

Bachelor Thesis

**CEGAR for Computing High-Level
Counterexamples for Probabilistic
Systems**

David Korzeniewski

March 28, 2014

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus fremden Schriften übernommen sind, sind als solche kenntlich gemacht.

Aachen, 28. März 2014

Abstract

In this thesis we present a technique to compute high-level counterexamples for probabilistic systems as described in [Wimmer et al., 2013]. High-level counterexamples operate on the description of probabilistic automata in a guarded command language inspired by the input language of the probabilistic model checker PRISM [Kwiatkowska et al., 2011]. While minimal high-level counterexamples provide valuable feedback for debugging a probabilistic model, their computation is computationally hard. Hence, we aim at deriving such counterexamples on an abstraction of the model. Following the ideas of [Wachter et al., 2007], we show how to apply the well-known paradigm of counterexample guided abstraction refinement to probabilistic systems in a way that enables us to derive a high-level counterexample for the concrete model on a potentially coarse abstraction without having to build an explicit representation of the full system. We show the correctness of the approach and present experimental results showing the possible speed-up obtained in practice.

Contents

1. Introduction	1
1.1. Probabilistic Model Checking	1
1.2. Scope	2
1.3. Structure	2
2. Preliminaries	5
2.1. Probabilistic Automata	5
2.2. Reachability Probabilities on Probabilistic Automata	7
2.3. Probabilistic Programs	10
2.3.1. Probabilistic Guarded Command Language	11
2.3.2. Semantics of Probabilistic Programs	12
2.4. Probabilistic Reachability Properties	13
3. Counterexamples	15
3.1. Path Set Counterexamples	15
3.2. Smallest Critical Command Set	16
3.3. Computing Smallest Critical Command Sets	16
4. Counterexample Guided Abstraction Refinement	19
4.1. Overview CEGAR Algorithm	19
4.2. Program Abstraction	21
4.3. Model Checking the Abstraction and Computing a Counterexample	23
4.4. Counterexample Analysis	23
4.4.1. Checking Realizability	24
4.4.2. Generating New Predicates	26
4.5. Simulation and Path Lifting	29
5. Computing High-Level Counterexamples on Abstractions	31
5.1. CEGAR for Computing Smallest Critical Command Sets	31
5.1.1. Program Abstraction	32
5.1.2. Model Checking the Abstract System and Computing a High- Level Counterexample	32
5.1.3. Counterexample Analysis for High-Level Counterexamples .	33
5.2. Correctness	33

Contents

5.3. Experimental Results	36
6. Conclusion	39
6.1. Future Work	39
References	40
A. Appendix	43

List of Figures

2.1.	A probabilistic automaton \mathcal{E} , represented by a digraph.	6
2.2.	The quotient \mathcal{E}/R of the PA \mathcal{E} , for an equivalence R , represented by a digraph	7
2.3.	A DTMC, \mathcal{E}^σ represented by a digraph.	10
2.4.	The probabilistic automaton $\llbracket \mathfrak{E} \rrbracket$, the semantics of the probabilistic program \mathfrak{E} , represented by a digraph.	13
4.1.	Flowchart of CEGAR algorithm.	20
4.2.	Abstract system in first iteration. (Self loops generated by third command omitted)	27
4.3.	Abstract system in second iteration. (Self loops generated by third command omitted)	28

1. Introduction

Formal verification of computer systems becomes more and more important as computer systems are used in important parts of our lives. Today, nearly all of our communication infrastructure, transportation infrastructure and medical technology heavily rely on computer systems. Failure or malfunctions of these systems can have severe consequences. Thus it is crucial to verify the algorithms used in these systems. To ensure the correctness of computer systems *model checking* is one common formal method.

Most of these systems exhibit certain unpredictable behaviors due to unreliable hardware or unpredictable environmental influences. Hence, we cannot assume them to be 100% reliable. However, the probability of random events often can be quantified. Other systems explicitly use random influences, such as randomized algorithms. In this case the probabilities are also known.

For these *probabilistic systems* we can apply *probabilistic model checking*. In probabilistic model checking the systems may exhibit probabilistic behaviors and properties may contain probabilities. In case a certain property is violated by the system we are interested in counterexamples hinting at the erroneous part of the system to track down the error and correct it.

High-level counterexamples are counterexamples that operate on the level of symbolic representation of probabilistic systems as *probabilistic programs*. In this thesis we present a new approach to speed up the computation of *high-level counterexamples*.

1.1. Probabilistic Model Checking

Systems that exhibit probabilistic as well as nondeterministic behaviors can be modeled as *probabilistic automata* or *Markov decision processes*. For these systems there exist well known techniques such as value iteration or policy iteration [Bellman, 1957] [Puterman, 1994] to check probabilistic properties. These systems are often described in a high-level formal language. In this thesis we will use PRISM's [Kwiatkowska et al., 2011] *probabilistic guarded command language* to describe probabilistic automata. The program written in the probabilistic guarded command language and a set of properties are passed to a model checking

1. Introduction

tool which then checks satisfaction of the properties. In case a property is violated a counterexample is generated.

1.2. Scope

Wimmer et al. described a novel approach to counterexamples for probabilistic systems [Wimmer et al., 2013]. They introduced *high-level counterexamples* for probabilistic systems that operate on the level of the description in a guarded command language. They showed how to obtain a smallest set of commands of a probabilistic program, that induce an erroneous system, using mixed integer linear programming. However their approach requires solving an NP-Hard problem. Their experiments showed that this leads to problems with the computation time, which often exceeded the used time limit.

Hermanns et al. [2008] described how to apply the well known technique of *counterexample guided abstraction refinement (CEGAR)* to probabilistic systems. CEGAR can be used to save memory and speed up model checking of very large systems by generating an abstraction that is iteratively refined until the satisfaction of a property for the original system can be decided.

In this thesis we present an approach to speed up the computation of high-level counterexamples for probabilistic systems. This approach utilizes CEGAR such that high-level counterexamples only have to be computed on an abstraction of the system. We describe the algorithm in detail and show that optimality, i.e. we compute smallest counterexamples, of the counterexamples is preserved, if the used algorithm to compute the counterexamples of the abstraction is optimal in that sense. Finally we evaluate experimental results with a prototypical implementation of the presented technique. The experiments show that in some cases the computation time is reduced by multiple orders of magnitude but in other cases the performance is worse than with the MILP-based approach.

1.3. Structure

In the second chapter we will lay the foundations for the following chapters. We will formally define probabilistic automata, the guarded command language and probabilistic reachability properties. We will also provide some additional definitions and state some known theorems about these theories, that are used in the course of this thesis.

In Chapter 3 we formally define two different types of counterexamples and briefly sketch how they are computed. In Chapter 4 we will describe counterexample guided abstraction refinement in detail and discuss some properties of the

algorithm that are used in the following parts. In Chapter 5 at first counterexample guided abstraction refinement for computing high-level counterexamples is described. After that we show that the algorithm is correct and, given optimality of the algorithm used for computing high-level counterexamples on the abstraction, yields smallest counterexamples. Finally we discuss experimental results of a prototype implementation of the algorithm. In the last Chapter we recap the presented techniques and summarize the results of this work.

2. Preliminaries

2.1. Probabilistic Automata

Probabilistic automata are a state based model for probabilistic systems. They can model systems that exhibit probabilistic as well as nondeterministic behavior. The following definition is based on the definitions in [Wimmer et al., 2013].

Let S be a finite set. A function $\mu : S \rightarrow [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$ is called a *probability distribution over S* . By $Dist(S)$ we denote the set of all probability distributions over S .

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

A probabilistic automaton intuitively works as follows. Initially the PA is in its initial state s_{init} . From there an action and a distribution (α, μ) , in the course of this thesis referred to as transition, are chosen nondeterministically from $\mathbf{P}(s_{init})$. Then a next state s' is chosen probabilistically according to μ . In s' this process is repeated and yields a next state and so on. We require $\mathbf{P}(s) \neq \emptyset$ for all states $s \in S$ to prevent deadlocks.

Example Probabilistic automata can be represented as a digraph. Figure 2.1 shows an automaton $\mathcal{E} = (\{0, 1, 2, 3\}, 1, \{\alpha, \beta, \tau\}, \mathbf{P})$. The large circles represent the states, the initial state is marked with an arrow. The probabilistic transition relation is represented by the edges of the digraph. Edges that overlap up to a small black dot are part of the same transition. The label before the black dot indicates the action of the transition, the labels after the black dot represent the probability of reaching the target state of that edge in the distribution of the transition. For example $\mathbf{P}(0) = \{(\alpha, \mu), (\beta, \mu')\}$, where $\mu(1) = 0.6$, $\mu(2) = 0.4$, $\mu(0) = \mu(3) = 0$, $\mu'(2) = 1$ and $\mu'(0) = \mu'(1) = \mu'(2) = 0$.

Let $succ_{\mathcal{A}}(s, \alpha, \mu) = \{s' \in S \mid \mu(s') > 0\}$ for all $(\alpha, \mu) \in \mathbf{P}(s)$, i.e. the set of all possible successors of a state for a given action and distribution, and $succ_{\mathcal{A}}(s) =$

2. Preliminaries

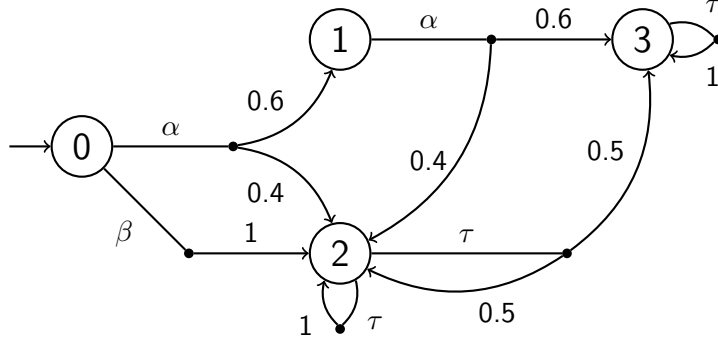


Figure 2.1.: A probabilistic automaton \mathcal{E} , represented by a digraph.

$\bigcup_{(\alpha, \mu) \in \mathbf{P}(s)} \text{succ}_{\mathcal{A}}(s, \alpha, \mu)$. The subscript \mathcal{A} may be omitted if the PA is clear from the context.

An infinite *path* π in a PA \mathcal{A} is an infinite sequence of the form $s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1) \dots$, such that $(\alpha_i, \mu_i) \in \mathbf{P}(s_i)$ and $s_{i+1} \in \text{succ}_{\mathcal{A}}(s_i, \alpha_i, \mu_i)$ for all $i \in \mathbb{N}$. A *finite path* ρ is a finite prefix $s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1) \dots s_n$ of an infinite path π . The *length* of a finite path is denoted by $\text{len}(\rho) = n$. By $\pi[i]$ we denote the i -th state and by $\text{act}(\pi[i])$ the i -th action on a path π . Analogously $\rho[i]$ and $\text{act}(\rho[i])$ denote the i -th state and action respectively on a finite path ρ . Further let $\text{last}(\rho) = \rho[n]$ denote the last state of a finite path ρ . The set of all infinite paths in \mathcal{A} starting in a state s is denoted by $\text{Paths}_{\mathcal{A}}(s)$, and the set of all finite paths by $\text{Paths}_{\mathcal{A}}^{\text{fin}}(s)$. A finite or infinite path is called *initial* if its first state is s_{init} .

Let S be a set and $R \subseteq S \times S$ an equivalence relation. For $s \in S$ we denote the equivalence class of s by $[s]_R = \{s' \in S \mid (s, s') \in R\}$. The partition of S into equivalence classes is denoted by $S/R = \{[s]_R \mid s \in S\}$.

