| | |
|---|---|
| **Project no.:** | **PIRSES-GA-2011-295261** |
| **Project full title:** | **Mobility between Europe and Argentina applying Logics to Systems** |
| **Project Acronym:** | **MEALS** |
| **Deliverable no.:** | **1.1 / 2** |
| **Title of Deliverable:** | **Symbolic Counterexample Generation for** |

**Large    Discrete-Time Markov Chains**

| | |
|---|---|
| **Contractual Date of Delivery to the CEC:** | **31-Mar-2015** |
| **Actual Date of Delivery to the CEC:** | **31-Mar-2015** |
| **Organisation name of lead contractor for this deliverable:** | **UNC** |
| **Author(s):** | **Nils Jansen, Ralf Wimmer, Erika Ábrahám, Barna Zajzon, Joost-Pieter Katoen, Bernd Becker, Johann Schuster** |
| **Participants(s):** | **RTW** |
| **Work package contributing to the deliverable:** | **WP1** |
| **Nature:** | **R** |
| **Dissemination Level:** | **Public** |
| **Total number of pages:** | **42** |
| **Start date of project:** | 1 Oct. 2011    **Duration:** 48 month |

Abstract:

This paper presents several symbolic counterexample generation algorithms for discrete-time Markov chains (DTMCs) violating a PCTL formula. A counterexample is (a symbolic representation of) a sub-DTMC that is incrementally generated. The crux to this incremental approach is the symbolic generation of paths that belong to the counterexample. We consider two approaches. First, we extend bounded model checking and develop a simple heuristic to generate highly probable paths first. We then complement the SAT-based approach by a fully (multi-terminal) BDD-based technique. All symbolic approaches are implemented, and our experimental results show a substantially better scalability than existing explicit techniques. In particular, our BDD-based approach using a method called *fragment search* allows for counterexample generation for DTMCs with billions of states (up to $10^{15}$).

Note:

# Contents

# 1 Introduction

*Model checking* is a very successful technique to automatically analyze the correctness of a system. During the last two decades, a lot of work has been done to develop model checking techniques for different kinds of systems like digital circuits and hybrid or probabilistic systems.

An important feature which made model checking for digital circuits a standard technology in industry is the ability to deliver a *counterexample* if a desired property is violated. Counterexamples, which provide an explanation for the violation, are indispensable for reproducing and fixing errors in the design. They are also crucial for so-called CEGAR (counterexample-guided abstraction refinement) frameworks [1, 2, 3], where a system is verified on an abstraction, which is gradually refined using (possibly spurious) counterexamples. The importance of counterexamples was stated by the Turing award winner Edmund Clarke in his talk at the celebration of 25 years of model checking [4]:

> *It is impossible to overestimate the importance of the counterexample feature. The counterexamples are invaluable in debugging complex systems. Some people use model checking just for this feature.*

This paper addresses counterexample generation for probabilistic systems modeled as *discrete-time Markov chains (DTMCs)* and properties formalized in *probabilistic computation tree logic (PCTL)* [5]. PCTL model checking of DTMCs has been widely studied and has successfully been employed for applications from distributed computing, security, hardware, and systems biology, to mention a few. Standard model checking algorithms for PCTL properties of DTMCs are based on probabilistic reachability analysis: the probability of reaching a given set of states is computed by solving a linear equation system [6]. These methods are implemented in popular probabilistic model checkers like PRISM [7] or MRMC [8].

However, if a PCTL property is violated, e. g., if the probability to reach a set of unsafe states is larger than a certain value, these model checking algorithms are not able to provide further information about this violation. Therefore, in the last years intensive research was carried out to develop methods which allow to generate *counterexamples* for PCTL properties of DTMCs. For digital circuits, a single execution that leads from an initial state to a safety-critical state suffices as a counterexample. Contrary, for DTMCs a *set* of such executions is required whose accumulated probability mass exceeds the maximally tolerated value. Some of the available counterexample generation methods [9, 10, 11, 12] represent counterexamples as sets of paths. Since the number of paths can be extremely large (or even infinite), alternative representations have been devised like regular expressions [12], winning strategies for probabilistic games [13, 14], or subsets of the state space [15, 16, 17]. Some of these approaches have been implemented in tools like DiPro [18], COMICS [19], and LTLSubsys [17], but scalability remains a serious issue.

Practically relevant systems are often too large to be represented explicitly, i. e., by enumerating their states and transitions. To overcome this problem, *symbolic model checking* using a representation by *binary decision diagrams (BDDs)* [20, 21] was introduced [22]. Sets of states and transitions are encoded by acyclic directed graphs, representing the elements in a set by paths in the graph. Symbolic model checking has been successfully established for DTMCs [23, 24].

For most of the available large case studies, as provided by PRISM [7], symbolic representations are smaller by orders of magnitude than explicit ones.

Given the enormous success of symbolic model checking techniques, our aim is to adopt these techniques for counterexample generation for DTMCs. In order to take full advantage of efficient symbolic representations, the applied path search methods should make full use of symbolic data structures, even for intermediate results. Some preliminary attempts towards this have been done [9, 10, 11], but they still rely on explicit representations at some points. For very large systems, these approaches are not scalable, as a counterexample may consist of a very huge or even infinite number of paths.

These partially symbolic approaches can be divided into two groups: Wimmer et al. [9, 10] apply bounded model checking [25] as path search method. Thereby the existence of a path of a certain length leading from the initial state to a target state is formulated as a *satisfiability problem*, which is solved by an appropriate solver. The drawback is that the computed paths are enumerated explicitly. Thus this method scales only to large systems, if the number of paths required for a counterexample is small. An alternative symbolic path search algorithm was introduced in [11]. This algorithm uses a *BDD-based representation* of the DTMC under consideration and calculates the $k$ most probable paths using an algorithm similar to Dijkstra's shortest path algorithm [26]. The number $k$ of paths is thereby adapted on the fly until a counterexample has been found. This approach enables a fully symbolic counterexample generation, but suffers from an exponential blow-up because the underlying graph essentially doubles for each found path by introducing two new BDD variables per path. Therefore it is also restricted to counterexamples consisting of few paths.

We show that also in counterexample generation, symbolic techniques can boost the scalability by several orders of magnitude. The central contribution of this paper is thus the development of *fully symbolic* algorithms, which overcome the main disadvantages of previous approaches:

- No explicit representation of states is needed during the counterexample generation.

- As in [15, 16], the counterexample is not represented by an enumeration of paths, but as a subsystem of the input DTMC, which yields a counterexample that is smaller by orders of magnitude.

- In comparison to other approaches we are now able to generate counterexamples for systems with billions of states.

In detail our technical contributions are as follows:

We first adapt SAT-based bounded model checking to support the search algorithms presented in [16] and suggest a heuristic for SAT-solving that allows to influence the SAT search to find more probable paths first, without the need to invoke SMT-solving. Furthermore, we do not restrict the search to paths of a fixed length as suggested by standard bounded model checking, but search for paths whose length is between a given lower and upper bound.

As a second approach, we propose novel fully symbolic methods for the generation of counterexamples for DTMCs and PCTL properties. Our methods take as input a DTMC which is symbolically represented by BDDs. We propose a symbolic version of the so-called *global search*

*approach* [16] to compute a symbolically represented subsystem of the original DTMC, whose paths form a counterexample. For this we adapt the symbolic *k*-shortest path search presented in [11] to find most probable paths of a DTMC. As this method suffers from very high memory consumption, we present an improved global search method which avoids the exponential blow-up of the symbolic *k*-shortest path search [11]. As our best approach, we adapt the idea for *fragment search*, also presented in [16]: Instead of searching for most probable paths from the initial state to a target state, we search for most probable path fragments extending the current subsystem. This scales as well as the improved global search, but yields typically more compact counterexamples.

We give a detailed experimental evaluation of all proposed algorithms. This includes a comparison to explicit methods considering the quality of results, running times, and the memory consumption. As the experiments will show, we were able to generate counterexamples for DTMCs of up to $10^{15}$ states, which poses a very difficult task even for mere model checking.

This paper is an extended version of the conference paper [27]. The extension encompasses the more extensive treatment of the foundations and an ongoing example. Most importantly, we present an improved version of the symbolic global search method which avoids an exponential blow-up of the underlying graph as well as an improved version of the symbolic fragment search. We give a more detailed experimental evaluation, comparing our approaches with the available explicit methods.

In Section 2 we introduce some theoretical foundations. Section 3 describes the general framework of our symbolic methods for counterexample generation. The usage of SAT-based path search is described in Section 4 and the application of BDD-based graph search algorithms in Section 5. Related work and connections or differences to other approaches are discussed in Section 6. All approaches are evaluated experimentally on a number of case studies in Section 7 including a detailed comparison with other tools and methods. We conclude our work and discuss future work in Section 8.

# 2   Preliminaries

We start with introducing some basic definitions and concepts used in this paper. For more details we refer to, e. g., [6, Chapter 10].

## 2.1   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 discrete probability distributions as follows.

**Definition 1** *A* discrete-time Markov chain (DTMC) *is a tuple* $M = (S, s_I, P, L)$ *with S being a finite set of states,* $s_I \in S$ *the initial state,* $P : S \times S \rightarrow [0, 1] \subseteq \mathbb{Q}^1$ *a matrix of transition*

---

[1]In theory, the definition of a DTMC allows probabilities from $[0, 1] \subseteq \mathbb{Q}$. However, due to algorithmic reasons we use only values from $\mathbb{Q}$.

*probabilities such that $\sum_{s' \in S} P(s, s') \leq 1$ for all $s \in S$, and $L$ a labeling function with $L : S \to 2^{AP}$ with AP a denumerable set of atomic propositions.*

Please note that we generalize the standard definition and allow *sub-stochastic* distributions $\sum_{s' \in S} P(s, s') \leq 1$ for all $s \in S$. Usually, these sums of probabilities are required to be exactly 1. This can be obtained by defining the transformation $\alpha_{s_\perp}$ of $M$ as the DTMC $\alpha_{s_\perp}(M) = M' = (S', s'_I, P', L')$ with

- $S' = S \,\dot\cup\, \{s_\perp\}$ for a fresh sink state $s_\perp \notin S$,

- $s'_I = s_I$,

- $P'(s, s') = \begin{cases} P(s, s') & \text{for } s, s' \in S, \\ 1 - \sum_{s'' \in S} P(s, s'') & \text{for } s \in S \text{ and } s' = s_\perp, \\ 1 & \text{for } s = s' = s_\perp, \\ 0 & \text{otherwise (for } s = s_\perp \text{ and } s' \in S), \end{cases}$ and

- $L'(s) = L(s)$ for $s \in S$ and $L'(s_\perp) = \emptyset$.

According to the DTMC semantics below, the reachability probabilities in $M$ and $\alpha_{s_\perp}(M)$ 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.

Assume in the following a DTMC $M = (S, s_I, P, L)$. We say that there is a *transition* $(s, s')$ from a state $s \in S$, the source, to a state $s' \in S$, the target, iff $P(s, s') > 0$. A *path* of $M$ is a finite or infinite sequence $\pi = s_0 s_1 \ldots$ of states $s_i \in S$ such that $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. We say that the transitions $(s_i, s_{i+1})$ are *contained* in the path $\pi$, written $(s_i, s_{i+1}) \in \pi$. Starting with $i = 0$, we write $\pi^i$ for the $i^{\text{th}}$ state $s_i$ on path $\pi$; the position $i$ is called its *depth*. The *length* $|\pi|$ of a finite path $\pi = s_0 \ldots s_n$ is the number $n$ of its transitions. The last state of $\pi$ is denoted by $\text{last}(\pi) = s_n$.

By $Paths_{inf}^M$ we denote the set of all infinite paths of $M$, by $Paths_{inf}^M(s)$ those starting in $s \in S$. Similarly, $Paths_{fin}^M$ is the set of all finite paths of $M$, $Paths_{fin}^M(s)$ those starting in $s \in S$, and $Paths_{fin}^M(s, t)$ those starting in $s \in S$ and ending in $t \in S$. A state $t \in S$ is *reachable* from another state $s \in S$ iff $Paths_{fin}^M(s, t) \neq \emptyset$.

We follow the standard way [28] to define for each state $s \in S$ a probability space $(\Omega_s^M, \mathcal{F}_s^M, Pr_s^M)$ on the infinite paths of a DTMC starting in $s$. The sample space $\Omega_s^M$ is the set $Paths_{inf}^M(s)$. The set $\mathcal{F}_s^M$ of events is defined as follows: The *cylinder set* of a finite path $\pi$ of $M$ is defined as $Cyl(\pi) = \{\pi' \in Paths_{inf}^M \mid \pi \text{ is a prefix of } \pi'\}$. The set $\mathcal{F}_s^M$ of events is the unique smallest $\sigma$-algebra that contains the cylinder sets of all finite paths in $Paths_{fin}^M(s)$. $Pr_s^M$ (or short $Pr$) is the unique probability measure on $\mathcal{F}_s^M$ such that the probabilities of the cylinder sets are given by

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

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

## 2.2   Probabilistic CTL and Critical Subsystems

*Probabilistic computation tree logic (PCTL)* [5] enriches CTL with an operator $\mathcal{P}$ arguing about the total probability of paths satisfying some properties. The syntax of PCTL is given by the following context-free grammar:[2]

$$\Phi ::= \text{true} \mid p \mid \neg\Phi \mid (\Phi \wedge \Phi) \mid \mathcal{P}_{\triangleleft\lambda}(\Phi \, \mathcal{U} \, \Phi)$$

for (state) formulae $\Phi$ with $p \in AP$, $\lambda \in [0,1] \subseteq \mathbb{Q}$, and $\triangleleft \in \{<, \leq, \geq, >\}$. For a PCTL state formula $\varphi$ we define the "finally"-operator $\Diamond$ as $\mathcal{P}_{\triangleleft\lambda}(\Diamond\varphi) = \mathcal{P}_{\triangleleft\lambda}(\text{true} \, \mathcal{U} \, \varphi)$ and the "globally"-operator $\Box$ as $\mathcal{P}_{\triangleleft\lambda}(\Box\varphi) = \mathcal{P}_{\triangleright 1-\lambda}(\text{true} \, \mathcal{U} \, \neg\varphi)$ where $\triangleright$ is $>, \geq, \leq, <$ if $\triangleleft$ is $<, \leq, \geq, >$, respectively.

We define a state $s$ of the DTMC $M$ to satisfy a PCTL formula $\varphi$, written $M, s \vDash \varphi$, recursively as follows:

| | | |
|---|---|---|
| $M, s \vDash \text{true}$ | | always, |
| $M, s \vDash p$ | $\Leftrightarrow$ | $p \in L(s)$, |
| $M, s \vDash \neg\varphi$ | $\Leftrightarrow$ | $M, s \nvDash \varphi$, |
| $M, s \vDash (\varphi_1 \wedge \varphi_2)$ | $\Leftrightarrow$ | $M, s \vDash \varphi_1$ and $M, s \vDash \varphi_2$, |
| $M, s \vDash \mathcal{P}_{\triangleleft\lambda}(\varphi_1 \, \mathcal{U} \, \varphi_2)$ | $\Leftrightarrow$ | $Pr(\{\pi \in Paths_{inf}^M(s) \mid \exists i \geq 0. \, (M, \pi^i \vDash \varphi_2 \wedge \forall 0 \leq j < i. \, M, \pi^j \vDash \varphi_1)\}) \triangleleft \lambda$. |

A DTMC $M$ satisfies a PCTL-Formula $\varphi$, written $M \vDash \varphi$, if its initial state $s_I$ does, i.e., if $M, s_I \vDash \varphi$.

