On Probabilistic CEGAR

by
Holger Hermanns, Björn Wachter, Lijun Zhang

AVACS Technical Report No. 33
December 2007
ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog,
Andreas Podelski, Reinhard Wilhelm
ATRs (AVACS Technical Reports) are freely downloadable from `www.avacs.org`

On Probabilistic CEGAR

Holger Hermanns, Björn Wachter, Lijun Zhang

Universität des Saarlandes, Saarbrücken, Germany
{hermanns,bwachter,zhang}@cs.uni-sb.de

Abstract. Counterexample-guided abstraction-refinement (CEGAR) has been *en vogue* for the automatic verification of very large systems in the past years. When trying to apply CEGAR to the verification of probabilistic systems, various foundational questions and practical tradeoffs arise. This paper explores them in the context of predicate abstraction.

1 Introduction

Probabilistic behavioral descriptions are widely used to analyze and verify systems that exhibit "quantified uncertainty", such as embedded, networked, or biological systems. The semantic model for such systems often are Markov chains or Markov decision processes. We here consider homogeneous discrete-time Markov chains (MCs) and Markov decision processes (MDPs). Properties of these systems can be specified by formulas in temporal logics such as PCTL [16, 3], where for instance quantitative probabilistic reachability ("the probability to reach a set of bad states is at most 3%") is expressible. Model checking algorithms for such logics have been devised mainly for finite-state MCs [16] and MDPs [3], and for certain infinite-state models [11, 12], and effective tool support is provided by probabilistic model checkers such as Prism [19] or Mrmc [20]. Despite the remarkable versatility of this approach, its power is limited by the state space explosion problem.

Predicate abstraction [14] is an attractive method for the verification of very large or infinite-state *non-probabilistic* systems, where sets of 'concrete' states are mapped to abstract states according to their valuation under a finite set of predicates. The predicates thus induce an 'abstract' model that is submitted to a model checker together with a property to be verified. Since the abstraction overapproximates the concrete model, the model checker might return *spurious* negative verdicts (false negatives).

Predicate abstraction combines well [2, 18] with *counterexample-guided abstraction refinement* (CEGAR) [4]. In this approach, a negative verification result - in the form of an abstract counterexample - is lifted to the concrete system, where it is inspected in order to decide whether it is spurious or realizable. If spurious, diagnostic information that aids in refining the abstraction via additional predicates is derived. The refined abstraction is again submitted to the model checker. Iterating this process leads to a sequence of abstractions which, once sufficiently precise, can be used to decide if the property is satisfied, or violated by the concrete model.

In this paper, we discuss how counterexample-guided abstraction refinement can be developed in a probabilistic setting. Predicate abstraction has been presented in [29] for a guarded command language whose concrete semantics maps on MDPs – more precise, to probabilistic automata [28] – and where predicates needed to be generated manually. This is the natural basis for our present work. We restrict to probabilistic reachability and thus aim to determine whether the probability to reach a set of bad states exceeds a given threshold.

The core complication when trying to develop probabilistic CEGAR lies in the notion and interpretation of counterexamples. In the traditional setting, an abstract counterexample is a single *finite path* (leading to some bad state) which corresponds to a possibly infinite set of concrete finite paths. The abstract counterexample is spurious if that set is empty, which is routinely checked using an SMT solver these days. In contrast, a counterexample of an MDP reachability probability can be viewed as a *finite* but generally cyclic *Markov chain* [21], which in turn corresponds to possibly infinitely many concrete MCs. Now, each of these MCs represents a possibly infinite tree of probabilistic experiments (leading to bad states). This additional infinite dimension stems from the need for possibly unbounded unwinding of the transition relation in order to accumulate enough probability mass (to refute the given threshold of reachability probability).

One may try to check whether the set of concrete MCs is empty, but since this involves considering cycles, reflecting the above infinite dimension, the standard SMT-approach will not work. We circumvent this problem, by preprocessing the abstract counterexample using the strongest evidence idea of [15]: We generate a finite set *afp* of abstract finite paths that together carry enough abstract probability mass, and formulate the problem of computing the realizable probability mass of *afp* in terms of a weighted MAX-SMT problem [25]. The set *afp* is built incrementally in an *on-the-fly* manner, until either enough probability is realizable, or *afp* cannot be enriched with sufficient probability mass to make the probability threshold realizable, in which case the counterexample is spurious. This procedure can, owed to its incremental nature, exploit *clause learning* [10] along invocations of the MAX-SMT solver. To discover new predicates, we employ interpolants [24].

From a pragmatic perspective, the amazing facet of this approach is that *it works*. All these ingredients result in a theory and tool for probabilistic CEGAR which we implemented in a prototypical form. We experimentally evaluated the approach on the bounded retransmission protocol [6], and on a sliding-window protocol over lossy channels. Compared to the predicate abstraction technique presented in [29], where predicates had to be given manually, CEGAR indeed entirely mechanises the verification process.

Related Work. We are not aware of other works on CEGAR and predicate abstraction for probabilistic systems. However, different abstraction-refinement approaches for finite state probabilistic models [5, 7, 22, 13] have been pursued, aiming at sufficiently precise (upper and lower) bounds on (minimum and maximum) reachability probabilities. While these approaches do not exploit counterexamples, a complete CEGAR approach has been proposed by Chatterjee et. al [21] for finite probabilistic two-player game structures, where each iteration of the refinement loop refines a single abstract state. While finite MDPs are a special case thereof, our work differs since it considers infinite-state models, stays more in the classical predicate abstraction realm, and is supplemented with a running implementation.

Outline. The paper is organized as follows. We briefly review our previous work on predicate abstraction for probabilistic programs in Section 2. The contributions of this paper, counterexample analysis and abstraction refinement for probabilistic programs, are presented in Section 3. Experimental results are presented in Section 4. We finally conclude.

2 Preliminaries

Probabilistic Programs. We model systems by probabilistic programs in a guarded command language [29] which is inspired by the PRISM input language, but supports infinite data domains, see Figure 4 for an example. We fix a finite set of program variables $\mathtt{X}$ and a finite set of actions Act. Variables are typed in a definition such as `i : int`. We denote the set of expressions over the set of variables V by $Expr_V$ and we denote the set of Boolean expression over V by $BExpr_V$. An *assignment* is a total function $E : \mathtt{X} \to Expr_{\mathtt{X}}$ that maps variables $\mathtt{x} \in \mathtt{X}$ to expressions $E(\mathtt{x})$. Given an expression $\mathtt{e} \in Expr_{\mathtt{X}}$ and an assignment E, we denote by $\mathtt{e}[\mathtt{X}/E(\mathtt{X})]$ the expression obtained from $\mathtt{e}$ by substituting each occurrence of a variable $\mathtt{x}$ with $E(\mathtt{x})$.

A *guarded command* $\mathtt{c}$ consists of an action name $a_{\mathtt{c}}$, a guard $\mathtt{g} \in BExpr_{\mathtt{X}}$ and assignments $E_1, ..., E_k$ weighted with probabilities $p_1, ..., p_k$ where $\sum_{i=1}^{k} p_i = 1$. We use the notation $\mathtt{X}' = E$ for the simultaneous update E of variables $\mathtt{X}$. Updates are syntactically separated by a "+":

$$[a]\ \mathtt{g} \to \mathtt{p}_1 : \mathtt{X}'\texttt{=}E_1\ \texttt{+}\ \ldots\ \texttt{+}\ \mathtt{p}_k : \mathtt{X}'\texttt{=}E_k$$

If the guard is satisfied, update $\mathtt{X}' = E_i$ will be executed with probability p_i. For $\mathtt{c}$, we write $a_{\mathtt{c}}$ for its action, $\mathtt{g}_{\mathtt{c}}$ for its guard. Moreover, we use $\mathtt{u}_{\mathtt{c}_i} \in \mathtt{U}$ to denote the i-th update of $\mathtt{c}$. If $\mathtt{c}$ is clear from the context, we write simply $a, \mathtt{g}$ and $\mathtt{u}_i$ instead.

A *program* $\mathtt{P} = (\mathtt{X}, \mathtt{I}, \mathtt{C})$ consists of a Boolean expression $\mathtt{I} \in BExpr_{\mathtt{X}}$ that defines the set of initial states and a set of guarded commands $\mathtt{C}$. Without loss of generality, we assume that the program has distinctly labeled guarded commands – two different commands have distinct action and update labels.