Definition 2 (Probabilistic Automaton Quotient). *For a probabilistic automaton $\mathcal{A} = (S, s_{\text{init}}, \text{Act}, \mathbf{P})$ let $R \subseteq S \times S$ be an equivalence relation on its states. The quotient of \mathcal{A} with respect to R is defined as $\mathcal{A}/R = (S/R, [s_{\text{init}}]_R, \text{Act}, \mathbf{P}/R)$, where $\mathbf{P}/R : S/R \rightarrow 2^{(\text{Act} \times \text{Dist}(S/R))}$ such that $(\alpha, \mu/R) \in \mathbf{P}/R([s]_R)$ if and only if $(\alpha, \mu) \in \mathbf{P}(s')$ for some $s' \in [s]_R$ and $\mu/R([t]_R) = \sum_{t' \in [t]_R} \mu(t')$ for all $t \in S$.*

Intuitively the quotient is built by lumping together abstract states. We will refer to the resulting abstract states of \mathcal{A}/R as *blocks* to distinguish them from states of \mathcal{A} . When joining a state into a block, the outgoing transitions are added to the transitions of the block and for incoming transitions the probabilities of reaching the states in the block are added to get the probability of reaching the block.

Example Figure 2.2 shows the quotient \mathcal{E}/R of the PA \mathcal{E} for the equivalence $R = \{(0, 0), (0, 1), (1, 0), (1, 1), (0, 3), (3, 0), (1, 3), (3, 1), (3, 3), (2, 2)\}$.

2.2. Reachability Probabilities on Probabilistic Automata

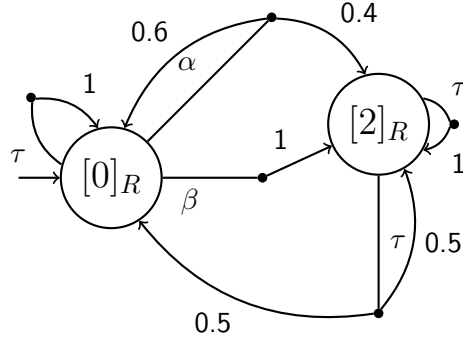


Figure 2.2.: The quotient $\mathcal{E}/_R$ of the PA \mathcal{E} , for an equivalence R , represented by a digraph

A quotient $\mathcal{A}/_R$ simulates the original automaton \mathcal{A} [Wachter et al., 2007]. Intuitively this means that for every step that the original automaton \mathcal{A} can do in some state s , the quotient $\mathcal{A}/_R$ can do an equivalent step in the block $[s]$ with at least the same probability. In this context we say that a step is equivalent if a transition with the same action was chosen and if \mathcal{A} moves to s' the quotient moves to $[s']$.

Definition 3 (Discrete Time Markov Chain). *A discrete time Markov chain (DTMC) is probabilistic automaton $\mathcal{M} = (S, s_{init}, Act, \mathbf{P})$ such that $|\mathbf{P}(s)| = 1$ for all $s \in S$.*

Intuitively a discrete time Markov chain is a system that selects a next state probabilistically based on its current state, starting from its initial state. So, it is a probabilistic automaton that does not exhibit any nondeterministic behavior. In the course of this thesis we will use the terms Markov chain and DTMC synonymously to probabilistic automaton without nondeterminism.

2.2. Reachability Probabilities on Probabilistic Automata

In probabilistic model checking we are interested in the probability of events in probabilistic systems. So we have to define a probability measure on probabilistic automata. At first we define measures for the special case of Markov chains. From there we then generalize the definitions to probabilistic automata.

2. Preliminaries

Reachability Probabilities on Markov Chains

For DTMCs a standard probability measure can be defined [Baier and Katoen, 2008]. Infinite paths in the Markov chain are the outcomes¹ and cylinder sets of finite paths serve as events in the associated probability space. A cylinder set of a finite path ρ is the set of all infinite paths π , such that ρ is a prefix of π . We write $Cyl(\rho)$ for the cylinder set of a finite path ρ .

For DTMC \mathcal{A} and a finite path $\rho = s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1) \dots s_n$ the probability of its cylinder set is defined as

$$Pr_{\mathcal{A}}(Cyl(\rho)) = \prod_{j=0}^{n-1} \mu_j(s_{j+1})$$

if $s_0 = s_{init}$. Otherwise, i.e. for paths that don't start in the initial state, $Pr_{\mathcal{A}}(Cyl(\rho)) = 0$. In the course of this thesis we will use $Pr_{\mathcal{A}}(\rho)$ as a shorthand notation for $Pr_{\mathcal{A}}(Cyl(\rho))$. For brevity we will also speak of the probability of finite paths; formally that always means the probability of cylinder sets.

For sets of finite paths, whose cylinder sets are pairwise disjoint, also a probability is implicitly defined. Let P be a set of finite paths, such that for all $\rho, \rho' \in P$ ρ is a prefix of ρ' implies $\rho = \rho'$. The cylinder sets of the elements of R are pairwise disjoint. As the union of the cylinder sets of the paths in P is a union of outcomes, its probability is defined by a, possibly infinite, series.

$$Pr_{\mathcal{A}}(P) = \sum_{\rho \in P} Pr_{\mathcal{A}}(Cyl(\rho))$$

Again in the following we will for brevity talk about the probability of sets of finite paths.

Based on this definition we can now define the probability of reaching state set. Intuitively this is the probability of taking any path that eventually leads to a state in the set.

Let $T \subseteq S$ and $P_T = \{\rho \in Paths_{\mathcal{A}}^{fin}(s_{init}) \mid last(\rho) \in T \wedge \forall 0 \leq j < n : \rho[j] \notin T\}$ the set of all finite paths ending in a state in T and don't contain a state in T before the last state. Note that the above condition for having a well defined probability is satisfied by P_T , since all elements contain precisely one state from T and that one is the last state in the path. The probability of reaching T in \mathcal{A} is defined as a, possibly infinite, series.

$$Pr_{\mathcal{A}}(\diamond T) = \sum_{\rho \in P_T} Pr_{\mathcal{A}}(Cyl(\rho))$$

¹We require PAs to be deadlock-free, so we always get infinite paths, when "running" a DTMC.

2.2. Reachability Probabilities on Probabilistic Automata

For subsets $P \subseteq P_T$ also the cylinder sets of the contained paths are pairwise disjoint and thus the probability $Pr_{\mathcal{A}}(P)$ is defined. Other cases do not arise with the problems discussed in this thesis and are therefore not relevant here.

Reachability Probabilities on General Probabilistic Automata

For general probabilistic automata the above definition is not applicable. The probabilities in a general probabilistic automaton depend on the nondeterministic choices taken. We will therefore now define the notion of *schedulers*. A scheduler is a function that resolves the nondeterminism of a probabilistic automaton by fixing a nondeterministic choice in the last state of a finite path.

Definition 4 (Scheduler). *A scheduler for a probabilistic automaton $\mathcal{A} = (S, s_{init}, Act, \mathbf{P})$ is a function $\sigma : Paths_{\mathcal{A}}^{fin}(s_{init}) \rightarrow Act \times Dist(S)$, such that $\sigma(\rho) \in \mathbf{P}(last(\rho))$.*

Let σ be a scheduler. If for all finite paths $\rho, \rho' \in Paths_{\mathcal{A}}^{fin}(s_{init})$ $last(\rho) = last(\rho')$ implies $\sigma(\rho) = \sigma(\rho')$, i.e. the selected choice only depends on the current state, then σ is called *simple*. In this case we use the short hand notation $\sigma(s)$, denoting $\sigma(\rho)$ for some $\rho \in Paths_{\mathcal{A}}^{fin}(s_{init})$ with $last(\rho) = s$. The set of all simple schedulers for \mathcal{A} is denoted by $sched(\mathcal{A})$.

A simple scheduler σ for a probabilistic automaton $\mathcal{A} = (S, s_{init}, Act, \mathbf{P})$ induces a new probabilistic automaton $\mathcal{A}^\sigma = (S, s_{init}, Act, \mathbf{P}^\sigma)$, with $\mathbf{P}^\sigma(s) = \{\sigma(s)\}$. Intuitively \mathcal{A}^σ is obtained by removing all transitions except those chosen by σ . Note that \mathcal{A}^σ is a DTMC, since $|\mathbf{P}^\sigma(s)| = 1$ for all $s \in S$.

Example In Figure 2.3 a DTMC \mathcal{E}^σ , represented by a digraph, is depicted. The DTMC is obtained from the probabilistic automaton in Figure 2.1 by applying a simple scheduler σ that selects the transition with the action α in state 0, the transition with the action τ in state 2 and the only existing transition in the other two states.

Recall that we aim at solving questions of the form “is the probability of reaching a bad state at most λ ”. To answer that question we don’t have to consider every possible resolution of the nondeterminism. It suffices to look at the worst case, i.e. a resolution of the nondeterminism where the probability of reaching a bad state is maximal. Thus we have to take the maximal probability of reaching a bad state for all schedulers. Luckily we only have to consider simple schedulers, since there always exists a simple scheduler that achieves the maximal probabilities [Baier and Katoen, 2008].

2. Preliminaries

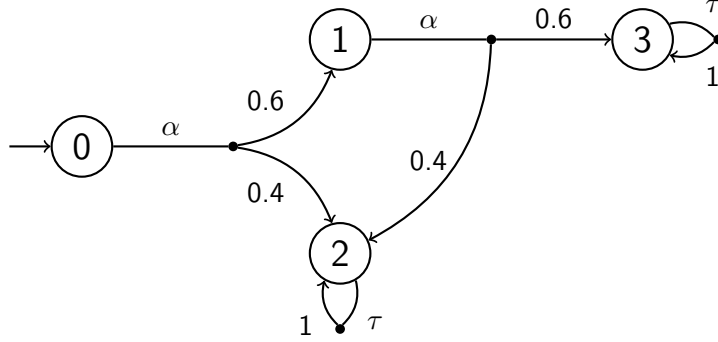


Figure 2.3.: A DTMC, \mathcal{E}^σ represented by a digraph.

Given a simple scheduler, we can apply the definition of probabilities for DTMCs on the DTMC induced by the scheduler. So we can define path probabilities in probabilistic automata with respect to a simple scheduler. Let \mathcal{A} be a probabilistic automaton and σ a simple scheduler for \mathcal{A} . The probability of a finite path² $\rho = s_0(\alpha_0, \mu_0)s_1(\alpha_1, \mu_1) \dots s_n$ in \mathcal{A} with respect to σ is denoted by $Pr_{\mathcal{A}}^\sigma(\rho) = Pr_{\mathcal{A}^\sigma}(\rho)$. Equally we denote the probability of reaching a state set $T \subseteq S$ with respect to σ by $Pr_{\mathcal{A}}^\sigma(\diamond T) = Pr_{\mathcal{A}^\sigma}(\diamond T)$ and we denote the probability of a path set R , whose elements are not prefixes of each other, by $Pr_{\mathcal{A}}^\sigma(R) = Pr_{\mathcal{A}^\sigma}(R)$.

The maximal probability of reaching a state set $T \subseteq S$ is defined as

$$Pr_{\mathcal{A}}^+(T) = \max_{\sigma \in \text{sched}(\mathcal{A})} Pr_{\mathcal{A}}^\sigma(T)$$

respectively. For a DTMC \mathcal{A} $Pr_{\mathcal{A}}^+$ is equal to $Pr_{\mathcal{A}}$, since there is only one simple scheduler of a DTMC.

If \mathcal{A} is clear from the context, it may be omitted. So we write e.g. $Pr(\rho)$ instead of $Pr_{\mathcal{A}}$ or $Pr^+(\diamond T)$ for $Pr_{\mathcal{A}}^+(\diamond T)$.

2.3. Probabilistic Programs

In the previous section we introduced probabilistic automata as a formal model for systems with both nondeterministic and probabilistic behavior. In this section we describe a probabilistic guarded command language, as defined in [Wachter, 2011] and [Wimmer et al., 2013], inspired by the input language of PRISM [Kwiatkowska et al., 2011], as a more intuitive formalism to describe probabilistic automata. The

²Actually the probability of the cylinder set of the finite path; for brevity however we speak about the probabilities of paths as stated before.

semantics of the probabilistic programs in this probabilistic guarded command language are again probabilistic automata. So we can think of the probabilistic programs as a high-level description of the formal systems introduced in the previous section.

