# Generation of Counterexamples for Model Checking of Markov Decision Processes

Husain Aljazzar and Stefan Leue
*Department of Computer and Information Science*
*University of Konstanz*
*D-78457 Konstanz, Germany*
{*Husain.Aljazzar, Stefan.Leue*}*@uni-konstanz.de*

***Abstract*—The practical usefulness of a model checker as a debugging tool relies on its ability to provide diagnostic information, sometimes also referred to as a counterexample. Current stochastic model checkers do not provide such diagnostic information. In this paper we address this problem for Markov Decision Processes. First, we define the notion of counterexamples in this context. Then, we discuss three methods to generate informative counterexamples which are helpful in system debugging. We point out the advantages and drawbacks of each method. We also experimentally compare all three methods. Our experiments demonstrate the conditions under which each method is appropriate to be used.**

***Keywords*-Stochastic Model Checking; Markov Decision Processes; Counterexamples; $k$-Shortest-Paths Search; Directed Search; K$^*$**

## I. Introduction

The success of model checking [1], [2] as a debugging tool lies in the diagnostic information, also referred to as counterexamples, it returns to the user when a fault is detected. Due to the numerical nature of the algorithms used in stochastic model checking [3], [4], [5], [6], [7], stochastic model checkers such as PRISM [8] and MRMC [9] cannot provide such diagnostic information. In precursory work [10], [11] we have addressed the generation of counterexamples for the violation of timed probabilistic reachability properties for *discrete-* and *continuous-time Markov Chains* (DTMCs and CTMCs) using heuristics guided, on-the-fly explicit state space search. That approach can not be directly applied to *Markov Decision Processes* (MDPs).

An MDP is a discrete time probabilistic model similar to a DTMC. However, it possesses both probabilistic and non-deterministic transitions: each transition of an MDP consists of a non-deterministic choice of actions and, after an action has been selected, a discrete time probabilistic transition to some successor state. The semantics of an MDP depends on an assumed scheduler which resolves the non-deterministic choices [12]. Consequently, the probability mass of executions also depends on the chosen scheduler. Intuitively, a scheduler turns an MDP into a DTMC by removing the non-determinism. This insight motivates a meta approach for providing counterexamples for MDPs. For a given MDP and a property one computes an adequate, i.e., a maximizing scheduler by a policy iteration procedure which originates from dynamic programming, such as proposed in [7], [13]. The MDP together with the determined scheduler represents a DTMC for which we can generate a counterexample using one of the existing approaches. The first method (**A**) is to apply the counterexample generation method presented in [14] which is based on *$k$-shortest-paths* (KSP). In Method A we just replace the the recursive enumeration algorithm (REA) [15], which is used in [14], with the more efficient lazy variant of Eppstein's algorithm [16]. The original version of Eppstein's algorithm is presented in algorithm [17]. We propose a new method (**B**) which is an improvement of Method A. Method B uses our directed KSP search algorithm K$^*$ [18] instead of the lazy variant of Eppstein's algorithm. The computation of an adequate scheduler for a very large MDP as well as computing the induced DTMC can be very expensive. Hence, we propose a third method (**C**) which can be applied to very large MDPs under tight memory constraints. This method performs K$^*$ search directly on the MDP. It avoids generating and processing the whole state space of the MDP.

The contributions of this work are:

- Defining the notion of a counterexample for an MDP, relative to a property specified in *PCTL logic* [4], [7], taking schedulers into account.
- Devising Methods B and C for generating counterexamples.
- Implementing and experimentally comparing all three methods regarding their performance in different settings using a set of case studies.

As it will become clear in this paper, counterexamples in the context of stochastic model checking can be very complex. Presenting and analysing such complex counterexamples are big challenges. We emphasize that this work does not discuss solutions to these issues. We presented an approach using interactive visualization techniques to support the human user in this task in [19].

**Related Work.** First results regarding counterexamples for MDPs are discussed in [20]. Method C presented in this paper is an improvement of that work. In [21] a

variant of Method A was proposed where the counterexample generation method proposed by [14], [22] is applied to the obtained DTMC. A recent proposal [23] addresses the problem of generating and analysing counterexamples for MDPs against LTL formulae. It follows the idea of computing a scheduler and generating a counterexample for the induced DTMC, following the method from [21] and Method A mentioned above. However, it applies KSP search to the graph of *strongly connected components* (SCCs) of the derived DTMC. It eliminates cyclic paths which are a big challenge for KSP search algorithms. However, this has got a negative side effect. Once an SCC is touched by a selected path the complete SCC will be added to the counterexample. Notice that a single SCC may be very large and could even comprise the whole model in which case the counterexample would be useless for debugging. In [23], the authors also present a method to facilitate the analysis of complex counterexamples based on partitioning the counterexample such that each partition conveys a similar set of diagnostic information.

In [11] we proposed a highly scalable method based on *eXtended Best-First* search (XBF) to generate counterexamples for DTMCs and CTMCs. It could be adopted as a further method for generating counterexamples for MDPs in two different ways. First, we could apply XBF, instead of KSP search, to the DTMC derived by computing the scheduler. The counterexample is delivered in the form of a diagnostic subgraph. The probability of the diagnostic subgraph is computed using a model checker. Second, we could apply XBF directly to the MDP in order to select a diagnostic subgraph. The non-determinism is removed from the seldiagnostic subgraph by the model checker. We expect both methods, in particular the second one, to be very efficient. In both cases, the counterexample is provided as a subgraph (or sub DTMC), whereas KSP based approaches provide a list of offending paths. The form of the provided counterexample determines the subsequent analysis techniques applied to it for the purpose of debugging. Therefore the application context of these methods differs from the context of the three methods we consider here. Thus, we do not consider them in this paper and refer their investigation to future research.

**Structure of the Paper.** In Section II we introduce the formal foundations of MDPs. We define the notion of a counterexample in Section III. The three methods for generating counterexamples are presented in Section IV. Section V contains our experimental evaluation of these methods. We conclude and discuss further research goals in Section VI.

## II. Markov Decision Processes

*Markov decision processes* (MDPs) allow the modelling of both non-deterministic and probabilistic system behaviour. They are very useful when modelling the asynchronous, concurrent composition of probabilistic models, such as the ones given by *discrete-time Markov chains* (DTMCs) [24], [12]. An MDP is formally defined as follows:

*Definition 1:* A labelled Markov decision process (MDP) $\mathcal{M}$ is a tuple $(\mathcal{S}, \hat{s}, \mathcal{A}, \mathcal{L})$, where $\mathcal{S}$ is a finite set of states, $\hat{s} \in \mathcal{S}$ is the initial state, $\mathcal{A} : \mathcal{S} \longrightarrow 2^{Distr(\mathcal{S})}$ is a transition function, and $\mathcal{L} : \mathcal{S} \longrightarrow 2^{\mathsf{AP}}$ is a labelling function.
The transition function $\mathcal{A}$ maps each state $s$ to a non-empty finite subset of $Distr(\mathcal{S})$, which is the set of all probability distributions over $\mathcal{S}$. In other words, for a state $s$ an element $\alpha$ of $\mathcal{A}(s)$ is a function $\alpha : \mathcal{S} \to [0, 1]$ such that $\sum_{s' \in \mathcal{S}} \alpha(s') = 1$. We call elements of $\mathcal{A}(s)$ *actions* of $s$. A transition leaving an arbitrary state $s$ begins with a non-deterministic choice between the actions available in $\mathcal{A}(s)$. After an action $\alpha$ is chosen, a probabilistic choice will be made between all possible successors, i.e., states for which $\alpha$ is not zero.