The model checking and counterexample generation problems for $\mathcal{P}_{\leq\lambda}(\varphi_1 \, \mathcal{U} \, \varphi_2)$ can be reduced to a reachability problem as follows: We transform the DTMC $M = (S, s_I, P, L)$ to a DTMC $M' = (S, s_I, P', L)$ by removing all outgoing transitions from states satisfying $\neg\varphi_1 \vee \varphi_2$, i.e., for all $s \in S$ we have $P'(s, s') = 0$ if $s$ satisfies $\neg\varphi_1 \vee \varphi_2$ and $P'(s, s') = P(s, s')$ otherwise. Then $M$ satisfies $\mathcal{P}_{\leq\lambda}(\varphi_1 \, \mathcal{U} \, \varphi_2)$ iff $M'$ satisfies $\mathcal{P}_{\leq\lambda}(\Diamond \, \varphi_2)$. In the following we concentrate on this reduced problem and also write $\mathcal{P}_{\leq\lambda}(\Diamond \, T)$ instead of $\mathcal{P}_{\leq\lambda}(\Diamond \, \varphi_2)$, where $T = \{s \in S \mid M, s \vDash \varphi_2\}$ is the set of those *target states* that satisfy $\varphi_2$.

Checking properties $\mathcal{P}_{\leq\lambda}(\Diamond \, T)$ consists of (1) computing the set of states $T = \{s \in S \mid M, s \vDash \varphi_2\}$ which satisfy $\varphi_2$, and (2) computing for each state $s \in S$ the probability $p_s$ to finally reach a state in $T$. These probabilities are the unique solution of the linear equation system [6, Theorem 10.19]:

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

containing one equation for each $s \in S$.

Consider a DTMC $M = (S, s_I, P, L)$ and a PCTL property $\mathcal{P}_{\leq\lambda}(\Diamond \, T)$ specifying an upper bound on the probability that, starting from the initial state, a state from $T$ will be reached. If this property is violated, a *counterexample* is a set $C \subseteq Paths_{fin}^M(s_I)$ of finite paths starting in $s_I$ such that all paths of the cylinder sets *satisfy* $\Diamond \, T$ and $Pr_{fin}(C) > \lambda$. For $\mathcal{P}_{<\lambda}(\Diamond \, T)$, the probability mass

---

[2]In this paper we do not consider properties involving bounded reachability.

has to be at least $\lambda$. We consider only upper bounds here; see [12] for the reduction of lower bounds to this case.

In [16] we proposed to represent counterexamples as so-called *critical subsystems* instead of large, possibly infinite, sets of paths. Intuitively, a critical subsystem is a part of the original system in which the given probability bound is already exceeded, no matter what happens outside the subsystem. The advantages of taking a subsystem instead of a set of paths as representation of a counterexample concern both the computation time and the representation size: In [16] we utilized the $k$-shortest path algorithm as proposed in [12] for the incremental construction of such a subsystem. Thereby, every new path is added to the current subsystem. The probability mass of the whole subsystem including all occurring loops is taken into account which significantly improved the running times. In many cases, only a few paths are needed to form a critical subsystem while a counterexample represented as a set of paths needs millions of paths. Furthermore, already in [12] it was shown that the size of a counterexample may be double exponential in the problem size, while the number of states of a critical subsystem is always bounded by the size of the input system.

**Definition 2** *A subsystem of a DTMC $M = (S, s_I, P, L)$ is a DTMC $M' = (S', s_I, P', L')$ such that $S' \subseteq S$, $s_I \in S'$, $P'(s, s') \in \{P(s, s'), 0\}$ and $L'(s) = L(s)$ for all $s, s' \in S'$.*
*For a violated PCTL property $\varphi$ with $M, s_I \not\models \varphi$, we call a subsystem $M'$ of $M$ critical for $\varphi$ iff $M', s_I \not\models \varphi$.*

Note that the set $\bigcup_{t \in T} Paths_{fin}^{M'}(s_I, t)$ of all finite paths of a critical subsystem $M'$ from $s_I$ to a target state forms a counterexample. We will conclude with an example illustrating the concepts introduced so far.

**Example 1** *Consider the DTMC $M$ depicted in Figure 1(a). The unique initial state is $s_I = s_0$, indicated by the incoming arrow, and the only target state is $s_3$. We are interested in the probability of reaching $s_3$ from the initial state, i. e., the probability of all paths leading from $s_0$ to $s_3$. Solving the linear equation system (1) yields a reachability probability $p_{s_0} = 0.55$. Therefore, the PCTL property $\varphi = \mathcal{P}_{\leq 0.3}(\lozenge s_3)$ is violated in the initial state.*

*Collecting the most probable paths as in [12] to generate a counterexample yields the following paths:*

$$\pi_1 = s_0 s_1 s_3 \qquad\qquad\qquad \textit{probability: } 0.25$$
$$\pi_2 = s_0 s_1 s_2 s_1 s_3 \qquad\qquad \textit{probability: } 0.0625$$
$$\pi_3 = s_0 s_5 s_3 \qquad\qquad\qquad \textit{probability: } 0.05$$

*The set $C = \{\pi_1, \pi_2, \pi_3\}$ has probability $p_C = 0.3625$ and forms a counterexample for $\varphi$. Please note that for a probability bound $\lambda = 0.55 - \varepsilon$ for a small $\varepsilon > 0$, the number of required paths heads to infinity for $\varepsilon \to 0$. Most of these paths only differ in the number and order of iterations of the same loops.*

*However, $\pi_1$ and $\pi_2$ already induce a subsystem $M'$ of $M$, depicted in Figure 1(b), inside which the probability of reaching $s_3$ from $s_0$ is $p_{s_0} = 0.\bar{3}$. Therefore, $M'$ is a critical subsystem for $M$ and $\varphi$.*
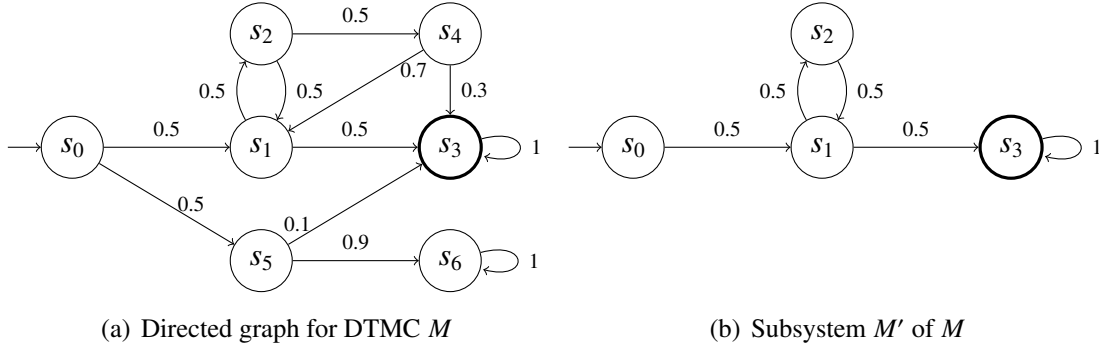
(a) Directed graph for DTMC $M$                    (b) Subsystem $M'$ of $M$

Figure 1: The graph induced by a DTMC $M$ and a subsystem $M'$ which violates $\mathcal{P}_{\leq 0.3}(\Diamond\, s_3)$.

## 2.3   Symbolic Representation of DTMCs

In an *explicit* representation of Markov chains, the transition probabilities are stored as a sparse matrix, which contains one entry per transition with non-zero probability. Its size is therefore linear in the number of states and transitions. This representation is used, for instance, by the probabilistic model checker MRMC [8].

   We use *symbolic* representations to encode state and transition sets, e. g., as paths in a graph or as solutions of a certain formula. Symbolic representations are in practice often smaller by orders of magnitude than explicit ones and allow to reduce not only the memory consumption but also the computational costs for operations on the data structures.

   As a symbolic data structure for the representation of DTMCs we choose *binary decision diagrams* (BDDs) [20] and *multi-terminal binary decision diagrams* (MTBDDs) [21]. (MT)BDDs have been applied very successfully for verification of digital circuits [22] and also play a role in verification of probabilistic and stochastic systems [29, 30, 24]. However, they have the drawback that for some systems the representation is large (e. g., for multiplier circuits [31]), and that their size can strongly depend on the ordering of the variables. An optimal ordering, however, is hard to find [32], but good heuristics are available [33, 24].

**Definition 3** *Let Var be a finite set of Boolean variables. A* binary decision diagram *(BDD) over Var is a rooted, acyclic, directed graph $B = (V, n_{\mathrm{root}}, E)$ with a finite set $V$ of nodes, a root node $n_{\mathrm{root}} \in V$ and edges $E \subseteq V \times V$. Each node is either an* inner *node or a* leaf *node. Leaf nodes $n \in V$ have no outgoing edges and are labeled with $label(n) \in \{0, 1\}$. Inner nodes $n \in V$ have exactly* two *successor nodes, denoted by $hi(n)$ and $lo(n)$, and are labeled with a variable $label(n) \in Var$.*

   *A* multi-terminal binary decision diagram *(MTBDD) is like a BDD but it labels leaf nodes $n \in V$ with rational values $label(n) \in \mathbb{Q}$.*

   Let $B$ be a BDD over *Var* and $\mathcal{V}(Var) = \{v : Var \to \{0, 1\}\}$ the set of all variable valuations. Each $v \in \mathcal{V}(Var)$ induces a unique path in $B$ from the root to a leaf node by moving from each inner node $n$ to $hi(n)$ if $v(label(n)) = 1$ and to $lo(n)$ otherwise. A BDD $B$ represents a function $f_B : \mathcal{V}(Var) \to \{0, 1\}$ assigning to each $v \in \mathcal{V}(Var)$ the label of the leaf node reached in $B$ by following the path induced by $v$. We often identify $B$ with $f_B$ and write $B(v)$ instead of $f_B(v)$. Analogously, each MTBDD $B$ represents a function $f_B : \mathcal{V}(Var) \to \mathbb{Q}$.

9

(a) BDD $B_1$ representing the state space of DTMC $M'$    (b) BDD $B_2$, reduced version of $B_1$
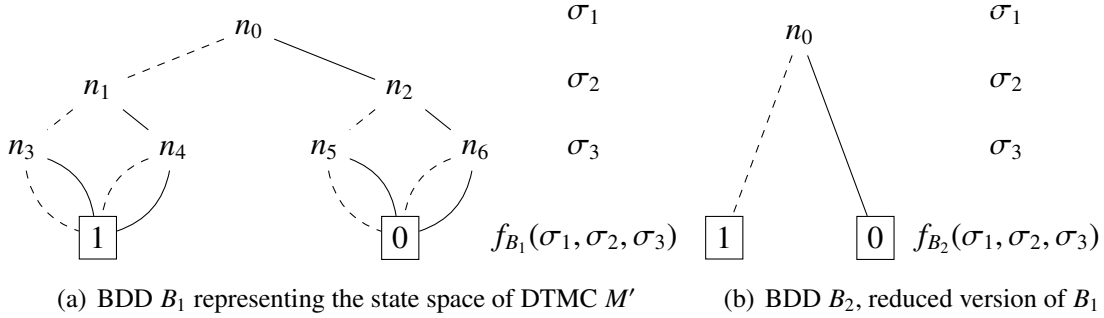
Figure 2: This figure shows two equivalent BDD representations of the state space of the subsystem $M'$ of $M$, depicted in Figure 1.

An (MT)BDD is *ordered* if there is a linear order $< \subseteq Var \times Var$ on the variables such that for all inner nodes $n$ either $hi(n)$ is a leaf node or $label(n) < label(hi(n))$, and the same for $lo(n)$. An (MT)BDD is *reduced* if all functions rooted at different nodes are different. For a fixed variable order, reduced (MT)BDDs are canonical data structures for representing functions $f : \mathcal{V}(Var) \to \{0, 1\}$ resp. $f : \mathcal{V}(Var) \to \mathbb{Q}$ [20]. In the following we assume all (MT)BDDs to be reduced and ordered with respect to a fixed variable order.

By $Var'$ we denote the variable set $Var$ with each variable $x \in Var$ renamed to some $x' \in Var'$ such that $Var \cap Var' = \emptyset$. Our algorithms use standard BDD operations like ITE (if-then-else) to implement union $B_1 \cup B_2$ and intersection $B_1 \cap B_2$, variable renaming $B[x \to x']$, and existential quantification $\exists x.B$ for $x \in Var$, $x' \in Var'$. For MTBDDs additionally APPLY and ABSTRACT are used to perform numerical operations. For details on these operations in the context of symbolic path search we refer to [34].

BDDs and MTBDDs can be used to represent DTMCs symbolically as follows: Let $M = (S, s_I, P, L)$ be a DTMC and $Var$ a set of Boolean variables such that for each $s \in S$ there is a unique binary encoding $v_s : Var \to \{0, 1\}$ with $v_s \neq v_{s'}$ for all $s, s' \in S$, $s \neq s'$. For $s, s' \in S$ we also define $v_{s,s'} : Var \,\dot{\cup}\, Var' \to \mathbb{Q}$ with $v_{s,s'}(x) = v_s(x)$ and $v_{s,s'}(x') = v_{s'}(x')$ for all $x \in Var$, $x' \in Var'$. A target state set $T \subseteq S$ is represented by a BDD $\hat{T}$ over $Var$ such that $\hat{T}(v_s) = 1$ iff $s \in T$. Similarly for the initial state, $\hat{I}(v_s) = 1$ iff $s = s_I$. The probability matrix $P : S \times S \to [0, 1] \subseteq \mathbb{Q}$ is represented by an MTBDD $\hat{P}$ over $Var \,\dot{\cup}\, Var'$ such that $\hat{P}(v_{s,s'}) = P(s, s')$ for all $s, s' \in S$. For an MTBDD $B$ over $Var$ we use $B_{\text{bool}}$ to denote the BDD over $Var$ with $B_{\text{bool}}(v) = 1$ iff $B(v) > 0$ for all valuations $v$.

The transition matrices of practically relevant systems are usually sparse and well-structured with relatively few different probabilities; therefore the symbolic MTBDD representation is in many cases more compact by several orders of magnitude than explicit representations.