In a model checking tool the guarded command language can be used as input language to describe the systems to be checked. The probabilistic program is then converted to a probabilistic automaton for which standard model checking techniques can be applied.

2.3.1. Probabilistic Guarded Command Language

Let Var be a finite set of Boolean variables. A *valuation* is a function $\nu : Var \rightarrow \mathbb{Z}$. The set of all valuations, that respect the domains of the variables, is denoted by \mathcal{N}_{Var} .

Definition 5 (Probabilistic Program, Command). A probabilistic program is a tuple $\mathcal{P} = (Var, \nu_{init}, Act, C)$, where Var is a finite set of integer variables with finite domain, $\nu_{init} \in \mathcal{N}_{Var}$ is the initial valuation of the variables, Act is a finite set of actions and C is a finite set of commands.

Let τ be the internal non-synchronizing action. A command $c \in C$ is of the form

$$c = [\alpha] g \rightarrow p_1 : f_1 + \dots + p_n : f_n$$

where $\alpha \in Act \dot{\cup} \{\tau\}$ is the action of the command, g is a boolean predicate over Var . g is called the guard of the command. For $1 \leq j \leq n$ $f_j : \mathcal{N}_{Var} \rightarrow \mathcal{N}_{Var}$ is an update function that assigns new values to the program variables based on the valuation of all program variables and $p_j \in [0, 1]$ is the probability of an update, such that $\sum_{j=1}^n p_j = 1$.

Intuitively we can describe the meaning probabilistic programs as follows. A state of the program is given by a valuation of its variables. The complete state space is given by \mathcal{N}_{Var} , the set of all variable valuations. The *commands* define the behavior of a program. In each execution step the program nondeterministically selects an *activated* command. We say a command is activated if the guard is true for the current valuation of the variables. After that an update of the command is selected probabilistically according to the probabilities of the updates. Then the update is applied to the current valuation of the variables to get the valuation of the successor state. This procedure is repeated indefinitely.

In the course of this thesis we assume that an update function f_j is given as an expression of the form

2. Preliminaries

$$(v_1 := E_{v_1, f_j}) \& \dots \& (v_l := E_{v_l, f_j})$$

where $v_1, \dots, v_l \in Var$ and $E_{v_1, f_j}, \dots, E_{v_l, f_j}$ are expressions over Var , evaluating to a value in the domain of the respective variable. In the following we will not distinguish between the update function and the expression.

Example Listing 2.1 shows a probabilistic program $\mathfrak{E} = (\{a, b\}, a = b = 0, Act, C)$. The command names c_1, \dots, c_4 are not part of the program but will be used later to refer to the commands individually.

Listing 2.1: A probabilistic Program \mathfrak{E}

```

a : [0,1] init 0
b : [0,1] init 0

c1 := [α] b=0 → 0.4 : (a:=0) & (b:=1) + 0.6 : (a:=1) & (b:=a)
c2 := [β] a=0 ∧ b=0 → 1 : (a:=0) & (b:=1)
c3 := [] a=0 ∧ b=1 → 0.5 : (b:=b) + 0.5 (a:=b)
c4 := [] b=1 → 1 : (a:=a)

```

By $\mathbf{C}(\mathfrak{P})$ we denote the set of commands of a program and by $\mathbf{Act}(\mathfrak{P})$ we denote the set of all actions of a program. For a command c we denote the action of the command by $\alpha(c)$ and the guard by $g(c)$.

2.3.2. Semantics of Probabilistic Programs

The semantics of a probabilistic program is a probabilistic automaton. We will give a procedure how to transform a probabilistic program into a probabilistic automaton.

The semantics of a probabilistic program $\mathfrak{P} = (Var, \nu_{init}, Act, C)$ are defined as a probabilistic automaton $\llbracket \mathfrak{P} \rrbracket = (S, s_{init}, Act', \mathbf{P})$. In the course of this thesis we will refer to $\llbracket \mathfrak{P} \rrbracket$ as *explicit model* for the program, in contrast to *abstract models*, e.g. quotients of $\llbracket \mathfrak{P} \rrbracket$, which only approximate \mathfrak{P} . We will now discuss the construction of the parts of the explicit model one by one.

A variable valuation represents the current state during the execution of a probabilistic program. Thus the states of the automaton are all possible variable valuations of the program. Formally we have $S = \mathcal{N}_{Var}$. The initial variable valuation serves as initial state $s_{init} = \nu_{init}$.

2.4. Probabilistic Reachability Properties

As action set of the automaton, we define the set of commands of the program, formally $Act' = C$. By doing so we can easily map from transitions of the PA to commands of the program. This detail will be important later.

Finally, we let the probabilistic transition relation

$$\mathbf{P}(s) = \{(c, \mu_c) \mid c \in C(\mathfrak{B}) \wedge s \models g(c)\}$$

where the distribution of a transition $(c, \mu_c) \in P(s)$ with the associated command $c = [\alpha]g \rightarrow p_1 : f_1 + \dots + p_n : f_n$ is defined such that

$$\mu_c(s') = \sum_{\{i \mid f_i(s)=s'\}} p_i$$

Intuitively, we add a transition to a state for each *activated* command, i.e. commands whose guards are true in that state. The probability of reaching the state s' is the accumulated probability of updates leading to s' from the current state.

Example In Figure 2.4 the semantics of the probabilistic program \mathfrak{E} , shown in Listing 2.1, is depicted. The actions are denoted by the name of the command assigned in Listing 2.1. Up to the naming of the states and the actions it is equivalent to the probabilistic automaton \mathcal{E} .

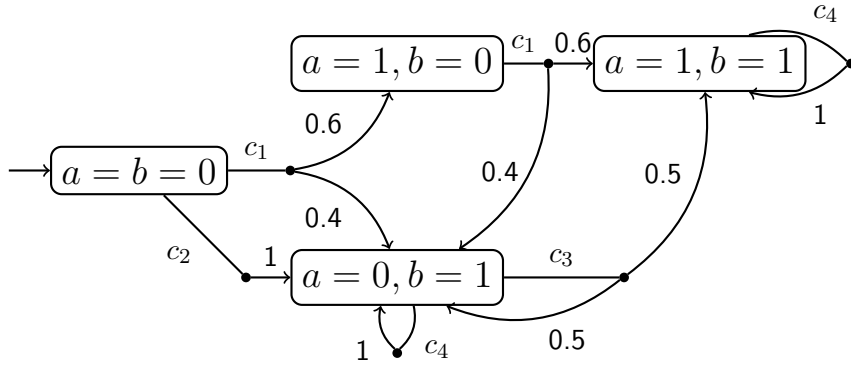


Figure 2.4.: The probabilistic automaton $\llbracket \mathfrak{E} \rrbracket$, the semantics of the probabilistic program \mathfrak{E} , represented by a digraph.

2.4. Probabilistic Reachability Properties

In model checking the question whether a certain *bad* state is reachable is very common. In the setting of probabilistic model checking we extend this to questions

2. Preliminaries

by asking for the probability of reaching a bad state. This leads to questions of the form “is the probability of reaching a bad state at most 1%”. To formally express these properties we use a notion inspired by PCTL [Baier and Katoen, 2008].

A *probabilistic reachability property*, or short reachability property, is of the form $P_{\leq\lambda}(\diamond T)$. T is an arbitrary set of states, and λ is the *threshold* or *bound*.

Let \mathcal{A} be a probabilistic automaton. The satisfaction relation for probabilistic reachability properties is defined as

$$\mathcal{A} \models P_{\leq\lambda}(\diamond T) \Leftrightarrow Pr_{\mathcal{A}}^+(\diamond T) \leq \lambda$$

So the property is satisfied if the maximal probability of reaching a state in T is less or equal to the bound λ .

In our setting we want to check properties of probabilistic programs. A *bad* state of a probabilistic program is a set of variable valuations. For example the set of all variable valuations where $a = 1$ for the program given in Listing 2.1.

A probabilistic reachability property for a probabilistic programs is of the form $P_{\leq\lambda}(\diamond e)$, where e is a boolean predicate over variables $Var(e)$. Given a program $\mathfrak{P} = (Var, \nu_{init}, Act, C)$ with $Var(e) \subseteq Var$ the satisfaction relation is given by the semantics of the program. The predicate e corresponds to the subset of the states of $\llbracket \mathfrak{P} \rrbracket$ that satisfy e .³

$$\mathfrak{P} \models P_{\leq\lambda}(\diamond e) \Leftrightarrow \llbracket \mathfrak{P} \rrbracket \models P_{\leq\lambda}(\diamond \{\nu \in \mathcal{N}_{Var} \mid \nu \models e\})$$

In the following we will not distinguish between a boolean predicate over the program variables and the set of states satisfying the predicate. So we may write $\llbracket \mathfrak{P} \rrbracket \models P_{\leq\lambda}(\diamond e)$ as a shorthand for $\llbracket \mathfrak{P} \rrbracket \models P_{\leq\lambda}(\diamond \{\nu \in \mathcal{N}_{Var} \mid \nu \models e\})$.

³States in the semantics of a program are valuations of the program variables.

3. Counterexamples

Counterexamples play an important role in model checking. In case a property is violated by a system, counterexamples are used as proof for that fact and help debugging the system. Depending on the application different types of counterexamples may be used.

Path set counterexamples show the violation of a property on the level of probabilistic automata. They show in a straight forward way how the system reaches a bad state. Additionally they play an important role in our variant of CEGAR, which is presented in the next chapter. However the number of paths in a counterexample may be arbitrarily large, which makes them hard to understand for humans.

High-level counterexamples operate on the level of programs in PRISM's guarded command language. In this chapter we will look at *critical command sets*, a type of high-level counterexample that points out a part of a program that already violates the property.

3.1. Path Set Counterexamples

For reachability properties paths or sets of paths are a common type of counterexamples. If an initial finite path to some target state exists, the target state is reachable with at least the probability of that path. Given a set of initial finite paths to some set of target states, the probability of reaching that set is at least the accumulated probability of these paths, according to the definition of reachability probabilities in Chapter 2.

Let $\mathcal{A} = (S, s_{init}, Act, \mathbf{P})$ be a probabilistic automaton and $P_{\leq \lambda}(\diamond T)$ a probabilistic reachability property that is violated by \mathcal{A} .

Definition 6 (Path Set Counterexample). *A path set counterexample for $P_{\leq \lambda}(\diamond T)$ in the PA \mathcal{A} is a set $P \subseteq Paths_{\mathcal{A}\sigma}^{fin}(s_{init})$ for some simple scheduler σ , such that for all $\rho \in P$ $last(\rho) \in T$ and for all $0 \leq j < n$ holds $\rho[j] \notin T$. It is called finite if the set P contains finitely many elements.*

According to Han and Katoen there always exists a finite path set counterexample if \mathcal{A} is a DTMC. On a DTMC such a finite path set counterexample can be

3. Counterexamples

computed using graph algorithms [Han and Katoen, 2007]. For general probabilistic automata it can easily be shown that there also is always a finite counterexample. Let σ be an arbitrary simple scheduler, such that $P_{\mathcal{A}^\sigma} > \lambda$. By applying the theorem to \mathcal{A}^σ we get that there is a finite counterexample. Also, given such a scheduler, we can compute a path set counterexample on \mathcal{A}^σ as for any other DTMC.

However in general we can only guarantee that a finite path set counterexample exists. There are probabilistic automata and properties, such that for any number $n \in \mathbb{N}$ there is a $\lambda \in [0, 1]$ such that the smallest path set counterexample contains at least n paths.

3.2. Smallest Critical Command Set

If parts of a system already violate a probabilistic reachability property, the whole system also violates the property. High-level counterexamples follow this approach, but in contrast to path set counterexamples operate on the level of probabilistic programs. High-level counterexamples are meant to be easily understandable and also helpful for people when debugging probabilistic programs.

