) + 0.4p_{s_0} + 0.5p_{s_4} \\ \sigma_{s_2,\tau} = x_{s_2} & p_{s_2} \leq (1 - \sigma_{s_2,\tau}) + 0.5p_{s_0} + 0.4p_{s_4} \end{array}$$

The assignment mapping (i) 1 to x_{s_0}, x_{s_1}, x_{s_4}, p_{s_4}, $\sigma_{s_0,\alpha}$ and $\sigma_{s_1,\tau}$, (ii) $5/6 \approx 0.83$ to p_{s_0} and p_{s_1}, and (iii) 0 to all other variables is a solution to the above constraint system, specifying a scheduler choosing action α in state s_0, and determining a smallest critical subsystem of the induced DTMC with state set $\{s_0, s_1, s_4\}$. ■

Example 34. To illustrate the need for the special handling of problematic states, consider again the example MDP in Figure 7 on page 28, but assume that at state s_0 there would be an additional distribution with action γ, looping on s_0 with probability 1.

Without the constraints (3g)–(3i), the assignment mapping (i) 1 to x_{s_0}, p_{s_0} and $\sigma_{s_0,\gamma}$ and (ii) 0 to all other variables would satisfy the remaining MILP constraints, however, it would specify a subsystem containing the single state s_0, i. e., having the probability 0 to reach the target state s_4.

The MILP with the constraints (3g)–(3i) exclude this possibility. Intuitively, (3g)–(3i) exclude subsystems that have a bottom SCC containing problematic states only. ■

6 Description-Language-Based Counterexamples

Typically, probabilistic models are not explicitly given at the state space level, but rather in a symbolic format that is able to succinctly capture large and complex system behavior.

`Prism`'s guarded command language One example of such a symbolic modeling formalism is the *guarded-command language* employed by the well-known

probabilistic model checker `Prism` [12]. It is a stochastic variant of Alur and Henzinger's reactive modules [33].

In this language, a *probabilistic program* consists of a set of *modules*. Each module declares a set of module *variables* and a set of *guarded commands*. A module has read and write access to its own variables, but only read access to the variables of other modules. The guarded commands have the form

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

where α is a *command label* being either an action name or τ, the *guard* g is a predicate over the variables in the program, f_i are the *variable update functions* that specify how the values of the module's variables are changed by the command, and the p_i are the *probabilities* with which the corresponding updates happen. A command with action label τ is executed asynchronously in the sense that no other command is executed simultaneously. In contrast, commands labeled with an action name $\alpha \neq \tau$ synchronize with all other modules that also have a command with this label, i. e., each of them executes a command with label α simultaneously.

Example 35. The top of Figure 15 shows an example guarded-command program in `Prism`'s input language. The program involves two modules `coin` and `processor`.

Initially the module `coin` can (asynchronously) do a coin flip (command c_1). The variable f stores the fact whether the coin has been already flipped ($f = 1$) or not ($f = 0$). After the coin flip, the variable c stores whether the coin shows tails ($c = 0$) or heads ($c = 1$)

After the coin flip, both modules can process some data by synchronizing on the `proc` action (c_3 and c_4). The variable p is used to make a bookkeeping whether processing has taken place ($p = 1$) or not ($p = 0$). However, the processing step can by mistake set the coin to show heads with probability 0.01 (c_3).

Additionally, if the coin is flipped and it shows tails, the coin flip can be undone by the synchronizing `reset` action (c_2 and c_6), leading the system back to its initial state.

Finally, if data has been processed the system may loop forever (c_5). ■

Several modules can be casted to a single module using *parallel composition*. The variable set of the composition is the union of the variable sets of the composed modules. Each non-synchronizing command is also a command in the composition. For each combination of synchronizing commands, the composition contains a single command whose guard is the conjunction of the involved guards, and whose updates are all possible combinations of joint updates with the product of the involved probabilities.