**Example 2** *In order to represent the 7 states of the DTMC $M$ from Figure 1(a) symbolically, we use the variable set $Var = \{\sigma_1, \sigma_2, \sigma_3\}$ with $\sigma_1 < \sigma_2 < \sigma_3$. To represent its transitions, we extend this set by a copy of itself: $Var \,\dot{\cup}\, Var' = \{\sigma_1, \sigma_2, \sigma_3, \sigma'_1, \sigma'_2, \sigma'_3\}$. A possible unique encoding of the states as well as the transitions is given by the following assignments:*

|       | $\sigma_1$ | $\sigma_2$ | $\sigma_3$ |
|-------|------------|------------|------------|
| $s_0$ | 0 | 0 | 0 |
| $s_1$ | 0 | 0 | 1 |
| $s_2$ | 0 | 1 | 0 |
| $s_3$ | 0 | 1 | 1 |
| $s_4$ | 1 | 1 | 1 |
| $s_5$ | 1 | 1 | 0 |
| $s_6$ | 1 | 0 | 0 |

|                       | $\sigma_1$ | $\sigma_2$ | $\sigma_3$ | $\sigma'_1$ | $\sigma'_2$ | $\sigma'_3$ |
|-----------------------|------------|------------|------------|-------------|-------------|-------------|
| $s_0 \rightarrow s_1$ | 0 | 0 | 0 | 0 | 0 | 1 |
| $s_0 \rightarrow s_5$ | 0 | 0 | 0 | 1 | 1 | 0 |
| $s_1 \rightarrow s_2$ | 0 | 0 | 1 | 0 | 1 | 0 |
| $s_1 \rightarrow s_3$ | 0 | 0 | 1 | 0 | 1 | 1 |
| $s_2 \rightarrow s_1$ | 0 | 1 | 0 | 0 | 0 | 1 |
| $s_2 \rightarrow s_4$ | 0 | 1 | 0 | 1 | 1 | 1 |
| $s_3 \rightarrow s_3$ | 0 | 1 | 1 | 0 | 1 | 1 |
| $s_4 \rightarrow s_1$ | 1 | 1 | 1 | 0 | 0 | 1 |
| $s_4 \rightarrow s_3$ | 1 | 1 | 1 | 0 | 1 | 1 |
| $s_5 \rightarrow s_3$ | 1 | 1 | 0 | 0 | 1 | 1 |
| $s_5 \rightarrow s_6$ | 1 | 1 | 0 | 1 | 0 | 0 |
| $s_6 \rightarrow s_6$ | 1 | 0 | 0 | 1 | 0 | 0 |

Based on the this encoding, the BDD $B_1$ in Figure 2(a) represents the state space of the subsystem $M'$ of $M$ from Figure 1(b). Node $n_0$ is labeled with the variable $\sigma_1$, $n_1$ and $n_2$ are labeled with $\sigma_2$, and $n_3, n_4, n_5, n_6$ with $\sigma_3$. Thus each level corresponds to the choice of the value for exactly one variable. The leaves labeled with 0 and 1 indicate whether the function $f_{B_1}(\sigma_1, \sigma_2, \sigma_3)$ is evaluated to 0 or 1. Dashed edges indicate that the variable at whose level the edge starts, is set to 0, solid edges that it is set to 1.

Consider the path $n_0, n_1, n_3, 1$ which results from choosing the low successor for each inner node. This path is induced by the assignment $\nu_{s_0}$ with $\nu_{s_0}(\sigma_1) = \nu_{s_0}(\sigma_2) = \nu_{s_0}(\sigma_3) = 0$ and has the evaluation $f_{B_1}(0, 0, 0) = 1$. Thus, state $s_0$ is part of the set encoded by this BDD. Consider furthermore the path $n_0, n_2, n_6, 0$. As this corresponds to state $s_4$ and evaluates to 0, the state $s_4$ is not included in this set.

The BDD $B_2$ in Figure 2(b) encodes the same state set as $B_1$ but it is reduced. Since in $B_1$ the choice of assignment for variable $\sigma_1$ already determines the evaluation of the whole function, all intermediate nodes after $n_0$ can be eliminated.

Finally, the transition matrix of the DTMC $M'$ can be encoded by the MTBDD $B_3$ in Figure 3. For each $s, s' \in S$, the path induced by the assignment $\nu_{s,s'}$ leads to a leaf that is labeled with the probability $P(s, s')$ to move from $s$ to $s'$ in $M'$. For example, the path $n_0, n_1, n_2, n_4, n_8, n_{13}, 0.5$ is induced by the assignment $\nu_{s_0, s_1}$, which corresponds to the transition between the states $s_0$ and $s_1$ with probability $0.5$. This MTBDD is already reduced. Please note, that in our implementation we use an interleaved variable ordering for the transition MTBDD, i. e., the levels would be in the order $\sigma_1, \sigma'_1, \sigma_2, \sigma'_2, \sigma_3, \sigma'_3$. We refrained from this ordering as a transition is easier to read with a non-interleaved ordering.

# 3   Symbolic Counterexample Generation Framework

In this section we present our framework for the generation of probabilistic PCTL counterexamples using symbolic data structures.
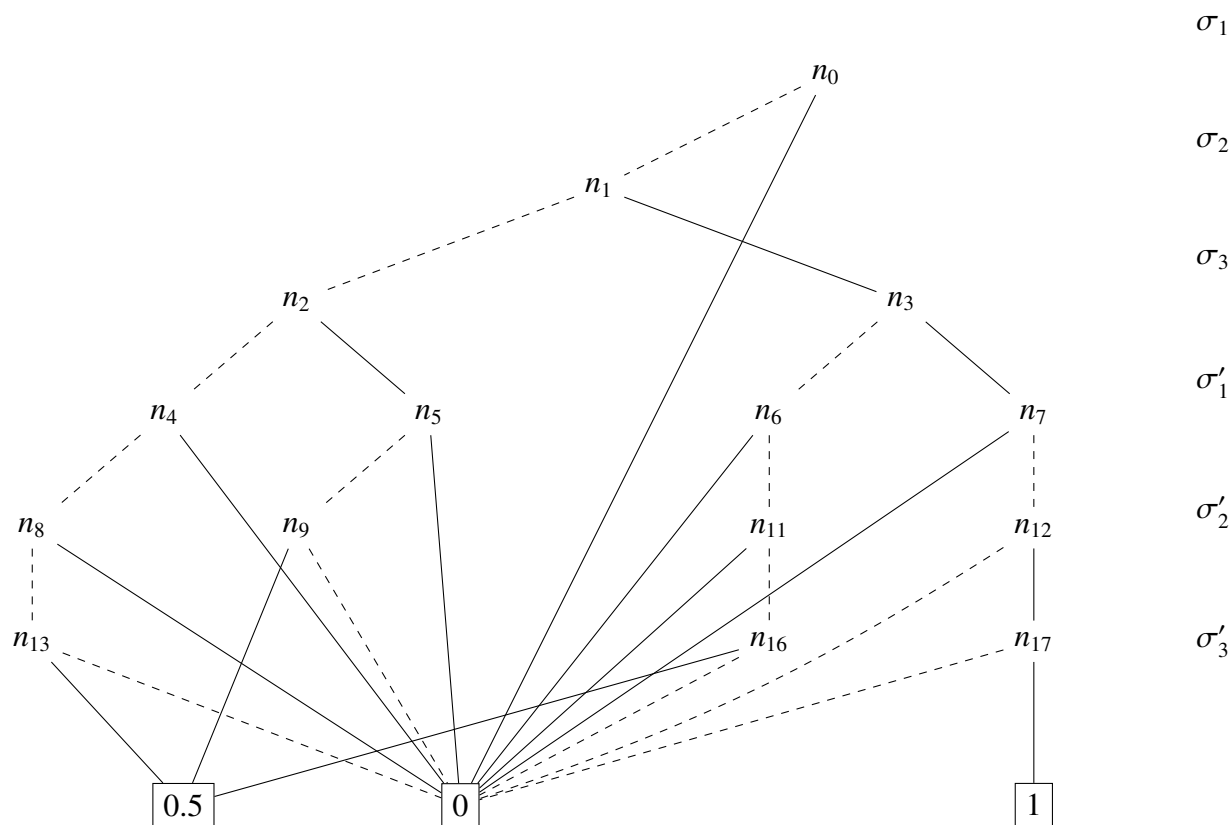
Figure 3: MTBDD $B_3$ representing the transition matrix of the DTMC $M'$.

We provide an algorithm that computes, for the symbolic representation of a DTMC as input, a critical subsystem, which is again symbolically represented. As the most significant ingredient, this algorithm needs a *symbolic path search* method, which returns paths of the input DTMC. The critical subsystem is initially empty and gets incrementally extended with the states of found paths and the transitions between the (old and new) states. Symbolic implementations of the path search method will be described in Sections 4 and 5.

## 3.1 The Framework

The algorithm for finding a symbolic counterexample is depicted in Algorithm 1. The parameters specify the input DTMC symbolically by the MTBDD $\hat{P}$ for the transition probability matrix, the BDD $\hat{I}$ for the initial state and the BDD $\hat{T}$ for the target states, as well as a probability bound $\lambda \in [0, 1] \subseteq \mathbb{R}$, which shall be exceeded by the resulting critical subsystem. The BDD *States* is used to symbolically represent the set of states which are part of the current subsystem, while *NewStates* is used to store the states occurring on a path or on a set of paths which shall extend the current subsystem. The MTBDD *SubSys* stores the transition MTBDD of the current subsystem. The algorithm uses the following methods:

ModelCheck(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$) performs symbolic probabilistic model checking [23,

---

**Algorithm 1** Incremental generation of critical subsystems

---

**FindCriticalSubsystem**(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$, double $\lambda$)
**begin**
    BDD *States* := 0, *NewStates* := 0;   MTBDD *SubSys* := 0;              (1)
    **if** `ModelCheck`$(\hat{P}, \hat{I}, \hat{T}) > \lambda$ **then**                        (2)
        **while** `ModelCheck`(*SubSys*, $\hat{I}, \hat{T}$) $\leq \lambda$ **do**           (3)
            *NewStates* := `FindNextPath`($\hat{P}, \hat{I}, \hat{T}$, *SubSys*);     (4)
            **if** *NewStates* $\neq 0$ **then**                    (5)
                *States* := *States* $\cup$ *NewStates*;          (6)
                *SubSys* := `ToTransitionBDD`(*States*) $\cdot \hat{P}$;   (7)
            **end if**                            (8)
        **end while**                         (9)
    **end if**                              (10)
    **return** *SubSys*                         (11)
**end**

---

    24] and returns the probability of reaching states in $\hat{T}$ from states in $\hat{I}$ via transitions in $\hat{P}$.

`FindNextPath`(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$, MTBDD *SubSys*) computes a set of states which occur on a path leading through the DTMC represented by the transition MTBDD $\hat{P}$, the initial state $\hat{I}$, and the set of target states $\hat{T}$. Which path is found next depends on the current subsystem *SubSys* and therefore on the set of previously found paths. The method can return states occurring on one path or on a set of paths. Different symbolic implementations of this method will be discussed in Sections 4 and 5.

`ToTransitionBDD`(BDD *States*) first computes the BDD *States'* by renaming each variable $x \in$ *Var* occurring in *States* to $x' \in$ *Var'* and returns the transition BDD *States* $\cap$ *States'* in which there is a transition between all pairs of states occurring in *States*, i.e., (*States* $\cap$ *States'*)($v_{s_1,s_2}$) = 1 iff *States*($v_{s_1}$) = *States*($v_{s_2}$) = 1. Intuitively, this yields a BDD inducing the complete directed graph over *States*, i.e., all states are connected to each other. Multiplying this BDD with the transition probability matrix $\hat{P}$ removes all transitions from $\hat{P}$ which do not connect two states of the subsystem.

    The algorithm proceeds as follows. First, the three empty objects *States*, *NewStates*, and *SubSys* are created in line (1). If `ModelCheck`($\hat{P}, \hat{I}, \hat{T}$) shows that $\lambda$ is exceeded, the reachability property is violated and the search for a counterexample starts (line 2). Otherwise, the algorithm just terminates, returning an empty subsystem since no counterexample exists if the property is not violated. The condition of the `while`-loop in line (3) invokes model checking for the current subsystem *SubSys* and the initial states and target states. The loop runs until `ModelCheck`(*SubSys*, $\hat{I}, \hat{T}$) returns a value which is greater than $\lambda$. In this case, the current subsystem is *critical*. Please note that calling a model checking algorithm in each iteration is quite costly. Depending on the input system, we search for a certain number of paths until we

invoke model checking. In every iteration, first the method FindNextPath($\hat{P}, \hat{I}, \hat{T}, SubSys$) in line (4) returns a set of states which occur on a path or a set of paths through the system. If this set is not empty, the current set of states is extended by these new states (line 6). Afterward, the current subsystem is extended (line 7): ToTransitionBDD(*States*) generates a transition relation between all found states. Multiplying the resulting BDD and the original transition MTBDD $\hat{P}$ yields a probability matrix $P' \subseteq P$ restricted to transitions between the states in *States*. These transitions define the updated subsystem *SubSys*.

## 3.2  Path Search Concepts

We distinguish between two basic concepts of searching for paths: *global search* and *fragment search* [16]. Global search finds paths that start in the initial state $s_I$ of the system and end in a target state $t \in T$. Fragment search searches for paths that connect already found states but visit only new states in between. Each symbolic search method presented in this paper will follow one of these concepts.

**Example 3** *Consider again the DTMC M in Figure 1(a). If we search for paths in decreasing order of their probability, the following two most probable paths are found:*

$$\longrightarrow s_0 \xrightarrow{\ 0.5\ } s_1 \xrightarrow{\ 0.5\ } s_3 \circlearrowleft 1$$

$$\longrightarrow s_0 \xrightarrow{\ 0.5\ } s_1 \xrightarrow{\ 0.5\ } s_2 \xrightarrow{\ 0.5\ } s_1 \xrightarrow{\ 0.5\ } s_3 \circlearrowleft 1$$

*The second path contains a state repetition of $s_1$ as it uses the corresponding loop. Note, that for the global approach paths may differ only in the order and/or the number of loop iterations. For instance, the second pat might occur many times with arbitrary many unrollings of the loop $s_1 s_2 s_1$.*

*The first two paths according to fragment search for most probable path fragments are as follows:*

$$\longrightarrow s_0 \xrightarrow{\ 0.5\ } s_1 \xrightarrow{\ 0.5\ } s_3 \circlearrowleft 1 \qquad s_1 \xrightarrow{\ 0.5\ } s_2 \xrightarrow{\ 0.5\ } s_1$$

*The first path is the same as for global search, as initially only initial and target states can be connected. The second path connects already found states: The most probable connection is the loop between $s_1$ and $s_2$. In this example the global and the fragment search both build the same subsystem presented in Figure 1(b).*

## 3.3  Complexity of the Framework

The complexity of BDD-based algorithms strongly depends on the size of the BDDs. In terms of variables, their size can in general only be bounded by $O(\frac{2^n}{n})$ where $n$ is the number of variables the BDD depends on [35]. In terms of non-zero elements, every matrix of dimension $m \times m$ with $k$ non-zero elements can be represented—independently from the variable order—with $O(k \cdot \log m)$

nodes [21], which is even in the worst-case competitive to explicit sparse matrix representations. For practical cases, which often contain many symmetries, the size is typically much smaller than this upper bound.

The SAT problem, which we utilize to search for paths, is NP-complete [36], i. e., all available algorithms have an exponential worst-case running time in the number of variables. Nevertheless, practical problems inducing millions of clauses and hundreds of thousands of variables can be often solved quickly using modern SAT solvers [37].

Due to the great gap between the worst case and the practically experienced complexity, a worst-case analysis of these algorithms makes little sense. We will instead give an experimental evaluation in Section 7.

# 4   Searching Paths Using SAT Solving

In this section we present two implementations for the method `FindNextPath(...)` (as invoked by Algorithm 1) using bounded model checking and SAT solving. First, an existing method which searches for paths of certain lengths is adapted to our symbolic framework, giving us a global search procedure. Second, we present a new method which looks for path fragments that extend a subsystem. Finally, we describe a new SAT-solving heuristic which guides the SAT solver to prefer more probable path fragments.

## 4.1   Adapting Bounded Model Checking for Global Search

In [9], a bounded model checking (BMC) approach for DTMCs was developed. The input is a symbolic representation of a DTMC $M = (S, s_I, P, L)$ and a set of target states, i. e., BDDs $\hat{I}$ and $\hat{T}$, and an MTBDD $\hat{P}$ over a variable set $Var = \{\sigma_1, \ldots, \sigma_m\}$ as described earlier. We assume target states to be *deadlock states*, i. e., without outgoing transitions.