Let $\mathcal{P} = (Var, \nu_{init}, Act, C)$ be a probabilistic program and $P_{\leq \lambda}(\diamond T)$ a reachability property.

Definition 7 (Smallest Critical Command Set). *A critical command set is set $\mathcal{C} \subseteq C$ such that $\llbracket \mathfrak{P} \rrbracket_{|\mathcal{C}|} \not\models P_{\leq \lambda}(\diamond T)$, with $\mathfrak{P}_{|\mathcal{C}|} = (Var, \nu_{init}, Act, \mathcal{C})$.*

A smallest critical command set is a critical command set \mathcal{C} such that for all critical command sets \mathcal{C}' $|\mathcal{C}| \leq |\mathcal{C}'|$.

A smallest critical command set is a high-level counterexample. Intuitively, it is a set of commands of a probabilistic program such that a target state is reachable, with probability greater than λ in the sub-PA induced by \mathcal{C} .

3.3. Computing Smallest Critical Command Sets

Computing smallest critical command sets is NP-hard and can be solved using mixed integer linear programming [Wimmer et al., 2013]. In this section we sketch the idea for the mixed integer linear program (MILP) used to compute the smallest critical command set.

Let $\mathfrak{P} = (Var, \nu_{init}, Act, C)$ be a probabilistic program in guarded command language and $P_{\leq \lambda}(\diamond e)$ a reachability property. $\llbracket \mathfrak{P} \rrbracket = (S, s_{init}, Act', \mathbf{P})$ is the probabilistic automaton defined by the program.

3.3. Computing Smallest Critical Command Sets

The MILP to compute the smallest command set works by encoding a scheduler for $\llbracket \mathfrak{P} \rrbracket$ and the commands that generated the selected transitions in variables of the MILP. The objective function then minimizes the number of selected commands. For each transition at each state a boolean variable in the MILP encodes whether the scheduler selects this transition and a constraint for each state ensures that at most one transition is chosen at a state. It is ensured that in the MILP the boolean variable for the command is 1 if at least one action generated by that command is taken. Additional constraints are used to ensure that the probability of reaching a target state in $\llbracket \mathfrak{P} \rrbracket$ from the initial state is greater than λ . As we want to minimize the number of commands, the objective function is the sum of the boolean variables encoding whether a command is selected.

4. Counterexample Guided Abstraction Refinement

Checking properties of programs defined in guarded command language is possible by constructing a PA from the program as described in Chapter 2 and then applying standard techniques like value- or policy-iteration [Bellman, 1957] [Puterman, 1994] to this *explicit model*. However the state space of these systems can be very large. This leads to problems as the system might not fit into the available memory, and the computation time increases dramatically.

Counterexample Guided Abstraction Refinement [Hermanns et al., 2008] circumvents this problem by checking properties on abstractions, i.e. a quotient automaton of the explicit model in our case, and refining this abstraction until a result for the explicit model can be inferred without actually building it.

In the first Section of this Chapter we given an overview over the algorithm and sketch the main steps. In the following Sections we then go over all main steps and explain them in detail. In the last Section we show that the abstractions used by CEGAR overapproximate the program in a sense that realizable paths, we will define realizability later, in the abstract model have counterparts in the explicit model. This is an important property for the correctness of the algorithm itself and the main reasons that optimality is preserved, when CEGAR is used to generate high-level counterexamples.

4.1. Overview CEGAR Algorithm

The *Counterexample Guided Abstraction Refinement* algorithm described in this thesis is an iterative algorithm. Figure 4.1 gives an overview over the algorithm.

The algorithm works on a program in guarded command language $\mathfrak{P} = (Var, \nu_{init}, Act, C)$ and a property of the form $\psi = P_{\leq \lambda}(\diamond e)$ where e is a boolean expression over the variables of the program. Additionally, it uses a set of predicates Φ which initially only contains the expression e from the property to be checked. This predicate set induces the used abstraction and is extended as the abstraction is refined. In each iteration a quotient of the PA $\llbracket \mathfrak{P} \rrbracket$ is considered as abstraction. The abstract system is generated by computing a quotient with respect to an equivalence relation \equiv_{Φ} induced by Φ . After that the property ψ is checked

4. Counterexample Guided Abstraction Refinement

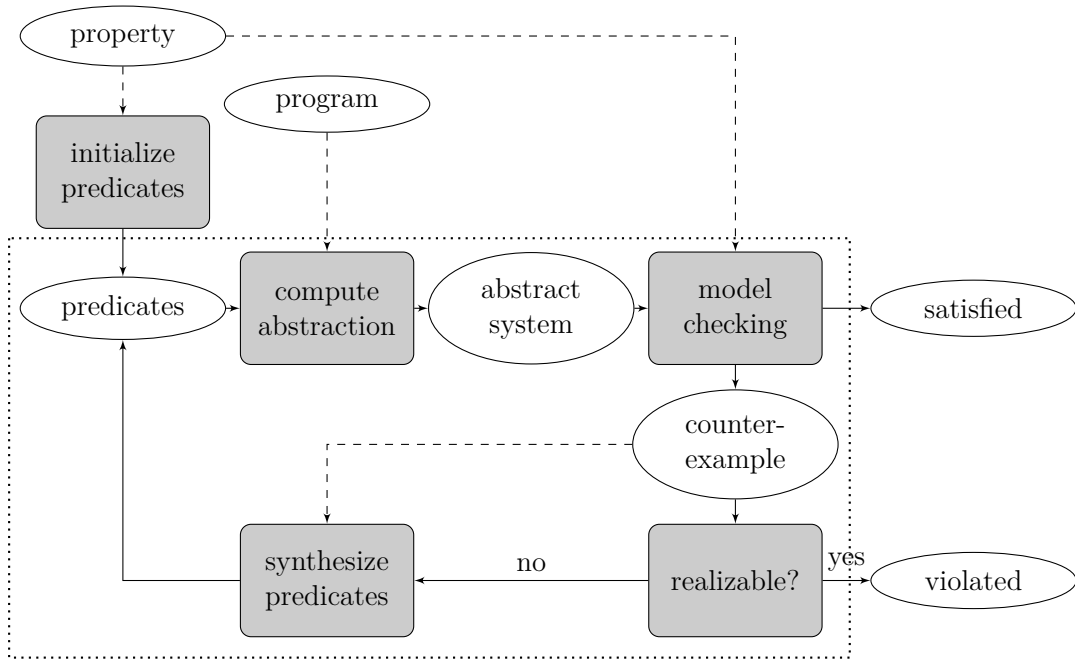


Figure 4.1.: Flowchart of CEGAR algorithm.

on the abstraction. If the property is fulfilled it is also fulfilled in the program \mathfrak{P} , since a quotient simulates the original automaton and simulation preserves satisfaction¹ of probabilistic reachability properties [Wachter et al., 2007]. Otherwise a counterexample is computed and checked for realizability. If the counterexample is *realizable* that means that the property ψ does not hold on the original system $\llbracket \mathfrak{P} \rrbracket$ and the realization of the counterexample is returned. Otherwise, if the counterexample is not realizable, we say the counterexample is *spurious*. Then new predicates are computed from the counterexample and added to Φ . The next iteration uses this larger set of predicates and therefore uses a finer abstraction.

In summary the CEGAR algorithm consists a main loop and an initialization step where Φ is assigned its initial value. The main loop of CEGAR consists of the three main steps:

- (1) Build abstract PA
- (2) Model check abstract PA
- (3) Analyze counterexample
 - a) Check realizability

¹but not violation

b) Synthesize predicates

Note that step (3) is only executed if model checking in step (2) found the property violated. Otherwise the algorithm terminates in step (2). Also step (3) b) is only executed if the counterexample is not realizable, otherwise the algorithm stops after step (3) a). In the following we will look all three steps in more detail.

4.2. Program Abstraction

The predicate set Φ induces an equivalence relation \equiv_Φ on the states of $\llbracket \mathfrak{P} \rrbracket$. Two states are related if and only if they satisfy the same predicates from Φ . This is obviously reflexive, symmetric and transitive. The equivalence class of a state s under the relation \equiv_Φ is denoted by $[s]_\Phi$ as a shorthand for $[s]_{\equiv_\Phi}$. The quotient of the PA $\llbracket \mathfrak{P} \rrbracket$ with respect to \equiv_Φ is denoted by $\llbracket \mathfrak{P} \rrbracket_\Phi$ as a shorthand for $\llbracket \mathfrak{P} \rrbracket_{/\equiv_\Phi}$. Recall that we refer to the states of a quotient, i.e. the equivalence classes, as blocks to distinguish them from states of the explicit model.

This quotient can be computed directly from the program \mathfrak{P} by enumerating models of SMT formulas [Wachter, 2011], as we will describe in this section. Computing $\llbracket \mathfrak{P} \rrbracket_\Phi$ directly from the program \mathfrak{P} without building $\llbracket \mathfrak{P} \rrbracket$ is essential, as one main purpose of CEGAR is to overcome limitations in the size of $\llbracket \mathfrak{P} \rrbracket$.

The command set of $\llbracket \mathfrak{P} \rrbracket_\Phi$ is the command set of \mathfrak{P} according to the definitions quotient automata and semantics of programs in Chapter 2. According to our definition of the quotient, a block, i.e. state, is an equivalence class of the relation \equiv_Φ . In the following, we assume an arbitrary but fixed ordering of the predicates in Φ is given. Therefore we may write $\Phi = \{\varphi_0, \dots, \varphi_{n-1}\}$, where $n = |\Phi|$. An equivalence class of \equiv_Φ , i.e. a block of $\llbracket \mathfrak{P} \rrbracket_\Phi$, then can be written as a bit-vector B of length n , representing all valuations of the program variables of \mathfrak{P} , i.e. states of $\llbracket \mathfrak{P} \rrbracket$, that satisfy the predicate φ_i iff $B[i] = 1$. In the following we will use bit-vectors and equivalence classes interchangeably. The state set of $\llbracket \mathfrak{P} \rrbracket_\Phi$ then can be seen as the set of all bit-vector of length n .

In the remainder of this section, we describe how to compute the initial state and the transition relation of $\llbracket \mathfrak{P} \rrbracket_\Phi$.

We now introduce the *block constraint* \mathcal{B}_Φ . For a bit-vector B and a variable valuation ν the block constraint is consistent if and only if the state ν is in the block B . We let

$$\mathcal{B}_\Phi = \bigwedge_{i=0}^k \varphi_i \leftrightarrow B[i] \quad (4.1)$$

4. Counterexample Guided Abstraction Refinement

where $B[i]$ are the entries of a bit-vector B and $\varphi_i \in \Phi$. This constraint forces that precisely those predicates hold that hold for states in B . For an arbitrary boolean predicate Ψ over Var , the conjunction $\Psi \wedge \mathcal{B}_\Phi$ is consistent for blocks B , such that there exists a state in B that satisfies Ψ . By letting B be a variable in \mathcal{B}_Ψ , we can enumerate blocks containing a state satisfying Ψ by enumerating models² of $\Psi \wedge \mathcal{B}_\Phi$. Using this method we now construct formulas to find the initial block and enumerate transitions.

Let I be a formula encoding the initial valuation of the program variables. A model of I is the initial state of $\llbracket \mathfrak{P} \rrbracket$ and thus a model of the formula $I \wedge \mathcal{B}_\Phi$ encodes the initial state of $\llbracket \mathfrak{P} \rrbracket_\Phi$ in the valuation of B .

To compute the transition relation of $\llbracket \mathfrak{P} \rrbracket_\Phi$ we compute the transitions generated by each command in isolation. A command $c = [\alpha] g \rightarrow p_1 : f_1 + \dots + p_n : f_n$ is *activated* in a block B if and only if its guard $g(c)$ is satisfied for some state s in B . Enumerating these blocks can be done by constructing a formula as described above. Let

$$g(c) \wedge \mathcal{B}_\Phi \tag{4.2}$$

where $g(c)$ is the guard of the command c . The distinct valuations B for models of Formula (4.2) are the blocks that have the command c activated.