Figure 2 depicts an MDP which was obtained as the result of the asynchronous concurrent composition of two instances of the DTMC in Figure 1 using the composition semantics proposed in [12]. The non-deterministic transitions represent the decision which of the concurrent processes is executing the next step. We assume that the composed system crashes if one of the concurrent processes crashes. Hence, any global state of the composed MDP is labelled by the atomic proposition *crash*, if one of the processes is at its local state $s_2$. Note that for enhanced readability we do not display the labelling function in Figure 2.

A *path* is a concrete execution of the system and corresponds to a sequence of state transitions. Since we consider reactive systems, paths are assumed to be infinite. However, sometimes we need to refer to finite path prefixes. Let $\mathcal{M} = (\mathcal{S}, \hat{s}, \mathcal{A}, \mathcal{L})$ be an MDP, then a path through $\mathcal{M}$ is formally defined as follows:

*Definition 2:* An **(infinite) path** in the MDP $\mathcal{M}$ is an infinite sequence $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$ with $\alpha_i \in \mathcal{A}(s_i)$ and $\alpha_i(s_{i+1}) > 0$ for all $i \geq 0$. A **finite path** is a finite prefix of an infinite path.

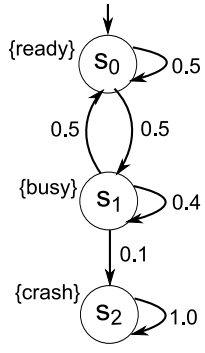


Figure 1: DTMC

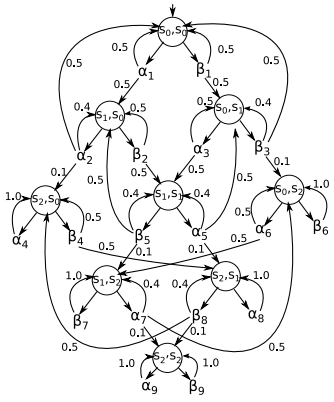

Figure 2: MDP

For a finite or an infinite path $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$, $l(\pi)$ denotes the length of $\pi$ determined by the number of states touched along $\pi$. Note that for an infinite path $\pi$, $l(\pi)$ is equal to $\infty$. For a natural number $k$ such that $0 \leq k < l(\pi)$, $\pi[k]$ refers to the $k$-th state of $\pi$, namely $s_k$ and $\pi^{(k)}$ denotes the $k$-th prefix of $\pi$, i.e., the prefix of $\pi$ consisting of the first $k$ transitions, namely $s_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{k-1}} s_k$. For $0 \leq k < l(\pi) - 1$, $\mathcal{A}_\pi(k)$ is the $k$-th action in $\pi$, namely $\alpha_k$. For a finite path $\pi$, $last(\pi)$ denotes the last state of $\pi$. $Paths^{\mathcal{M}}$ denotes the set of all infinite paths and $Paths^{\mathcal{M}}_{fin}$ denotes the set of all finite paths in $\mathcal{M}$. For any state $s$, $Paths^{\mathcal{M}}(s)$ and $Paths^{\mathcal{M}}_{fin}(s)$ refer to the sets of infinite or finite paths which start at $s$. When $\mathcal{M}$ is clear from the context we usually omit the corresponding superscript.

### A. Schedulers And Probability Measures

The non-deterministic choices in an MDP are made by a *scheduler* (also called *policy* or *adversary*) [12]. A scheduler constrains the set of allowed executions of an MDP by selecting an action based on the execution history of the system. Formally, for an MDP $\mathcal{M} = (\mathcal{S}, \hat{s}, \mathcal{A}, \mathcal{L})$, a scheduler $d$ is a function mapping every finite path $\pi$ in $\mathcal{M}$ onto an action $d(\pi) \in \mathcal{A}(last(\pi))$. Based on results from [25], [26] we can focus on *deterministic* schedulers which deterministically select an action. Such schedulers induce the maximal and minimal probability measures which are of interest when model checking MDPs, as we shall show later.

Paths which are allowed under a scheduler $d$ are called *valid* under $d$, as captured by the following definition.

*Definition 3:* A finite or an infinite path $\pi$ in an MDP is **valid** under a given scheduler $d$ iff for all $0 \leq k < l(\pi) - 1$ it holds that $\mathcal{A}_\pi(k) = d(\pi^{(k)})$ and $\mathcal{A}_\pi(k)(s_{k+1}) > 0$. Otherwise, we say that $\pi$ is **invalid** under $d$.

We refer to the set of all infinite paths which are valid under a scheduler $d$ as $Paths_d$.

A scheduler $d$ resolves the non-determinism of an MDP $\mathcal{M}$. By doing so it transforms $\mathcal{M}$ into a DTMC for which the probability of paths is measurable (c.f. [12] for a detailed discussion of this transformation). This transformation induces a probability measure $Pr_d$ over the paths of the MDP. The underlying $\sigma$-algebra is formed by the cylinder sets which are induced by finite paths in $\mathcal{M}$. Each finite path $s_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{k-1}} s_k$ induces a cylinder set $cyl_d(s_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{k-1}} s_k) = \{\pi \in Paths_d \mid \pi^{(k)} = s_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{k-1}} s_k\}$. For any $\pi \in Paths_{fin}$, the probability of the cylinder set $cyl_d(\pi)$ is defined as follows:

$$Pr_d(cyl_d(\pi)) = \begin{cases} Pr_d(cyl_d(\pi^{(0)})) \cdot \prod_{i=1}^{l(\pi)} \mathcal{A}_\pi(i-1)(\pi[i]), \\ \qquad \text{if } \pi \text{ is valid under } d \\ 0, \quad \text{otherwise} \end{cases}$$

where $Pr_d(cyl_d(\pi^{(0)})) = 1$ if $\pi^{(0)} = \hat{s}$, or 0 otherwise. Consequently, the cylinder set induced by any finite path $\pi$ possesses two possible probabilities. The first is 0 for all schedulers under which $\pi$ is invalid. The second is

$$\gamma(\pi) = \gamma(\pi^{(0)}) \cdot \prod_{i=1}^{l(\pi)} \mathcal{A}_\pi(i-1)(\pi[i]), \tag{1}$$

with $\gamma(\pi^{(0)}) = 1$ if $\pi^{(0)} = \hat{s}$, or 0 otherwise, for all schedulers under which $\pi$ is valid. The function $\gamma$ will be useful later when we introduce our approach to the generation of counterexamples.

### B. Model Checking of MDPs

The logic PCTL offers operators which allow reasoning about the occurrence probability and time of system states and events of interest in an MDP [4]. A stochastic model checker can then be used to verify a PCTL formula on the given MDP. The interesting PCTL formulae are those of the form $\mathcal{P}_{\bowtie p}(\varphi)$, where $\bowtie$ is a comparison operator out of $\{<, \leq, >, \geq\}$ and $p \in [0, 1]$ is a probability bound. Such a formula asserts that the probability to satisfy $\varphi$ fulfils the comparison $\bowtie p$. The sub-formula $\varphi$ is an *Until* formula like $(\vartheta\ U\ \varphi)$ or a *time-bounded Until* formula like $(\vartheta\ U^{\leq t}\ \varphi)$, for two state formulae $\vartheta$ and $\varphi$ and $t \in \mathbb{N}$.