First, Tseitin's transformation [38] is applied to generate formulae in conjunctive normal form (CNF) for the BDDs $\hat{I}$, $\hat{T}$ and $\hat{P}_{\text{bool}}$, where the latter represents the BDD for the induced non-probabilistic transition relation for $\hat{P}$ without any outgoing transitions from target states. We denote the resulting CNF predicates by $\check{I}(Var)$, $\check{T}(Var)$ and $\check{P}(Var, Var')$, respectively. The BMC formula is parametric in $k \in \mathbb{N}$ and has the following structure:

$$BMC(k) \; = \; \check{I}(Var_0) \; \wedge \; \bigwedge_{i=0}^{k-1} \check{P}(Var_i, Var_{i+1}) \; \wedge \; \check{T}(Var_k) \,. \tag{2}$$

The solution set of $BMC(k)$ corresponds to the set of paths of length $k$ from the initial to a target state, where for each $i = 0, \ldots, k$ the set $Var_i = \{\sigma_{i,1}, \ldots, \sigma_{i,m}\}$ of Boolean variables is used to encode the state at depth $i$ on a path. That means, a satisfying assignment $v : \bigcup_{i=0}^{k} Var_i \rightarrow \{0, 1\}$ encodes the $i^{\text{th}}$ state on the path by $v_i : Var_i \rightarrow \{0, 1\}$ with $v_i(\sigma_j) = v(\sigma_{i,j})$ for each $j = 1, \ldots, m$. If there is no satisfying assignment, there is no such path.

Usually multiple paths need to be found in order to form a counterexample, thus the solver has to enumerate satisfying solutions for $BMC(k)$, $k = 0, 1, \ldots$, until enough probability mass

has been accumulated. Note that target states have no outgoing transitions in the encoding, i. e., paths end in the first target state that is reached and therefore two different paths from the initial to a target state are never prefixes of each other. Therefore their corresponding cylinder sets are disjoint and their joint probability is the sum of their individual probabilities.

To assure that a path is not considered several times, each time a solution is found it is excluded from further search by adding new clauses to the SAT solver's clause database. Assume that the solver has found a solution $v : \bigcup_{i=0}^{k} Var_i \to \{0, 1\}$ for $BMC(k)$. The found path is uniquely described by the following conjunction:

$$\bigwedge_{i=0}^{k} \bigwedge_{j=1}^{m} \sigma_{i,j}^{v(\sigma_{i,j})} , \tag{3}$$

where $\sigma_{i,j}^{1} = \sigma_{i,j}$ and $\sigma_{i,j}^{0} = \neg \sigma_{i,j}$. To exclude the found path from the solution space of $BMC(k)$, the negation of the above conjunction is added to the solver's clause database:

$$\bigvee_{i=0}^{k} \bigvee_{j=1}^{m} \sigma_{i,j}^{1-v(\sigma_{i,j})} . \tag{4}$$

This ensures that for a new path at least one state variable has to be assigned differently as it is done by $v$.

Termination of the iterative construction of a critical subsystem is guaranteed, as the SAT solver finds *all* paths of length $k$. Eventually, the subsystem will consist of all states that are part of paths from initial to target states. This subsystem induces the whole probability mass of reaching a target state in the original system. As the counterexample generation in Algorithm 1 only starts if the probability bound is exceeded, the probability mass of this system will also exceed the bound. Therefore, the algorithm always terminates.

**Example 4** *Assume the symbolic representation of the DTMC M of Figure 1(a) as explained in Example 2. We use the same set of variables $Var = \{\sigma_1, \sigma_2, \sigma_3\}$ while we add another index for the depth of the path at which each variable is used to encode a state. For example, the formula $\sigma_{2,1}^{0} \wedge \sigma_{2,2}^{1} \wedge \sigma_{2,3}^{0}$ encodes state $s_2$ at depth 2 of a path. As the shortest path that leads from the initial state $s_I$ to the target state $s_4$ has length 2, there will be no satisfying assignments for $BMC(0)$ and $BMC(1)$. For $k = 2$, the formula*

$$\underbrace{\sigma_{0,1}^{0} \wedge \sigma_{0,2}^{0} \wedge \sigma_{0,3}^{0}}_{s_0} \quad \wedge \quad \underbrace{\sigma_{1,1}^{0} \wedge \sigma_{1,2}^{0} \wedge \sigma_{1,3}^{1}}_{s_1} \quad \wedge \quad \underbrace{\sigma_{2,1}^{0} \wedge \sigma_{2,2}^{1} \wedge \sigma_{2,3}^{1}}_{s_3}$$

*encodes the first path $s_0 s_1 s_3$ of the global search in Example 3. The predicates $\check{P}$, $\check{I}$, and $\check{T}$ are all satisfied. Adding the negation of this formula to $BMC(2)$ prevents the SAT solver from finding this path again.*

## 4.2   Adapting Bounded Model Checking for Fragment Search

The previously described approach of using a SAT solver to find paths leading from the initial state of the DTMC to the target states is now extended according to the *fragment search* approach

as described in Section 3.2. We therefore aim at finding *path fragments* that extend the already found system iteratively.

The intuition is as follows: In search iteration 0, the CNF formula given to the SAT solver is satisfied if and only if the assignment corresponds to a path of arbitrary but bounded length (by some predefined $n \in \mathbb{N}$ which will be increased later if necessary) through the input DTMC leading from the initial state $s_I$ to a target state $t \in T$. This path induces the initial subsystem. Subsequently, this system is extended by paths whose first and last states are included in the current subsystem, while all states in between are fresh states.

For this we need to consider already found states for all possible depths $d$ of a path, $0 \leq d \leq n$. For a state $s$ let $v_s^d : Var_d \rightarrow \{0, 1\}$ be the unique assignment of $Var_d$ corresponding to state $s$.

We introduce a flag $f_s^d$ for each state $s$ and each depth $d$. This flag is assigned 1 if and only if the assignment of the state variables at depth $d$ corresponds to the state $s$. Thereby, we can "switch" the occurrence of a state $s$ at level $d$ by setting its flag $f_s^d$ to 0 or 1.

$$f_s^d \leftrightarrow (\sigma_{d,1}^{v_s^d(\sigma_{d,1})} \wedge \cdots \wedge \sigma_{d,m}^{v_s^d(\sigma_{d,m})}) \, . \tag{5}$$

The next variable $K_j^d$ describes the whole set of states which have been found so far, namely in the iterations $0, \ldots, j$ of the search process (again in terms of the variables $Var_d$ for depth $d$). Note, that these are exactly the states of the current subsystem *SubSys* after iteration $j$. We set $K_{-1}^d := f_{s_I}^d \vee \bigwedge_{t \in T} f_t^d$ to allow for paths initially leading from the initial state to all target states. Note that we assume all target states to have no outgoing transitions. Assume that in iteration $j$ of the search process the path $\pi_j = s_0 s_1 \ldots s_n$ is found. We then define

$$K_j^d \leftrightarrow \left( K_{j-1}^d \vee \bigvee_{i=0}^{n} f_{s_i}^d \right) . \tag{6}$$

All flags for the states $s_0 \ldots s_n$ can satisfy the right-hand side of this formula as well as the ones hidden in $K_{j-1}^d$. $K_j^d$ is thereby true iff the assignment corresponds to at least one of the states that were encountered so far.

In the first search iteration $j = 0$ we need a formula which is true iff the variable assignment corresponds to a path of arbitrary length—again bounded by $n$—leading from the initial state to a target state of the DTMC.

$$\check{I}(Var_0) \; \wedge \; \bigvee_{i=0}^{n} \check{T}(Var_i) \wedge \tag{7a}$$

$$\bigwedge_{i=0}^{n-1} \left[ (\neg \check{T}(Var_i) \rightarrow \check{P}(Var_i, Var_{i+1})) \; \wedge \; (\check{T}(Var_i) \rightarrow (Var_i = Var_{i+1})) \right] . \tag{7b}$$

Assume that $\nu$ is an assignment corresponding to the path $\pi = s_0 s_1 \ldots s_n$. Formula (7a) states that the first state $s_0$ is the initial state and that one of the states $s_0, \ldots s_n$ is a target state. Formula (7b) ensures, that if a state $s_i$ is not a target state, a valid transition will be taken to the next state. On the other hand, if $s_i$ is a target state, all following state variables will be assigned $s_i$, which creates an implicit self-loop on this state. This is useful to detect when a target state is

reached, since otherwise the solver would be free to assign arbitrary values to the states following a target state. The path returned to Algorithm 1 ends with the first target state $s_n$.

For the following iterations $j > 1$ we require that each solution corresponds to a path fragment that starts and ends in the current subsystem and contains at least one new state in between. For this we need the previously defined variables $K_d^j$:

$$K_{j-1}^0 \;\wedge\; \check{P}(Var_0, Var_1) \;\wedge\; \neg K_{j-1}^1 \;\wedge\; \bigvee_{d=2}^{n} K_{j-1}^d \tag{8a}$$

$$\wedge \bigwedge_{d=1}^{n-1} \left[ \left( \neg K_{j-1}^d \rightarrow \check{P}(Var_d, Var_{d+1}) \right) \wedge \left( K_{j-1}^d \rightarrow Var_d = Var_{d+1} \right) \right] . \tag{8b}$$

Formula (8a) ensures that the first state $s_0$ of a solution path $\pi_j = s_0 \ldots s_n$ is contained in the set $K_{j-1}^0$ of previously found states, that a transition is taken from this state to a not yet found state $s_1$ and that one of the following states $s_d$, $d \geq 2$, is again contained in $K_{j-1}^d$. Formula (8b) enforces valid transitions from all not yet found states $s_i$ to $s_{i+1}$. If $s_i$ was already included in previous paths, then all following states are assigned to $s_i$, thereby again creating an implicit self-loop on this state.

Termination is guaranteed, as the length of the paths is bounded by $n$. If no further satisfying assignments are found, this number has to be increased. However, the diameter, i. e., the longest cycle-free path of the underlying graph, is an upper bound on the length of loop-free paths from $s_I$ to target states. Therefore, $n$ needs to be increased only finitely many times, such that a critical subsystem is always determined in finite time.

**Example 5** *Consider again the assignment $\sigma_{0,1} \mapsto 0, \sigma_{0,2} \mapsto 0, \sigma_{0,3} \mapsto 0, \sigma_{1,1} \mapsto 0, \sigma_{1,2} \mapsto 0, \sigma_{1,3} \mapsto 1, \sigma_{2,1} \mapsto 0, \sigma_{2,2} \mapsto 1, \sigma_{2,3} \mapsto 1$ which encodes the first path $s_0 s_1 s_3$ for the fragment search as in Example 3. Having this in iteration 0, (7a) is satisfied, as the assignment of the variables in $Var_0 = \{\sigma_{0,1}, \sigma_{0,2}, \sigma_{0,3}\}$ encodes the initial state. The assignment of the variables in $Var_2 = \{\sigma_{2,1}, \sigma_{2,2}, \sigma_{2,3}\}$ the corresponds to the target state. (7b) is also satisfied, as for the states encoded by the variables $Var_0$ and $Var_1$, which are not target states, transitions are available leading to the state at the next depth. As the variables from $Var_2$ are assigned to a target state, all following variable sets $Var_m$ with $2 \leq m \leq n$ will be assigned equally, thereby again encoding the target state. This causes an implicit self-loop on the target state. According to (6), we build:*

$$K_0^0 \leftrightarrow (f_{s_0}^0 \vee f_{s_1}^0 \vee f_{s_3}^0), \quad K_0^1 \leftrightarrow (f_{s_0}^1 \vee f_{s_1}^1 \vee f_{s_3}^1), \quad K_0^2 \leftrightarrow (f_{s_0}^2 \vee f_{s_1}^2 \vee f_{s_3}^2) .$$

*Intuitively, $K_0^d$ is true for $0 \leq d \leq 2$ iff the variables at depth $d$ are assigned to any of $s_0$, $s_1$ or $s_3$.*

*For iteration 1, consider the assignment $\sigma_{0,1} \mapsto 0, \sigma_{0,2} \mapsto 0, \sigma_{0,3} \mapsto 1, \sigma_{1,1} \mapsto 0, \sigma_{1,2} \mapsto 1, \sigma_{1,3} \mapsto 0, \sigma_{2,1} \mapsto 0, \sigma_{2,2} \mapsto 0, \sigma_{2,3} \mapsto 1$. This encodes the second path $s_1 s_2 s_1$ of the fragment search (Example 3). First, (8a) is true: The variables from $Var_0$ are assigned such that $K_0^0$ is true as $f_{s_1}^0$ is true for $\sigma_{0,1} \mapsto 0, \sigma_{0,2} \mapsto 0, \sigma_{0,3} \mapsto 1$; a valid transition leads from $s_1$ to $s_2$; $s_2$ satisfies $\neg K_0^1$, and at $d = 2$ again a state satisfying $K_0^2$ is assigned, namely again $s_1$. (8b) is also satisfied, as for state $s_2$—not satisfying $K_0^1$—a valid transition is taken. Once $K_0^d$ for $0 < d \leq n$ is satisfied, all states at the following depths are assigned the same, again creating an implicit self-loop.*

## 4.3   A SAT Heuristics for Finding More Probable Paths

A drawback of the SAT-based search strategies is that paths are found without considering their probability beforehand. If paths or transitions with higher probabilities are preferred, the process can be accelerated.

SAT solvers have efficient variable selection strategies, i. e., strategies to decide which variable should be assigned next during the solving process. We therefore modify only the choice of the *value* the solver assigns to the selected variable, in order to prefer paths with higher probabilities.

The decision how to assign a variable is based on the transition probabilities. If a variable $\sigma_{i+1,j}$ is to be assigned at depth $0 < i + 1 \leq n$, its value partly determines $s_{i+1}$, being the target of a transition with source $s_i$. We choose the value for $\sigma_{i+1,j}$ which corresponds to the state $s_{i+1}$ to which the transition with the highest probability can be taken (under the current assignment). This can be applied for several consecutive transitions in the future up to the complete path. However, as this computation is very expensive, we restrict the number of time steps we look ahead. For our test cases, assigning variables for 3 possible consecutive transitions in one step led to the best results.

**Example 6** *Consider the DTMC M from Figure 1(a). Assume the binary encoding as described in Example 2. In the table below, a partial assignment $v_{part}$ of the variables for a state $s_i$ and its successor $s_{i+1}$ is shown; "?" indicates, that this variable is not yet assigned, next, that this variable will be assigned next.*

|               | $s_i$          |              |              | $s_{i+1}$        |                |                |
| ------------- | -------------- | ------------ | ------------ | ---------------- | -------------- | -------------- |
|               | $\sigma_{i,1}$ | $\sigma_{i,2}$ | $\sigma_{i,3}$ | $\sigma_{i+1,1}$ | $\sigma_{i+1,2}$ | $\sigma_{i+1,3}$ |
| $v_{part}$    | 1              | 1            | ?            | *next*           | ?              | ?              |

The current assignment determines state $s_i$ to be either $s_4$ or $s_5$. Assigning 1 to the next variable $\sigma_{i+1,1}$, which is the first variable for the successor state $s_{i+1}$, would only lead to the non-target absorbing state $s_6$. As the most probable transition outgoing from $s_4$ or $s_5$ would be the one leading to state $s_1$ with probability 0.7, we guide the SAT solver to assign 0 here.