Recall that the updates f_1, \dots, f_k of command c are given by expressions $(v_1 := E_{v_1, f_j}) \& \dots \& (v_l := E_{v_l, f_j})$, where v_1, \dots, v_l are the variables of \mathfrak{P} and E_{v_i, f_j} is an expression over the variables of \mathfrak{P} evaluating to the new value assigned to the variable v_i . By extending Formula (4.2) we can retrieve the blocks of $\llbracket \mathfrak{P} \rrbracket_\Phi$ that are reachable with the updates of c for a transition defined by c . Let

$$g(c) \wedge \mathcal{B}_\Phi[b/b_0] \wedge \left(\bigwedge_{1 \leq j \leq k} \mathcal{B}_\Phi[X/E_{f_j}][Var/Var_i][b/b_k] \right) \tag{4.3}$$

where $\mathcal{B}_\Phi[Var/E_{f_j}][Var/Var_i]$ denotes that each occurrence of a variable $v \in Var$ is replaced by E_{v, f_j} and after that each occurrence of a variable $v \in Var$ is replaced by a copy $v_i \in Var_i$ and Var_i is a copy of the program variables.

A model for this formula can be split into $k + 1$ valuations of the program variables ν, ν_1, \dots, ν_k , where ν is the valuation of Var , ν_1 the valuation of Var_1 and so on, and $k + 1$ blocks B, B_1, \dots, B_k . ν corresponds to a state in the block B such that c is activated in ν . Further $\nu_1 = f_1(\nu)$, i.e. the state reached from ν with the update f_1 , and ν_1 is in the block B_1 . Therefore the block B_1 is reachable with the update f_1 from block B . The same arguments can be applied to all other

²In this thesis a model of a formula is a valuation of its variables under which the formula evaluates to *true*.

4.3. Model Checking the Abstraction and Computing a Counterexample

updates. So, from a model of Formula (4.3) we can infer that there is a transition $(c, \mu) \in \mathbf{P}_\Phi(B)$ with

$$\mu(B') = \sum_{\{k|B_k=B'\}} p_k$$

By enumerating all models of Formula (4.3) with distinct B, B_1, \dots, B_k we can thus infer all transitions generated by c . By iterating over all commands, we can construct the complete transition relation of the quotient $\llbracket \mathfrak{P} \rrbracket_\Phi$.

4.3. Model Checking the Abstraction and Computing a Counterexample

In this step the property ψ is checked on the abstract PA $\llbracket \mathfrak{P} \rrbracket_\Phi$ using standard techniques like value- or policy-iteration [Bellman, 1957] [Puterman, 1994]. If ψ holds on $\llbracket \mathfrak{P} \rrbracket_\Phi$ then it also holds on $\llbracket \mathfrak{P} \rrbracket$, since $\llbracket \mathfrak{P} \rrbracket_\Phi$ simulates $\llbracket \mathfrak{P} \rrbracket$. Therefore the property ψ also holds on the program \mathfrak{P} . The algorithm terminates with the result “ $\mathfrak{P} \models \psi$ ”. Otherwise, if the property is not fulfilled by $\llbracket \mathfrak{P} \rrbracket_\Phi$ a path set counterexample, as defined in Chapter 3 is computed.

We require the used model checking algorithm to provide a simple scheduler σ such that $\llbracket \mathfrak{P} \rrbracket_\Phi^\sigma$ violates ψ , in case the property is violated. A finite path set counterexample Π can be computed in $\llbracket \mathfrak{P} \rrbracket_\Phi^\sigma$ using a *k-shortest paths* algorithm [Han and Katoen, 2007]. The *recursive enumeration algorithm* [Jiménez and Marzal, 1999] suites well for this task, as the number of paths does not have to be known in advance. Instead, new paths can be computed on the fly as needed by the counterexample analysis.

4.4. Counterexample Analysis

In this step the path set counterexample Π computed in the previous step is tested for realizability. If the counterexample is not realizable, new predicates are computed and the loop starts over. If the counterexample is realizable the property is not fulfilled in $\llbracket \mathfrak{P} \rrbracket$ and the algorithm terminates with the result “ $\mathfrak{P} \not\models \psi$ ”.

In this step we have to extend the definition of probabilistic automata to *update labeled probabilistic automata*. In a nutshell an update labeled probabilistic automaton is a probabilistic automaton where the distributions have additional *update labels*. A path in the update labeled probabilistic automaton $\llbracket \mathfrak{P} \rrbracket'_\Phi$ is of the form $s_0(c_0, \mu_0, f_0)s_1(c_1, \mu_1, f_1) \dots$, a finite path is of the form $s_0(c_0, \mu_0, f_0)s_1(c_1, \mu_1,$

4. Counterexample Guided Abstraction Refinement

$f_1) \dots s_k$. The updates on a paths in $\llbracket \mathfrak{P} \rrbracket_\Phi$ are updates of the command that generated the transition, e.g. f_0 in the previously defined path is an update of the command c_0 . The probability of such a path³ is $\prod_{i=1}^k p_i$ where p_i is the probability of update f_i in the command c_i . Since multiplication distributes over addition the probability of all paths that only differ in the update labels adds up to the corresponding path in $\llbracket \mathfrak{P} \rrbracket_\Phi$. In the following, we will use $\llbracket \mathfrak{P} \rrbracket_\Phi$ and $\llbracket \mathfrak{P} \rrbracket'_\Phi$ interchangeably.

4.4.1. Checking Realizability

Before we describe how to check a counterexample for realizability we define what realizability means in this context.

Let $\pi = B_0(c_0, \mu_0, f_0)B_1(c_1, \mu_1, f_1) \dots B_k$ be a finite path in the (update labeled) probabilistic automaton $\llbracket \mathfrak{P} \rrbracket_\Phi$. In the following we refer to π as *abstract path*. We say π is *realizable*, if there is a finite path $\gamma(\pi) = s_0(c_0, \mu'_0)s_1(c_1, \mu'_1) \dots s_k$ in $\llbracket \mathfrak{P} \rrbracket$, such that $[s_i] = B_i$ for $0 \leq i \leq k$, $s_{i+1} = f_i(s_i)$ for $0 \leq i < k$. $\gamma(\pi)$ is called a *realization of π* . Observe that the probabilities for each step in the realization are at least as large as the probabilities in the abstract path. If no realization exists, the path is called *spurious*.

A path set counterexample Π on $\llbracket \mathfrak{P} \rrbracket_\Phi$ for the property ψ is called *realizable*, if there exists $\Pi' \subseteq \Pi$ such that all paths in Π' are realizable and $\sum_{\pi \in \Pi'} Pr(\pi) > \lambda$. In this case $\gamma(\Pi') = \{\gamma(\pi) \mid \pi \in \Pi'\}$ is a counterexample for ψ in $\llbracket \mathfrak{P} \rrbracket$ [Hermanns et al., 2008]. Otherwise Π is called *spurious*.

To test if a counterexample is realizable, we iterate over the paths $\pi \in \Pi$ and check each path in isolation. If the path is realizable, we add $Pr(\pi)$ to the *realizable probability* Pr_r . If the realizable probability exceeds λ , we found a realizable counterexample. If π is spurious, we synthesize new predicates from π , as described in the next subsection, and add $Pr(\pi)$ to the *spurious probability* Pr_s . Once we have $\sum_{\pi \in \Pi} Pr(\pi) - Pr_s \leq \lambda$ the bound λ can not be achieved and thus the counterexample is reported as spurious.

In practice we terminate an iteration of the CEGAR loop even earlier. Once a fixed number of spurious paths is exceeded, the loop is restarted. The idea behind this is to avoid checking high numbers of paths, once some progress is guaranteed by the predicates generated from the first few spurious paths.

In practice Π is not computed in advance, as we may restart the loop early and don't always have to consider all paths in Π . Instead new paths from Π are computed as needed.

³more precisely its cylinder set

4.4. Counterexample Analysis

Now we will go over the process of checking a fixed path $\pi = B_0(c_0, \mu_0, f_0)B_1(c_1, \mu_1, f_1) \dots B_k \in \Pi$ for realizability. To check if π is realizable or spurious, we have to check the existence of a realization $\gamma(\pi) = s_0(c_0, \mu'_0)s_1(c_1, \mu'_1) \dots s_k$.

The realization of the states B_i can be encoded in a formula by instances of the program variables Var_i . The valuation of these variables uniquely identify a state of $\llbracket \mathfrak{P} \rrbracket$. To verify that the valuation of Var_i actually encodes a state in B_i we use the block formula

$$\mathcal{F}(B_i) := \left(\bigwedge_{\varphi \in \Phi_{B_i}^+} \varphi \right) \wedge \left(\bigwedge_{\varphi \in \Phi_{B_i}^-} \neg \varphi \right) \quad (4.4)$$

where $\Phi_{B_i}^+$ is the set of predicates that hold in B_i and $\Phi_{B_i}^-$ the set of the predicates that don't hold in B_i . This formula is true for all states in B_i and no others. $\mathcal{F}(B_i)[Var/Var_i]$ denotes the block formula of B_i where all occurrences of the program variables are replaced by their copies from Var_i .

Next we look at a single step $B_i(c_i, \mu_i, f_i)B_{i+1}$ of π . We construct the *step formula* $\mathcal{R}_{c_i, f_i}(Var_i, Var_{i+1})$. The step formula is true if and only if there is a transition generated by the command c_i from the state s_i , encoded by Var_i , to the state s_{i+1} , encoded by Var_{i+1} , and the update f_i leads from s_i to s_{i+1} .

$$\mathcal{R}_{c_i, f_i}(Var_i, Var_{i+1}) := g(c_i)[Var/Var_i] \wedge (Var_{i+1} = f_i[Var/Var_i]) \quad (4.5)$$

where $g(c)[Var/Var_i]$ denotes the guard of c and $f_i[Var/Var_i]$ denotes the update function each with the occurrences of a program variable replaced by its copy in Var_i .

Finally we have to ensure that the realization of the path π starts in the initial state of $\llbracket \mathfrak{P} \rrbracket$. This is done by asserting that the program variables are equal to their initial values.

$$\mathcal{I} := \left(\bigwedge_{x \in Var} x = \nu_{init}(x) \right) \quad (4.6)$$

We now construct the *path formula* $PF(\pi)$ by asserting the block formula and step formula for each state and transition along the path π on the respective copies Var_i of the program variables. Additionally we assert the initial condition \mathcal{I} on the first copy of the program variables.

$$PF(\pi) := \mathcal{I}[Var/Var_0] \wedge \left(\bigwedge_{i=0}^k \mathcal{F}(B_i)[Var/Var_i] \right) \wedge \left(\bigwedge_{i=0}^{k-1} \mathcal{R}_{c_i, f_i}(Var_i, Var_{i+1}) \right) \quad (4.7)$$

If this formula is consistent the path π is realizable, otherwise it is spurious [Hermanns et al., 2008].

4.4.2. Generating New Predicates

Let $\pi \in \Pi$ be a spurious path from the path set counterexample. We split it into two parts π_j^- and π_j^+ such that $last(\pi_j^-) = \pi_j^+[0] = B_j$ where $0 \leq j < n$. If for none of the states B_j that are reachable with a realization of π_j^- a realization of π_j^+ starting in the same state exists, then we call j a cut point. In that case we have to split B_j to eliminate the spurious path π . In general there can be multiple cut points on a single spurious path [Wachter, 2011]. Suppose for a spurious path π the prefix of length j is realizable and the postfix of length $k - j$ is realizable, then we have to split block B_j since j is a cutpoint. If furthermore the prefix of length j' and the postfix of length $k - j'$ are realizable, we have another cutpoint. The intuition for this is that we have two “realization candidates” that break in different blocks.