The satisfaction of $\mathcal{P}_{\bowtie p}(\varphi)$ depends on the probability mass of the set of all paths satisfying $\varphi$. The probability of paths in MDPs is only defined in association with a given scheduler. A formula $\mathcal{P}_{\bowtie p}(\varphi)$ is satisfied on an MDP $\mathcal{M}$ iff for any scheduler $d$ it holds that $Pr_d(\varphi) \bowtie p$, where $Pr_d(\varphi)$ refers to the probability, under $d$, of the set of all paths satisfying $\varphi$. Notice that this probability is measurable for any possible PCTL path formula $\varphi$ [3].

This means we have to consider the extreme schedulers, *maximizing* and *minimizing schedulers*. Let $Pr_{max}(\varphi)$ and $Pr_{min}(\varphi)$ refer to the maximal and minimal probability of $\varphi$ which means $Pr_{max}(\varphi) = max\{\, Pr_d(\varphi) \mid d \in \mathfrak{D} \,\}$ and $Pr_{min}(\varphi) = min\{\, Pr_d(\varphi) \mid d \in \mathfrak{D} \,\}$, where $\mathfrak{D}$ is the set of all schedulers. It is trivial to deduce that $\mathcal{M} \not\models \mathcal{P}_{\leq p}(\varphi) \Leftrightarrow Pr_{max}(\varphi) > p$. Similar implications can easily be made for other probability bounds $<, \geq$ and $> p$.

## III. Notion of Counterexamples

In this section we define the notion of a counterexample for an MDP against a PCTL formulae $\mathcal{P}_{\bowtie p}(\varphi)$. In case $\bowtie$ is $<$ or $\leq$ we call the property *upper-bounded*, otherwise we call it *lower-bounded*. We define the notion of a counterexample for upper-bounded properties. We will demonstrate in Section IV-D how to deal with lower-bounded properties.

The validity of $\varphi$, as being an *Until* formula, can be proved by finite paths starting at $\hat{s}$ [14]. We refer to such finite paths as *diagnostic paths*.

*Definition 4:* A **diagnostic path** of $\mathcal{P}_{\bowtie p}(\varphi)$ is a finite path $\pi \in Paths_{fin}(\hat{s})$ with $\pi \models \varphi$, where $\models$ is the usual satisfaction relation from PCTL.

Let $X$ be a set of diagnostic paths. The probability $Pr_d\left(\bigcup_{\pi \in X} cyl_d(\pi)\right)$, or short $Pr_d(X)$, is measurable for

any scheduler $d$ since $X$ consists of finite paths [14]. In the case that $Pr_d(X)$ violates the upper-bound constraint $\bowtie p$ for some scheduler $d$, then $X$ is a proof of the violation of $\mathcal{P}_{\bowtie p}(\varphi)$. We call such a set a *counterexample* of $\mathcal{P}_{\bowtie p}(\varphi)$. Let $\widehat{Pr}(X)$ be the maximal probability of $X$, i.e.,

$$\widehat{Pr}(X) = sup\{\ Pr_d(X) \mid d \in \mathfrak{D}\ \}.$$

It is obvious that $X$ is a counterexample if and only if $\widehat{Pr}(X)$ violates the upper-bound constraint $\bowtie p$.

*Definition 5:* Let $\mathcal{M}$ be an MDP violating an upper-bounded formula $\mathcal{P}_{\bowtie p}(\varphi)$. A **counterexample** of $\mathcal{P}_{\bowtie p}(\varphi)$ is a set $X$ of diagnostic paths in $\mathcal{M}$ such that $\widehat{Pr}(X) \bowtie p$ does not hold.

In the case of a *non-strict* upper-bound ($\leq p$) property it is ensured that a counterexample exists which consists of a finite number of diagnostic paths. This is not guaranteed for the case of *strict* upper-bound ($< p$) properties. The approach presented in our paper aims at providing finite counterexamples. Hence, we approximate the case of a strict upper-bound $< p$ by a non-strict upper-bound $\leq (p - \varepsilon)$, for a given arbitrarily small $\varepsilon > 0$. From a practical point of view such an approximation is very adequate in order to support debugging and we hence focus on generating finite counterexamples for formulae of the form $\mathcal{P}_{\leq p}(\varphi)$ in the remainder of this paper.

## IV. Generating Counterexamples

Let $\mathcal{M}$ be an MDP which refutes the property $\mathcal{P}_{\leq p}(\varphi)$. Due to Definition 5, a counterexample is a set $X$ of diagnostic paths with $\widehat{Pr}(X) > p$. The task of selecting such a set can be represented as a $k$-Shortest-Paths problem (KSP). This is the problem of finding $k$ shortest paths, for an arbitrary natural number $k$, from a start node to a target node in a weighted directed graph. Notice that an MDP and also a DTMC can be viewed as a *state transition graph* which is a weighted directed graph defined as follows. The nodes of the graph represent the states and the edges represent the transitions of the MDP or DTMC. The weight of an edge $(s, s')$ is equal to the transition probability. The start node of the search is the initial state $\hat{s}$. We get the KSP algorithm to select diagnostic paths using the following construction. The path formula $\varphi$ is of the form $(\phi_1\ U\ \phi_2)$ or $(\phi_1\ U^{\leq t}\ \phi_2)$. Since any finite path ending in a state violating $\phi_1$ can not be completed to a diagnostic path, such states are turned into sinks by ignoring all their outgoing transitions. This ensures that any finite path ending in a state satisfying $\phi_2$ is a diagnostic path. W.l.o.g., we can assume that there is a unique state satisfying $\phi_2$ that represents a target node $g$. Otherwise, we add an absorbing state $g$ as a target and replace all outgoing transitions of each state satisfying $\phi_2$ by a single transition to $g$ with the weight 1.0. Any $\hat{s}$-$g$ path then represents exactly one diagnostic path, modulo the last transition that can easily be removed. In order to select the most probable paths instead of the shortest ones we define the weight of a path as the product, instead of the sum, of the weights of its edges. Further, we inverse the meaning of "shortest" and search for paths with maximal instead of minimal weights. Note that the definition of transition and path weights that we use here differs from the approach presented in [14], [22]. Our approach is possible since the weight product forms a cost algebra [27].

We can generate a counterexample for the MDP by applying a KSP search algorithm to its state transition graph. The KSP search algorithm will find the most probable diagnostic paths. We only consider KSP algorithms which do not require the number $k$ to be specified in advance. Examples for such algorithms are Eppstein's algorithm [17], the recursive enumeration algorithm (REA) [15] and K* [18]. These algorithms enumerate diagnostic paths one by one until the user decides to stop. The diagnostic paths are delivered starting with the most probable ones. Notice that diagnostic paths indicating high probabilities represent the executions which most significantly contribute to the property violation. At any time of the search let $\mathcal{R}$ denote the set of already found diagnostic paths. The set $\mathcal{R}$ is a counterexample once $\widehat{Pr}(\mathcal{R}) > p$. However, the computation of $\widehat{Pr}(\mathcal{R})$ is not trivial. Note that in general $\widehat{Pr}(\mathcal{R})$ is not equal to the sum of the individual probabilities, more precisely the $\gamma$-values, of the paths in $\mathcal{R}$. This is because it is not guaranteed that a scheduler exists under which all paths in $\mathcal{R}$ are valid. Also, $\mathcal{R}$ is not very convenient for debugging since it contains a lot of "noise". We recall that $\widehat{Pr}(\mathcal{R}) = Pr_d(\mathcal{R})$ for some scheduler $d$. All elements from $\mathcal{R}$ which are invalid under $d$ do not contribute to $\widehat{Pr}(\mathcal{R})$. Hence, they should not be included in the counterexample. The goal is to provide a counterexample which is "noise free" and to state its probability. In the following three sections we present three methods to achieve this goal.