# 5   BDD-based Symbolic Path Search

In this section we present new *BDD-based graph algorithms* to implement the path search procedure `FindNextPath(...)` as invoked by Algorithm 1. We first explain, how one can find the most probable path through a symbolically represented DTMC using a set-theoretic variant of Dijkstra's algorithm, called Flooding Dijkstra [11]. This method is extended to allow the computation of the *k* most probable paths of a DTMC. This procedure can be directly embedded into the symbolic framework from Section 3, resulting in a *symbolic global search*. However, the direct application leads to an exponential blow-up of the search graph. Therefore we introduce an improved variant which—amongst other improvements—avoids this growth, called *adaptive global search*. Afterward we present a new search method which symbolically searches for the most probable path fragments that extend the current subsystem. We call this approach the *symbolic fragment search*.

---

**Algorithm 2** The Flooding Dijkstra algorithm for symbolic DTMCs

---

**FloodingDijkstra**(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$)
**begin**

    BDD $UD := \hat{I}$;                                                                   (1)

    MTBDD $PR_1 := \hat{I}$, $PR_2 := 0$, $SP := 0$, $SPG := 0$;                       (2)

    **while** $UD \neq 0$ **do**                                                                   (3)

        $PR_2 := \mathtt{CalcProbs}(UD, PR_1, \hat{P})$;                             (4)

        $UD := \mathtt{GetStates}(PR_2, PR_1)$;                                  (5)

        $PR_1 := \mathtt{UpdatePR}(UD, PR_1, PR_2)$;                            (6)

        $SPG := \mathtt{UpdateSPG}(UD, \hat{P}, SPG)$;                        (7)

    **end while**                                                                        (8)

    $SP := \mathtt{GetPath}(SPG, \hat{I}, \hat{T})$;                                         (9)

    **return** $SP, SPG$                                                     (10)

**end**

---

## 5.1   Flooding Dijkstra Algorithm

The *Flooding Dijkstra* algorithm was introduced in [11]. As it is used in all of our BDD-based symbolic algorithms, we give a short explanation. The algorithm computes a *shortest* path, which is in our context a *most probable* path, from the initial state of a DTMC to a target state. This is done by a forward fixed point computation, iteratively improving for all states $s$ of the DTMC an under-approximation of the largest path probability from the initial state $s_I$ to $s$. Initially, the under-approximation is 1 for the initial state and 0 for all other states. An *update set*, which initially consists of the initial state, stores those states whose approximation was improved and needs to be propagated to their successors. The difference to the standard Dijkstra algorithm [26] for computing shortest paths in a directed graph is that Flooding Dijkstra updates in each iteration the approximations of the successors of *all* states from the update set, instead of restricting the propagation to an *optimal* element with the minimal currently known cost (highest probability). That means, in contrast to the depth-first search of the standard Dijkstra algorithm, the Flooding Dijkstra algorithm operates in a breadth-first-style over sets of states. Therefore it can be efficiently implemented using MTBDD operations. For details on the differences between the Flooding and standard Dijkstra variant cf. [34, Section 5.2.1].

    The Flooding Dijkstra algorithm is sketched in Algorithm 2. The *parameters* $\hat{P}$, $\hat{I}$ and $\hat{T}$ specify the input DTMC with the target states. The BDD *UD* stores the *update set* of those states that gained higher probabilities in the last iteration, whereas the *probability approximations* before resp. after a propagation step are stored in the MTBDDs $PR_1$ resp. $PR_2$. A *directed acyclic graph* (*DAG*) *SPG* (short for *shortest path graph*) is maintained to contain all most probable paths with minimal length from the initial state to all other states (w. r. t. the current approximation). Please note, that *SPG* is not a tree, as there may be two or more paths of the same highest probability and length leading to the same state. After the fixed point has been reached, i. e., when the approximation becomes exact, the last step of the algorithm extracts a single *most probable path*

(represented by the set *SP* of contained states) from the initial state to a target state. The following methods are used:

`CalcProbs(BDD `*UD*`, MTBDD `$PR_1$`, MTBDD `$\hat{P}$`)` propagates the improved probability values of states in *UD* to their successors. It calculates for all states $s'$ with at least one predecessor $s \in UD$ the maximal currently known path probability to go from $s_I$ to a state in *UD* (as stored in $PR_1$) and from there in one step to $s'$ (according to $\hat{P}$). The name "flooding" indicates that hereby the maximum is formed over all states $s \in UD$ with $P(s, s') > 0$.

Using (MT)BDD operations this is done as follows: $PR_1$ stores the probabilities of the most probable paths detected so far. Initially, only the initial state has probability 1 and all other states 0. $PR_1 \cdot UD$ restricts the probabilities to the states in *UD*. $PR_1 \cdot UD \cdot \hat{P}$ yields an MTBDD defined over *Var* and *Var'*. For an assignment $v_{s,s'}$ this MTBDD gives the probability to go from $s_I$ to $s$ (according to $PR_1$) and then takes the direct transition from $s$ to $s'$. We quantify over the source states, i.e., the variables *Var*, taking the maximum over all possibilities. Since the resulting MTBDD is defined over *Var'*, we rename these variables to *Var*. This yields $PR_2$.

`GetStates(MTBDD `$PR_2$`, MTBDD `$PR_1$`)` determines those states whose probability approximations were improved during the last propagation step. The resulting BDD contains those states whose probability in $PR_2$ is higher than in $PR_1$. In detail, the operation APPLY($>$, $PR_2$, $PR_1$) is carried out.

`UpdatePR(BDD `*UD*`, MTBDD `$PR_1$`, MTBDD `$PR_2$`)` computes the maximum over $PR_1$ and $PR_2$, where *UD* is assumed to contain those states whose values in $PR_2$ are higher than in $PR_1$. This function is implemented using ITE($UD, PR_2, PR_1$).

`UpdateSPG(BDD `*UD*`, MTBDD `$\hat{P}$`, MTBDD `*SPG*`)` maintains the DAG according to the improved probabilities for the update set *UD*. Those transitions of *SPG* that lead to a state in *UD*, i.e., to a state whose probability was improved, are removed. The transitions that cause the higher probabilities in $PR_2$ are added.

`GetPath(MTBDD `*SPG*`, BDD `$\hat{I}$`, BDD `$\hat{T}$`)` extracts one most probable path from the DAG *SPG* by walking backward from $\hat{T}$ to one of its predecessors until $\hat{I}$ is reached. This is straightforward and will not be explained further.

When the while-loop terminates, the MTBDD *SPG* contains, for each state $s$, *all* most probable paths with a minimal number of transitions from $s_I$ to $s$.

For both path search methods that we describe in the following, it is often beneficial not to return only a single path, but all paths in *SPG* to a target state. Then we perform a backward breadth-first search in *SPG* starting from $\hat{T}$ in order to have only states from which the target state is reached inside *SPG*. Therefore, the return statement of the Algorithm returns both *SP* and *SPG*.

**Example 7** *If the Flooding Dijkstra algorithm is run on the DTMC from Figure 1(a), a DAG containing all paths of maximal probability from the initial state to all other states is computed.*

*This graph and the most probable path to the target state $s_3$ of probability $0.25$ and length $2$ are depicted below. The framed values above the nodes of the DAG show the computed probability values from $PR_1$.*
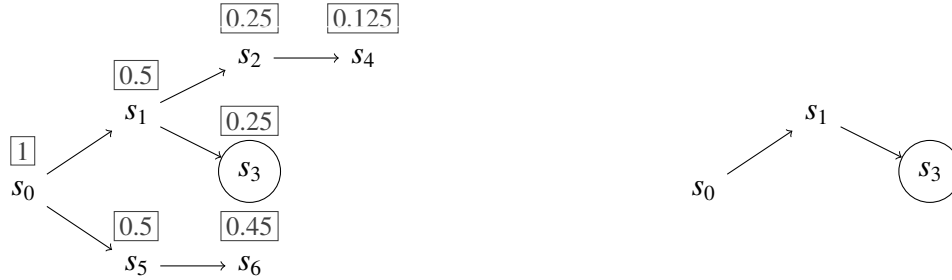


The path is determined by invoking a backward breadth-first search from the target state. Please note that in case there was another path of the same probability and length to the target state $s_3$ in the DTMC, there would be another path from $s_0$ to $s_3$ in the DAG. Please note also that standard Dijkstra would compute the same result, only the way of computation differs.

## 5.2   Adaptive Symbolic Global Search

In [11], a symbolic version of a *k*-shortest path search was presented. This corresponds to the *k* most probable paths, leading from the initial state to a target state ordered by their probabilities. Utilized for a counterexample search, the value of *k* is not fixed beforehand but the search terminates if enough probability mass is accumulated [12]. The main components are the calculation of a most probable path by the Flooding Dijkstra, see Section 5.1, and a transformation of the DTMC such that the most probable path in the altered system corresponds to the second-most probable path in the original system.

The adaption to our symbolic framework for the computation of critical subsystems is straightforward. Intuitively, for every new path the states on this path are available in BDD-representation and returned to Algorithm 1 as the BDD *NewStates*. As long as still new states are needed to form a critical subsystem, the *k*-shortest path search continues to deliver the next shortest path. This adaption is shown in Algorithm 3.

---

**Algorithm 3** The global search algorithm for symbolic DTMCs

---

**SymbolicGlobalSearch**(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$, BDD $SP$)
**begin**
    **if** $SP \neq 0$ **then** $(\hat{P}, \hat{I}, \hat{T}) := \texttt{Change}(\hat{P}, \hat{I}, \hat{T}, SP)$;         (1)
    $SP := \texttt{ShortestPath}(\hat{P}, \hat{I}, \hat{T})$;         (2)
    **return** $SP$         (3)
**end**

---

The parameters $\hat{P}$, $\hat{I}$ and $\hat{T}$ represent the input DTMC and the target states, whereas $SP$ stores
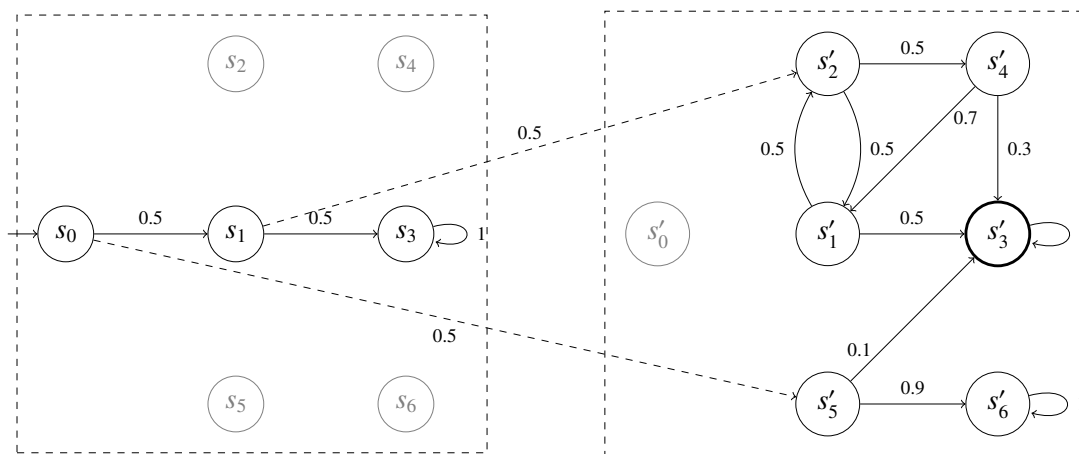
Figure 4: Altered system.

the states of the shortest path. The following methods are used (for details on the MTBDD operations we refer to the appendix of [11]):

ShortestPath(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$) is a symbolic implementation of the Flooding Dijkstra algorithm described in Section 5.1. It returns a BDD that represents the states occurring on a most probable path from the initial state represented by $\hat{I}$ to a target state from the set represented by $\hat{T}$.

Change(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$, MTBDD $SP$) changes the DTMC $(\hat{P}, \hat{I}, \hat{T})$ such that the most probable path in the new DTMC corresponds to the second-most probable path of the original DTMC.

The idea is to use two copies of the DTMC. The initial state is in the first copy while the target states are in the second copy. In the first copy, only edges of $SP$ remain unchanged; all other edges are redirected to the corresponding states in the second copy, in which all edges remain unchanged. Then all paths—with the exception of $SP$—lead from the initial state to a target state in this modified graph. As a consequence, every path to a target state needs to take at least one transition at a certain depth that does not occur in $SP$ at the same depth. Therefore, the most probable path in the modified graph corresponds to the second-most probable path in the original model. These modifications can be performed symbolically by adding an additional state variable that indicates which copy is used. The MTBDD $\hat{P}$ is therefore extended by two variables: One for the source and one for the target state. The adaptation of the transition relation is straightforward.

**Example 8** *To explain the procedure of altering the system using the above method, assume that the first global path $s_0 s_1 s_3$ of the DTMC shown in Figure 1(a) is found (the first global path in Example 3). The altered system is depicted in Figure 4. The two copies of the DTMC are marked by dashed rectangles. The initial state is still $s_0$ while the new target state is the copy $s_3'$ of $s_3$. Only*

23

*the transitions of the most probable path reside in the left copy. States, that are not reachable any more are drawn gray and we omit the transitions. The dashed transitions are the ones that do not belong to the most probable path and lead from the left to the right copy. The most probable path in this altered system is now the path $s_0 s_1 s_2' s_1' s_3'$. This corresponds to the second-most probable path of the original system as in Example 3. Note, that in order to find the next path, this whole altered system is again copied.*

As the whole modified system is copied again in every iteration (after each path), this procedure leads to an exponential blow-up in the system size. The MTBDD resulting from the iterative application of this altering grows also rapidly and renders this method inapplicable to systems which require a large number of paths, as our test cases will show. A further drawback is that many of the computed paths do not extend the subsystem and therefore do not lead to any progress. We have implemented this approach in order to compare it to other ones, and call it *symbolic global search*.

To present a symbolic global search approach that is usable for practical instances, we developed a new improved variant. In comparison to the straightforward approach this on the one hand avoids the exponential blow-up of the system size and on the other hand saves many search iterations by adding sets of paths. Furthermore, the search algorithm uses an adaptive strategy in order to find small counterexamples. We call this approach the *adaptive symbolic global search*, depicted in Algorithm 4.

---

**Algorithm 4** The adaptive global search algorithm for symbolic DTMCs

---

**AdaptiveSymbolicGlobalSearch**(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$, BDD *SubSys*)
**begin**
  BDD $SPG = 0$;   MTBDD $\hat{P}'$;   BDD $\hat{I}', \hat{T}'$;             (1)
  **if** $SubSys = 0$ **then** $(\hat{P}', \hat{I}', \hat{T}') := (\hat{P}, \hat{I}, \hat{T})$;          (2)
  **else** $(\hat{P}', \hat{I}', \hat{T}') := \texttt{Change}(\hat{P}, \ \hat{I}, \ \hat{T}, \ SubSys)$;      (3)
  $SPG := \texttt{ShortestPath}(\hat{P}', \hat{I}', \hat{T}')$;           (4)
  **return** $SPG$                      (5)
**end**

---

Parameters are $\hat{P}$, $\hat{I}$, $\hat{T}$ and *SubSys* as well as an BDD *SPG* which stores the current DAG. The methods differ from the ones for Algorithm 3 as follows.

$\texttt{ShortestPath}$(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$) returns the BDD representation of the DAG *SPG* containing all most probable paths of minimal length from a state of $\hat{I}$ to a state of $\hat{T}$. A symbolic implementation of the Flooding Dijkstra algorithm as described in Section 5.1 is used to obtain the DAG; a backward reachability analysis starting from $\hat{T}$ yields *SPG*.