Probabilistic Update Automata. The semantics of a probabilistic program is a probabilistic automaton. To be able to reconstruct both program commands and updates from the semantics, automata are decorated with labels from two alphabets: beside the action alphabet Act for commands, we assume a finite, fixed update alphabet $\mathtt{U}$. A *simple distribution* π over Σ is a function $\pi : \Sigma \to [0, 1]$ such that $\pi(\Sigma) = \sum_{s \in \Sigma} \pi(s) = 1$. Let $Distr_{\Sigma}$ denote the set of all simple distributions over Σ. For two sets $\mathtt{U}$ and Σ, a relation $Z \subseteq \mathtt{U} \times \Sigma$ from $\mathtt{U}$ to Σ is called right-unique if $(x, y) \in Z$ and $(x, y') \in Z$ implies that $y = y'$. An *(update-labelled) distribution* π over Σ with respect to $\mathtt{U}$ is a distribution $\pi \in \mathit{Distr}_Z$ where Z is a right-unique relation from $\mathtt{U}$ to Σ. In this paper, we use Z to denote an arbitrary right-unique relation from $\mathtt{U}$ to Σ. Let $\mathit{Distr}_{(\mathtt{U},\Sigma)} = \cup_Z \mathit{Distr}_Z$ denote the set of distributions over right-unique relations from $\mathtt{U}$ to Σ. For $\pi \in \mathit{Distr}_{(\mathtt{U},\Sigma)}$, $\mathtt{u} \in \mathtt{U}$, the set of states $\mathit{Supp}_{\mathtt{u}}(\pi) = \{s \in \Sigma \mid \pi(\mathtt{u}, s) > 0\}$ is called the support of π with respect to $\mathtt{u}$. The set of states $\mathit{Supp}(\pi) = \bigcup_{\mathtt{u} \in \mathtt{U}} \mathit{Supp}_{\mathtt{u}}(\pi)$ is called the support of π.

Definition 1. *A probabilistic update automaton $\mathcal{M}$ is a tuple $(\Sigma, I, \mathit{Act}, \mathtt{U}, R)$ where Σ is a set of states, $I \subseteq \Sigma$ is the set of initial states, Act is the set of actions, $\mathtt{U}$ is the update alphabet, and $R \subseteq \Sigma \times \mathit{Act} \times \mathit{Distr}_{(\mathtt{U},\Sigma)}$ is the probabilistic transition relation.*

A finite path of length n is a finite sequence $(s_0, a_0, \mathtt{u}_0, \pi_0), (s_1, a_1, \mathtt{u}_1, \pi_1), \ldots s_n$ such that $s_0 \in I$, $(s_i, a_i, \pi_i) \in R$, and $s_{i+1} \in \mathit{Supp}_{\mathtt{u}_i}(\pi_i)$ for all $i = 0, \ldots, n-1$. Let $\mathit{Path}_{\mathit{fin}}(\mathcal{M})$ denote the set of all finite paths over $\mathcal{M}$. We write $\sigma \leq \sigma'$, if the finite path σ is a prefix of σ'. A finite path σ is maximal if $\sigma \leq \sigma'$ implies that $\sigma = \sigma'$. An infinite path σ is an infinite sequence $(s_0, a_0, \mathtt{u}_0, \pi_0), (s_1, a_1, \mathtt{u}_1, \pi_1), \ldots$ starting with an initial state $s_0 \in I$, $(s_i, a_i, \pi_i) \in R$, and $s_{i+1} \in \mathit{Supp}_{\mathtt{u}_i}(\pi_i)$ for all $i = 0, 1, \ldots$. Let $\mathit{Path}_{\infty}(\mathcal{M})$ denote the set of all infinite or maximal paths over $\mathcal{M}$. $\mathcal{M}$ is called finite if Σ is finite. For $\sigma \in \mathit{Path}_{\mathit{fin}}$, let $C(\sigma) = \{\sigma' \in \mathit{Path}_{\infty}(\mathcal{M}) \mid \sigma \text{ is a prefix of } \sigma'\}$ denote the *cylinder set* for σ.

Our definition of probabilistic update automata is essentially the same as probabilistic automata [28], except that we label distributions with an update alphabet. This doesn't give any additional modeling power, it rather allows us to develop the Cegar approach succinctly. Dropping the labels of the distributions, $\mathcal{M}$ induces a probabilistic automaton $\mathit{ind}(\mathcal{M})$ in the style of [28] as follows: replace every update-labelled distribution π by its induced distribution $\mathit{ind}(\pi)$, defined by $\mathit{ind}(\pi)(s) = \sum_{\mathtt{u} \in \mathtt{U}} \pi(\mathtt{u}, s)$. If the context is clear, we use $\mathcal{M}$ and $\mathit{ind}(\mathcal{M})$ interchangingly.

MCs and MDPs are special cases of probabilistic automata. An MC is a deterministic probabilistic automaton, i.e. an automaton where for every state s there is at most one transition $(s, a, \pi) \in R$. An MDP is an action-deterministic probabilistic automaton, i.e. an automaton where for every pair $s \in \Sigma$ and $a \in \mathit{Act}$, there exists at most one π with $(s, a, \pi) \in R$.

Program Semantics. A *state* over variables $\mathtt{X}$ is a type-consistent mapping from variables in $\mathtt{X}$ to their semantic domains. We denote the set of states by $\Sigma(\mathtt{X})$ or Σ for short and a single state by s. For an expression $\mathtt{e} \in \mathit{Expr}_{\mathtt{X}}$, we denote by $[\![\mathtt{e}]\!]_s$ its valuation in state s. The valuation of a Boolean expression $\mathtt{e}$ is a value $[\![\mathtt{e}]\!]_s \in \{0, 1\}$ (0 for "false", 1 for "true"). For a Boolean expression $\mathtt{e}$ and a state s, we write $s \vDash \mathtt{e}$ iff $[\![\mathtt{e}]\!]_s = 1$. Semantic brackets around a Boolean expression $\mathtt{e}$ without a subscript denote the set of states fulfilling $\mathtt{e}$, i.e. $[\![\mathtt{e}]\!] = \{s \in \Sigma \mid s \vDash \mathtt{e}\}$.

Given a program $\mathtt{P} = (\mathtt{X}, \mathtt{I}, \mathtt{C})$, a probabilistic update automaton $\mathcal{M} = (\Sigma, I, \mathit{Act}, R)$ can be defined with set of states $\Sigma = \Sigma(\mathtt{X})$, set of initial states $I = [\![\mathtt{I}]\!]$, set of actions $\mathit{Act} = \{a_{\mathtt{c}} \mid \mathtt{c} \in \mathtt{C}\}$, and transitions induced by the guarded commands $R = \bigcup_{\mathtt{c}\in\mathtt{C}} [\![\mathtt{c}]\!]$ where $(s, a, \pi) \in [\![\mathtt{c}]\!]$ if $s \vDash \mathtt{g}$ and π such that $\pi(\mathtt{u}_i, s') = p_i$ if $s'(\mathtt{x}) = [\![E_i(\mathtt{x})]\!]_s$ for all $\mathtt{x} \in \mathtt{X}$. The semantics of $\mathtt{P}$ is $\mathcal{M}$.

Weakest Preconditions. Dijkstra introduced weakest preconditions for imperative programs [9]: the weakest precondition $\mathtt{WP}_c(Q) = Q'$ of a Boolean expression Q with respect to program statement $\mathtt{c}$ is the weakest Boolean expression (w.r.t. implication order) that guarantees Q to hold after executing $\mathtt{c}$, typically this is written as a triple $\{Q'\}\mathtt{c}\{Q\}$. For an update $\mathtt{X}'$=E, the corresponding triple is $\{Q[\mathtt{X}/E(\mathtt{X})]\}\mathtt{X}'$=$E\{Q\}$, i.e. the weakest precondition of expression Q is obtained by substituting within Q the left-hand side variables with the right-hand side expressions yielding $Q[\mathtt{X}/E(\mathtt{X})]$. Therefore, given a set of states characterized by a Boolean expression P, satisfiability of the conjunction $P \wedge Q[\mathtt{X}/E(\mathtt{X})]$ guarantees the existence of a state transition from P to Q. We abbreviate the weakest precondition of an expression e with respect to an assignment E as $\mathtt{WP}_E(e) = e[\mathtt{X}/E(\mathtt{X})]$.

Properties. Let $p \in [0, 1]$ denote a real constant, and let $\mathtt{e} \in BExpr_{\mathtt{X}}$ be a Boolean expression. In this paper we consider probabilistic reachability properties which can be represented as $Reach_{\leq p}(\mathtt{e})$. For a given Markov chain, such a property is satisfied if the probability measure of the set of paths (henceforth denoted $\rightsquigarrow\mathtt{e}$) reaching states satisfying $\mathtt{e}$ is smaller or equal p. Probabilistic reachability properties are simple safety PCTL formulas [16, 3]. To interpret such properties on MDPs, a probability measure on paths is required, which in turn requires to resolve non-determinism.