### A. Method A

The idea of this method is to eliminate the non-determinism before a counterexample is generated. Hence, a maximizing scheduler $d$ is determined using a policy iteration procedure as proposed in [7], [13]. This scheduler turns $\mathcal{M}$ into a DTMC $\mathcal{M}_d$ by removing the non-determinism. A counterexample of $\mathcal{P}_{\leq}(\varphi)$ on the DTMC $\mathcal{M}_d$ is generated using a method which is similar to the one represented in [14], [22]. The main idea of the approach from [14], [22] is to apply KSP search to the state transition graph of the obtained DTMC. In [14], [22] the REA algorithm [15] is recommended. However, we use the lazy version of Eppstein's algorithm [16] since it has a better asymptotic complexity and also has been proven to perform better in practice.

Algorithm 1 illustrates the algorithmic structure of this method. The result is a set $X$ of diagnostic paths with $Pr_d(X) > p$. Note that all paths from $X$ exist in the original MDP $\mathcal{M}$ and that $Pr_d(X) > p$ implies $\widehat{Pr}(X) > p$. Thus, $X$

**Algorithm 1**: Method A

**Input**: An MDP $\mathcal{M}$ and a property $\mathcal{P}_{\leq p}(\varphi)$

1 Compute a maximizing scheduler $d$ for $\mathcal{M}$ and $\mathcal{P}_{\leq p}(\varphi)$. Let $\mathcal{M}_d$ be the DTMC induced by $d$.
2 Run Eppstein's algorithm on the state transition graph of $\mathcal{M}_d$.
3 $X \leftarrow \{\}$
4 **while** $Pr_d(X) \leq p$ **do**
5 Ask Eppstein's algorithm for the next probable diagnostic path $\pi$.
6 Add $\pi$ to $X$
7 **return** $X$ as a counterexample.

is also a counterexample of $\mathcal{M}$. Further, all paths from $X$ are valid under the scheduler $d$ which means that $X$ is free of "noise".

### B. Method B

We propose a new method B which is an improvement of Method A. It has the same algorithmic structure as Method A, cf. Algorithm 1. Method B computes the DTMC induced by a maximizing scheduler. It then generates a counterexample by running a KSP algorithm on the derived DTMC. The difference compared to Method A is that we use K* instead of Eppstein's algorithm, see Algorithm 1, Line 2. K* is a directed algorithm for solving the KSP problem [18]. For a proper understanding of this paper it is fully sufficient to consider K* as an adoption of the lazy variant of Eppstein's algorithm [16], [17]. The main feature of K* is that it works on-the-fly and can also be guided using heuristic functions. In other words, unlike Eppstein's algorithm, K* does not need to process the entire state transition graph to provide a solution.

### C. Method C

Both methods A and B compute a maximizing DTMC in advance which requires to process the entire state space of the MDP. This can be very expensive for large models. Method C avoids this effort. It directly applies K* to the state transition graph of the MDP. All diagnostic paths discovered by K* are stored in the set $\mathcal{R}$. In order to provide a "noise-free" counterexample we filter out all "noisy" paths from $\mathcal{R}$. We explain soon how to do this.

*Definition 6:* Let $d$ be a **maximizing scheduler** of $\mathcal{R}$, i.e. $Pr_d(\mathcal{R}) = \widehat{Pr}(\mathcal{R})$. The set of all diagnostic paths from $\mathcal{R}$ which are valid under $d$ is called a **maximum** of $\mathcal{R}$.
We can easily prove for a maximum $X \subset \mathcal{R}$ that $\widehat{Pr}(\mathcal{R}) = \widehat{Pr}(X) = \sum_{\pi \in X} \gamma(\pi)$. Consequently, if $\mathcal{R}$ is a counterexample, then each maximum of $\mathcal{R}$ is a "noise-free" counterexample. This leads to the strategy illustrated in Algorithm 2. Now, we need a method to compute a maximum $X$ of $\mathcal{R}$

**Algorithm 2**: Strategy for generating counterexamples according to Method C

**Input**: An MDP $\mathcal{M}$ and a property $\mathcal{P}_{\leq p}(\phi)$

1 $\mathcal{R} \leftarrow \{\}$
2 $X \leftarrow \{\}$
3 **while** $\widehat{Pr}(X) \leq p$ **do**
4 Ask K* for the next probable diagnostic path $\pi$.
5 Add $\pi$ to $\mathcal{R}$
6 Compute a maximum of $\mathcal{R}$ and assign it to $X$.
7 **return** $X$ as a counterexample.

and to compute its probability $\widehat{Pr}(X)$ which can be used at Line 6 in Algorithm 2.

*Computing a Maximum.:* In computing a maximum of $\mathcal{R}$ we have to take the following issues into account. First, $\mathcal{R}$ grows with each new diagnostic path found by K*. As we see in Algorithm 2, we have to (re-)compute a maximum of $\mathcal{R}$ and compute its probability, whenever $\mathcal{R}$ grows. Hence, our technique should be designed as an online algorithm which efficiently updates the maximum computed so far when new diagnostic paths are added into $\mathcal{R}$. Second, we have to compute a maximum without knowing a concrete maximizing scheduler for $\mathcal{R}$. To this end we need first to introduce the notion of *(scheduler) compatible* paths.

*Definition 7:* A set of paths $C$ is **(scheduler) compatible** iff there is a scheduler $d$ such that all paths in $C$ are valid under $d$.
The relevance of this notion arises from the insight that each scheduler compatible subset $X$ of $\mathcal{R}$ with a maximal probability is a maximum of $\mathcal{R}$.

Recall that a scheduler is a deterministic function mapping a finite path of the MDP onto an action. Thus, for diagnostic paths, the first branching (after a common prefix) is decisive for scheduler compatibility. Diagnostic paths can not be compatible when branching in a state. On the other hand, it is always possible to define a scheduler that allows a set of diagnostic paths which branch in actions. Thus, diagnostic paths which branch in an action are compatible. Consequently, checking for scheduler compatibility can be accomplished as follows. We implement the set $\mathcal{R}$ as an AND/OR tree. Each diagnostic path is stored in $\mathcal{R}$ by mapping its states and actions to nodes. We map states to OR nodes and actions to AND nodes.