Wachter describes the following procedure to generate predicates for a cutpoint. To generate a predicate from the cutpoint j the path formula $PF(\pi)$ is split into two parts ψ_j^- and ψ_j^+ , representing π_j^- and π_j^+ .

$$\psi^- := \mathcal{I}[Var/Var_0] \wedge \left(\bigwedge_{0 \leq i \leq j} \mathcal{F}(B_i)[Var/Var_i] \right) \wedge \left(\bigwedge_{0 \leq i < j} \mathcal{R}_{c,f_i}(Var_i, Var_{i+1}) \right) \quad (4.8)$$

$$\psi^+ := \left(\bigwedge_{j < i \leq k} \mathcal{F}(B)[Var/Var_i] \right) \wedge \left(\bigwedge_{j \leq i < k} \mathcal{R}_{c_i,f_i}(Var_i, Var_{i+1}) \right) \quad (4.9)$$

To generate a predicate, interpolation is used. The interpolant φ of ψ_j^- and ψ_j^+ is a formula over the common variables of ψ_j^- and ψ_j^+ such that

- ψ_j^- implies φ
- $\varphi \wedge \psi_j^+$ is inconsistent

An interpolant always exists if $\psi_j^- \wedge \psi_j^+$ is inconsistent, which is guaranteed because the trace formula is inconsistent for spurious paths. As the interpolant only contains common variables of ψ_j^- and ψ_j^+ , it only contains one copy of the program variables, namely Var_j . Thus φ can be evaluated for single states and can be used as new predicate to eliminate a cut points. By adding the interpolants for all cut points to the predicate set Φ the elimination of the spurious path π is ensured.

This is however not sufficient in all cases. Suppose there exists an index h such that ψ_h^- and ψ_h^+ are inconsistent. This means that from all of the states in the block B_0 no state in B_h is reachable and from all of the states in block B_h no state in B_k is reachable. Then for all $j \leq h$ the formula ψ_j^+ is inconsistent as it is a conjunction with the inconsistent formula ψ_h^+ . In this case *true* is a valid interpolant, since

all of its variables are contained in ψ_j^- and ψ_j^+ , *true* is implied by any formula and the conjunction of *true* and an inconsistent formula is inconsistent. By the same arguments for all $j \geq h$ the formula ψ_j^- is inconsistent. Then *false* is a valid interpolant, since inconsistent formulas imply *false*, *false* only contains common variables from ψ_j^- and ψ_j^+ and a conjunction with *false* is inconsistent. So in this case we only obtain *true* and *false* as predicates from a spurious path. As neither of them splits blocks in the abstract model, the spurious path is not eliminated.

We now provide an example that shows that this case can actually arise in practice. After that, we describe a method to generate predicates that are different from *true* and *false*.

Example We will show the computations for the program given in Listing 4.1 and the property $P_{\leq 0.5}(\diamond s = 3)$.

Listing 4.1: Counterexample showing that interpolating the path formula is not always sufficient.

```

s: [0..3] init 0;
f: [0,1] init 0;

[] s=1 → 1: (s := 2) & (f := 1)
[] s=2 ∧ f=0 → 1: (s := 3)
[] true → 1: (s := s)

```

For the first iteration we initialize the predicate set as $\{(s = 3)\}$. This leads to an abstract system depicted in Figure 4.2. The probability for eventually reaching a state where $(s = 3)$ is obviously 1 in the abstract system. A scheduler is generated that selects the transition for the initial state that leads to the other state. The only path π leading to the final state in the resulting Markov chain is obviously not realizable as the guard of the command is not true for the initial state of the program. Since the length of π is 1, there is only one pair of ψ_j^- and ψ_j^+ formulas.

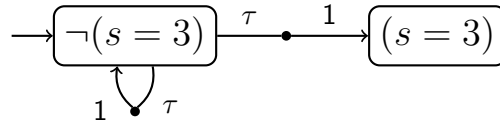


Figure 4.2.: Abstract system in first iteration. (Self loops generated by third command omitted)

These are

$$\begin{aligned} \psi_0^- &= ((s_0 = 0) \wedge (f_0 = 0)) \wedge (\neg(s_0 = 3)) \\ \psi_0^+ &= ((s_0 = 2 \wedge f_0 = 0) \wedge (s_1 = 3 \wedge f_1 = f_0)) \wedge (s_1 = 3) \end{aligned}$$

4. Counterexample Guided Abstraction Refinement

An interpolant for ψ_0^- and ψ_0^+ is $\neg(s_0 = 2)$, so we add $\neg(s = 2)$ as new predicate and start a new iteration, as paths with the probability mass 1 are not realizable and thus the bound of 0.5 can not be achieved.

For the next iteration we obtain the system depicted in Figure 4.3. In this system again the probability for eventually reaching a state where $(s = 3)$ is 1. A scheduler that achieves that probability has to select the two transitions depicted in the figure. Again only one path exists that eventually reaches a target state,

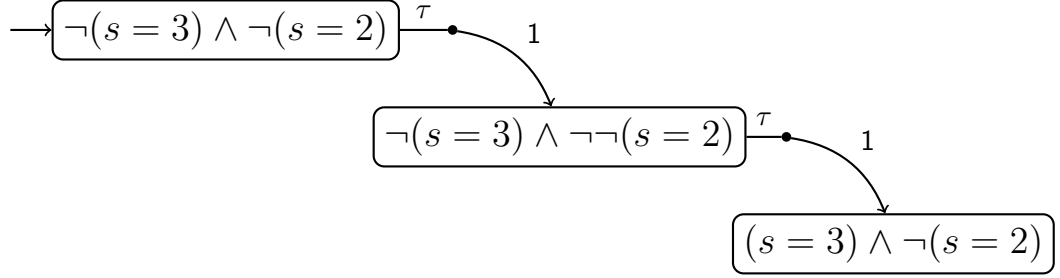


Figure 4.3.: Abstract system in second iteration. (Self loops generated by third command omitted)

this time of length 2. The following formulas are considered to generate predicates.

$$\begin{aligned}\psi_0^- &= ((s_0 = 0) \wedge (f_0 = 0)) \wedge (\neg(s = 3) \wedge \neg(s = 2)) \\ \psi_0^+ &= ((s_0 = 1) \wedge (s_1 = 2 \wedge f_1 = 1)) \wedge (\neg(s_1 = 3) \wedge \neg\neg(s_1 = 2)) \\ &\quad \wedge ((s_1 = 2 \wedge f_1 = 0) \wedge (s_2 = 3 \wedge f_2 = f_1)) \wedge ((s_2 = 3))\end{aligned}$$

Here the formula ψ_0^+ is inconsistent, since the clauses $(f_1 = 1)$ and $(f_1 = 0)$ are contained in the conjunction. Therefore *true* is a valid interpolant for ψ_0^- and ψ_0^+ .

$$\begin{aligned}\psi_1^- &= ((s_0 = 0) \wedge (f_0 = 0)) \wedge (\neg(s = 3) \wedge \neg(s = 2)) \\ &\quad \wedge ((s_0 = 1) \wedge (s_1 = 2 \wedge f_1 = 1)) \wedge (\neg(s_1 = 3) \wedge \neg\neg(s_1 = 2)) \\ \psi_1^+ &= ((s_1 = 2 \wedge f_1 = 0) \wedge (s_2 = 3 \wedge f_2 = f_1)) \wedge ((s_2 = 3))\end{aligned}$$

Here, the formula ψ_1^- is inconsistent, since the clauses $(s_0 = 0)$ and $(s_0 = 1)$ are contained in the conjunction. Therefore *false* is a valid interpolant for ψ_1^- and ψ_1^+ .

With the single path all not being realizable the probability bound of 0.5 can not be achieved any more. So the current iteration terminates and a new iteration is started. But we have not derived any new predicates that could split a block of the abstract PA in Figure 4.3. So in the next iteration an equivalent abstract system is built and the same path is selected as counterexample for the refinement. Thus, no progress is made.

Generating New Predicates Revised A path of length one is always realizable, unless we require that the path is an initial path. This is guaranteed by the construction of the quotient. For every transition in the quotient there exists a witness in the original model. The prefix formula ψ_j^- is a complete path formula for the prefix. Therefore the prefix of length j is realizable if and only if ψ_j^- is satisfiable.

To guarantee the generation of predicates different from *true* and *false* we select the largest index j such that ψ_j^- is satisfiable. We then set ψ_{j+1}^- as new path formula $PF(h)_{|j+1}$ and apply the interpolation algorithm described previously to this formula. With this formula we have that for the index j the prefix formula ψ_j^- and the postfix formula ψ_j^+ in isolation are satisfiable. Therefore, neither *true* nor *false* are valid interpolants. But the conjunction is inconsistent, so an interpolant is guaranteed to exist.

4.5. Simulation and Path Lifting

In this section we will show that all realizable paths in the abstract PA $\llbracket \mathfrak{P} \rrbracket_{\Phi}$ have an exact counterpart in $\llbracket \mathfrak{P} \rrbracket$. We will later need this fact to show that optimality is preserved when CEGAR is used to compute high-level counterexamples.

Lemma 1. *Let \mathfrak{P} be a program and $\llbracket \mathfrak{P} \rrbracket_{\Phi}$ its quotient as computed in the program abstraction step of CEGAR. For all finite paths $s_0(c_0, \mu_0)s_1(c_1, \mu_1) \dots s_k \in Paths_{\llbracket \mathfrak{P} \rrbracket}^{fin}(s_{init})$ there exists a path $[s_0](c_0, \mu'_0)[s_1](c_1, \mu'_1) \dots [s_k]$ in $Paths_{\llbracket \mathfrak{P} \rrbracket_{\Phi}}^{fin}([s_{init}])$, such that $\mu_i(s_{i+1}) \leq \mu'_i([s_{i+1}])$ for $0 \leq i < k$.*

The relation \equiv_{Φ} is an equivalence relation and the PA $\llbracket \mathfrak{P} \rrbracket_{\Phi}$ computed by the program abstraction step is the quotient of $\llbracket \mathfrak{P} \rrbracket$ with respect to \equiv_{Φ} . As noted in Section 2.1 a quotient simulates the original PA. Thus we have that $s_1 \in succ_{\llbracket \mathfrak{P} \rrbracket_{\Phi}}(s_0, c_0, \mu_0)$ implies $[s_1] \in succ_{\llbracket \mathfrak{P} \rrbracket}([s_0], c_0, \mu'_0)$ for some $\mu'_0 \in \mathbf{P}_{\Phi}$ such that $\mu_0(s_0) \leq \mu'_0([s_0])$. By successively applying this fact on each step in the path, we get can derive that the lemma holds.

5. Computing High-Level Counterexamples on Abstractions

In the previous two chapters we looked at counterexample guided abstraction refinement and high-level counterexamples. Counterexample guided abstraction refinement, described in Chapter 4, is a technique to speed up model checking and overcome limitations in the size of the models. It works by checking an abstraction which is iteratively refined until satisfaction of the property in question can be decided. High-level counterexamples are counterexamples on the level of probabilistic programs, designed to be easily understandable. Critical command sets, described in Chapter 3, are an example of such high-level counterexamples. Intuitively speaking they point out the commands that make a program violating the property in question. However computing smallest critical command sets is computationally hard.

In this chapter we will first show that smallest critical command sets can be computed on the abstraction used by counterexample guided abstraction refinement. By computing the smallest critical command set on an abstraction, with less states than the original system, we can potentially speed up the computation. After the theoretical consideration we provide experimental results of a prototype implementation of the proposed concept.

5.1. CEGAR for Computing Smallest Critical Command Sets

In this section we describe how the CEGAR algorithm from Chapter 4, in this section referred to as basic algorithm, can be adapted to compute smallest critical command sets. We will again go through the CEGAR algorithm composed of an initialization and the main loop consisting of the steps, program abstraction, model checking the abstract system and counterexample analysis.

Given a probabilistic program \mathfrak{P} and a probabilistic reachability property $P_{\leq b}(\diamond e)$, where e is a boolean predicate over the variables of \mathfrak{P} , CEGAR checks if $\mathfrak{P} \models P_{\leq b}(\diamond e)$. Additionally we now want to compute a smallest critical command set.

5. Computing High-Level Counterexamples on Abstractions

The initialization phase of CEGAR is just as in the basic algorithm. The set $\Phi = \{e\}$ is used as initial set of predicates.