An adversary is a resolution of non-determinism. In general, an adversary A of an automaton $\mathcal{M}$ is a function from paths to products of actions and distributions. We let $\mathcal{D}_\delta$ denote the Dirac distribution defined by: $\mathcal{D}_\delta(\delta) = 1$ where δ is a special symbol for termination. An adversary A is called *simple* if it only looks at the last state in a path, i.e. if it is a function $A : \Sigma \to (Act \times Distr_\Sigma) \cup \{\mathcal{D}_\delta\}$. Note if $A(s) = \mathcal{D}_\delta$, the adversary A decides to stop at state s. For PCTL formulas, thus also for $Reach_{\leq p}(\mathtt{e})$, it suffices to consider *simple* adversaries, as extremal probabilities are already attained among them [3]. Given a probabilistic automaton $\mathcal{M}$, a simple adversary A induces a MC $\mathcal{M}_A = (\Sigma, I, Act, R_A)$ where $R_A = \{(s, a, \pi) \in R \mid A(s) = (a, \pi)\}$. For a given state $s \in \Sigma$ and a path $\sigma = (s_0, a_0, \mathtt{u}_0, \pi_0), \ldots s_n$, we define $Prob_s^{\mathcal{M}_A}(\sigma)$ by $\prod_{i=0}^{n-1} \pi_i(\mathtt{u}_i)(s_{i+1})$ if $s_0 = s$ and 0 otherwise. A unique probability measure can be extended [26] from $Prob_s^{\mathcal{M}_A}$. For an adversary A, let $Prob_s^{\mathcal{M}_A}(\rightsquigarrow\mathtt{e}) = Prob_s^{\mathcal{M}_A}(\{\sigma \in Path_\infty(\mathcal{M}) \mid$ some state along σ satisfies $\mathtt{e}\})$ denote the probability of set of paths reaching an $\mathtt{e}$-state. If $\mathcal{M}$ is clear from the context, we write $Prob_s^A(\rightsquigarrow\mathtt{e})$ instead. Thus, a state s satisfies $Reach_{\leq p}(\mathtt{e})$, denoted by $s \models Reach_{\leq p}(\mathtt{e})$, iff $Prob_s^A(\rightsquigarrow\mathtt{e}) \leq p$ for *every* simple adversary A. To check if a state s satisfies a safety PCTL formula $\mathcal{P}_{\leq p}(\phi)$, it suffices to compute $Prob_s^{max}(\phi) = \sup_A Prob_s^A(\phi)$ where A ranges over all simple adversaries. Therefore, $s \models Reach_{\leq p}(\mathtt{e})$ iff $Prob_s^{max}(\rightsquigarrow\mathtt{e}) \leq p$. Let $Prob_I^{max}(\rightsquigarrow\mathtt{e}) = \sup\{Prob_s^{max}(\rightsquigarrow\mathtt{e}) \mid s \in I\}$. We say that $\mathcal{M}$ satisfies $Reach_{\leq p}(\mathtt{e})$, denoted by $\mathcal{M} \models Reach_{\leq p}(\mathtt{e})$, iff $Prob_I^{max}(\rightsquigarrow\mathtt{e}) \leq p$.

Predicates. Predicates are Boolean expressions over the program variables. A predicate φ stands for the set of states satisfying it, namely $[\![\varphi]\!]$. We fix a set of predicates $\mathcal{P} = \{\varphi_1, ..., \varphi_n\}$. A set of predicates $\mathcal{P}$ determines an abstract probabilistic automaton. The set $\mathcal{P}$ induces an equivalence relation over states and a homomorphism. More precisely, two states in Σ are *equivalent* if they satisfy the same set of predicates in $\mathcal{P}$. The equivalence classes partition the states into disjoint sets characterized by which predicates hold and which don't. An equivalence class can therefore be represented by a bit vector of length n. We call such a bit-vector an *abstract state*. We define a homomorphism, which is a total function mapping concrete states to a finite set of abstract states, by: $h_{\mathcal{P}}(s) = (\llbracket\varphi_1\rrbracket_s, ..., \llbracket\varphi_n\rrbracket_s)$. For a given abstract state $s^\sharp$, we call the corresponding equivalence class the concretization of $s^\sharp$ and denote it by $\gamma(s^\sharp)$. Further, we logically characterize the states in the concretization of $s^\sharp$ by a Boolean expression[1] $F(s^\sharp)$ such that $\gamma(s^\sharp) = \{s \mid s \in \llbracket F(s^\sharp)\rrbracket\}$.

Quotient Automaton. Function $h_{\mathcal{P}}$ induces a quotient automaton, denoted by $\mathcal{M}^\sharp(h_{\mathcal{P}})$, or $\mathcal{M}^\sharp$ if $h_{\mathcal{P}}$ is clear. The transitions of the quotient automaton are chosen such that $h_{\mathcal{P}}$ preserves the transition structure. We define a *quotient automaton* such that safety properties are preserved:

Definition 2 (Quotient Automaton). *Let $\mathcal{M}$ be a probabilistic update automaton. Further, let $\Sigma^\sharp$ be a finite set of* abstract states *and $h : \Sigma \to \Sigma^\sharp$ a homomorphism. The homomorphism h induces an automaton $\mathcal{M}^\sharp$, called the quotient of $\mathcal{M}$ (under h), with state set $\Sigma^\sharp$, the same set of actions, initial states $I^\sharp = \{h(s) \mid s \in I\}$, and transitions $R^\sharp = \{(h(s), a, h(\pi)) \mid (s, a, \pi) \in R\}$ where $h(\pi) = \{(\boldsymbol{u}, h(s)) : p \mid \pi(\boldsymbol{u}, s) = p\}$.*

For a given set of predicates $\mathcal{P}$, a quotient automaton $\mathcal{M}^\sharp$ can be constructed, as shown in [29]. Moreover, the *soundness* of the abstraction [29] enables us to check a safety properties on the quotient automaton $\mathcal{M}^\sharp$: If the safety property holds for the quotient automaton $\mathcal{M}^\sharp$, we can safely conclude that it holds for the original model $\mathcal{M}$ as well. The fact that we use right-uniqueness with respect to updates – different from [29] – does not harm these results, but will become instrumental later.

3 Refinement

In this section, we present a novel refinement scheme for probabilistic programs based on CEGAR (counterexample-guided abstraction refinement). The plain CEGAR approach is the obvious strategy also to follow in the probabilistic case: start with a coarse abstraction and successively refine it using predicates learnt from spurious counterexamples until either a realizable counterexample is found or the abstract model is precise enough to establish the property. However, in order to put refinement to work for probabilistic models, several questions of both principal and practical nature need to be answered. We *(i)* need to identify what an abstract counterexample constitutes, *(ii)* lift it to the concrete system,

[1] $F(s^\sharp)$ is the conjunction containing exactly the satisfied predicates in positive and the unsatisfied ones in negated form.

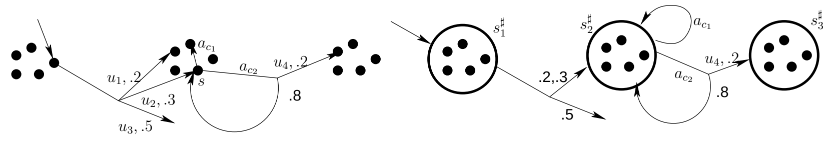


Fig. 1. A probabilistic automaton and its corresponding quotient automaton

(iii) decide if it is spurious, and *(iv)* identify appropriate predicates to refine the abstract quotient automaton.

3.1 Counterexamples for quotient automata

We first introduce the notion of counterexamples for probabilistic automata. Intuitively, a counterexample is a pair of an initial state and an adversary which violates the property to be checked. This pair induces an MC in the abstract setting. We assume we are given some fixed probabilistic update automaton $\mathcal{M}$ and some fixed reachability property $\mathit{Reach}_{\leq p}(\mathsf{e})$ in the sequel.

Definition 3 (Counterexample for quotient automata). *Let $\mathcal{M}^\sharp$ be the quotient automaton of $\mathcal{M}$. A counterexample for $\mathit{Reach}_{\leq p}(\mathsf{e})$ is a pair $(s^\sharp, A^\sharp)$ where $s^\sharp \in I^\sharp$ is an initial state and $A^\sharp$ is an adversary such that $\mathit{Prob}_{s^\sharp}^{A^\sharp}(\rightsquigarrow \mathsf{e}) > p$. In this case, $A^\sharp$ is called a* counter-adversary.

3.2 Spurious counterexamples

In the non-probabilistic setting, a counterexample is a path, which is called spurious if there does not exist a corresponding concrete path. We now introduce the notion of spurious counterexamples for probabilistic automata, which is based on the concretization of an abstract counter-adversary and abstract counterexample.

Concretization of counter-adversaries. Let $A^\sharp$ be a counter-adversary in the quotient $\mathcal{M}^\sharp$. Its concretization, denoted by $\gamma(A^\sharp)$, is an adversary in M defined by:

- $\gamma(A^\sharp)(s) = (a_\mathsf{c}, \pi)$ if $A^\sharp(s^\sharp) = (a_\mathsf{c}, \pi^\sharp)$ with $s^\sharp = h(s)$ and $\pi^\sharp = h(\pi)$.
- $\gamma(A^\sharp)(s) = \mathcal{D}_\delta$ otherwise.