$\texttt{Change}$(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$, MTBDD *SubSys*) In the improved version, the idea is to not only exclude the most probable path from the system but *all* transitions from the current subsystem. This is done by applying the transformation to the original DTMC and the

current subsystem *SubSys* instead of only the most probable path *SP*. In every step, we redirect the transitions such that at least one transition not present in *SubSys* has to be found.

This avoids doubling the search graph after each path and therefore the exponential blow-up, as in every step again the original system is used; the system size only increases linearly. Furthermore, we always obtain a path having a transition that is not yet contained in the subsystem at each time. Since the subsystem contains all transitions of the original DTMC connecting two states in the subsystem, each new transition also contributes a new state. Therefore the subsystem is extended in each iteration. Note that $\hat{P}$ is always unmodified.

Returned to Algorithm 1 is *SPG* which represents not only a single shortest path but a set of shortest paths. To further speed up the calculation, we add all states of this DAG to the subsystem at once. As we will see in our experiments, the search process is accelerated by orders of magnitude.

Adding many paths to the subsystem at once involves the risk that the computed counterexample has more states than needed and is of a probability that is not close to the probability bound. We overcome this problem by using an *adaptive search strategy*: In case the current subsystem is *critical*, i. e., its probability exceeds the probability bound, we measure the difference of these probabilities. If the difference is higher than a predefined constant $\delta > 0$, we perform backtracking to the state of the search procedure before the last spanning tree was added. We now add only a single path at a time and terminate as soon as the probability bound is again exceeded.

## 5.3   Symbolic Fragment Search

In contrast to the previous approach, where we search for whole paths through the system, we now aim at finding most probable *path fragments* as described in Section 3.2. This approach was successfully implemented for explicit graph representations [16] and is now adapted to symbolic representations, depicted in Algorithm 5.

---

**Algorithm 5** The adaptive fragment search for symbolic DTMCs

---

**AdaptiveSymbolicFragmentSearch**(MTBDD $\hat{P}$, BDD $\hat{I}$, BDD $\hat{T}$, MTBDD *SubSys*)
**begin**
    BDD *SPG*, *SubSysStates*;                                                                                      (1)
    **if** *SubSys* = 0 **then**                                                                                         (2)
        *SPG* := ShortestPath($\hat{P}$, $\hat{I}$, $\hat{T}$);                                  (3)
    **else**                                                                                                             (4)
        *SubSysStates* := ToStateBDD(*SubSys*);                                                (5)
        *SPG* := ShortestPath($\hat{P} \setminus SubSys$, *SubSysStates*, *SubSysStates*);        (6)
    **end if**                                                                                                          (7)
    **return** *SPG*                                                                                                   (8)
**end**

---

We need again a BDD *SPG* to store the DAG resulting from the Flooding Dijkstra algorithm. The subsystem is now represented as an MTBDD *SubSys*, its states as a BDD *SubSysStates*. The following methods are used:

`ShortestPath(MTBDD` $\hat{P}$`, BDD` $\hat{I}$`, BDD` $\hat{T}$`)` returns again the DAG *SPG* describing all paths of the highest probability leading to a target state.

`ToStateBDD(MTBDD` *SubSys*`)` computes for the transition MTBDD *SubSys* a BDD describing all states that occur as source state or target state for one of the transitions of *SubSys*. When *SubSys* is defined over the variables $Var = \{x_1, \ldots, x_n\}$ and $Var' = \{x'_1, \ldots, x'_n\}$, this is done by first building the set $OUT := \exists x'_1, \ldots, x'_n. SubSys_{\text{bool}}$ of all states with an outgoing transition. Afterward, the set $IN' := \exists x_1, \ldots, x_n. SubSys_{\text{bool}}$ of states with incoming transitions is built. These resulting BDDs have to be defined over the same variable set, therefore we perform variable renaming for the set of states with incoming transitions: $IN := IN'[x'_1 \to x_1] \ldots [x'_n \to x_n]$. Building the union $IN \cup OUT$ yields the needed BDD.

The symbolic fragment search checks whether the parameter *SubSys* is empty, i. e., whether this is the first search iteration. If this is the case, the base paths leading from the initial state to a target state are computed by invoking the most probable path search. The resulting paths, stored in the BDD *SPG* are returned to Algorithm 1. If *SubSys* is not empty, then a part of the subsystem has already been determined. In this case we compute the state BDD *SubSysStates* by invoking `ToStateBDD(`*SubSys*`)`. The most probable path algorithm is called to find the most probable paths from a state in *SubSysStates* to a state in *SubSysStates* inside the DTMC induced by $\hat{P}$ without using direct transitions from *SubSysStates* to *SubSysStates*. Note again that the resulting DAG might describe a large number of such paths.

In contrast to the symbolic global search described in Section 5.2, the MTBDD for the transition relation needs no significant modification. We only need to exclude the current subsystem from the further search in every iteration, which didn't lead to any remarkable overheads in our experiments. We also use the adaptive search algorithm in order to gain small critical subsystems and call this the *adaptive symbolic fragment search*.

**Example 9** *To illustrate the advantages of the adaptive fragment search, consider the following toy example DTMC with a single target state $s_3$.*
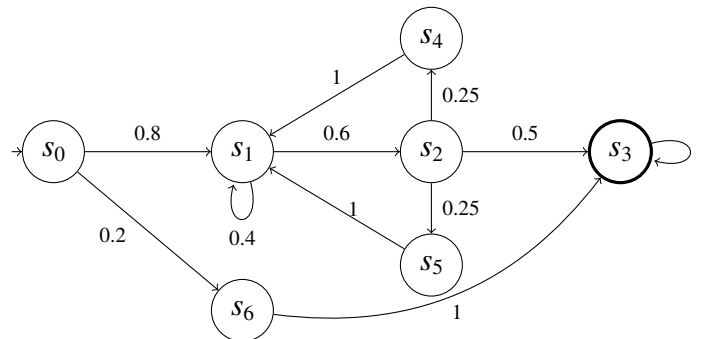
*Using the adaptive global search, first the path $\pi_1 = s_0 s_1 s_2 s_3$ of probability $0.24$ is found. The self-loop on $s_1$ is a transition starting and ending at states of the above path and will thus be automatically contained in the DAG SPG. The next path is $\pi_2 = s_0 s_6 s_3$ having probability $0.2$. Each of the next steps will extend $\pi_1$ by traversing the loops $\pi_3 = s_1 s_2 s_4 s_1$ and $\pi_3 = s_1 s_2 s_5 s_1$.*

*In contrast, the fragment search will first find the path $\pi_1$ and then the path fragments $\pi_3$ and $\pi_4$, both in one step. If the probability bound was not higher than $\lambda = 0.8$, this suffices to form a critical subsystem. As in most of the available benchmarks such symmetric loop-behavior is very common, this example is illustrative.*

# 6   Related Work

This paper builds (amongst others) on the concepts used in [16, 11]. Jansen et al. [16] introduced the global and fragment search techniques for the explicit generation of counterexamples. In this paper, we adapted these techniques and developed both a SAT-based (cf. Section 4) and a (MT)BDD-based (cf. Section 5) approach thereof. The experiments show a significant increase in scalability over [16] by several orders of magnitude.

Günther et al. [11] presented a symbolic $k$-shortest path algorithm and exploited this in the context of counterexample generation for DTMCs represented by MTBDDs. We adopted this for the global search approach, but as opposed to [11] we do not obtain an MTBDD representing a set of $k$ shortest paths, but rather a representation as a critical subsystem of the DTMC (cf. Section 5.2). In addition, we improved the running time of [11] by reducing the number of variable shiftings for the transition MTBDD in our implementation.

The idea to use a $k$-shortest path algorithm in counterexample generation stems from Han et al. [12]. They show that the $k$ most probable paths in a DTMC (forming a counterexample) correspond to the $k$ shortest paths in a related weighted digraph. Using a $k$ shortest path algorithm by Jiménez and Marzal [39], the number of paths is not determined beforehand but on-the-fly by an external condition. In this case that means that the search terminates once a counterexample has been found. This avoids fixing some (arbitrary) $k$ in advance, and allows for finding the smallest $k$ yielding a counterexample.

Aljazzar and Leue [15] exploit a best-first search for various search algorithms. The main advantage is that this can be pursued in an on-the-fly manner, avoiding an a priori generation of the state space. Using the simulation engine of PRISM [7], a successor relation on states delivers for one state exactly the explicit representation of its successors. Starting from an initial state of the system, the system is thereby successively extended along most probable local paths. Additionally, a heuristic function enables the user to use specific knowledge about benchmarks to prefer or penalize certain states of the system. As there might be cases where it is be beneficial to represent a counterexample by a subsystem instead of a set of paths, an *extended best-first search (XBF)* is used, where for each node not only the predecessor inducing the optimal path is stored but also the other connections. Thereby, a whole system instead of a single path is obtained. The approaches are implemented in the tool DIPRO [18]. Furthermore, the authors introduce an on-the-fly algorithm called $K^\star$ [40] for finding the $k$ shortest paths, which is based on the similar concepts.

Next we shortly discuss, how the approach implemented in DiPro could be adapted to the symbolic setting. First of all, we always use subsystems as representations of counterexamples and not sets of paths. As was already argued and evaluated in [15, 16], providing counterexamples in form of subsystems is clearly superior to enumerating the paths in terms of running time and size of the representation as well. Furthermore, having a subsystem, a path-based counterexample is always induced as the set of all finite paths going from initial to target states inside the subsystem forms a counterexample.

Adapting the best-first approaches as in [15, 40] to symbolic data structures is not directly possible. Both our symbolic approach and the best-first approach tackle the state explosion problem, the first one by using symbolic data structures, the second one by building the state space successively. If the state space is already fully generated in form of an MTBDD representation, it seems more reasonable to use search algorithms that take the whole system into account as in our approach and not only locally optimal transition choices. Using heuristics for preferring certain states as in these approaches could be adapted as follows: If a state is to be preferred or penalized, the probability of its adjacent transitions can be scaled by a factor $\delta \in [0, 1] \subset \mathbb{Q}$, while the probability of all other transitions is scaled by $1 - \delta$. Thereby, a path leading through this state becomes more probable or less probable, respectively, which would be taken into account by our algorithms.

In [17], a method to compute a *minimal subsystem* inducing a counterexample was presented. Unlike the path-based methods, solving techniques like SAT modulo theories (SMT) and mixed integer linear programming (MILP) are used to encode and compute these minimal subsystems. The implementation is available as part of the tool LTLSubsys. An adaption to the symbolic setting poses the difficulty, that in order to minimize the number of states a variable for each state needs to be introduced. With increasing explicit size of a system, these approaches become infeasible.

Finally, in [41] the strongly connected components (SCCs) of the graph induced by a DTMC are abstracted to single transitions leading through these SCCs. This is achieved by using Tarjan's algorithm [42] for finding SCCs and computing the overall probabilities of walking through SCCs and exiting them at certain states. In [16], this concept was extended to arbitrarily nested connected components combined with path search on abstract systems. Adapting these approaches would be possible, as efficient symbolic algorithms for finding strongly connected components exist [43]. However, while this is straightforward for [41], defining a nested abstraction scheme as in [16] is not obviously feasible for MTBDD representations and needs to be investigated further.

# 7   Case Studies

In this section we present an experimental evaluation of the approaches introduced in this paper. We compare them with existing methods, which all rely on an explicit representation of the state space. Measuring how useful counterexamples in practice are is difficult. Obviously the size of a counterexample plays a crucial role: If the counterexample is much larger than necessary, it will be of little help for debugging. Therefore we focus on comparing the sizes of the counterexamples computed by different tools. Additionally we measure the time for their computation and the

memory consumption of the tools. We will see that our novel symbolic methods can handle much larger state spaces than all tools relying on explicit representations.

## 7.1  Implementation

We have implemented a prototypical tool in C++ for the approaches presented in this paper. It uses the model checker Prism 4.0.3, the BDD package Cudd 2.5.0 [44], and the SAT solver MiniSAT 2.2.0 [45].

We are going to compare our tool with DiPro [18], COMICS [19], and LTLSubsys [17], which are—to the best of our knowledge—the only tools that support counterexample generation for DTMCs in form of critical subsystems. In order to obtain comparable results, we use the same Prism models for all tools. For COMICS and LTLSubsys, which cannot read Prism models, we use Prism to convert the state spaces into Mrmc's input format, which is essentially a list of reachable states and transitions. DiPro, on the other hand, internally calls Prism to generate the state space. For our tool chain, we modified Prism such that it is able to write its (MT)BDD representation of the reachable states, the initial and target states, and the transition probability matrix into a file that can be read by our tool.

Regarding the variable order of the BDDs, we use the order generated by Prism: state and next state variables are interleaved, the Boolean variables which encode a certain integer variable of the Prism model are kept together. For more information on Prism's model representation, we refer to [24]. Furthermore we did not use dynamic reordering [33] to reduce the size of the BDDs during computations since the improvement in terms of running times due to slightly smaller BDDs did not compensate the additional overhead for sifting.

For the SAT-based approaches, we also use the MTBDD of the transition probability matrix. As described in Section 4.1, to obtain a SAT formula, we map all leaves with a positive value to 1. This results in a BDD describing the possible transitions of the system. By applying Tseitin's transformation [38] we obtain a propositional formula in conjunctive normal form whose length is linear in the number of nodes of the BDD.

For solving the formulae, we use MiniSAT. We extended it with a callback function which is called each time a satisfying assignment has been found. This way the solver can continue its search after reporting a solution without performing a restart from the beginning.

## 7.2  Models

We present results for the Probabilistic Contract Signing protocol [46], the Crowds protocol [47], and a synchronous leader election protocol [48]. We used the Prism models [49] of these protocols, which are publicly available at the Prism web page [50].

*Probabilistic Contract Signing* is a network protocol targeting the *fair* exchange of critical information between two parties *A* and *B*. In particular, whenever *B* has obtained *A*'s commitment to a *contract*, *B* should not be able to prevent *A* from getting *B*'s commitment. The PCTL property $\mathcal{P}_{\leq 0.5}(\lozenge\,[knowA \wedge \neg knowB])$ we are investigating describes an unfair situation where *A* knows *B*'s secrets while *B* doesn't know *A*'s secrets. The target states in our model are those which

29

| Model | N-K | States | Transitions | Probability | $\lambda$ |
|---|---|---:|---:|---:|---:|
| crowds | 5-6 | 18 817 | 30 158 | 0.426153 | 0.25 |
| | 8-15 | 50 445 495 | 88 120 216 | 0.850540 | 0.20 |
| | 10-20 | 4 163 510 716 | 10 172 513 716 | 0.931304 | 0.40 |
| | 20-30 | 10 173 177 100 089 080 | 38 403 575 234 221 120 | ?? | 0.20 |
| contract | 5-2 | 33 790 | 34 814 | 0.515625 | 0.50 |
| | 5-8 | 156 670 | 157 694 | 0.515625 | 0.50 |
| | 7-2 | 737 278 | 753 662 | 0.503906 | 0.50 |
| | 7-4 | 1 654 783 | 1 671 166 | 0.503906 | 0.50 |
| leader | 4-8 | 12 302 | 16 397 | 1.000000 | 0.50 |
| | 8-4 | 458 847 | 524 382 | 1.000000 | 0.50 |

Table 1: Model statistics. PRISM was not able to compute the reachability probabilities for crowds20-30 within 2 hours.

carry the label *knowA*, but not the label *knowB*. The model is parametric in the number *N* of data pieces to exchange and in the size *K* of each data piece.