Algorithm 3 illustrates the steps of adding diagnostic paths into $\mathcal{R}$ and how scheduler compatible subsets are identified and their probabilities are computed. The steps up to Line 4 add a diagnostic path $\pi$ to $\mathcal{R}$. Notice that the construction ensures that $\mathcal{R}$ is an AND/OR tree rooted at the OR node $\hat{s}$. The remaining steps beginning at Line 5 perform the identification of scheduler compatible subsets of $\mathcal{R}$ and the computation of their probabilities in an online fashion upon adding $\pi$ to $\mathcal{R}$. The AND/OR tree structure makes all

**Algorithm 3**: Adding diagnostic paths to the AND/OR tree implementing $\mathcal{R}$

**Input**: A newly found diagnostic path $\pi = s_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{k-1}} s_k$.

1 Interpret $\pi$ as an alternating sequence of OR and AND nodes $S_\pi = \langle s_0, \alpha_0, \ldots, \alpha_{k-1}, s_k \rangle$.
2 **if** $\mathcal{R}$ *is empty* **then** Add all nodes and the edges $\{ (s_0, \alpha_0), (\alpha_0, s_1), \ldots, (\alpha_{k-1}, s_k) \}$ to $\mathcal{R}$ and go to Line 5.
3 Determine, starting from the root $\hat{s}$ [1], the longest prefix of $S_\pi$ which is already contained in $\mathcal{R}$, i.e., the longest prefix which $S_\pi$ shares with any path in $\mathcal{R}$.
4 Attach the remainder of $S_\pi$ to the last node of the prefix as a new subgraph of $\mathcal{R}$. During the adding $S_\pi$ is not folded.
5 In a bottom-up manner, assign to each node on the newly added path two marks, a probability mark $M_p$ and an integer mark $M_i$, as follows:
6 Assign to the leaf, i.e., the node $s_k$, the marks $M_p = \gamma(\pi)$ and $M_i = 1$.
7 **foreach** *inner node* $n$ *on* $S_\pi$ **do**
8  **if** $n$ *is an AND node* **then**
9  Add the $M_p$ marks assigned to the children of $n$ and assign the sum to $n$ as a probability mark $M_p$.
10  Add the $M_i$ marks assigned to the children of $n$ and assign the sum to $n$ as an $M_i$ mark.
11  **else** `// If n is an OR node`
12  Select the child $n'$ of $n$ with the maximal $M_p$ mark. If $n$ has more than one child the $M_p$ mark of which is maximal, then select that one with the minimal $M_i$ mark.
13  Mark $n$ with identical $M_p$ and $M_i$ marks as $n'$.
14  Attach to $n$ a reference MAX referring to $n'$.

scheduler compatible subsets of $\mathcal{R}$ available. Our algorithm compares these subsets in terms of probability and size at Line 12 and selects, using the reference MAX, the action which leads to the scheduler compatible subset with maximal probability and minimal size. The aim of minimizing the size is to compute practically useful counterexamples carrying a maximal probability with a minimal number of diagnostic paths.

Note that a global processing of the entire AND/OR tree is never required. The computational effort of Algorithm 3 is limited to traversing the tree only along the newly inserted path, which is important for the performance of our method and its adequacy as an online technique.

As from now let $\mathcal{X} \subseteq \mathcal{R}$ be the set of the diagnostic paths corresponding to those paths in the tree, from the root to the leaves, which contain only AND and MAX references. Our algorithm ensures that $\mathcal{X}$ is a *smallest maximum* of $\mathcal{R}$. This means, $\mathcal{X}$ is a maximum of $\mathcal{R}$ with a minimal number of diagnostic paths. So $\mathcal{X}$ is the most informative counterexample which can be extracted from $\mathcal{R}$. In other words, $\mathcal{X}$ is an optimal counterexample over $\mathcal{R}$. It carries a maximal probability with minimal size. However, it is possible to find a counterexample $\mathcal{X}'$ consisting of elements from $\mathcal{R}$ together with other diagnostic paths which have not been delivered yet by K$^*$. The set $\mathcal{X}'$ might carry a higher probability and might contain fewer diagnostic paths than $\mathcal{X}$. However, the probability of the diagnostic paths from $\mathcal{X}'$ which are not elements of $\mathcal{R}$ may be very low such that they are insignificant for debugging. $\mathcal{X}$ has the advantage that it includes **only** diagnostic paths from $\mathcal{R}$ which are the most probable and crucial ones. In our experiments we did not observe any penalty in the counterexample quality compared to the methods A and B. Note that A and B provide optimal counterexamples.

In addition to computing the smallest maximum of $\mathcal{R}$, the AND/OR tree includes an explicit representation of the scheduler corresponding to the computed maximum. The MAX references represents the scheduler's decision for each non-deterministic choice. This scheduler can be useful in debugging since it indicates a set of extreme choices which lead to property violation.

*Example.:* Let $\phi$ be the path formula $(true\ U^{\leq 3}\ crash)$. The MDP from Figure 2 violates the property $\mathcal{P}_{\leq 0.09}(\phi)$ because $Pr_{max}(\phi) = 0.095 > 0.09$. We use our method to generate a counterexample. K$^*$ searches the state space for most probable diagnostic paths which are then added into $\mathcal{R}$ which is implemented as an AND/OR tree. The obtained AND/OR tree is illustrated in Figure 3. Some nodes of interest are additionally labelled with the marks used in Algorithm 3. We see that the root is marked with $0.095$ and $3$, which means that the found maximum (highlighted in the figure by bold lines) has the probability $0.095$ and consists of $3$ diagnostic paths. Since $0.095 > 0.09$, the maximum that was found is a counterexample. Out of a total of 14, which is the number of all paths stored in the tree, our counterexample consists of only 3 compatible paths. This counterexample facilitates gaining insight why the property is violated. For instance, it shows that a large portion of the total probability mass of the counterexample of $0.095$, namely $0.05$, is caused by the immediate *crash* of the second component after it becomes *busy*. Another major contributor is the *crash* of the first component after the loop $(s_0, s_0) \xrightarrow{\beta_1} (s_0, s_0)$ with a mass of $0.025$. The remainder is related to the loop $(s_0, s_1) \xrightarrow{\beta_3} (s_0, s_1)$ which increases the likelihood of a *crash* after staying *busy* for a while. To debug

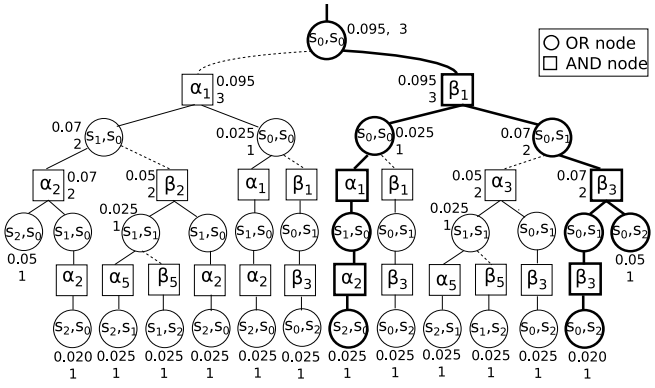


Figure 3: The AND/OR tree from the example in Section IV-C

the system, meaning to make it immune against violations of the property $\mathcal{P}_{\leq 0.09}\ (true\ U^{\leq 3}\ crash)$, one has to reduce the probability of the paths making up the counterexample. The effective approach would be to redesign the system such that the transition probability from *busy* to *crash* is reduced. If this was not sufficient, then one has to lessen the impact of the loops at the *ready* and *busy* states in terms of the likelihood to *crash*.