In case of $\gamma(A^\sharp)(s) = \mathcal{D}_\delta$, the adversary $A^\sharp$ chooses the distribution $\pi^\sharp$ from $s^\sharp$, however s does not satisfy the guard $\mathsf{g_c}$ associated with c. Thus, $\gamma(A^\sharp)$ decides to stop at state s, as no corresponding concretization exists. For illustration, consider the fragment of a probabilistic automaton and its corresponding quotient automaton in Figure 1. If the adversary $A^\sharp$ chooses a_{c_2} at state $s_2^\sharp$, the concretization $\gamma(A^\sharp)$ chooses also action a_{c_2} at state s. By our assumption of right-uniqueness the possible guarded commands are labeled distinctly, and thus we can choose at most one corresponding outgoing concrete transition.

Concretization of counterexamples. Now consider a counterexample $(s^\sharp, A^\sharp)$. Its concretization, denoted $\gamma(s^\sharp, A^\sharp)$, is the set: $\{(s, A) \mid A = \gamma(A^\sharp) \wedge s \in (I \cap \gamma(s^\sharp))\}$. Directly linked to the cardinality of the initial state set, the concretization can contain many (even infinitely many) elements, and thus induce many MCs. The reachability probability $Prob(\rightsquigarrow \mathtt{e})$ may differ from element to element. A counterexample $(s^\sharp, A^\sharp)$ is spurious if its concretization does not contain a pair (s, A) such that the probability threshold is exceeded. In other words, a spurious counterexample does not induce any concrete MC for which the probability measure of reaching concrete e-states exceeds the specified threshold.

Definition 4. *Let* $(s^\sharp, A^\sharp)$ *be a counterexample for* Φ *in* $\mathcal{M}^\sharp$*. Then,* $(s^\sharp, A^\sharp)$ *is called* realizable *if there exists* $(s, A) \in \gamma(s^\sharp, A^\sharp)$ *such that* $Prob_s^{\mathcal{M}_A}(\rightsquigarrow \mathtt{e}) > p$*. Otherwise, we say that the counterexample is* spurious*.*

Consider again Figure 1, and property $Reach_{\leq 0.07}(goal)$ where *goal* is only satisfied in those states inside $s_3^\sharp$. The above counterexample is realizable, since in $\mathcal{M}$, under the induced adversary A, the probability of reaching *goal* states is 0.3 which is greater than 0.07.

3.3 Checking Counterexamples

Checking realizability of counterexamples is a key element of the refinement procedure: If a counterexample turns out to be realizable, the property is refuted with A playing the role of a counter-adversary in the concrete model, which can be used for debugging purposes. Otherwise, the abstract model is too coarse and additional predicates will need to be added to eliminate the false negative.

Overall Idea. In the non-probabilistic setting, an abstract counterexample is a single finite abstract path $\sigma^\sharp$ starting in an abstract initial state. Its concretization is a set of corresponding paths in the concrete model each of which starts in some concrete initial state and respects the concrete transition relation. Like in our case, this set might potentially be infinite. If it is empty, the counterexample is spurious. It is common practice to check emptiness of the concretization by expressing the behavior enforced on the concrete program by the abstract path implicitly by a formula and checking the satisfiability of that formula [2, 18]. If the formula is satisfied, then the concretization is non-empty, and we have found a concrete counterexample violating the property. In this case, the counterexample is realizable. Otherwise, it is spurious, and additional predicate can be extracted from the path $\sigma^\sharp$ for refinement.

In the probabilistic setting, however, the situation is much more involved. What makes the counterexample $(s^\sharp, A^\sharp)$ realizable is a concrete initial state s and adversary A such that the probability of reaching an e-state in the thus induced concrete MC exceeds the given threshold p. All candidates (s, A) are contained in $\gamma(s^\sharp, A^\sharp)$ but this set might be infinite. What we need is an emptiness check, since then the counterexample is spurious. Unfortunately, we cannot just encode the behaviour enforced by the counterexample by a formula and use satisfiability checking, basically because the formula will in general not be of finite length. This is because the induced MCs may have cycles, which are needed for possibly unbounded unwinding of the transition relation in order to accumulate the threshold probability p of reaching an $\mathtt{e}$-state.

Thus it is not a good idea to check directly whether the set $\gamma(s^\sharp, A^\sharp)$ is empty. What we do is different. We instead preprocess the abstract counterexample using the strongest evidence idea of Han & Katoen [15]. They have devised a method that, for a given MC, can be used to construct the smallest set of paths reaching $\mathtt{e}$-states with an accumulated probability measure above p. This fits well to our needs.

As the abstract counterexample $(s^\sharp, A^\sharp)$ induces an abstract MC, we can apply the algorithm from [15] yielding *a finite* set of *finite paths* starting from $s^\sharp$ in the quotient automaton, such that the probability measure exceeds p. To check whether the counterexample is spurious, our goal is then to compute how much measure out of this set of paths can be reproduced in *any* concrete model. If that is indeed larger than the threshold p, we have found a realizable counterexample, since the set $\gamma(s^\sharp, A^\sharp)$ is nonempty. Otherwise, we may be able to conclude that it is spurious, or conclude that more work is needed, as we will explain below.

Spuriousness of Abstract Paths. To get this approach working, we must first explain how it can be checked if a single abstract path is realizable or spurious. Let $(s^\sharp, A^\sharp)$ be a counterexample and let $\sigma^\sharp = (s_0^\sharp, a_0, \mathtt{u}_0, \pi_0^\sharp)\ (s_1^\sharp, a_1, \mathtt{u}_1, \pi_1^\sharp) \ldots s_k^\sharp$ be a path in $\mathcal{M}^\sharp_{A^\sharp}$ where $s_0^\sharp = s^\sharp$ and $s_k^\sharp$ satisfies $\mathtt{e}$. The concretization $\gamma(\sigma^\sharp)$ of an abstract path $\sigma^\sharp$ is a (possibly infinite) set of finite paths in $\mathcal{M}_{\gamma(A^\sharp)}$ with consistent states, and the same update and action labels, i.e. $\gamma(\sigma^\sharp) = \{(s_0, a_0, \mathtt{u}_0, \pi_0) \ldots s_k \mid (s_0, \ldots, s_k) \vDash TF(\sigma^\sharp)\}$ where $TF(\sigma^\sharp)$ is the trace formula which is defined as follows:

$$TF(\sigma^\sharp) = \mathtt{I}(\mathtt{X}_0) \wedge \bigwedge_{i=0}^{k} F(s_i^\sharp)(X_i) \wedge \bigwedge_{i=1}^{k-1} (\mathtt{g}_{a_i}(\mathtt{X}_i) \wedge \mathtt{X}_{i+1} = E_{u_i}(\mathtt{X}_i)) \wedge \mathtt{e}(\mathtt{X}_k)\ .$$

The measure of $\sigma^\sharp$ under $(s^\sharp, A^\sharp)$ is $\prod_{i=0}^{k-1} \pi_i^\sharp(\mathtt{u}_i, s_{i+1}^\sharp)$. Note that the paths in the concretization of $\sigma^\sharp$ share the same measure. The path $\sigma^\sharp$ is called *realizable* if its concretization is non-empty, $\gamma(\sigma^\sharp) \neq \emptyset$, otherwise it is called *spurious.* As for the non-probabilistic setting [18, 2], checking spuriousness of an abstract path can be done by checking satisfiability of the trace formula or, equivalently, the satisfiability of the weakest precondition of the path (see algorithm in Figure 2).

```
WP($\sigma^\sharp = (s_0^\sharp, a_{\mathtt{c}_0}, \mathtt{u}_0, \pi_0^\sharp) \ldots s_k^\sharp$)

1: $exp_{\sigma^\sharp} \leftarrow F(s_k^\sharp) \wedge \mathtt{e}$
2: for ($j = k$ .. 0) do
3:     $exp_{\sigma^\sharp} \leftarrow \mathtt{g}_{\mathtt{c}_j} \wedge F(s_j^\sharp) \wedge \mathtt{WP}_{\mathtt{u}_j}(exp_{\sigma^\sharp})$
4: return $exp_{\sigma^\sharp} \wedge I$
```

Fig. 2. WP of an abstract path $\sigma^\sharp$

Lemma 1. *For an abstract path $\sigma^\sharp$, the following statements are equivalent:*

i. The weakest precondition $\mathrm{WP}(\sigma^\sharp)$ *of path $\sigma^\sharp$ is satisfiable.*
ii. The trace formula $TF(\sigma^\sharp)$ of $\sigma^\sharp$ is satisfiable.

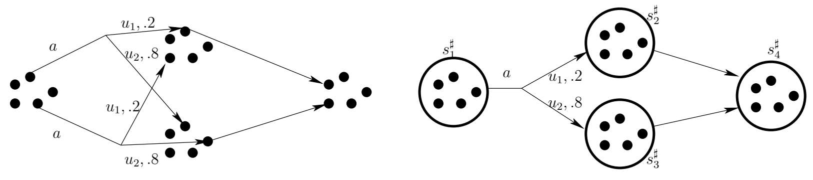


Fig. 3. Why summing up probabilities of abstract paths is insufficient.

iii. The path $\sigma^\sharp$ is realizable, i.e. $\gamma(\sigma^\sharp) \neq \emptyset$.