Example 36. The parallel composition of the two modules of the example probabilistic program at the top of Figure 15 is given at the bottom of the same figure. ■

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

module `coin` $\|$ `processor`
f: **bool init** 0; c: **bool init** 0; p: **bool init** 0;
`[`τ`]` $\neg f \to 0.5 : (f' = 1)\&(c' = 1) + 0.5 : (f' = 1)\&(c' = 0);$ $(\hat{c}_1)$
`[reset]` $f \wedge \neg c \to 1 : (f' = 0)\&(p' = 0);$ $(\hat{c}_2)$
`[proc]` $f \wedge \neg p \to 0.99 : (f' = 1)\&(p' = 1) + 0.01 : (c' = 1)\&(p' = 1);$ $(\hat{c}_3)$
`[`τ`]` $p \to 1 : (p' = 1);$ $(\hat{c}_4)$
endmodule

Fig. 15. Top: The probabilistic program from Example 35, specified in `Prism`'s guarded-command language; Bottom: The parallel composition of the two modules

The *semantics* of a module is given in terms of a probabilistic automaton [47]. The state space of the automaton is the set of all valuations of the variables that appear in the program. The transitions between states are determined by the module's commands. More specifically, for every state and every guarded command with label α whose guard evaluates to true in the given state, the transition relation contains a pair (α, μ) such that μ defines for each update a transition to the state after the update with the probability of the update.

Example 37. The (reachable part of the) probabilistic automaton specifying the meaning of the probabilistic program in Figure 15 is depicted in Figure 16.

Note that the coin will finally show heads with probability 1 for all schedulers which choose the `reset` action with a non-zero probability if it is enabled. This behavior can be modified by, e. g., defining a scheduler with memory, which bounds the number of `reset` executions by some finite bound. ■

A probabilistic program satisfies a reachability property iff the PA specifying its semantics does so. Thus explanations for the violation could be given by path- and subsystem-based counterexamples at the state-space level. However, such counterexamples tend to be too large and structureless to be easily interpretable in the probabilistic program, and therefore they are not well suited to help the designer to eliminate the unwanted behavior at the command level of modules.

Therefore, [34] proposed to naturally extend the computation of smallest critical subsystems to probabilistic programs by determining a *subset of the*

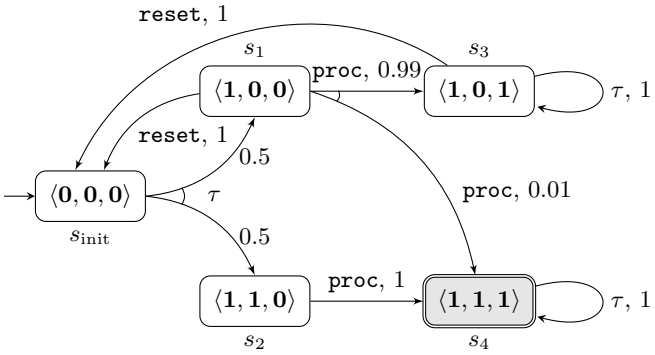


Fig. 16. The PA specifying the semantics of the probabilistic program in Figure 15 (cf. Example 37)

commands that gives rise to a sub-PA that still violates the property in question. More precisely, the task is to compute a minimal number of commands such that the reachability probability in the semantics of the restricted program exceeds the threshold λ. Moreover, we aim at finding a *smallest critical command set* which maximizes the reachability probability under all subsets of the commands. It thus acts as a counterexample by pointing to a set of commands that already generate erroneous behavior. Additionally, to increase usefulness, the commands in a smallest critical command set can be further reduced by removing some of their update branches without which the property is still violated.

Example 38. Consider our example probabilistic program from Figure 15 and a reachability property $\mathbb{P}_{\leq\lambda}\big(\Diamond\{s_4\}\big)$ (where s_4 describes the state in which all variables evaluate to 1).

If $0 < \lambda < 0.5$, at the level of the composed module `coin` $\|$ `processor` (at the bottom of Figure 15), the commands $\hat{c}_1$ and $\hat{c}_3$ would build a smallest critical command set. At the level of the modules `coin` and `processor`, we need to include c_1, c_3 and c_4.