The *Crowds Protocol* aims at anonymous communication in networks, where a crowd of *n* users is divided into *good members* and *bad members*. A good member delivers a message to its destination with probability $1 - p_f$ and forwards it to another member, randomly chosen, with probability $p_f$. This guarantees that no bad member knows the original sender of the message. Each *session* describes the delivery of a message to a sender. If a user is identified twice by a bad member as the sender of a message, we assume that anonymity is no longer guaranteed. This is called *positively identified* (*Pos*). The PCTL property we consider is $\mathcal{P}_{\leq\lambda}(\Diamond Pos)$. The models are parametric in the size *N* of the crowd and in the number *K* of sessions.

The synchronous *Leader Election Protocol* is run on a ring-structured network of *N* identical nodes. The goal is to randomly elect a leader node, which can later serve, e. g., as a coordinator. For the election, each node randomly draws a number in the range from 1 to *K*. If at least one node draws a unique number, the one with the highest unique number becomes the leader. Otherwise a new round starts. We investigate the property whether the probability to finally elect a leader exceeds the bound $\lambda$.

The three benchmark classes have different structures: while the Crowds protocol contains nested loops, the protocol for contract signing is completely acyclic. The leader benchmark contains non-nested loops which correspond to the rounds of the protocol. This will also be reflected in the results.

Table 1 contains information about the different instances. The first column contains the benchmark class, the second the values of the parameters *N* and *K*. The next three columns list the number of states, transitions, and the actual probability (computed by PRISM [7]) to reach a state satisfying property $\varphi$ from the initial state. The column titled "$\lambda$" contains the upper bound on the allowed probability.

## 7.3   Experimental Setting

All experiments were performed on an Intel Xeon E5-2643 CPU (3.3 GHz) with 32 GB RAM running Ubuntu Linux 12.04. The timeout (TO) for counterexample generation was defined as

2 hours. We made 30 GB of memory available to the program, leaving 2 GB for the operating system.

We do a comparison of the methods described in this paper with the following three tools:

The developers of DɪPʀᴏ [18] provided us with the most recent version of DɪPʀᴏ, which contains a series of improvements and bug fixes compared to the published version. For comparison with our tool we used three different algorithms, supported by DɪPʀᴏ on DTMCs: eXtended Best First search (XBF), Eppstein's $k$-shortest paths algorithm [51], and the K* algorithm by Aljazzar and Leue [40] with X optimization (`-kxsol` switch). Model checking is performed every 50 iterations to check whether the computed subsystem is already critical, which is the default setting of DɪPʀᴏ.

It is possible to extend DɪPʀᴏ with user-defined heuristics, which can considerably speed-up the search. Such a heuristic, however, has to exploit the user's knowledge about the structure of the model under consideration. Since all other approaches work for arbitrary models without knowing their internals, we did not develop any heuristics for DɪPʀᴏ to make the comparison fair. As mentioned in Section 6, we could extend our approaches to use a heuristic like in DɪPʀᴏ.

The tool LTLSᴜʙsʏs can be used to compute minimal critical subsystems for DTMCs. It uses mixed integer linear programming (MILP) to obtain a critical subsystem with a minimum number of states [17]. It allows to add redundant constraints, which often speed up the solution process by strengthening the linear relaxation of the MILP. We use the default settings which add forward and backward cuts as well as SCC input cuts. The MILPs are solved using the commercial solver Cplex 12.4 by IBM. Although Cplex supports multi-threading, we ran LTLSᴜʙsʏs only with a single thread since all other tools work sequentially.

`COMICS` [19] is able to construct hierarchical, refineable counterexamples by abstracting strongly connected components as well as critical subsystems of the DTMC. It can apply both global search, i. e., the explicit-state counterpart of the method described in Section 5.2 using the $k$-shortest paths algorithm by Jiménez and Marzal [39], and an explicit-state variant of the fragment search method described in Section 5.3. We use both global and fragment search on the non-abstracted system for our experiments.

Using `COMICS` and our symbolic algorithms, model checking is invoked for every 10th found path. If the probability mass of the subsystem has reached around 95% of the needed mass, the probability is checked for every new path. Using the adaptive strategy as explained in Section 5.2, the number of subsequent iterations is heuristically determined with respect to the probability of the paths contained in the spanning tree.

## 7.4   Results

In Table 2 we have collected a number of results we achieved with our symbolic methods on the different instances of the described case studies with properties of the form $\mathcal{P}_{\leq \lambda}(\Diamond \varphi)$.

We tested the methods for symbolic counterexample generation described in this paper as well as the original bounded model checking approach [9], which computes a set of paths. We provide results for the following symbolic algorithms:
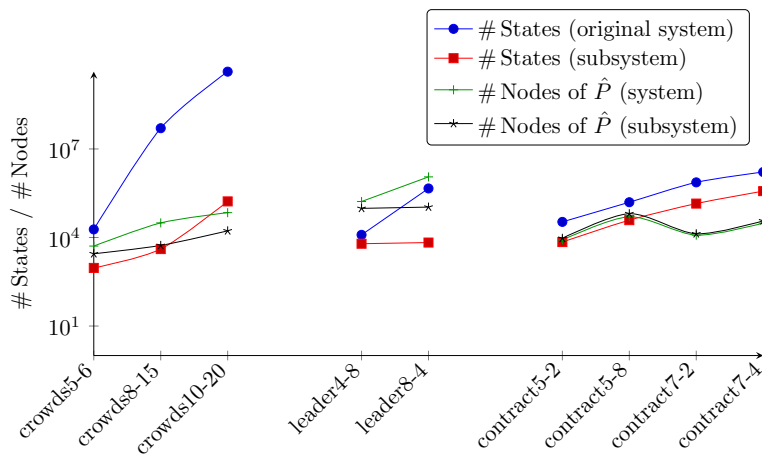
Figure 5: Number of states and MTBDD nodes of the original system and the computed subsystem.

¡

Table 2: Results using symbolic methods (TO > 2 h, MO > 30 GB).

- BDD global: The naive BDD-based symbolic global search approach without optimizations, described in Section 5.2.

- Adaptive BDD global: The BDD-based symbolic global search with optimizations also described in Section 5.2.

- Adapt. BDD fragment: The BDD-based symbolic fragment search approach from Section 5.3 with adaptive strategy and adding sets of paths at one time.

- SAT global: The global search approach using SAT solvers, see Section 4.1.

- SAT fragment: The fragment search approach using SAT solvers, Section 4.2.

- SAT fragment + H: The SAT-based fragment search approach together with the SAT heuristic preferring more probable paths, Section 4.3.

- BMC classic: The original bounded model checking approach for DTMCs as described in [9].

For the resulting subsystems we present their number of states (# states) and the number of performed path searches (# paths) or the number of iterations (# iter.) for the symbolic adaptive strategies. Additionally we report the reachability probability within this subsystem (prob.), the computation time (time) in seconds, and memory consumption (memory) in megabytes.

The classic BMC-approach of [9] does not compute a subsystem, but a set of acyclic paths which are annotated with loops. In order to make its result comparable with the subsystems computed by the other approaches, we give the total number of involved states, which occur on any of the computed paths or loops (# inv. states).

| Model | *N-K* | LTLSubsys | COMICS global | COMICS fragment | DiPro XBF | DiPro Eppstein | DiPro K* w. X opt. |
|---|---|---|---|---|---|---|---|
| | | # states lower bound prob. time memory | # states # paths prob. time memory | # states # paths prob. time memory | # states # vertices prob. time memory | # states # vertices prob. time memory | # states # vertices prob. time memory |
| crowds | 5-6 | 487 457.19 0.25 TO 2382 | **2038** **55369** **0.25** **0.30** **56** | **803** **487** **0.25** **7.54** **32** | **1114** **3051** 0.254 14.04 333 | TO | **2076** **8549** 0.292 193.88 4142 |
| crowds | 8-15 | MO | MO | MO | 3274 11390 0.211 33.64 1392 | TO | 3877 25413 0.202 237.79 3523 |
| crowds | 10-20 | MO | MO | MO | TO | TO | TO |
| contract | 5-2 | **6683** **—** **0.501** **0.21** **24** | **6832** **513** **0.501** **0.12** **31** | **6684** **513** **0.501** **3.98** **164** | **6683** **15911** **0.501** **1060.70** **5770** | **6770** **23863** **0.501** **34.58** **999** | **7015** **23863** **0.516** **99.49** **1056** |
| contract | 5-8 | 37463 — 0.501 1.12 106 | 37674 513 0.501 1.25 141 | 37464 513 0.501 53.89 904 | 37463 77342 0.501 5363.96 28087 | 37610 115123 0.501 163.27 4805 | 38755 115123 0.516 622.36 5117 |
| contract | 7-2 | 139302 — 0.501 8.29 411 | 140050 8193 0.501 62.14 642 | TO | TO | TO | TO |
| contract | 7-4 | 368705 — 0.501 301.12 1032 | TO | TO | TO | TO | TO |
| leader | 4-8 | 6150 — 0.50 61.94 123 | 6160 2049 0.50 0.70 16 | 6426 2137 0.50 22.29 565 | 6221 12302 0.505 23.75 1306 | 6160 12302 0.505 18.62 630 | 7342 12302 0.596 52.08 720 |
| leader | 8-4 | TO | TO | TO | TO | TO | TO |

Table 3: Results using explicit methods (TO > 2 h, MO > 30 GB).

With the exception of LTLSᴜʙꜱʏꜱ, all tools extend the subsystem until its probability measure exceeds the bound $\lambda$. LTLSᴜʙꜱʏꜱ, however, starts with the whole DTMC as critical subsystem and tries to reduce it, until a minimal critical subsystem (MCS) is obtained. At the same time it maintains a lower bound on the size of the MCS which is successively improved. Optimality is detected as soon as the lower bound and the size of the computed subsystem coincide. While LTLSᴜʙꜱʏꜱ computes the smallest possible critical subsystem, all other tools apply heuristics to efficiently compute small, but not necessarily minimal critical systems.

All results which were finished within the resource limits of 2 hours and 30 GB of main memory are printed in **boldface**. For unfinished cases we give, if possible, the results that were achieved so far. Note that the probability for these unfinished benchmarks lies under the probability threshold. TO means that the time limit was exceeded, MO stands for exceeding the memory limit.

We first demonstrate the effectiveness of the BDD-based representation of the original system, which is the starting point for both the SAT-based and the BDD-based methods, and of the computed subsystem. In Figure 5 we show for the considered benchmarks the number of states both of the original DTMC and of the subsystem computed by the adaptive global search approach. Additionally the figure contains the number of MTBDD nodes of the transition probability matrix $\hat{P}$ of the original DTMC and of the subsystem. Note that the vertical axis is logarithmically scaled.

We can observe that the number of MTBDD nodes is often smaller by orders of magnitude than the represented state space. In particular for the crowds instances the difference between the number of states and the number of MTBDD nodes spans several orders of magnitude. This is because practical benchmarks are not random, but exhibit regularities in their system structure, which is exploited by the BDD-based representation. Exceptions are possible, e. g., only for quite large instances of the leader election protocol the BDD-based representation pays off.

Comparing the size of the original system and the size of the computed subsystem, we can see that in most cases the subsystem is so small that it can easily be represented in an explicit way, which opens the possibility to post-process it using exact minimization, e. g., using LTLSᴜʙꜱʏꜱ, or further heuristic minimization using COMICS, DɪPʀᴏ, or a greedy approach.

The results in Table 2 show that the adaptive BDD-based global and fragment search significantly outperform all other symbolic approaches on our benchmarks sets. We can compute critical subsystems for benchmarks consisting of billions of states. The memory consumption stays below 250 MB for all instances in Table 2 with the exception of leader8-4, while BDD-based global search used up to 429 MB and the SAT-based approaches more than 4.7 GB.

For the leader instances, all paths corresponding to one successful election round have the same probability and length. Therefore the adaptive strategies find all these paths in a single iteration. For leader8-4 this yields a subsystem with a probability of 0.76 in about 2 seconds. Since this probability exceeds the bound by more than 10 %, this step is undone, and the search continues from scratch by computing a single path at a time. This explains the large number of iterations and the relatively high running times for the leader election protocol. Additionally the large BDD representation contributes to the high running times.

To evaluate the limits of adaptive BDD-based search strategies, we generated the instance crowds20-30 with more than $10^{16}$ states and $3.8 \cdot 10^{16}$ transitions. Adaptive BDD-based fragment

search computed, for a probability bound $\lambda = 0.2$, a subsystem with 76 007 states, probability 0.208446 within 2972.36 seconds using less than 873 MB of main memory. It needed 96 iterations for this. The adaptive global search returns a subsystem with 82 944 states and probability 0.207726 within 2497.89 seconds, using roughly the same amount of memory. It needed 43 iterations.

None of the explicit-state tools was able to handle this instance: COMICS and LTLSUBSYS failed because we were not even able to store the explicit state space on hard disk. DIPRO did not immediately fail due to the limited memory, but ran into a timeout with all three search methods. DIPRO was able to at least start the search on this large instance. As the state space is not constructed fully but on the fly using the extended best-first search, often relatively little memory is required in spite of the explicit representation of the state space.

The adaptive BDD-based fragment search is superior to the adaptive BDD-based global search: The former yields typically smaller subsystems than the latter. This is due to the fact that most probable local extensions of the current subsystem are added, while the global search finds paths that can traverse completely different parts of the state space. Nevertheless, depending on the problem at hand this might also be of advantage.

We also measured the difference between the original BDD-based global search and the improved variant which does not need to double the graph after each iteration. While the former is hardly able to handle any of the instances besides the smallest ones, the improved version succeeds also when we only insert a single path at a time instead of all most probable paths of minimal length. The reason for this behavior can be seen in Figure 6. The horizontal axis shows the number of paths computed by the two variants of BDD-based global search. The vertical axis of the graph on the left-hand side contains the computation time of each path in seconds, the graph on the right-hand side the number of nodes of the BDD representation. We can observe that, for the original global search, the computation time per path rapidly grows during the search process and at the same time the number of BDD nodes increases substantially. This effect is caused by the strategy used to exclude the already found paths from the search space: each time a path has been found, the state space is doubled and the transition relation adapted to exclude the found path. This makes the introduction of two additional BDD-variables necessary. Thus the size of the transition BDD grows, making the operations thereon more expensive. Our improved variant does not suffer from this effect. It only needs to maintain two copies of the DTMC. Therefore the size of the BDD for the graph on which the shortest path computation is performed and the time per path stay almost constant.

When comparing the SAT- and the BDD-based approaches one can recognize that the former performs much worse—in particular in comparison with the adaptive BDD-based search strategies. The SAT-based approaches ignore the actual transition probabilities, while the BDD-based approaches always compute the most probable paths. Therefore the BDD-based methods need in general fewer paths to reach a critical subsystem. Additionally, the Flooding Dijkstra algorithm computes not only one path at a time but all most probable paths of minimal length. Therefore the adaptive strategies need few iterations to reach the probability bound. Not only the computation time is higher for the SAT-based approaches, but also the memory consumption: for each found path an additional clause has to be added to the solver's clause database to exclude it from the search space. Moreover a large number of conflict clauses is computed by the solver during the
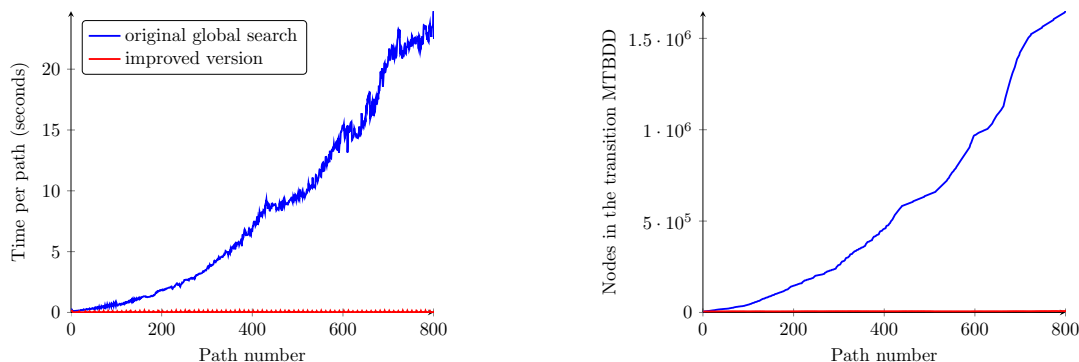
Figure 6: Evaluation of the symbolic global search on crowds5-6.

search process, which significantly contribute to the memory consumption.