5.1.1. Program Abstraction

The first step of the main loop of CEGAR is computing the abstraction \mathfrak{P}_Φ . This step does not have to be adapted, as the quotient contains the commands as actions of the transitions, we have everything that is needed for the computation of a minimal command set.

5.1.2. Model Checking the Abstract System and Computing a High-Level Counterexample

In this step we model check the abstract system and generate a counterexample that is analyzed in the third step of the CEGAR loop.

In the basic algorithm we used standard techniques to compute $Pr_{\mathfrak{P}_\Phi}^+(e)$ and a scheduler σ such that $Pr_{\llbracket \mathfrak{P} \rrbracket_\Phi}^\sigma(e) = Pr_{\mathfrak{P}_\Phi}^+(e)$ the probability of reaching a state in which e is satisfied and a scheduler reaching that probability. In the Markov chain $\llbracket \mathfrak{P} \rrbracket_\Phi^\sigma$ we then computed a finite path set counterexample using graph algorithms.

Now that we want to compute a high-level counterexample, we have to adapt this approach. Our ultimate goal is to find a smallest critical command set of \mathfrak{P} . So we compute the smallest critical command set \mathcal{C}' on $\llbracket \mathfrak{P} \rrbracket_\Phi$ and check it for realizability. A critical labeling is realizable, if it actually is a critical command set of \mathfrak{P} .

Realizability of a critical command set \mathcal{C}' can be decided by building and model checking $\llbracket \mathfrak{P} \rrbracket_{\mathcal{C}'}$, the explicit model of the program restricted to \mathcal{C}' . In our experiments this turned out to be the most efficient way. Another method, closely related to the basic algorithm, is to check a path set counterexample computed on $(\llbracket \mathfrak{P} \rrbracket_\Phi)_{\mathcal{C}'}$, the restriction of the quotient to transitions with an action in \mathcal{C}' . If there are a scheduler σ and a set of realizable paths Π in $(\llbracket \mathfrak{P} \rrbracket_\Phi)_{\mathcal{C}'}^\sigma$ with a probability mass greater than λ , then \mathcal{C}' is a critical command set of \mathfrak{P} . In Section 5.2, we will show the correctness of this statement and that we even compute the smallest critical command set with this algorithm. The advantage of this approach is that we can reuse the counterexample analysis from the basic algorithm. However, note that if the path set Π is not realizable with a probability greater than λ , other path set counterexamples might still be realizable and thus \mathcal{C}' might still be realizable.

In summary this step now consists of the following substeps:

- (1) Model check $\llbracket \mathfrak{P} \rrbracket_\Phi$. If the property is satisfied, exit the CEGAR loop and return “property satisfied”.

- (2) Compute a smallest critical command set \mathcal{C}' on $\llbracket \mathcal{P} \rrbracket_{\Phi}$
- (3) Compute $(\llbracket \mathcal{P} \rrbracket_{\Phi})|_{\mathcal{C}'}$, the restriction of $\llbracket \mathcal{P} \rrbracket_{\Phi}$ to transitions with actions in \mathcal{C}' .
- (4) Model check $(\llbracket \mathcal{P} \rrbracket_{\Phi})|_{\mathcal{C}'}$ and compute a path set as counterexample

The first model checking step is necessary to verify that there can exist a critical command set. The second step computes the high-level counterexample. In step number three we cut down our quotient to the abstraction of the system induced by the high-level counterexample. In step number 4 we then do the computations from the basic algorithm, this time on the restricted quotient.

Note that the probability on the restricted quotient may be less than on the unrestricted quotient, since we minimize the command set under the constraint that we exceed the bound. Allowing transitions with all actions, i.e. commands, might result in a higher probability.

5.1.3. Counterexample Analysis for High-Level Counterexamples

In this step we check the counterexample from the previous step for realizability and generate new predicates, if it is not realizable.

In the basic algorithm we built SMT formulas from the paths in the counterexample. If the path formula $\mathcal{PF}(\pi)$ of a finite path π is satisfiable, the path is realizable. If realizable paths with a probability mass greater than the bound are found, they form a realizable counterexample. For spurious paths, i.e. paths that are not realizable, new predicates are generated by using interpolation on parts of the path formula.

Since we check the realizability of a critical command set by checking the realizability of a path set as described in the previous Subsection 5.1.2, we can reuse the approach from the basic algorithm. In case that a realizable path set with a probability mass greater than the bound is found, the CEGAR loop exists with the result “the property is violated” and returns the critical command set, which is a smallest critical command set, as high-level counterexample. If the necessary probability mass cannot be achieved, new predicates are generated from the spurious paths and the next iteration is started.

5.2. Correctness

In this section we show the correctness of the algorithm for computing high-level counterexamples using CEGAR presented in the previous section.

5. Computing High-Level Counterexamples on Abstractions

At first we show that the quotient of the restriction of a PA to an command set is equivalent to the restriction of the quotient of a PA to that command set, i.e. the order of taking the quotient and restricting the commands is irrelevant. Having that, we will show that a critical command set of a PA is also a critical command set in its quotient. We then conclude that a realizable minimal critical command set on a quotient of the semantics of a program represents a minimal critical command set in the program.

Let \mathfrak{P} be a probabilistic program, $\llbracket \mathfrak{P} \rrbracket = \mathcal{A} = (S, s_{init}, Act, \mathbf{P})$ be a PA, $R \subseteq S \times S$ an equivalence relation and $\mathcal{C}' \subseteq Act$. Since $\llbracket \mathfrak{P} \rrbracket = \mathcal{A}$ we have that that $Act = C(\mathfrak{P})$. Thus we will use actions of the PA and commands of the program interchangeably.

Lemma 2. *Let \mathcal{A} , R and \mathcal{C}' as above. $(\mathcal{A}_{|\mathcal{C}'})_{/R} = (\mathcal{A}_{/R})_{|\mathcal{C}'}$ i.e., the order of taking the quotient and restricting to a label set is irrelevant.*

Obviously the state set, the initial state and the action set of $(\mathcal{A}_{|\mathcal{C}'})_{/R}$ and $(\mathcal{A}_{/R})_{|\mathcal{C}'}$ are equal, since restricting to a command set only changes the probabilistic transition relation. So it remains to show that the transition relations are equal.

Let $[s]$ be an arbitrary state of $(\mathcal{A}_{|\mathcal{C}'})_{/R}$ and $(\mathcal{A}_{/R})_{|\mathcal{C}'}$. If a transition $(\alpha, \mu_{/R})$ is in the transitions of $[s]$ for either of the PAs, then there is $(\alpha, \mu) \in \mathbf{P}(s')$ for some $s' \in [s]$ and $\mu_{/R}([t]) = \sum_{t' \in [t]} \mu(t')$ for all $t \in S$. This comes from the fact, that restricting to a label set only removes transitions. The set of all $(\alpha, \mu_{/R})$ pairs that satisfy that requirement are called transition candidates. So in either case, a transition requires a witness (α, μ) in \mathcal{A} . In $(\mathcal{A}_{|\mathcal{C}'})_{/R}$ a transition candidate is present as transition if and only if all witnesses have the a label $\alpha \in \mathcal{C}'$. Otherwise the witnesses are not present in $\mathcal{A}_{|\mathcal{C}'}$ and don't generate a transition in the quotient of that automaton. In $(\mathcal{A}_{/R})_{|\mathcal{C}'}$ a transition candidate is present as a transition if the transition candidate has an action label $\alpha \subseteq \mathcal{C}'$. Otherwise it would have been eliminated when restricting the quotient to the label set.

So we can conclude that in both cases a state $[s]$ has the transitions $(\alpha, \mu) \in \mathbf{P}(s')$ for some $s' \in [s]$ such that $\mu_{/R}([t]) = \sum_{t' \in [t]} \mu(t')$ for all $t \in S$, with $\alpha \in \mathcal{C}'$.

Since state set, initial state, action set and probabilistic transition relation are equal, the probabilistic automata $(\mathcal{A}_{|\mathcal{C}'})_{/R}$ and $(\mathcal{A}_{/R})_{|\mathcal{C}'}$ are equal.

Let \mathcal{A} , R , and \mathcal{C}' as above. Let $P_{\leq \lambda}(\diamond T)$ be a probabilistic reachability property.

Lemma 3. *If \mathcal{C}' is a critical command set on \mathcal{A} , then \mathcal{C}' is a critical command set on $\mathcal{A}_{/R}$.*

If \mathcal{C}' is a critical labeling on \mathcal{A} , then $P_{\leq \lambda}(\diamond T)$ is violated on $\mathcal{A}_{|\mathcal{C}'}$. Since a quotient simulates the original system, we have that the the probability for reaching a state in T is greater in $(\mathcal{A}_{|\mathcal{C}'})_{/R}$ compared to $\mathcal{A}_{|\mathcal{C}'}$. Therefore $(\mathcal{A}_{|\mathcal{C}'})_{/R}$ violates $P_{\leq \lambda}(\diamond T)$.

By Lemma 2 we have that $(\mathcal{A}/R)_{|\mathfrak{C}'}$ violates $P_{\leq\lambda}(\diamond T)$. By definition of critical command set, \mathfrak{C}' is a critical label set on \mathcal{A}/R .

Let \mathcal{A} , R , \mathfrak{C}' and $P_{\leq\lambda}(\diamond T)$ as above.

Lemma 4. *A realizable smallest critical label set \mathfrak{C}' on \mathcal{A}/R is a smallest critical command set of \mathfrak{P} .*

At first we show that a realizable critical command set \mathfrak{C}' is a critical command set on \mathfrak{P} . Since the critical command set is realizable, we have a set of paths in $\mathcal{A}_{|\mathfrak{C}'}$, and thus in \mathcal{A} , with a probability mass greater than λ . Therefore the property is violated in \mathcal{A} and thus also by definition in \mathfrak{P} . So \mathfrak{C}' is a critical command set.

It further is a smallest critical command set. Assume there is a critical command set \mathfrak{C}'' of \mathfrak{P} with $|\mathfrak{C}''| < |\mathfrak{C}'|$. Then \mathfrak{C}'' is a critical command set on \mathcal{A} . By Lemma 3 it also is a critical command set on \mathcal{A}/R . This is a contradiction to \mathfrak{C}' being the smallest critical command set.

Now we show that checking realizability of a critical labeling can be done by checking the realizability of a path set counterexample in the restriction of the quotient.

Let \mathcal{A} , R be as above. Let $P_{\leq\lambda}(\diamond T)$ be a probabilistic reachability property violated by \mathcal{A}/R .

Lemma 5. *Let \mathfrak{C}' be a critical command set on \mathcal{A}/R , σ a scheduler such that $(\mathcal{A}/R)_{|\mathfrak{C}'}$ violates $P_{\leq\lambda}(\diamond T)$ and Π a path set counterexample on $(\mathcal{A}/R)_{|\mathfrak{C}'}$. If Π is realizable then \mathfrak{C}' is a critical command set on \mathcal{A} .*

If Π is realizable, there exists Π' and σ' such that all paths of Π' are in $\mathcal{A}^{\sigma'}$ and the probability mass of Π' is greater than λ . Since the paths of Π are in $(\mathcal{A}/R)_{|\mathfrak{C}'}$ and thus in $(\mathcal{A}/R)_{|\mathfrak{C}'}$ and by Lemma 2 in $(\mathcal{A}_{|\mathfrak{C}'})/R$, we by Lemma 1 have that they are in $\mathcal{A}_{|\mathfrak{C}'}$. Thus $\mathcal{A}_{|\mathfrak{C}'}$ violates the property $P_{\leq\lambda}(\diamond T)$. Therefore \mathfrak{C}' is a critical command set on \mathcal{A} .

Observe that the other direction in general does not hold. So there may be path set counterexamples that are not realizable although the command set is critical. This can easily be seen when looking at $C(\mathfrak{P})$ i.e., the set of all commands. This is critical, if the program violates the property. But a quotient may contain unrealizable paths.