For $\lambda \geq 0.5$, a smallest critical command set would be $\{\hat{c}_1, \hat{c}_2, \hat{c}_3\}$ at the composed level `coin` $\|$ `processor`. At the level of the modules, we can only exclude c_5. ■

Linear programming approach In [34], the authors show the problem of finding a smallest critical command set to be NP-hard and present a *mixed integer linear programming* (*MILP*) approach to solve it. The basic idea is to describe the PA semantics of a smallest critical command set of a probabilistic program, together with a maximal scheduler, by an MILP formulation. This MILP formulation can be disposed to a state-of-the-art MILP solver to get an optimal solution.

Assume a probabilistic program and let $\mathcal{M} = (S, s_{\text{init}}, \text{Act}, \hat{P}, L)$ be the PA generated by it after removing all irrelevant states, and assume that the

reachability property $\mathbb{P}_{\leq\lambda}(\Diamond T)$ is violated by $\mathcal{M}$. For each state $s \in S$ and transition $(\alpha, \mu) \in \hat{P}(s)$ let $L(s, \alpha, \mu)$ denote the set of commands that generate the given transition.[6] Note that in case of synchronization several commands together create a certain transition.

The idea to encode the selection of smallest critical command sets as an MILP problem is similar to the MILP encoding of smallest critical subsystems for PAs (see Section 5.1). However, now we want to select a minimal number of commands of a probabilistic program instead of a minimal number of states of a PA. The selected commands should induce a PA, for which there is a memoryless deterministic scheduler inducing a critical subsystem of the PA.

Additionally to the variables used for the smallest critical subsystem encoding for PAs, we encode the selection of a smallest critical command set using a variable $x_c \in \{0, 1\}$ for each command c, which is 1 iff c is part of the smallest critical command set. Using these variables, the MILP for a smallest critical command set is as follows:

$$\text{minimize} \quad -\frac{1}{2} \cdot p_{s_{\text{init}}} + \sum_c x_c \tag{4a}$$

such that

$$p_{s_{\text{init}}} > \lambda \tag{4b}$$

$$\forall s \in S \setminus T: \quad \sum_{(\alpha,\mu)\in P(s)} \sigma_{s,\alpha,\mu} \leq 1 \tag{4c}$$

$$\forall s \in S\ \forall (\alpha, \mu) \in P(s)\ \forall c \in L(s, \alpha, \mu): \quad x_c \geq \sigma_{s,\alpha,\mu} \tag{4d}$$

$$\forall s \in T: \quad p_s = 1 \tag{4e}$$

$$\forall s \in S \setminus T: \quad p_s \leq \sum_{(\alpha,\mu)\in P(s)} \sigma_{s,\alpha,\mu} \tag{4f}$$

$$\forall s \in S \setminus T\ \forall (\alpha, \mu) \in P(s): \quad p_s \leq \sum_{s'\in \text{supp}(\mu)} \mu(s') \cdot p_{s'} + (1 - \sigma_{s,\alpha,\mu}) \tag{4g}$$

$$\forall (s, \alpha, \mu) \in \hat{P}_p: \quad \sigma_{s,\alpha,\mu} \leq \sum_{s'\in \text{supp}(\mu)} t_{s,s'} \tag{4h}$$

$$\forall s \in S_p\ \forall (\alpha, \mu) \in P(s)\ \forall s' \in \text{supp}(\mu): \quad r_s < r_{s'} + (1 - t_{s,s'})\ . \tag{4i}$$

By (4b) we ensure that the the subsystem induced by the selected scheduler is critical. For reachability properties, we can restrict ourselves to memoryless deterministic schedulers. So for each state at most one action-distribution pair is selected by the scheduler (4c). Note that there may be states where no such pair is chosen, which we call deadlocking. If the scheduler selects an action-distribution pair, all commands involved in its generation have to be chosen (4d). For all target states $s \in T$ the probability p_s is set to 1 (4e), while the probability is set

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

to zero for all deadlocking non-target states (4f). Constraint (4g) is responsible for assigning a valid probability to p_s under the selected scheduler. The constraint is trivially satisfied if $\sigma_{s,\alpha,\mu} = 0$. If (α, μ) is selected, the probability p_s is bounded from above by the probability to go to one of the successor states of (α, μ) and to reach the target states from there.