Checking Spuriousness. We now turn our attention to the algorithm that analyzes if an abstract counterexample is realizable or spurious. It is guaranteed to be realizable if there is a corresponding concrete counterexample that has a sufficiently high measure. We assume a nonempty set *afp* of abstract paths respecting the counterexample. Note that corresponding concrete paths may start in different initial states, so that the probability in the concrete model is possibly lower. Let us consider an abstract path $\sigma^\sharp$. For all $\sigma \in \gamma(\sigma^\sharp)$, the measure of the cylinder set $C(\sigma)$ under $(s, A) \in \gamma(s^\sharp, A^\sharp)$ is given by $\prod_{i=0}^{k-1} \pi_i(\mathtt{u}_i, s_{i+1})$ if $s = s_0$, which is the same as $\prod_{i=0}^{k-1} \pi_i^\sharp(\mathtt{u}_i, s_{i+1}^\sharp)$. For a set *afp* of abstract path we let $\gamma(\mathit{afp}) = \bigcup_{\sigma^\sharp \in \mathit{afp}} \gamma(\sigma^\sharp)$ denote the union of the concretizations. Now an interesting issue arises: what is the maximal probability measure of the set $\gamma(\mathit{afp})$ under some concretization of $\gamma(s^\sharp, A^\sharp)$. For illustration, consider Figure 3 where *afp* consists of two disjoint abstract paths $\sigma_1^\sharp, \sigma_2^\sharp$, but the intersection is empty: $\mathit{exp}_{\sigma_1^\sharp} \wedge \mathit{exp}_{\sigma_2^\sharp} = \emptyset$, hence only the maximum of both can be achieved. We resolve this problem by using weakest preconditions of abstract paths. Given an abstract path $\sigma^\sharp$, the backwards algorithm in Figure 2 computes its weakest precondition, i.e. those initial states in which a path from the concretization of $\sigma^\sharp$ starts. We use these weakest preconditions to obtain subsets of the given set of abstract paths sharing a common concrete initial state. The subset with maximal probability gives us the actual measure in the concrete model.

For $\mathit{afp} = \{\sigma_1^\sharp, \ldots, \sigma_n^\sharp\}$, let $\mathit{exp}_1, \ldots, \mathit{exp}_n$ denote the weakest preconditions returned by $\mathrm{WP}(\sigma_i^\sharp)$. Moreover, for each of them the probability measure of path $\sigma_i^\sharp$ is given as a weight, denoted by p_i, which corresponds to the probability of the set $\gamma(\sigma_i^\sharp)$ starting from some initial state in exp_i. We now formulate the problem of computing the realizable probability mass of a set of abstract paths in terms of a weighted MAX-SMT problem. The *weighted MAX-SMT* [25] problem consists in finding an assignment of X such that the total weight of the satisfied expression is maximal. Formally, it is defined by: $\textsc{MaxSmt}(\mathit{exp}_1, \ldots, \mathit{exp}_n) = \max\left\{\sum_{i=1}^{n} [\![\mathit{exp}_i]\!]_s \cdot p_i \mid s \in [\![\mathtt{I} \wedge F(s^\sharp)]\!]\right\}$.

Lemma 2. *Let $(s^\sharp, A^\sharp)$ be a counterexample and let $\mathit{afp} = \{\sigma_1^\sharp, \ldots, \sigma_n^\sharp\}$ be a set of abstract paths refuting the safety property Φ. It holds:*

– $\textsc{MaxSmt}(\mathit{exp}_1, \ldots, \mathit{exp}_n) > p$ *implies that $(s^\sharp, A^\sharp)$ is realizable,*

– $\textsc{MaxSmt}(exp_1, \ldots, exp_n) + Prob_{s^\sharp}^{A^\sharp}(\rightsquigarrow \mathtt{e}) - Prob_{s^\sharp}^{A^\sharp}(afp) \leq p$ *implies that* $(s^\sharp, A^\sharp)$ *is spurious.*

Let $\varepsilon = Prob_{s^\sharp}^{A^\sharp}(\rightsquigarrow \mathtt{e}) - Prob_{s^\sharp}^{A^\sharp}(afp)$ denote the probability of the set of abstract paths which violate the property Φ, but are not part of the set *afp*. The lemma indicates that the decision algorithm is only partial: if the value $\textsc{MaxSmt}(exp_1, \ldots, exp_n)$ lies in the interval $(p - \epsilon, p]$, we are not sure whether the counterexample $(s^\sharp, A^\sharp)$ is spurious or realizable. By enlarging the set *afp*, the ε can be made arbitrarily small. We will see in Section 3.5 how this is exploited.

3.4 Obtaining Predicates

There are two sources of imprecision: spurious abstract paths and a too coarse abstraction of the initial states.

Predicates to remove spurious paths. Let $(s^\sharp, A^\sharp)$ be a counterexample in $\mathcal{M}^\sharp_{A^\sharp}$. Let $\sigma^\sharp = (s_0^\sharp, a_0, \mathtt{u}_0, \pi_0^\sharp)(s_1^\sharp, a_1, \mathtt{u}_1, \pi_1^\sharp) \ldots s_k^\sharp$ be a path such that $s_0^\sharp = s^\sharp$ and $\sigma^\sharp$ satisfies $\rightsquigarrow \mathtt{e}$. Assume that $\sigma^\sharp$ is spurious. Our goal is to find predicates to eliminate the spurious abstract path. The abstract path resolves both nondeterministic choice between different commands, and probabilistic choice between different updates. That enables us to use standard techniques developed in the non-probabilistic setting to find predicates. Here we employ interpolation and apply it to the trace formula of the abstract path along the lines of [17].

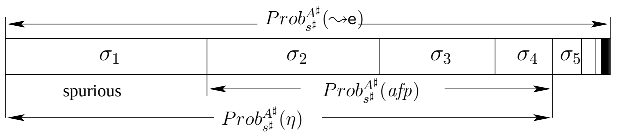
Predicates to separate initial states. Observe the case where no path in *afp* is spurious but the realizable probability of the paths lower than the probability threshold p given by the property, i.e., $\textsc{MaxSmt}(exp_1, \ldots, exp_n) \leq p$. In this case, the abstraction of the initial states may be too coarse. To this end, we choose the maximal solution obtained from $\textsc{MaxSmt}$. Let ψ^- denote the conjunction of non-satisfied exp_i, and ψ^+ denote the conjunction of satisfied exp_i. Obviously, $\psi^- \wedge \psi^+$ is not satisfiable. Hence, the interpolants can be found to refine the abstraction of the initial states. Note that this is a heuristic choice and does not guarantee removal of the abstract counterexample.

3.5 CEGAR algorithm

For a given probabilistic program $(\mathtt{X}, \mathtt{I}, \mathtt{C})$ and some fixed reachability property $Reach_{\leq p}(\mathtt{e})$ the probabilistic CEGAR algorithm proceeds as follows. Each iteration of the CEGAR loop first performs a construction of the abstract model $\mathcal{M}^\sharp$, then a model checking run on the abstract model, and, if the property is not established, finally an analysis of the abstract counterexample $(s^\sharp, A^\sharp)$. After this counterexample analysis the algorithm either reports a realizable counterexample, or it generates new predicates to refine the abstract model. The non-standard component in our instance of the CEGAR loop is an *on-the-fly* counterexample analysis, which uses the criteria (Lemma 1 and Lemma 2) developed in Section 3.3.

CEGAR loop in detail. At the beginning of each iteration of the CEGAR loop, the quotient automaton $\mathcal{M}^\sharp$ is built using the current set of predicates. We submit the quotient automaton and the property to a probabilistic model checker. If the probability exceeds the threshold, a counterexample $(s^\sharp, A^\sharp)$ is produced and is passed to the next phase. Otherwise, we have established the property and report success.

Counterexample analysis constitutes the next phase: Along the ideas of strongest evidence [15] we compute an ordered sequence $\eta = \langle \sigma_1, \sigma_2, ..., \sigma_n \rangle$ of abstract paths reaching an e-state – in the MC generated by $(s^\sharp, A^\sharp)$ on $\mathcal{M}^\sharp$. The sequence η is ordered by decreasing probability mass, and can be computed incrementally, by a variant of best first search [8] in a weighted graph obtained from MC by replacing each probability q by $\ln(q)$ and with additive cost function [1, 15]. The sequence η can be illustrated by the following diagram in which the length of the segments indicate the respective probability measure of the path. The length of all segments (or, measure over all paths appearing in η) is bounded by $Prob^{A^\sharp}_{s^\sharp}(\rightsquigarrow \mathsf{e})$ which is greater than p. Initially η contains only the highest probability path (σ_1).