### D. Lower-Bounded Properties

A given MDP $\mathcal{M}$ refutes a lower-bounded property $\mathcal{P}_{\bowtie p}(\varphi)$ ($\bowtie$ is either $>$ or $\geq$) iff $Pr_{min}(\varphi)$ does not reach/exceed the specified lower-bound $p$. Thus, in both methods A and B a minimizing scheduler $d$ instead of a maximizing one is computed. The DTMC $\mathcal{M}_d$ induced by the minimizing scheduler refutes $\mathcal{P}_{\bowtie p}(\varphi)$ too. Then, a counterexample is generated for the DTMC $\mathcal{M}_d$. In order to provide a proof that a DTMC refutes a lower-bounded property we have to show that the probability of all diagnostic paths in the model does not reach/exceed the specified lower-bound. In contrast to the upper-bounded case, any strict subset of diagnostic paths does fulfil this requirement. Thus, a lower-bounded property is reformulated into an equivalent upper-bounded one as shown in [14]. This reformulation requires the computation of the *bottom strongly connected components* (BSCCs) of the DTMC.

In Method C we have to perform similar preparations as in Methods A and B. We can reformulate $\mathcal{P}_{\bowtie p}(\varphi)$ as an upper-bounded property on the MDP directly. We need then to compute the *maximum end components* of the MDP [13]. These are the counterpart of BSCCs in a DTMC. For this computation the entire state space has to be generated. This weakens the advantage of the on-the-fly feature of Method C. This result is a similar to the fact that directed methods are more successful for safety than for liveness properties [28]. Notice that lower-bounded properties here are the probabilistic counterpart of liveness properties.

## V. Experimental Evaluation

In the experiments we compare the performance of all three methods investigating which method is adequate under which conditions. We report here the experimental results for two significant case studies which are both available from the PRISM web page[2]. We implemented the three methods in Java. All measurements were made on a machine with an Intel Pentium 4 CPU (3.2 GHz)[3] and 2 GB RAM. Java code was executed by the Java runtime environment JDK 1.6.

### A. IPv4 Zeroconf Protocol

This case study is a model of a dynamic configuration protocol for IPv4 addresses. We are interested in the probability that a host picks an IP address already in use. We configured the model with the parameters $NO\_RESET$, $N = 1000$ (number of hosts) and $loss = 0.1$ (probability of message loss). In order to study the scalability of the methods, we considered three variants of the model with different sizes. For this purpose, we set the parameter $K$ (number of probes to send) to $K = 2$, $K = 4$ and $K = 8$. We list essential characteristics of these model variants in Table I. Next to the number of states and transitions of the MDP we give in the column labelled with "Probability" the probability which PRISM computed for the property for each model variant. The columns "MC Time" and "MC Memory" record the runtime (in seconds) and memory (in KB) needed for building and checking the models using PRISM.

| K | 2 | 4 | 8 |
|---|---|---|---|
| States | 77,279 | 276,781 | 1,774,403 |
| Transitions | 180,442 | 640,721 | 4,021,084 |
| Probability | 0.0030424 | 1.075912E-4 | 1.396195E-7 |
| MC Time | 24.868 | 77.430 | 928.728 |
| MC Memory | 3,584.0 | 11,878.4 | 58,368.0 |

Table I: IP – Properties of the MDPs

| K | 2 | 4 | 8 |
|---|---|---|---|
| States | 77,279 | 276,781 | 1,774,403 |
| Transitions | 95,272 | 335,325 | 2,119,827 |
| Gen. Time | 16.150 | 55.705 | 522.893 |
| Memory | 7,757.6 | 27,715.5 | 177,329.5 |

Table II: IP – Computing maximizing DTMCs

Both methods A and B need to generate the DTMC induced by a maximizing scheduler. Table II shows for each model variant the size of the computed DTMC and the effort required by PRISM to generate it. Column "Gen. Time" indicates the generation time (in seconds) and Column "Memory" indicates the memory space (in KB) needed to store the DTMC. Each DTMC has the same number of states as the original MDP. Resolving the non-determinism reduces the number of transitions in each variant of this model approximately to the half. This is a clear sign for a high degree of non-determinism in the original MDP.

We ran the three methods A, B and C on all three variants of the model and recorded the results in Table III. We report the runtime (in seconds) and memory consumption (in KB) of generating counterexamples for probability upper-bounds 10%, 40%, 80%, 95% and 100% (see Column "P" in Tables IIIa and IIIb) of the total probability which is given in Table I. Table cells with the content of the pattern "$? > x$", for some value $x$, mean that the corresponding method failed to provide a counterexample for the corresponding probability bound although it ran for $x$ seconds or consumed $x$ KB of memory.

It holds for all three methods that the larger the model is the more effort is required to generate counterexamples.

[2]http://www.prismmodelchecker.org/

[3]The CPU had got 2 cores, but we do not exploit them because our implementation does not employ parallelism.

| K | P | A | B | C |
|---|---|---|---|---|
| 2 | 10 | 19.076 | 16.388 | 0.161 |
| | 40 | 19.076 | 16.388 | 5.090 |
| | 80 | 19.082 | 17.252 | 21.214 |
| | 95 | 19.098 | 17.255 | 1,515.341 |
| | 100 | ? > 836.697 | ? > 876.925 | ? > 7,260.011 |
| 4 | 10 | 63.040 | 58.630 | 135.943 |
| | 40 | 63.047 | 58.643 | 518.177 |
| | 80 | 63.085 | 58.741 | 2,163.076 |
| | 95 | 63.127 | 65.309 | 3,982.039 |
| | 100 | ? > 720.890 | 65.662 | ? > 7,260.012 |
| 8 | 10 | 566.330 | 527.506 | 33,683.118 |
| | 40 | 566.416 | 527.564 | 51,908.576 |
| | 80 | 567.053 | 530.905 | ? > 144,855.119 |
| | 95 | 568.332 | 531.497 | ? > 144,855.119 |
| | 100 | ? > 1,000.144 | ? > 1,040.076 | ? > 144,855.119 |

(a) Runtime [sec]

| K | P | A | B | C |
|---|---|---|---|---|
| 2 | 10 | 7,797.9 | 7,761.8 | 34.4 |
| | 40 | 7,798.4 | 7,762.2 | 463.5 |
| | 80 | 7,802.2 | 7,789.3 | 951.2 |
| | 95 | 7,811.3 | 7,798.4 | 16,263.5 |
| | 100 | ? > 264,128.1 | ? > 258,668.9 | ? > 52,690.0 |
| 4 | 10 | 27,849.2 | 27,734.6 | 3,017.2 |
| | 40 | 27,851.7 | 27,736.6 | 5,960.0 |
| | 80 | 27,869.7 | 27,751.5 | 12,412.7 |
| | 95 | 27,910.3 | 27,905.1 | 14,075.5 |
| | 100 | ? > 201,959.9 | 28,302.0 | ? > 17,416.1 |
| 8 | 10 | 177,874.1 | 177,400.0 | 59,329.6 |
| | 40 | 177,927.5 | 177,446.2 | 64,813.5 |
| | 80 | 178,311.8 | 177,834.0 | ? > 90,695.2 |
| | 95 | 179,177.5 | 178,575.4 | ? > 90,695.2 |
| | 100 | ? > 316,325.3 | ? > 316,128.6 | ? > 90,695.2 |

(b) Memory [KB]