For non-deadlocking problematic states, the reachability of at least one unproblematic state is ensured by (4h) and (4i). First, for every state s with a selected (α, μ) 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 or deadlocking state.

These constraints enforce that each satisfying assignment corresponds to a critical command set. By minimizing the number of the selected commands we obtain a size-minimal critical command set. By the additional term $-\frac{1}{2} \cdot p_{s_{\text{init}}}$ we obtain a smallest critical command set. The coefficient $-\frac{1}{2}$ is needed to ensure that the benefit from maximizing the probability is smaller than the loss by adding an additional command.

Example 39. We want to compute a smallest critical command set for the example probabilistic program in Figure 15 and $\mathbb{P}_{\leq 0.4}\big(\Diamond\{s_4\}\big)$. Since the induced PA is an MDP, for readability we write $\sigma_{s,\alpha}$ instead of $\sigma_{s,\alpha,\mu}$ for the scheduler choices in the following MILP formulation:

$$\text{minimize} \quad -\tfrac{1}{2}p_{s_{\text{init}}} + (x_{c_1} + x_{c_2} + x_{c_3} + x_{c_4} + x_{c_5} + x_{c_6}) \text{ such that}$$

$$p_{s_{\text{init}}} > 0.4$$

$$\begin{array}{ll}
\sigma_{s_{\text{init}},\tau} \leq 1 & \sigma_{s_{\text{init}},\tau} \leq x_{c_1} \quad \sigma_{s_2,\texttt{proc}} \leq x_{c_3} \\
\sigma_{s_1,\texttt{proc}} + \sigma_{s_1,\texttt{reset}} \leq 1 & \sigma_{s_1,\texttt{proc}} \leq x_{c_3} \quad \sigma_{s_2,\texttt{proc}} \leq x_{c_4} \\
\sigma_{s_2,\texttt{proc}} \leq 1 & \sigma_{s_1,\texttt{proc}} \leq x_{c_4} \quad \sigma_{s_3,\texttt{reset}} \leq x_{c_2} \\
\sigma_{s_3,\texttt{reset}} + \sigma_{s_3,\tau} \leq 1 & \sigma_{s_1,\texttt{reset}} \leq x_{c_2} \; \sigma_{s_3,\texttt{reset}} \leq x_{c_6} \\
 & \sigma_{s_1,\texttt{reset}} \leq x_{c_6} \; \sigma_{s_3,\tau} \leq x_{c_5}
\end{array}$$

$$\begin{array}{ll}
p_{s_4} = 1 & p_{s_{\text{init}}} \leq 0.5p_{s_1} + 0.5p_{s_2} + (1 - \sigma_{s_{\text{init}},\tau}) \\
 & p_{s_1} \leq 0.99p_{s_3} + 0.01p_{s_4} + (1 - \sigma_{s_1,\texttt{proc}}) \\
p_{s_{\text{init}}} \leq \sigma_{s_{\text{init}},\tau} & p_{s_1} \leq p_{s_{\text{init}}} + (1 - \sigma_{s_1,\texttt{reset}}) \\
p_{s_1} \leq \sigma_{s_1,\texttt{proc}} + \sigma_{s_1,\texttt{reset}} & p_{s_2} \leq p_{s_4} + (1 - \sigma_{s_2,\texttt{proc}}) \\
p_{s_2} \leq \sigma_{s_2,\texttt{proc}} & p_{s_3} \leq p_{s_{\text{init}}} + (1 - \sigma_{s_3,\texttt{reset}}) \\
p_{s_3} \leq \sigma_{s_3,\texttt{reset}} + \sigma_{s_3,\tau} & p_{s_3} \leq p_{s_3} + (1 - \sigma_{s_3,\tau})
\end{array}$$

$$\begin{array}{l}
\sigma_{s_3,\tau} \leq t_{s_3,s_3} \\
r_{s_3} < r_{s_3} + (1 - t_{s_3,s_3})
\end{array}$$

It is easy to check that the assignment mapping 1 to $\sigma_{s_{\text{init}},\tau}$, $\sigma_{s_2,\texttt{proc}}$, x_{c_1}, x_{c_3}, x_{c_4} and p_{s_4}, 0.5 to $p_{s_{\text{init}}}$ and p_{s_2}, and 0 to all other variables is a satisfying solution, encoding the smallest critical command set $\{c_1, c_3, c_4\}$. ■

7 Tools and Implementations

In this section we give a short overview on public tools and prototype implementations for some of the approaches that were presented in this paper. We report on the scalability of the different approaches as far as there were comparisons made in the corresponding papers. We first present the publicly available tools.