Table 3 shows the experimental results for the following explicit algorithms and tools:

- LTLSᴜʙꜱʏꜱ [17] computes the smallest possible critical subsystem (in terms of states) using an approach based on mixed integer programming.

- COMICS [19] using both global and fragment search on explicitly represented state spaces.

- DɪPʀᴏ [18] using three different methods: extended best first search (XBF) [15], Eppstein's $k$ shortest path algorithm [51], and the K* algorithm [40], with $X$ optimizations enabled [52].[3]

We provide the following data: the number of states contained in the subsystem (# states), the number of paths (# paths) for COMICS, and the number of expanded vertices (# vertices) for DɪPʀᴏ. For LTLSᴜʙꜱʏꜱ we give—if the computed solution was not proven optimal—a lower bound on the size of the optimal subsystem. Additionally we give the reachability probability within the subsystem (prob.), the computation time (time) in seconds, and the memory consumption (memory) in megabytes.

   The approaches implemented in COMICS and DɪPʀᴏ are fast on small or medium-sized instances. Since they represent the state space by enumerating all transitions, their memory consumption is at least linear in the size of the DTMC. As DɪPʀᴏ does not construct the whole state space beforehand, it suffers far less from this problem than COMICS.

   The sizes of the subsystems computed by COMICS and DɪPʀᴏ are similar to those of the symbolic approaches. The adaptive BDD-based search strategies return slightly larger subsystems since they add several paths at once and perform model checking with a different frequency than DɪPʀᴏ. This can delay when the tool recognizes that the subsystem has become critical. The frequency of model checking has a great impact on the overall computation time since repeatedly computing the reachability probability for the subsystem makes up more than 50 % of the total running time.

---

[3]We omit the K* algorithm without $X$ optimizations because it performed worse on all of our instances.
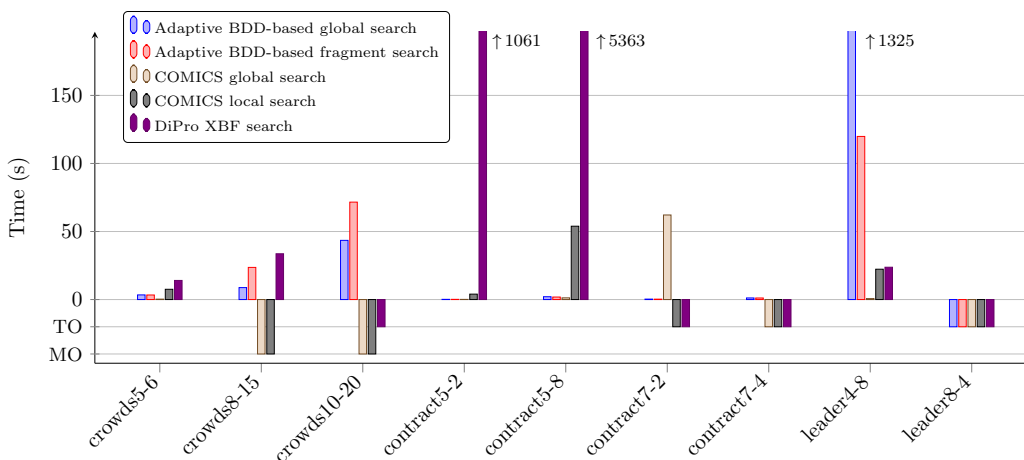
Figure 7: Comparing the running times of different tools.

The large differences in computation times of the most efficient methods are illustrated in Figure 7. We can clearly see that the two adaptive BDD-based approaches are much faster than the other methods, in particular for large instances.

LTLSubsys is restricted to small systems. In contrast to the other tools, it focuses on computing *optimal* counterexamples. Although the complexity of this problem is unknown, it seems to be hard. Therefore LTLSubsys is often not able to prove the optimality of its solution, but yields a relatively small subsystem when the time limit is exceeded. It is the only tool which can give information about the quality of the returned subsystem compared to the optimal one. LTLSubsys can be used to improve the heuristic solutions of the other tools by applying it to the returned subsystems. This will not yield a globally optimal counterexample, but a locally optimal one in the sense that it is the smallest critical subsystem which is contained in the heuristic subsystem.

# 8   Conclusion and Future Work

In this paper we presented a new framework for the symbolic generation of probabilistic counterexamples for discrete-time Markov chains. We suggested several methods; thereby the symbolic fragment search turned out to be the best alternative. Our experiments showed that using our framework the size of input systems feasible for counterexample generation is increased by orders of magnitude, compared to other approaches.

Binaries and source code of our `COMICS` tool and the benchmark sets used in this paper are available at the `COMICS`-web page `http://www-i2.informatik.rwth-aachen.de/i2/comics/`.

# Bibliography

[1] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: Proc. of CAV, Vol. 1855 of LNCS, Springer-Verlag, 2000, pp. 154–169.

[2] H. Hermanns, B. Wachter, L. Zhang, Probabilistic CEGAR, in: Proc. of CAV, Vol. 5123 of LNCS, Springer-Verlag, 2008, pp. 162–175.

[3] R. Chadha, M. Viswanathan, A counterexample-guided abstraction-refinement framework for Markov decision processes, ACM Transactions on Computational Logic 12 (1) (2010) 1–45.

[4] E. M. Clarke, The birth of model checking, in: 25 Years of Model Checking – History, Achievements, Perspectives, Vol. 5000 of LNCS, Springer-Verlag, 2008, pp. 1–26.

[5] H. Hansson, B. Jonsson, A logic for reasoning about time and reliability, Formal Aspects of Computing 6 (5) (1994) 512–535.

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

[7] M. Z. Kwiatkowska, G. Norman, D. Parker, Prism 4.0: Verification of probabilistic real-time systems, in: Proc. of CAV, Vol. 6806 of LNCS, Springer-Verlag, 2011, pp. 585–591.

[8] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, D. N. Jansen, The ins and outs of the probabilistic model checker MRMC, Performance Evaluation 68 (2) (2011) 90–104.

[9] R. Wimmer, B. Braitling, B. Becker, Counterexample generation for discrete-time Markov chains using bounded model checking, in: Proc. of VMCAI, Vol. 5403 of LNCS, Springer-Verlag, 2009, pp. 366–380.

[10] B. Braitling, R. Wimmer, B. Becker, N. Jansen, E. Ábrahám, Counterexample generation for Markov chains using SMT-based bounded model checking, in: Proc. of FMOODS/FORTE, Vol. 6722 of LNCS, Springer-Verlag, 2011, pp. 75–89.

[11] M. Günther, J. Schuster, M. Siegle, Symbolic calculation of *k*-shortest paths and related measures with the stochastic process algebra tool Caspa, in: Proc. of DYADEM-FTS, ACM Press, 2010, pp. 13–18.

[12] T. Han, J.-P. Katoen, B. Damman, Counterexample generation in probabilistic model checking, IEEE Transactions on Software Engineering 35 (2) (2009) 241–257.

[13] M. Kattenbelt, M. Huth, Verification and refutation of probabilistic specifications via games, in: Proc. of FSTTCS, Vol. 4 of LIPIcs, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2009, pp. 251–262.

[14] H. Fecher, M. Huth, N. Piterman, D. Wagner, PCTL model checking of Markov chains: Truth and falsity as winning strategies in games, Performance Evaluation 67 (9) (2010) 858–872.

[15] H. Aljazzar, S. Leue, Directed explicit state-space search in the generation of counterexamples for stochastic model checking, IEEE Transactions on Software Engineering 36 (1) (2010) 37–60.

[16] N. Jansen, E. Ábrahám, J. Katelaan, R. Wimmer, J.-P. Katoen, B. Becker, Hierarchical counterexamples for discrete-time Markov chains, in: Proc. of ATVA, Vol. 6996 of LNCS, Springer-Verlag, 2011, pp. 443–452.

[17] R. Wimmer, N. Jansen, E. Ábrahám, B. Becker, J.-P. Katoen, Minimal critical subsystems for discrete-time Markov models, in: Proc. of TACAS, LNCS, Springer-Verlag, 2012, pp. 299–314.

[18] H. Aljazzar, F. Leitner-Fischer, S. Leue, D. Simeonov, DiPro – A tool for probabilistic counterexample generation, in: Proc. of SPIN, Vol. 6823 of LNCS, Springer-Verlag, 2011, pp. 183–187.

[19] N. Jansen, E. Ábrahám, M. Volk, R. Wimmer, J.-P. Katoen, B. Becker, The COMICS tool – Computing minimal counterexamples for DTMCs, in: Proc. of ATVA, Vol. 7561 of LNCS, Springer-Verlag, 2012, pp. 349–353.

[20] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers 35 (8) (1986) 677–691.

[21] M. Fujita, P. C. McGeer, J. C.-Y. Yang, Multi-terminal binary decision diagrams: An efficient data structure for matrix representation, Formal Methods in System Design 10 (2/3) (1997) 149–169.

[22] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic model checking: $10^{20}$ states and beyond, Information and Computation 98 (2) (1992) 142–170.

[23] C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Z. Kwiatkowska, M. Ryan, Symbolic model checking for probabilistic processes, in: Proc. of ICALP, 1997, pp. 430–440.

[24] D. Parker, Implementation of symbolic model checking for probabilistic systems, Ph.D. thesis, University of Birmingham (2002).

[25] E. M. Clarke, A. Biere, R. Raimi, Y. Zhu, Bounded model checking using satisfiability solving, Formal Methods in System Design 19 (1) (2001) 7–34.

[26] E. W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.

[27] N. Jansen, E. Ábrahám, B. Zajzon, R. Wimmer, J. Schuster, J.-P. Katoen, B. Becker, Symbolic counterexample generation for discrete-time Markov chains, in: Proc. of FACS, Vol. 7684 of LNCS, Springer-Verlag, 2012, pp. 134–151.

[28] J. G. Kemeney, J. L. Snell, A. W. Knapp, Denumerable Markov Chains, Springer-Verlag, 1976.

[29] M. Z. Kwiatkowska, G. Norman, D. Parker, Probabilistic symbolic model checking with PRISM: A hybrid approach, in: J.-P. Katoen, P. Stevens (Eds.), Proc. of TACAS, Vol. 2280 of LNCS, Springer-Verlag, 2002, pp. 52–66.

[30] H. Hermanns, M. Z. Kwiatkowska, G. Norman, D. Parker, M. Siegle, On the use of MTBDDs for performability analysis and verification of stochastic systems, Journal of Logic and Algebraic Programming 56 (1–2) (2003) 23–67.

[31] R. E. Bryant, On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication, IEEE Transactions on Computers 40 (2) (1991) 205–213.

[32] B. Bollig, I. Wegener, Improving the variable ordering of OBDDs is NP-complete, IEEE Transactions on Computers 45 (9) (1996) 993–1002.

[33] R. Rudell, Dynamic variable ordering for ordered binary decision diagrams, in: Proc. of ICCAD, IEEE Computer Society, Santa Clara, CA, USA, 1993, pp. 42–47.

[34] J. Schuster, Towards faster numerical solution of continuous time Markov chains stored by symbolic data structures, Ph.D. thesis, Universität der Bundeswehr München, `http://d-nb.info/102057920X/34` (2012).

[35] Y. Breitbart, H. B. Hunt III, D. J. Rosenkrantz, On the size of binary decision diagrams representing boolean functions, Theoretical Computer Science 145 (1&2) (1995) 45–69.

[36] S. A. Cook, The complexity of theorem-proving procedures, in: Proc. of STOC, ACM Press, 1971, pp. 151–158.

[37] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, 2009.

[38] G. S. Tseitin, On the complexity of derivations in the propositional calculus, Studies in Mathematics and Mathematical Logic Part II (1968) 115–125.

[39] V. M. Jiménez, A. Marzal, Computing the $k$ shortest paths: A new algorithm and an experimental comparison, in: Int'l Workshop on Algorithm Engineering (WAE), Vol. 1668 of LNCS, Springer-Verlag, 1999, pp. 15–29.

[40] H. Aljazzar, S. Leue, K$^*$: A heuristic search algorithm for finding the $k$ shortest paths, Artificial Intelligence 175 (18) (2011) 2129–2154.

[41] M. E. Andrés, P. D'Argenio, P. van Rossum, Significant diagnostic counterexamples in probabilistic model checking, in: Proc. of HVC, Vol. 5394 of LNCS, Springer-Verlag, 2008, pp. 129–148.

[42] R. Tarjan, Depth-first search and linear graph algorithms, SIAM Journal on Computing 1 (2) (1970) 146–160.

[43] R. Gentilini, C. Piazza, A. Policriti, Computing strongly connected components in a linear number of symbolic steps, in: SODA, ACM/SIAM, 2003, pp. 573–582.

[44] F. Somenzi, Cudd: Cu decision diagram package release 2.5.0 (2013).

[45] N. Eén, N. Sörensson, An extensible SAT-solver, in: Proc. of SAT, Vol. 2919 of LNCS, Springer-Verlag, 2003, pp. 502–518.

[46] G. Norman, V. Shmatikov, Analysis of probabilistic contract signing, Journal of Computer Security 14 (6) (2006) 561–589.

[47] M. K. Reiter, A. D. Rubin, Crowds: Anonymity for web transactions, ACM Transactions on Information and System Security 1 (1) (1998) 66–92.

[48] A. Itai, M. Rodeh, Symmetry breaking in distributed networks, Information and Computation 88 (1) (1990) 60–87.

[49] M. Kwiatkowska, G. Norman, D. Parker, The PRISM benchmark suite, in: Proc. of QEST, IEEE Computer Society, 2012, pp. 203–204.

[50] PRISM Website, `http://prismmodelchecker.org` (Aug. 2013).

[51] D. Eppstein, Finding the $k$ shortest paths, SIAM Journal on Computing 28 (2) (1998) 652–673.

[52] H. Aljazzar, M. Kuntz, F. Leitner-Fischer, S. Leue, Directed and heuristic counterexample generation for probabilistic model checking – a comparative evaluation, in: Proc. of QUOVADIS, ACM Press, 2010, pp. 25–32.

# MEALS Partner Abbreviations

**SAU:** Saarland University, D

**RWT:** RWTH Aachen University, D

**TUD:** Technische Universität Dresden, D

**INR:** Institut National de Recherche en Informatique et en Automatique, FR

**IMP:** Imperial College of Science, Technology and Medicine, UK

**ULEIC:** University of Leicester, UK

**TUE:** Technische Universiteit Eindhoven, NL

**UNC:** Universidad Nacional de Córdoba, AR

**UBA:** Universidad de Buenos Aires, AR

**UNR:** Universidad Nacional de Río Cuarto, AR

**ITBA:** Instituto Técnológico Buenos Aires, AR