Table III: IP – Counterexample generation

We notice that the computational costs of both methods A and B is dominated by the costs of computing the DTMC. For example, to generate a counterexample with 10% of the total probability for the case with $K = 2$, method A needed 19.076 seconds and 7 797.9 KB. The most of that effort, precisely 16.150 seconds and 7 757.6 KB, is needed for computing and storing the DTMC. We observe this effect for both methods A and B and for all model variants. Method C needs very long time especially for large $K$ values and large probability bounds. It even failed in the case with $K = 8$ to provide a counterexample with 80% of the total probability after running for 144 885 seconds (i.e. 40 hours). However, it requires significantly less memory than both methods A and B. In most cases C required approximately half of the memory required by A or B. The economical memory consumption of method C is due to its on-the-fly nature. The long runtime of C in this case study can be explained by the high degree of non-determinism of the models, which has two consequences. First, the DTMC derived by resolving the non-determinism contains about half of the transitions of the original MDP. This implies that the reachable portion of the DTMC is very small compared to the original MDP. On the other hand, method C operates directly on the MDP. The portion $K^*$ had to explore was much bigger than the reachable part of the DTMC. We consider for example the case with $K = 8$. Method A explores 13 053 states and 22 662 transitions (according to figures which, due to space limitations, are not reported in the tables). This is the size of the reachable part of the DTMC since Eppstein's algorithm, as we know, has to explore the entire reachable graph before it delivers a solution. Method C explored 379 791 states and 1 286 235 transitions which is a large nuber compared to the 13 053 states and 22 662 transitions constituting the reachable part of the DTMC. Second, the MDP contains much more diagnostic paths than the DTMC. Notice that an MDP models more behaviour than the corresponding DTMC since it also represents nondeterminism. However, only few of these paths are scheduler compatible. Consequently, method C has to find and process many diagnostic paths to achieve a noticeable increase in the counterexample probability. For instance, for $K = 8$ method C delivered 68 202 diagnostic paths to provide a counterexample with 60% of the total probability. A lot of time is used to process this huge number of paths while only 349 of them form the counterexample.

### *B. Bounded Retransmission Protocol*

This case study is based on the bounded retransmission protocol (BRP) [29], a variant of the alternating bit protocol. The BRP protocol sends a file in a number of chunks, but allows only a bounded number of retransmissions of each chunk. The model is represented as an MDP. The parameter $N$ denotes the number of chunks and $MAX$ the maximum allowed number of retransmissions of each chunk. We set in our experiments $MAX$ to 5. We scaled the size of the state space by varying the value of $N$. We list essential characteristics of the considered variants of the model in Table IV. All tables in this section have the same structure as in the previous case study.

Table V shows for each model variant the size of the DTMC and the effort PRISM required to generate it. We can

| N | 640 | 3200 | 6400 |
|---|---|---|---|
| States | 103,962 | 518,682 | 1,037,082 |
| Transitions | 139,888 | 697,968 | 1,395,568 |
| Probability | 4.482E-7 | 2.241E-6 | 4.482E-6 |
| MC Time | 347.055 | 5,023.233 | 11,880.100 |
| MC Memory | 4,403.2 | 20,992.0 | 34,611.2 |

Table IV: BRP – Properties of the MDPs

| N | 640 | 3200 | 6400 |
|---|---|---|---|
| States | 103,962 | 518,682 | 1,037,082 |
| Transitions | 137,313 | 685,153 | 1,369,953 |
| Gen. Time | 36.592 | 776.584 | 4,792.394 |
| Memory | 8,919.0 | 44,499.0 | 88,974.0 |

Table V: BRP – Computing maximizing DTMCs

| N | P | A | B | C |
|---|---|---|---|---|
| 640 | 1 | 132.719 | 137.352 | 35.441 |
| | 4 | 133.890 | 139.704 | 52.852 |
| | 8 | 328.980 | 340.152 | 2,074.385 |
| | 9 | 765.808 | 740.193 | ? > 3,600.005 |
| | 10 | ? > 3,207.107 | ? > 3,352.839 | ? > 3,600.005 |
| 3200 | 0.5 | 1,401.751 | 892.909 | 85.227 |
| | 0.8 | 1,403.315 | 894.225 | 104.268 |
| | 1.6 | ? > 3,259.52 | 1,110.174 | 2,979.821 |
| | 1.8 | ? > 3,259.52 | ? > 4,470.727 | ? > 3,600.019 |
| | 2.0 | ? > 3,259.52 | ? > 4,470.727 | ? > 3,600.019 |
| 6400 | 0.1 | ? > 6,853.476 | 4,899.457 | 80.336 |
| | 0.3 | ? > 6,853.476 | 4,900.007 | 88.904 |
| | 0.5 | ? > 6,853.476 | 4,906.373 | 171.328 |
| | 0.8 | ? > 6,853.476 | 5,113.388 | 2,883.459 |
| | 1.0 | ? > 6,853.476 | ? > 7,273.858 | ? > 3,600.009 |

(a) Runtime [sec]

| N | P | A | B | C |
|---|---|---|---|---|
| 640 | 1 | 11,009.3 | 11,009.3 | 12,949.8 |
| | 4 | 12,767.9 | 12,767.9 | 13,839.1 |
| | 8 | 214,252.6 | 214,252.6 | 79,452.9 |
| | 9 | 585,077.0 | 585,077.0 | ? > 114,081.9 |
| | 10 | ? > 672,350.9 | ? > 658,306.1 | ? > 114,081.9 |
| 3200 | 0.5 | 54,962.6 | 47,431.9 | 30,430.4 |
| | 0.8 | 56,317.9 | 48,787.2 | 31,022.2 |
| | 1.6 | ? > 110,220.0 | 250,265.0 | 96,579.9 |
| | 1.8 | ? > 110,220.0 | ? > 435,597.5 | ? > 107,114.9 |
| | 2.0 | ? > 110,220.0 | ? > 435,597.5 | ? > 107,114.9 |
| 6400 | 0.1 | ? > 105,504.7 | 91,503.6 | 30,133.1 |
| | 0.3 | ? > 105,504.7 | 92,155.2 | 30,572.6 |
| | 0.5 | ? > 105,504.7 | 98,223.1 | 32,376.6 |
| | 0.8 | ? > 105,504.7 | 294,737.1 | 96,579.0 |
| | 1.0 | ? > 105,504.7 | ? > 439,987.9 | ? > 109,181.0 |

(b) Memory [KB]

Table VI: BRP – Counterexample generation

see that in each model variant, the DTMC still contains about 98% of the transitions. This is evidence for a low degree of non-determinism. The results of running all methods are reported in Table VI. Notice that the probability bounds listed here are very low (c.f. Column "P" in Tables VIa and VIb). We did so because all three methods failed to provide counterexamples for large probability bounds in realistic time. This phenomenon can be explained as follows. Single diagnostic paths in this model have very low individual probability. For instance in the case with $N = 3\,200$, method A selects 5 588 diagnostic paths the accumulated probability of which is only 1% of the total probability. That means, numerous diagnostic paths must be found and processed to achieve significant counterexample example probability. We expect XBF based methods, which we do not consider in this paper, to have a clear advantage in such situations.

Method A and B have comparable performance when $N = 640$. However, the larger the models are, the more inefficient method A is. Method C outmatches A and B in most cases with respect to both runtime and memory consumption. This holds particularly for large models with $N = 3\,200$ and $N = 6\,400$. We notice that the advantage of C regarding memory consumption is bigger for larger probability bounds. This is a result of using AND/OR trees. The number of found diagnostic paths is very big for large probability bounds. Hence, storing them compactly in an AND/OR tree pays off.