7.1 DiPro — A Tool for Probabilistic Counterexample Generation

DiPro [36] was the first official tool for the counterexample generation of probabilistic systems. Basically, most of the implemented approaches are based on variations of *best-first search*. An *extended best-first search* is used to generate critical subsystems of DTMCs and CTMCs, see Section 5 and the corresponding paper [30]. Moreover a K^* search [67] for finding the k most probable paths of a DTMC together with some optimizations is implemented, see Section 4. Finally, DiPro is able to compute a path-based counterexample together with a scheduler for MDPs, see Section 5 and [25].

Technically, the best-first search approaches of DiPro are implemented using the simulation engine of a previous version of the probabilistic model checker Prism [12]. Thereby, the state space is built incrementally and in many cases not to its full extend. That enables the generation of counterexamples for rather large graphs for many benchmarks.

In order to help the user understand the process of finding a counterexample, the tool offers a graphical user interface [60] where the search process is illustrated.

7.2 COMICS — Computing Smallest Counterexamples for DTMCs

COMICS [37] implements the approaches of [29], namely the *hierarchical counterexample generation* and the two search approaches called *global search* and *local search*, see Section 5. The core functionality is to offer the computation of counterexamples for reachability properties of DTMCs either automatically or user-guided. A graphical user-interface offers to depict every stage of the hierarchical counterexample generation. The user can interactively choose certain states of interest to be concretized, while there are also several heuristics available to automate this choice. Furthermore, several heuristics can be used for the search process, e. g., how many states to concretize in one step or how often model checking is performed. Moreover, the tool has a mere command-line version in order to perform benchmarking. It is always possible to compute smallest critical subsystems without the hierarchical concretization. Finally, the k shortest path approach, see Section 4 and [17], was implemented in order to provide comparisons regarding scalability.

7.3 Other Implementations

Basically, we did not have access to the implementations on foreign approaches as presented in this paper.

We are able to report on the implementations of the several approaches concerning the computation of *smallest critical subsystems*, see Sections 5 and 6. Parts of these implementations are summarized in a tool called LTLSubsys. The high-level approaches are mainly implemented into the framework of a successor of the probabilistic model checker MRMC [13]. These still prototypical implementations utilize the SMT-solver Z3 [68] and the MILP solvers SCIP [69], CPLEX [70] and GUROBI [71].

Moreover, we describe the scalability of the approaches to symbolic counterexample generation, see Section 5 and the publications [31, 32].

7.4 Comparison of the Tools

We will now shortly report on comparisons of DiPro, COMICS and LTLSubsys that were made in previous publications.

First, COMICS and DiPro were directly compared for reachability properties of DTMCs in [32, 37, 61]. Summarizing the results we observe that for benchmarks with up to one million states COMICS performs better in terms of running times and of the size of the generated subsystem. However, for larger benchmarks DiPro might be the better choice, as the state space is generated on the fly. Thus, if the critical subsystem generated by DiPro is of moderate size, a result is obtained even for very large graphs. Please keep in mind that each tool has its own advantages such as the animated search process of DiPro or the user-guided hierarchical counterexample search of COMICS.

LTLSubsys was compared to both publicly available tools in [61]. In terms of running times, the creation of a *smallest* critical subsystem is almost always worse than the heuristical tools. In terms of the system size, LTLSubsys naturally always generates the smallest possible critical subsystem. For the benchmarks tested in the paper, the *local search* approach in some cases generated critical subsystems that were only around 10% larger than the actual minimal subsystem while the running time was considerably lower. Note finally, that within an MILP solver such as GUROBI, an intermediate solution and a lower bound on the value of the optimal solution is maintained at every time. In many cases, the minimal solution is obtained within seconds while it is a very hard case to actually prove minimality. Thereby, if the intermediate result is already sufficiently small, the search process can be stopped at any time.