$Prob^{A^\sharp}_{s^\sharp}(\rightsquigarrow \mathsf{e})$
σ_1
σ_2
σ_3
σ_4
σ_5
spurious
$Prob^{A^\sharp}_{s^\sharp}(afp)$
$Prob^{A^\sharp}_{s^\sharp}(\eta)$

Throughout counterexample analysis, we maintain a set of abstract paths *afp* confirmed to be realizable, starting out with an empty *afp*. We select the path with highest probability, σ_1 in the diagram, and use Lemma 1 to check if it is realizable. If so, we have confirmed that path and we add σ_1 to *afp*, otherwise we do not. (In the diagram we assume σ_1 is spurious.) We select the path with second highest probability, σ_2, add it to η, and check if it is realizable. If realizable, we add σ_2 to *afp*. We continue in this way (assuming $afp = \{\sigma_2, \sigma_3, \sigma_4\}$ for the diagram) until we have a set *afp* with either

$$Prob^{A^\sharp}_{s^\sharp}(afp) > p \quad \text{ or } \quad Prob^{A^\sharp}_{s^\sharp}(afp) + Prob^{A^\sharp}_{s^\sharp}(\rightsquigarrow \mathsf{e}) - Prob^{A^\sharp}_{s^\sharp}(\eta) \leq p.$$

The left hand side of the right inequation gives us a bound on the realizable measure of $\rightsquigarrow \mathsf{e}$ in the concrete model. If this inequation becomes true first, we have a spurious counterexample, and thus we can use the trace formulas of spurious paths for predicate generation, which is the third phase of the CEGAR loop. If instead $Prob^{A^\sharp}_{s^\sharp}(afp) > p$ becomes true, we check for the paths in *afp* how much of their probability is actually realizable using Lemma 2 and MAX-SMT. There are three possible outcomes: (1) we find that the probability mass is high enough to exceed the threshold, in this case we refute the property since we can report a realizable counterexample, (2) the concrete probability mass is that low that the abstract counterexample must be spurious, in this case we go to the third phase, predicate generation, of the loop, or (3) the lemma is inconclusive. In the latter case we are in a tradeoff situation: we can either continue to add some

```
module loop
  N : int;
  invar : N>2;
  bad : bool;
  i : int;
  [a] !bad & i<N ->
   0.9: (i'=i+1) +
   0.1: bad'=(i=N-1)
end module
init !bad & i=0 endinit
```

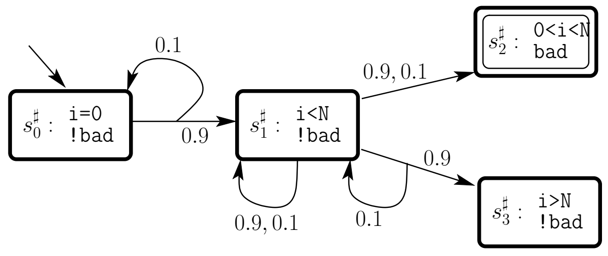


Fig. 4. Cycle program and the quotient automaton with respect to `i=0,bad,i<N`

abstract paths to *afp* and check the inequalities again, or go to the third phase (predicate generation). Since we are in a measurable setting the distances of the involved quantities can be used as obvious heuristics in deciding: If abstract paths with relatively high probability are available for enlarging *afp*, we are more probable to decide whether the counterexample is spurious in the present iteration. Further, if we decide to enlarge the set of abstract paths *afp*, we can incrementally invoke MaxSmt and benefit from clauses learnt so far.

Once we have quit counterexample analysis, the third phase generates predicates learnt from spurious paths or from weakest preconditions. Then the next iteration of the algorithm commences.

To illustrate the above mentioned tradeoff, i.e. when to abort counterexample analysis, consider again Figure 1, and assume we want to check $\mathit{Reach}_{\leq 0.07}(\mathit{goal})$ where *goal* is only satisfied in those states inside $s_3^\sharp$. The smallest set of paths with measure higher than 0.07 in this case contains two paths, $(s_1^\sharp, a_1, \mathtt{u}_1, \pi_1^\sharp), (s_2^\sharp, a_2, \mathtt{u}_4, \pi_2^\sharp), s_3^\sharp$ and $(s_1^\sharp, a_1, \mathtt{u}_2, \pi_1^\sharp), (s_2^\sharp, a_2, \mathtt{u}_4, \pi_2^\sharp), s_3^\sharp$. It gives probability 0.1 of reaching state $s_3^\sharp$, which is larger than the threshold 0.07. The second path is realizable, so we put it into *afp*. We find that the concretization $\gamma(\mathit{afp})$ has only probability measure 0.06. A refinement step seems necessary, but Lemma 2 is inconclusive. However, if we would add another path $(s_1^\sharp, a_1, \mathtt{u}_2, \pi_1^\sharp), (s_2^\sharp, a_2, \mathtt{u}_4, \pi_2^\sharp), (s_2^\sharp, a_2, \mathtt{u}_4, \pi_2^\sharp), s_3^\sharp$ into *afp*, we find out that the counterexample is realizable, thus, we save the refinement step.

As an example of the interpolant generation within CEGAR, we consider the program `Cycle` shown in the left part of Figure 4. The right part shows the quotient automaton with respect to predicates `i=0,bad,i<N` where we omitted the actions and updates. Assume we want to check $\mathit{Reach}_{\leq 0.3}(\mathit{bad})$. In the quotient automaton the probability of reaching the `bad` state is 1. Let $\mathtt{u}_0$ denote the update `i'=i+1`. We take the abstract path with (highest) probability 0.81 (actions and distributions are clear and thus omitted): $\sigma^\sharp = (s_0^\sharp, \mathtt{u}_0)(s_1^\sharp, \mathtt{u}_0)s_2^\sharp$. Obviously, this path is not realizable, which is confirmed by Lemma 1. *afp* is as yet empty. Since $0+1-0.81 \leq 0.3$ we apply Lemma 2 from which we conclude that we have a spurious counterexample. We want to find the interpolant at position $i = 1$. Thus, ψ_1^- is `N0>2`$\wedge$`i0=0`$\wedge$`bad0=false`$\wedge$`i1=i0+1`$\wedge$`N1=N0`$\wedge$`bad1=bad0`$\wedge$`i1<N1`$\wedge$`bad1=false`,

and ψ_1^+ is `i2=i1+1` $\wedge$ `N2=N1` $\wedge$ `bad2=bad1` $\wedge$ `!i2<N2` $\wedge$ `bad2=true`. The common variables are `i1,N1`. Hence, ψ_1^- can be simplified to `i1=1` $\wedge$ `N1>2`, and ψ_1^+ can be simplified to `!(i1<N1-1)`. Thus, we get the interpolant `i<N-1`, with which we can verify the property in the next iteration of the CEGAR loop.

4 Experimental results

In previous work [29], we have developed a predicate abstraction tool for probabilistic models called PASS. To construct abstract models, PASS uses the SMT solver Yices [10] which also provides MAX-SMT support. We have integrated a prototype of abstraction refinement in PASS based on interpolation using the interpolant-generating theorem prover FOCI [23] by Ken McMillan. All experiments were conducted on a Pentium^{TM} 4 2.6 GHz CPU with 1.5 GB of main memory running Linux in 32 bit mode.

	# preds	# paths	maxdepth	probabilities	time
BRP1	3/15	1/7	21/101	1 /1/1.26E-05	8.6s
BRP2	1	1	18	2.64E-05/7.89E-07	2.9s
BRP3	1/2/3/14	1/1/1/13	24/32/50/104	1/1/1/1/5.25E-06	11.2s
BRP4	3	1	8	8.0E-06/1.6E-07	2s
sliding1	1/3/6/2/12	1/1/1/1/1	1/4/16/25/35	1.0/1.0/0.98/0.49/ 3.7E-02/1.96E-02	197s
sliding2	2/12	1/1	4/20	1.0/0.98/1.43E-03	102.8s

Fig. 5. The refinement process is iterative and stops (with a positive verdict) once the probability falls below the threshold p specified by the reachability property. The table shows for each iteration (separated by slashes) the respective statistics: column 1 gives the number of predicates found in an iteration, column 2 the number of abstract paths that had to be explored, column 3 the maximal depth of these paths, column 4 gives the probability obtained in the respective iteration, and the last column gives the time spent in each iteration.

In this paper, we discuss two case studies conducted with PASS. We started with predicates given in the guards and the initial condition and used automatic refinement to obtain further predicates. First, we analyzed the same model of the Bounded Retransmission Protocol (BRP) as in our previous paper [29]. However, there predicates had to be given manually. Now PASS discovers predicates automatically. These predicates are fewer than in the manual process, and, furthermore, obtain the same precision, i.e. probabilities.