### C. Summary of Experimental Results

We summarize our experimental conclusions as follows. Method B outmatches A in most cases, in particular for large models. Thus, we recommend to use Method B instead of A. For large MDPs method C offers a significant advantage with respect to memory consumption. It also outperforms A and B with respect to runtime, in particular for small probability bounds. For models with a very high degree of non-determinism, method C can take a very long time to produce counterexamples for large probability bounds. However, it has a good runtime behaviour if the degree of non-determinism of the model is not very high. Hence, we recommend method C if the model is large with a low degree of non-determinism. If the model is small or it is highly non-deterministic we can use C for small probability bounds. B seems to be the more promising method for small models or model with high degree of non-determinism, in particular for large probability bounds. We summarize these conclusions in Table VII.

## VI. Conclusion

We first defined the notion of a counterexample for verifying MDPs against PCTL properties. We then demonstrated three methods, called A, B and C, for computing informative counterexamples based on $k$-Shortest-Paths search. All three methods optimize the selected counterexamples in terms of their probability and the number of included paths. Methods A and B compute an adequate scheduler for a given MDP and a property. They generate a counterexample for the DTMC induced by this scheduler. Method C generates a counterexample directly for the MDP on-the-fly and uses an AND/OR tree structure to accommodate the influence of schedulers. Method B outperforms Method A with respect to computational costs in most cases. Method C, due to its on-the-fly nature, shows promise for large models. The algorithmic core of Methods B and C is the K$^*$ algorithm, which is a directed algorithm for solving the $k$-Shortest-Paths problem. In particular, it is on-the-fly and can be guided by heuristics. We experimentally compared the performance of all three

| | | Model Size | |
|---|---|---|---|
| | | Small | Large |
| Degree of Non-Determ. | Low | **C** for small prob. bounds<br>**B** for larg prob. bounds | **C** |
| | High | **B** | **C** for small prob. bounds<br>**B** for larg prob. bounds |

Table VII: Which method under which conditions

methods. Our experimental evaluation shows which method is adequate under which conditions. Future work includes an adoption of XBF do MDPs.

**Acknowledgement.** This work was partially supported by DFG Research Training Group GK-1042 "Explorative Analysis and Visualization of Large Information Spaces".

## REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking (3rd ed.)*. The MIT Press, 2001.

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[3] M. Y. Vardi, "Automatic verification of probabilistic concurrent finite-state programs," in *26th Annual Symposium on Foundations of Computer Science (FOCS 1985)*. IEEE, 1985, pp. 327–338.

[4] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[5] C. Courcoubetis and M. Yannakakis, "The complexity of probabilistic verification." *J. ACM*, vol. 42, no. 4, pp. 857–907, 1995.

[6] A. Aziz, V. Singhal, and F. Balarin, "It usually works: The temporal logic of stochastic systems." in *CAV 1995*, ser. LNCS vol. 939. Springer, 1995, pp. 155–165.

[7] A. Bianco and L. de Alfaro, "Model checking of probabalistic and nondeterministic systems," in *FSTTCS 1995*, ser. LNCS vol. 1026. Springer, 1995, pp. 499–513.

[8] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A tool for automatic verification of probabilistic systems," in *TACAS'06*, ser. LNCS vol. 3920. Springer, 2006, pp. 441–444.

[9] J.-P. Katoen, M. Khattri, and I. S. Zapreev, "A Markov Reward Model Checker," in *QEST '05*. IEEE Computer Society, 2005, pp. 243–244.

[10] H. Aljazzar, H. Hermanns, and S. Leue, "Counterexamples for timed probabilistic reachability." in *FORMATS '05*, ser. LNCS vol. 3829. Springer, 2005, pp. 177–195.

[11] H. Aljazzar and S. Leue, "Extended directed search for probabilistic timed reachability." in *FORMATS '06*, ser. LNCS vol. 4202. Springer, 2006, pp. 33–51.

[12] C. Baier and M. Z. Kwiatkowska, "Model checking for a probabilistic branching time logic with fairness," *Distributed Computing*, vol. 11, no. 3, pp. 125–155, 1998.

[13] L. de Alfaro, "Formal verification of probabilistic systems," Ph.D. Dissertation, Stanford University, 1997.

[14] T. Han and J.-P. Katoen, "Counterexamples in probabilistic model checking," in *TACAS '07*, 2007.

[15] V. M. Jiménez and A. Marzal, "Computing the k shortest paths: A new algorithm and an experimental comparison," in *Algorithm Engineering*, 1999, pp. 15–29.

[16] ——, "A lazy version of eppstein's shortest paths algorithm," in *WEA '03*, ser. LNCS vol. 2647. Springer, 2003, pp. 179–190.

[17] D. Eppstein, "Finding the $k$ shortest paths," *SIAM J. Computing*, vol. 28, no. 2, pp. 652–673, 1998. [Online]. Available: http://dx.doi.org/10.1137/S0097539795290477

[18] H. Aljazzar and S. Leue, "K$^*$: A directed on-the-fly algorithm for finding the $k$ shortest paths," Univ. of Konstanz, Gemany, Tech. Rep. soft-08-03, March 2008, submitted for publication. [Online]. Available: http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-08-03.pdf

[19] ——, "Debugging of dependability models using interactive visualization of counterexamples." in *QEST '08*. IEEE Computer Society Press, 2008.

[20] ——, "Counterexamples for model checking of markov decision processes," Univ. of Konstanz, Gemany, Tech. Rep. soft-08-01, December 2007. [Online]. Available: http://www.ub.uni-konstanz.de/kops/volltexte/2008/4530/

[21] H. Hermanns, B. Wachter, and L. Zhang, "Probabilistic CEGAR," in *Computer Aided Verification, 20th International Conference, CAV 2008, Proceedings*, ser. Lecture Notes in Computer Science, vol. 5123. Springer, 2008, pp. 162–175.

[22] T. Han, J.-P. Katoen, and B. Damman, "Counterexample generation in probabilistic model checking," *IEEE Trans. Softw. Eng.*, vol. 35, no. 2, pp. 241–257, 2009.

[23] M. E. Andrés, P. R. D'Argenio, and P. van Rossum, "Significant diagnostic counterexamples in probabilistic model checking," *CoRR*, vol. abs/0806.1139, 2008.

[24] W. J. Stewart, *Introduction to the Numerical Solution of Markov Chains*. New Jersey, USA: Princeton University Press, 1994.

[25] S. Hart, M. Sharir, and A. Pnueli, "Termination of probabilistic concurrent program," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 356–380, 1983.

[26] R. Segala and N. A. Lynch, "Probabilistic simulations for probabilistic processes," in *CONCUR '94*, ser. LNCS vol. 836. Springer, 1994, pp. 481–496.

[27] S. Edelkamp, S. Jabbar, and A. Lluch-Lafuente, "Cost-algebraic heuristic search," in *AAAI '05*, 2005, pp. 1362–1367.

[28] S. Edelkamp, S. Leue, and A. Lluch-Lafuente, "Directed explicit-state model checking in the validation of communication protocols," *International Journal on Software Tools for Technology Transfer STTT*, vol. 5, no. 2–3, pp. 247–267, 2004.

[29] L. Helmink, M. Sellink, and F. Vaandrager, "Proof-checking a data link protocol," in *TYPES'93*, ser. LNCS vol. 806, H. Barendregt and T. Nipkow, Eds. Springer, 1994, pp. 127–165.