The symbolic counterexample generation based on graph algorithms as presented in [32] and Section 5 was compared to COMICS and DiPro. As expected, on smaller benchmarks the other tools perform better in terms of running times. The size of the subsystems was comparable to the results as obtained by COMICS as the same approaches were used only on the one hand for explicit graph representations and on the other hand for symbolic graph representations. For benchmarks with millions of states, DiPro and the symbolic algorithms

were the only ones to obtain results while the latter obtained better running times the larger the benchmarks were. Finally, the symbolic algorithms were able to generate counterexamples for systems with billions of states while all other approaches failed.

8 Conclusion

This paper surveyed state-of-the-art methods for counterexample generation for discrete-time Markov models. Three techniques have been covered: path-based representations, minimal critical subsystems, and high-level representations of counterexamples. In addition to techniques using explicit model representations, we addressed methods that use symbolic BDD-based model representations and symbolic computations.

It is fair to say, that probabilistic counterexamples are still at their infancy. Although dedicated tools such as DiPro and Comics support (some of) the techniques presented in this survey, the integration into mainstream probabilistic model checkers is still open. This could make the usage of probabilistic counterexamples more popular in other application domains like, e. g., robotics or security. Besides, it is a challenging task to consider counterexamples for continuous-time or hybrid probabilistic models, in particular for time-constrained reachability properties.

References

1. Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking. Volume 5000 of LNCS, Springer (2008) 1–26
2. Clarke, E.M., Veith, H.: Counterexamples revisited: Principles, algorithms, applications. In: Verification: Theory and Practice. Volume 2772 of LNCS, Springer (2003) 208–224
3. Fraser, G., Wotawa, F., Ammann, P.: Issues in using model checkers for test case generation. Journal of Systems and Software **82**(9) (2009) 1403–1418
4. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Performance Evaluation Review **32**(4) (2005) 34–40
5. Ngo, T.M., Stoelinga, M., Huisman, M.: Effective verification of confidentiality for multi-threaded programs. Journal of Computer Security **22**(2) (2014) 269–300
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (2003) 752–794
7. Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: Proc. of LICS, IEEE Computer Society Press (2002) 19–29
8. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: Proc. of DAC. (1995) 427–432
9. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
10. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Performance evaluation and model checking join forces. Commun. ACM **53**(9) (2010) 76–85

11. Kwiatkowska, M.Z.: Model checking for probability and time: From theory to practice. In: Proc. of LICS, IEEE Computer Society Press (2003) 351–360
12. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Proc. of CAV. Volume 6806 of LNCS (2011) 585–591
13. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. **68**(2) (2011) 90–104
14. Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic model checking. In: Lecture notes of SFM. Volume 4486 of LNCS, Springer (2007) 220–270
15. Katoen, J.P.: Model checking meets probability: A gentle introduction. In: Engineering dependable software systems. Volume 34 of NATO Science for Peace and Security Series - D: Information and Communication Security, IOS Press, Amsterdam (2013) 177–205
16. Han, T., Katoen, J.P.: Counterexamples in probabilistic model checking. In: Proc. of TACAS. Volume 4424 of LNCS, Springer (2007) 72–86
17. Han, T., Katoen, J.P., Damman, B.: Counterexample generation in probabilistic model checking. IEEE Transactions on Software Engineering **35**(2) (2009) 241–257
18. Aljazzar, H., Leue, S.: Extended directed search for probabilistic timed reachability. In: Proc. of FORMATS. Volume 4202 of LNCS, Springer (2006) 33–51
19. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Proc. of VMCAI. Number 5403 in LNCS, Springer (2009) 366–380
20. Braitling, B., Wimmer, R., Becker, B., Jansen, N., Ábrahám, E.: Counterexample generation for Markov chains using SMT-based bounded model checking. In: Proc. of FMOODS/FORTE. IFIP Advances in Information and Communication Technology, Springer (2011)
21. Günther, M., Schuster, J., Siegle, M.: Symbolic calculation of k-shortest paths and related measures with the stochastic process algebra tool CASPA. In: Proc. of DYADEM-FTS, ACM Press (2010) 13–18
22. Damman, B., Han, T., Katoen, J.P.: Regular expressions for PCTL counterexamples. In: Proc. of QEST, IEEE Computer Society Press (2008) 179–188
23. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Proc. of ICTAC. Volume 3407 of LNCS, Springer (2005) 280–294
24. Andrés, M.E., D'Argenio, P., van Rossum, P.: Significant diagnostic counterexamples in probabilistic model checking. In: Proc. of HVC. Number 5394 in LNCS, Springer (2008) 129–148
25. Aljazzar, H., Leue, S.: Generation of counterexamples for model checking of Markov decision processes. In: Proc. of QEST, IEEE Computer Society Press (2009) 197–206
26. Chadha, R., Viswanathan, M.: A counterexample-guided abstraction-refinement framework for Markov decision processes. ACM Transactions on Computational Logic **12**(1) (2010) 1–45
27. Wimmer, R., Becker, B., Jansen, N., Ábrahám, E., Katoen, J.P.: Minimal critical subsystems for discrete-time Markov models. In: Proc. of TACAS. Volume 7214 of LNCS, Springer (2012) 299–314
28. 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
29. Jansen, N., Ábrahám, E., Katelaan, J., Wimmer, R., Katoen, J.P., Becker, B.: Hierarchical counterexamples for discrete-time Markov chains. In: Proc. of ATVA. Volume 6996 of LNCS, Springer (2011) 443–452