Second, we analyzed a model of the Sliding Window Protocol. The sender and the receiver communicate via lossy channels. Unlike in the BRP, the sender continues sending messages without waiting for an acknowledgment from the receiver after each package, until he has exhausted his so-called sending window. Here so-called goodput properties are of interest which consider the difference between the number of sent and received packages. We want to know the probability that the number of sent packages exceeds the number of received packages by a particular constant. For example, PASS checked automatically that, at any time, the probability of the difference exceeding 3 is at most 2 percent for windows size 4.

The specifications of the models used are available from `http://depend.cs.uni-sb.de/pass-cegar.html` together with more detailed experimental results, including results with Rybalchenko's interpolant generator CLP-Prover [27] instead of FOCI.

One may wonder why we did not extensively draw upon the large collection of probabilistic programs available on the PRISM web site with documented properties and probabilities. The reason is that almost all the available models are results of heavy manual abstractions, to make the problem amenable to model checking, and, therefore, offer little opportunity for abstraction.

5 Conclusion

This paper has explored foundational questions and pragmatic issues of probabilistic counterexample-guided abstraction-refinement in the context of predicate abstraction. The main contribution lies in our treatment of abstract counterexamples which are finite Markov chains, instead of finite paths. To account for cyclicity, we have devised a procedure for deciding spuriousness of the counterexample, which is based on solving MAX-SMT problems for sets of abstract paths. This procedure may give an inconclusive answer, in which case we reiterate the procedure with more paths. Spurious counterexamples are analysed with interpolation-based predicate inference, leading to a refined model which closes the CEGAR loop. The resulting theory and tool works smoothly, as shown by our experimental evaluation.

References

1. H. Aljazzar, H. Hermanns, and S. Leue. Counterexamples for timed probabilistic reachability. In *FORMATS*, pages 177–195, 2005.
2. T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02*, pages 1–3. ACM Press, 2002.
3. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *FSTTCS*, pages 499–513. Springer, 1995.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.
5. P. R. D'Argenio, B. Jeannet, H. E. Jensen, and K. G. Larsen. Reduction and Refinement Strategies for Probabilistic Analysis. In *PAPM-PROBMIV*, pages 57–76, 2002.
6. P. R. D'Argenio, J.-P. Katoen, T. C. Ruys, and J. Tretmans. The Bounded Retransmission Protocol Must Be on Time! In *TACAS*, pages 416–431, 1997.
7. L. de Alfaro and P. Roy. Magnifying-lens abstraction for markov decision processes. In *CAV*, pages 325–338, 2007.

8. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality af a*. *J. ACM*, 32(3):505–536, 1985.
9. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
10. B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, pages 81–94, 2006.
11. J. Esparza, A. Kučera, and R. Mayr. Model Checking Probabilistic Pushdown Automata. *Logical Methods in Computer Science*, 2006.
12. K. Etessami and M. Yannakakis. Algorithmic Verification of Recursive Probabilistic State Machines. In *TACAS*, pages 253–270, 2005.
13. H. Fecher, M. Leucker, and V. Wolf. *Don't Know* in probabilistic systems. In *SPIN*, pages 71–88, 2006.
14. S. Graf and H. Saídi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, London, UK, 1997. Springer-Verlag.
15. T. Han and J.-P. Katoen. Counterexamples in probabilistic model checking. In *TACAS*, number 4424 in LNCS, pages 60–75, 2007.
16. H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.
17. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.
19. A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS*, pages 441–444, 2006.
20. J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *QEST*, pages 243–244, 2005.
21. R. J. Krishnendu Chatterjee, Thomas A. Henzinger and R. Majumdar. Counterexample-Guided Planning. In *UAI*, July 2005.
22. M. Kwiatkowska, G. Norman, and D. Parker. Game-based Abstraction for Markov Decision Processes. In *QEST*, pages 157–166, 2006.
23. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
24. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.
25. C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.*, 43(3):425–440, 1991.
26. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 1994.
27. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *VMCAI*, volume 4349, pages 346–362. Springer-Verlag, 2007.
28. R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.
29. B. Wachter, L. Zhang, and H. Hermanns. Probabilistic Model Checking Modulo Theories. In *Proceedings of the Fourth International Conference on the Quantitative Evaluation of Systems*, September 2007.

A Proof of Lemma 1

Proof. We show for $\mathrm{WP}(\sigma^\sharp)$ and $TF(\sigma^\sharp)$: $\mathrm{WP}\sigma^\sharp(\mathtt{X}_0) \equiv \exists \mathtt{X}_1, \ldots, \mathtt{X}_k . TF(\sigma^\sharp)$ which proves claim $(i) \Leftrightarrow (ii)$. To show the stronger equivalence, it is sufficient to establish the following loop invariant for the loop in Algorithm 2:

$$exp^{j}_{\sigma^\sharp}(\mathtt{X}_0) \equiv \exists \mathtt{X}_{j+1}, \ldots, \mathtt{X}_k. \bigwedge_{i=j}^{k} F(s^\sharp_i)(X_i) \wedge \bigwedge_{i=j}^{k-1} (\mathtt{g}_{\mathtt{c}_i}(\mathtt{X}_i) \wedge \mathtt{X}_{i+1} = E_{u_i}(\mathtt{X}_i)) \wedge \mathtt{e}(\mathtt{X}_k)$$

where $exp^{j}_{\sigma^\sharp}$ denotes the value of variable $exp_{\sigma^\sharp}$ and j is the loop iteration variable. On loop entry, $j = k$, we have $exp^{k}_{\sigma^\sharp}(\mathtt{X}_0) = F(s^\sharp_k)(\mathtt{X}_k) \wedge \mathtt{e}(\mathtt{X}_k)$. Let us consider a subsequent iteration j, we have $exp^{j-1}_{\sigma^\sharp} = \mathtt{g}_{\mathtt{c}_{j-1}} \wedge \mathtt{WP}_{\mathtt{u}_{j-1}}(exp^{j}_{\sigma^\sharp}) \wedge F(s^a_{j-1})$. The equivalence[2]: $\mathtt{WP}_{\mathtt{u}_{j-1}}(exp^{j}_{\sigma^\sharp}) \equiv \exists \mathtt{X}_j.\ (\mathtt{X}_j = E_{u_{j-1}} \wedge exp^{j}_{\sigma^\sharp})$ holds. By applying the loop invariant, we obtain the claim. The equivalence $(ii) \Leftrightarrow (iii)$ holds by definition.

B Proof of Lemma 2

Proof. The measure of the concretization $\gamma(\mathit{afp})$ is $\textsc{MaxSmt}(exp_1, \ldots, exp_n)$. If $\textsc{MaxSmt}(exp_1, \ldots, exp_n) > p$, a counterexample exists: Let s be the state achieving the maximum value for $\textsc{MaxSmt}(exp_1, \ldots, exp_n)$, thus, $\sum_{i=1}^{n} [\![exp_i]\!]_s \cdot p_i > p$. Starting from s under the adversary $\gamma(A^\sharp)$, the probability of refuting $\rightsquigarrow\mathtt{e}$ is higher than p, thus the counterexample $(s^\sharp, A^\sharp)$ is not spurious.

Assume that $\textsc{MaxSmt}(exp_1, \ldots, exp_n) + Prob^{A^\sharp}_{s^\sharp}(\rightsquigarrow\mathtt{e}) - Prob^{A^\sharp}_{s^\sharp}(\mathit{afp}) \leq p$. Let $\widetilde{\mathit{afp}}$ denote the set of paths belonging to $\rightsquigarrow\mathtt{e}$ but not in afp. Thus, it has measure $Prob^{A^\sharp}_{s^\sharp}(\rightsquigarrow\mathtt{e}) - Prob^{A^\sharp}_{s^\sharp}(\mathit{afp})$. First we observe that the probability of $\gamma(\widetilde{\mathit{afp}})$ in $\mathcal{M}$ under $(s, \gamma(A^\sharp))$ with $s \in \gamma(s^\sharp)$ is bounded by $Prob^{A^\sharp}_{s^\sharp}(\rightsquigarrow\mathtt{e}) - Pr^{A^\sharp}_{s^\sharp}(\mathit{afp})$. Since $\textsc{MaxSmt}(exp_1, \ldots, exp_n)$ is maximal probability of satisfying $\gamma(\mathit{afp})$, we have that the probability of satisfying $\rightsquigarrow\mathtt{e}$ in $\mathcal{M}$ is bounded by the sum of the two probabilities, which is smaller or equal to p. Hence, the counterexample $(s^\sharp, A^\sharp)$ is spurious.

C Case Study: Sliding Window Protocol

We have considered a model of the Sliding window protocol (go-back(N)). We have checked a goodput property and the probability of a timeout given by the following property file

```
//goodput:
//probability that 3 more packages have been sent than received
Pmax<=0.24 [ true U sent > expected + 2]

//timeout probability:
, Pmax<=1.3E-03  [true U clock >= TIMEOUT]
```

[2] Given a variable x, a term a and a formula f, we have that $\exists x.\ x = a \wedge f$ is logically equivalent to $f[a/x]$. This rule generalizes to multiple quantification variables.