30. Aljazzar, H., Leue, S.: Directed explicit state-space search in the generation of counterexamples for stochastic model checking. IEEE Transactions on Software Engineering **36**(1) (2010) 37–60
31. Jansen, N., Ábrahám, E., Zajzon, B., Wimmer, R., Schuster, J., Katoen, J.P., Becker, B.: Symbolic counterexample generation for discrete-time Markov chains. In: Proc. of FACS. Volume 7684 of LNCS, Springer (2012) 134–151
32. Jansen, N., Wimmer, R., Ábrahám, E., Zajzon, B., Katoen, J.P., Becker, B., Schuster, J.: Symbolic counterexample generation for large discrete-time Markov chains. Science of Computer Programming (2014) (accepted for publication).
33. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design **15**(1) (1999) 7–48
34. Wimmer, R., Jansen, N., Vorpahl, A., Ábrahám, E., Katoen, J.P., Becker, B.: High-level counterexamples for probabilistic automata. In: Proc. of QEST. Volume 8054 of LNCS, Springer (2013) 18–33
35. Katoen, J.P., van de Pol, J., Stoelinga, M., Timmer, M.: A linear process-algebraic format with data for probabilistic automata. Theor. Comput. Sci. **413**(1) (2012) 36–57
36. Aljazzar, H., Leitner-Fischer, F., Leue, S., Simeonov, D.: DiPro – A tool for probabilistic counterexample generation. In: Proc. of SPIN. Volume 6823 of LNCS, Springer (2011) 183–187
37. Jansen, N., Ábrahám, E., Volk, M., Wimmer, R., Katoen, J.P., Becker, B.: The COMICS tool – Computing minimal counterexamples for DTMCs. In: Proc. of ATVA. Volume 7561 of LNCS, Springer (2012) 349–353
38. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Proc. of CAV. Volume 5123 of LNCS, Springer (2008) 162–175
39. Komuravelli, A., Pasareanu, C.S., Clarke, E.M.: Assume-guarantee abstraction refinement for probabilistic systems. In: Proc. of CAV. Volume 7358 of LNCS, Springer (2012) 310–326
40. Grunske, L., Winter, K., Yatapanage, N., Zafar, S., Lindsay, P.A.: Experience with fault injection experiments for FMEA. Softw. Pract. Exper. **41**(11) (2011) 1233–1258
41. Aljazzar, H., Fischer, M., Grunske, L., Kuntz, M., Leitner-Fischer, F., Leue, S.: Safety analysis of an airbag system using probabilistic FMEA and probabilistic counterexamples. In: Proc. of QEST, IEEE Computer Society Press (2009) 299–308
42. Debbi, H., Bourahla, M.: Generating diagnoses for probabilistic model checking using causality. Journal of Computing and Information Technology **21**(1) (2013) 13–23
43. Debbi, H., Bourahla, M.: Causal analysis of probabilistic counterexamples. In: Proc. of MEMOCODE, IEEE (2013) 77–86
44. Leitner-Fischer, F., Leue, S.: Probabilistic fault tree synthesis using causality computation. Int'l Journal of Critical Computer-Based Systems **4**(2) (2013) 119–143
45. Bernardo, M., Hillston, J., eds.: 7th Int'l School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM). In Bernardo, M., Hillston, J., eds.: Lecture notes of SFM. Volume 4486 of LNCS, Bertinoro, Italy, Springer (2007)
46. Kemeney, J.G., Snell, J.L., Knapp, A.W.: Denumerable Markov Chains. Springer (1976)
47. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. Nordic Journal on Computing **2**(2) (1995) 250–273