The model is given below:

```
const N = 4;
const TIMEOUT = 8;

// belongs to channel2
ack_nr : int;

// belongs to receiver
// only visible to channel and receiver
expected : int;

module sender
sstate : [0..1];
// control state of sender
// 0 ... send & wait for ack
// 1 ... timeout / retransmit
nextseqnum : int;
base : int;
clock : int;
// number of packages that have been sent
sent : int;
// state 1 ... send & wait for acknowledgment
// -----------------------------------------
// (1) send first package
// (2) start the timer
[data_s2c] sstate = 0 & nextseqnum = base ->
(clock'=0) & (sent'=sent+1) & (nextseqnum'=nextseqnum + 1);
// timeout has happened
[] sstate = 0 & clock>=TIMEOUT -> (sstate'=1);
// sending some package
[data_s2c] sstate = 0 & nextseqnum < base + N - 1 &
   clock<TIMEOUT ->
(nextseqnum'=nextseqnum + 1) & (clock'=clock+1) &
(sent'=sent+1);
[data_s2c] sstate = 0 & nextseqnum = base + N - 1 &
clock<TIMEOUT ->
(nextseqnum'=nextseqnum + 1) & (sent'=sent+1);
// getting acknowledgement
[ack_c2s]  sstate = 0 & clock<TIMEOUT  ->
(base'= ack_nr + 1) & (clock'=0);

// timeout state ... handle timeout situation
// ------------------------------------------
[reset] sstate = 1 ->
(nextseqnum'=base) & (sstate'=0) & (clock'=0);
```

```
endmodule

//
// channel of the sender
//
module channel1
cstate1 : [0..2];
// 0 ... wait
// 1 ... send data to receiver ?
// 2 ... send data to receiver

// buffer contents
packet : int;
[data_s2c] cstate1 = 0 -> (packet'=nextseqnum) & (cstate1'=1);
[] cstate1=1 -> 0.98: (cstate1'=2) + 0.02: (cstate1'=0);
[data_c2r] cstate1=2 -> (cstate1'=0);
endmodule

//
// channel of the receiver
//
module channel2
cstate2 : [0..2];
// 0 ... wait
// 1 ... send ack to sender ?
// 2 ... send ack
[ack_r2c] cstate2=0 -> (ack_nr'=expected) & (cstate2'=1);
[] cstate2=1 -> 0.98: (cstate2'=2) + 0.02: (cstate2'=0);
[ack_c2s] cstate2 = 2 -> (cstate2'=0);
endmodule

module receiver
rstate : [0..1];
// 0 ... wait
// 1 ... ack
// read package
[data_c2r] rstate = 0 & packet = expected ->
(expected'= expected + 1) & (rstate'=1);
[ack_r2c] rstate=1 -> (rstate'=0);
endmodule

init
sent = 0 & ack_nr = -1 & expected = 0 & base = 0 &
nextseqnum = 0 & clock = 0 & sstate = 0 & cstate1 = 0 &
```

```
cstate2 = 0 & rstate = 0 & packet = 0
endinit
```

D Case Study: Bounded Retransmission Protocol

We have considered the following properties:

```
// property A: "Eventually the sender reports a certain unsuccessful
// transmission but the receiver got the complete file."
// property A: true U(srep=NOK & rrep=OK & recv=true)
Pmax = 0.0[ true U srep=1 & rrep=3 & recv ]

// property B: "Eventually the sender reports a certain successful
// transmission but the receiver did not get the complete file."
// true U (srep=OK & !(rrep=OK) & recv=true)
, Pmax = 0.0 [true U srep=3  & rrep!=3 & recv ]

// property 1: "Eventually the sender does not
// report a successful transmission."
// Property 1: true U (s=error & T=true)
 , Pmax <= 1.262E-05 [true U s=5 & T ]

// property 2: "Eventually the sender reports an uncertainty
// on the success of the transmission."
// Property 2: true U (s=error & T=true & srep=DK)
, Pmax <= 7.89E-07 [true U s=5 & T & srep=2 ]

// property 3: "Eventually the sender reports an unsuccessful
// transmission after more than 8 chunks have been sent
// successfully."
// Property 3: true U (s=error & T=true & srep=NOK & i>8)
, Pmax <= 5.25E-06 [ true U s=5 & T & srep=1 & i > 8  ]

// property 4: "Eventually the receiver does not receive any
// chunk and the sender tried to send a chunk."
// Property 4: true U (!(srep=0) & T=true & recv=false)
, Pmax <= 1.6E-07 [ true U srep!=0 & T& !recv]
```

The model is given below:

```
// maximum number of retransmissions
const int MAX=3;
//// number of chunks
const int N=16;

T : bool;
```

```
module sender
s : [0..6];
// 0 idle
// 1 next_frame
// 2 wait_ack
// 3 retransmit
// 4 success
// 5 error
// 6 wait_sync
srep : [0..3];
// 0 bottom
// 1 not ok (nok)
// 2 do not know (dk)
// 3 ok (ok)
nrtr : [0..MAX];
i : [0..N];
bs : bool;
s_ab : bool;
fs : bool;
ls : bool;
// idle
[NewFile] (s=0) -> (s'=1) & (i'=1) & (srep'=0);
// next_frame
[EF] (s=1) -> (s'=2) & (fs'=(i=1)) & (ls'=(i=N)) &
                      (bs'=s_ab) & (nrtr'=0);
// wait_ack
[B] (s=2) -> (s'=4) & (s_ab'=!s_ab);
[TO_Msg] (s=2) -> (s'=3);
[TO_Ack] (s=2) -> (s'=3);
// retransmit
[EF] (s=3) & (nrtr<MAX) -> (s'=2) & (fs'=(i=1)) & (ls'=(i=N)) &
                                 (bs'=s_ab) & (nrtr'=nrtr+1);
[] (s=3) & (nrtr=MAX) & (i<N) -> (s'=5) & (srep'=1);
[] (s=3) & (nrtr=MAX) & (i=N) -> (s'=5) & (srep'=2);
// success
[] (s=4) & (i<N) -> (s'=1) & (i'=i+1);
[] (s=4) & (i=N) -> (s'=0) & (srep'=3);
// error
[SyncWait] (s=5) -> (s'=6);
// wait_sync
[SyncWait] (s=6) -> (s'=0) & (s_ab'=false);
endmodule

module receiver
```

```
r : [0..5];
// 0 new_file
// 1 fst_safe
// 2 frame_received
// 3 frame_reported
// 4 idle
// 5 resync
rrep : [0..4];
// 0 bottom
// 1 fst
// 2 inc
// 3 ok
// 4 nok
fr : bool;
lr : bool;
br : bool;
r_ab : bool;
recv : bool;
// new_file
[SyncWait] (r=0) -> (r'=0);
[EG] (r=0) -> (r'=1) & (fr'=fs) & (lr'=ls) & (br'=bs) & (recv'=T);
// fst_safe_frame
[] (r=1) -> (r'=2) & (r_ab'=br);
// frame_received
[] (r=2) & (r_ab=br) & (fr=true) & (lr=false)  -> (r'=3) & (rrep'=1);
[] (r=2) & (r_ab=br) & (fr=false) & (lr=false) -> (r'=3) & (rrep'=2);
[] (r=2) & (r_ab=br) & (fr=false) & (lr=true)  -> (r'=3) & (rrep'=3);
[A] (r=2) & !(r_ab=br) -> (r'=4);
// frame_reported
[A] (r=3) -> (r'=4) & (r_ab'=!r_ab);
// idle
[EG] (r=4) -> (r'=2) & (fr'=fs) & (lr'=ls) & (br'=bs) & (recv'=T);
[SyncWait] (r=4) & (ls=true) -> (r'=5);
[SyncWait] (r=4) & (ls=false) -> (r'=5) & (rrep'=4);
// resync
[SyncWait] (r=5) -> (r'=0) & (rrep'=0);
endmodule

module checker
//T : bool;
[NewFile] (T=false) -> (T'=false);
[NewFile] (T=false) -> (T'=true);
endmodule

module channelK
```

```
k : [0..2];
// idle
[EF] (k=0) ->
0.98 : (k'=1) +
0.02 : (k'=2);
// sending
[EG] (k=1) -> (k'=0);
// lost
[TO_Msg] (k=2) -> (k'=0);
endmodule

module channelL
l : [0..2];
// idle
[A] (l=0) -> 0.99 : (l'=1) + 0.01 : (l'=2);
// sending
[B] (l=1) -> (l'=0);
// lost
[TO_Ack] (l=2) -> (l'=0);
endmodule

init
s = 0 & srep = 0 & nrtr = 0 & i = 0 & !bs &
!s_ab & !fs & !ls & r = 0 & rrep = 0 & !fr &
!lr & !br & !r_ab & !recv & k = 0 & l = 0 & !T
endinit
```