48. Eppstein, D.: Finding the k shortest paths. SIAM Journal on Computing **28**(2) (1998) 652–673
49. Jiménez, V.M., Marzal, A.: Computing the k shortest paths: A new algorithm and an experimental comparison. In: Proc. of WAE. Volume 1668 of LNCS, Springer (1999) 15–29
50. Aljazzar, H., Leue, S.: K*: A heuristic search algorithm for finding the k shortest paths. Artificial Intelligence **175**(18) (2011) 2129–2154
51. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003) 118–149
52. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic **Part 2** (1970) 115–125
53. Braitling, B., Wimmer, R., Becker, B., Ábrahám, E.: Stochastic bounded model checking: Bounded rewards and compositionality. In: Proc. of MBMV, Universität Rostock, ITMZ (2013) 243–254
54. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
55. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Proc. of ICTAC. Volume 3407 of LNCS, Springer (2004) 280–294
56. Han, Y.S., Wood, D.: Obtaining shorter regular expressions from finite-state automata. Theoretical Computer Science **370**(1–3) (2007) 110–120
57. Halpern, J.Y., Pearl, J.: Causes and explanations: A structural approach. Part I: Causes. British Journal on the Philosophy of Science **56** (2005) 843–887
58. Chockler, H., Halpern, J.Y.: Responsibility and blame: A structural-model approach. Journal of Artificial Intellelligence Research (JAIR) **22** (2004) 93–115
59. Leitner-Fischer, F., Leue, S.: On the synergy of probabilistic causality computation and causality checking. In: Proc. of SPIN. Volume 7976 of LNCS. Springer (2013) 246–263
60. Aljazzar, H., Leue, S.: Debugging of dependability models using interactive visualization of counterexamples. In: Proc. of QEST, IEEE Computer Society Press (2008) 189–198
61. Wimmer, R., Jansen, N., Ábrahám, E., Katoen, J.P., Becker, B.: Minimal counterexamples for refuting ω-regular properties of Markov decision processes (extended version). Reports of SFB/TR 14 AVACS 88 (2012) ISSN: 1860-9821, available at `http://www.avacs.org/fileadmin/Publikationen/Open/avacs_technical_report_088.pdf`.
62. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
63. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1984)
64. Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: Proc. of ICALP. (1997) 430–440
65. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems. PhD thesis, University of Birmingham (2002)
66. Ábrahám, E., Jansen, N., Wimmer, R., Katoen, J.P., Becker, B.: DTMC model checking by SCC reduction. In: Proc. of QEST, IEEE Computer Society Press (2010) 37–46
67. Aljazzar, H., Leue, S.: K*: A directed on-the-fly algorithm for finding the k shortest paths. Technical report, Chair of Software Engineering, University of Konstanz, Germany (2008)
68. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of TACAS. Volume 4963 of LNCS, Springer (2008) 337–340

69. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation **1**(1) (2009) 1–41
70. : IBM CPLEX optimization studio, version 12.4. http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/ (2012)
71. Gurobi Optimization, Inc.: Gurobi optimizer reference manual. http://www.gurobi.com (2013)