Using the above lemmas we can now argue that the CEGAR algorithm as presented in this chapter computes the smallest critical command set, or eventually finds the property satisfied.

5. Computing High-Level Counterexamples on Abstractions

Theorem 6. *Let \mathfrak{P} be a probabilistic program and $P_{\leq\lambda}(\diamond e)$ a probabilistic reachability property. If $\mathfrak{P} \not\models P_{\leq\lambda}(\diamond e)$ CEGAR returns a smallest critical command set \mathcal{C} of \mathfrak{P} , otherwise it returns that the property is satisfied.*

We have as a loop invariant for the CEGAR loop that the critical label set, that is computed as candidate for the smallest critical command set, is smaller or equal to all critical label sets in $\llbracket \mathfrak{P} \rrbracket$. Once the loop exits with the result $\mathfrak{P} \not\models P_{\leq\lambda}(\diamond e)$, we have that the candidate is a critical command set on \mathfrak{P} . Together with the loop invariant, we can conclude that it is a smallest one.

The refinement step uses path set counterexamples and the model checking of the quotient is unchanged. Thus we always refine the quotient as in each iteration at least one spurious path is removed. By the correctness of the basic algorithm we get the correctness in the case that $\mathfrak{P} \models P_{\leq\lambda}(\diamond e)$, since eventually the abstraction is fine enough that the probability in the quotient is less or equal λ .

5.3. Experimental Results

A prototype of the described CEGAR algorithm was implemented in the model checking tool StoRM¹ with about 1000 lines of C++ code in the course of this thesis. The implementation additionally uses existing classes from StoRM for model checking probabilistic automata and generating minimal command sets for given probabilistic automata, as well as some utility classes, e.g. for input/output. The recursive enumeration algorithm [Jiménez and Marzal, 1999] was implemented with about 300 additional lines of code, to compute the paths in the counterexample used for refinement. To solve the mixed integer linear program used for computing the minimal command set Gurobi [Gurobi Optimization, 2014] was used. As SMT-solver for computing the abstraction Z3 [De Moura and Bjørner, 2008] was used. For checking realizability and interpolation MathSAT [Cimatti et al., 2013] was the underlying SMT solver.

The experiments were performed on a machine with an Intel Core i7 processor and 16GB main memory, running Microsoft Windows 7 64-bit. The program was built using Microsoft Visual Studio.

For the experiments the crowds protocol [Reiter and Rubin, 1998] was used. The protocols purpose is to obscure the origin of a message. The sender of the message is a member of the crowd. He sends the message to some member of the crowd. That member then forwards the message with a certain probability to some member of the crowd or else delivers the message. Each member of the crowd can only observe that crowd member who most recently sent the message. Some crowd

¹StoRM is currently under development and not yet published.

5.3. Experimental Results

members however eavesdrop the communication. These members don't forward messages, instead they record who sent a message to them. Each time a message is delivered to some crowd member, it is randomly decided whether the crowd member is currently good or bad.

In the experiments we checked if the probability that bad members observe member 0, the true sender, more than once as sender of a message is at most $\lambda = 0.1$. The actual probability is about 0.11 up to 0.33 for the parameters used in the experiments, so that the existence of a counterexample is guaranteed.

The protocol has two parameters, the crowd size and the number of runs. The crowd size is encoded into the program, as it influences the commands and updates. The second parameter, the number of runs, is encoded as a constant in the program. The probabilistic program in the input language of StoRM is given in Listing A.1 on page 43 for a crowd size of 5 and in Listing A.2 for a crowd size of 10.

example	explicit model			CEGAR (abstract model)			
	states	transitions	time	states	transitions	iterations	time
crowds5_5	8607	15113	2147 ms	27	42	6	673 ms
crowds10_5	110562	262073	82664 ms	27	97	6	786 ms
crowds15_5	586242	1753883	1342069 ms	27	127	6	955 ms
crowds20_5	2036647	7362293	TO (>1 h)	27	151	6	1100 ms
crowds5_5 ($\lambda = 0.2$)	8607	15113	2095 ms	209	406	9	3135459 ms

Table 5.1.: Experimental results for computing high-level counterexamples on the explicit model vs. on an abstraction using CEGAR.

The results of the experiments are shown in Table 5.1. In the columns labeled *explicit models* show the results for computing the counterexample using the existing implementation in StoRM. That implementation generated an explicit probabilistic automaton and solves the mixed integer linear program described in [Wimmer et al., 2013] for that probabilistic automaton. The columns labeled *CEGAR (abstract model)* show the results for our new algorithm described in Chapter 5.

The columns “*states*” indicate the number of states for the explicit model and for the abstraction in the final iteration of our algorithm respectively. The columns “*states*” hold the number of non-zero entries in the transition matrix is in column. Again these numbers refer to the explicit model and the abstraction in the last iteration, respectively. The columns “*time*” indicate the total running time of the algorithm, measured by the program internally. *TO* indicates that the computation timed out, the timeout was set to one hour for each individual run. For CEGAR the additional column *iterations* holds the number of iterations of the algorithm.

The experiments clearly show that our algorithm has the potential of vastly reducing the computation time. Due to the structure of the crowds protocol for

5. Computing High-Level Counterexamples on Abstractions

a bound of $\lambda = 0.1$ and all parameter combinations used in the experiments, equivalent sets of 6 commands were sufficient as counterexample. Our algorithm based on counterexample guided abstraction refinement computed an abstraction that abstracted from the number of crowd members. This explains why the number of states in the abstraction is identical for all crowd sizes. On this very small abstraction a high-level counterexample could be computed in very short time.

step	time	percentage
Building Abstraction	701 ms	0.02 %
Model Checking	1160 ms	0.04 %
k-shortest Paths	2049947 ms	65.4 %
Checking Paths and Interpolation	1083525 ms	34.6 %

Table 5.2.: Time breakdown of generating a counterexample for the bound 0.2 on `crowds5_5` using CEGAR.

However, for some cases the performance is far worse than computing high-level counterexamples on the explicit model. For the `crowds5_5` example and a probability bound of $\lambda = 0.2$, the actual probability is about 0.33, the computation took about 52 minutes. In this case the smallest critical command set consists of 7 commands. To find such a command set a finer quotient was needed. The major bottleneck is finding and checking huge amounts of paths for realizability as shown in Table 5.2. Since the strongest evidence approach [Han and Katoen, 2007] is used to generate paths from the counterexample, the algorithm stalls if the most probable paths are all realizable and only very few, improbable but necessary paths are not realizable. Only after having checked thousands of realizable paths eventually spurious paths were found and the quotient could be refined.

Summarizing the results, our presented algorithm using CEGAR for computing high-level counterexamples could in all experiments determine correct high-level counterexamples on abstractions that are multiple orders of magnitude smaller than the explicit model. Although searching for spurious paths is still a severe bottleneck, the computation time was significantly reduced in some experiments.

6. Conclusion

In this thesis, we presented a technique to compute *high-level counterexamples* as introduced in [Wimmer et al., 2013]. These counterexamples operate on the level of probabilistic programs in PRISM’s *guarded command language*. They consist of commands from the program and thus can provide useful insights when debugging probabilistic programs.

We also presented counterexample guided abstraction refinement for probabilistic systems described in [Wachter et al., 2007] in detail. We pointed out a problem with the generation of new predicates when multiple disjoint parts of a path are not realizable. We presented a method that guarantees that predicates that are neither “true” nor “false” are generated. The presented method worked well for our experiments.

Based on counterexample guided abstraction refinement for probabilistic systems we developed an algorithm to speed up the computation of *smallest critical command sets*, a type of high-level counterexamples, and sketched a proof for its correctness. Our algorithm computes high-level counterexamples for the abstractions used in CEGAR. By reducing checking the realizability of high-level counterexamples to checking the realizability of path sets the refinement procedures from CEGAR can be used to refine the abstraction if necessary.

We implemented the described algorithm and performed experiments investigating the performance of our algorithm in comparison to the application of the mixed integer linear programming method to explicit models generated from probabilistic programs. The results are presented in this work and show that the proposed algorithm can reduce the computation time needed by multiple orders of magnitude.

6.1. Future Work

In future work, techniques to avoid explicitly checking realizability of individual paths could be investigated. Preliminary experiments showed that without the optimization of checking the realizability of high-level counterexamples by model checking explicit models for subprograms, the performance of CEGAR is unacceptably bad on small systems compared to the MILP-based approach.

One way to overcome this would be checking realizability of paths with cycles

6. Conclusion

such that the cycle is also a cycle in the realization of the path. In that case the cycle could be repeated arbitrarily often. The probability of the obtained set can be computed approximately by approximating the limit of a series.

With such optimizations algorithms based on CEGAR might be an even more powerful tool to reduce memory usage and computation time when generating high-level counterexamples.

Bibliography

- Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.
- Bellman, R. (1957). A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684.
- Cimatti, A., Griggio, A., Schaafsma, B., and Sebastiani, R. (2013). The MathSAT5 SMT Solver. In Piterman, N. and Smolka, S., editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer.
- De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.
- Gurobi Optimization, I. (2014). Gurobi optimizer reference manual.
- Han, T. and Katoen, J.-P. (2007). *Counterexamples in probabilistic model checking*. Springer.
- Hermanns, H., Wachter, B., and Zhang, L. (2008). Probabilistic cegar. In *Computer Aided Verification*, pages 162–175. Springer.
- Jiménez, V. M. and Marzal, A. (1999). Computing the k shortest paths: A new algorithm and an experimental comparison. In *Algorithm engineering*, pages 15–29. Springer.
- Kwiatkowska, M., Norman, G., and Parker, D. (2011). Prism 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification*, pages 585–591. Springer.
- Puterman, M. L. (1994). Markov decision processes: Discrete dynamic stochastic programming.
- Reiter, M. K. and Rubin, A. D. (1998). Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92.
- Wachter, B. (2011). *Refined probabilistic abstraction*. Logos Verlag Berlin GmbH.

Bibliography

- Wachter, B., Zhang, L., and Hermanns, H. (2007). Probabilistic model checking modulo theories. In *Quantitative Evaluation of Systems, 2007. QEST 2007. Fourth International Conference on the*, pages 129–140. IEEE.
- Wimmer, R., Jansen, N., Vorpahl, A., Ábrahám, E., Katoen, J.-P., and Becker, B. (2013). High-level counterexamples for probabilistic automata. *CoRR*, abs/1305.5055.

A. Appendix

Listing A.1: crowds5_5 in the input language of StoRM

```
mdp

// probability of forwarding
const double PF = 0.8;
const double notPF = .2; // must be 1-PF
// probability that a crowd member is bad
const double badC = .167;
// probability that a crowd member is good
const double goodC = 0.833;
// Total number of protocol runs to analyze
const int TotalRuns = 5;
// size of the crowd
const int CrowdSize = 5;

module crowds
  // protocol phase
  phase: [0..4] init 0;

  // crowd member good (or bad)
  good: bool init false;

  // number of protocol runs
  runCount: [0..TotalRuns] init 0;

  // observe_i is the number of times the attacker observed crowd member i
  observe0: [0..TotalRuns] init 0;

  observe1: [0..TotalRuns] init 0;

  observe2: [0..TotalRuns] init 0;

  observe3: [0..TotalRuns] init 0;

  observe4: [0..TotalRuns] init 0;

  // the last seen crowd member
  lastSeen: [0..CrowdSize - 1] init 0;

  // get the protocol started
  [] phase=0 & runCount<TotalRuns -> 1: (phase'=1) & (runCount'=runCount+1)
    & (lastSeen'=0);
  [] phase=0 & runCount=TotalRuns -> 1: (phase'=0);

  // decide whether crowd member is good or bad according to given
  probabilities
  [] phase=1 -> goodC : (phase'=2) & (good'=true) + badC : (phase'=2) & (
    good'=false);
```

A. Appendix

```
// if the current member is a good member, update the last seen index (
  chosen uniformly)
[] phase=2 & good -> 1/5 : (lastSeen'=0) & (phase'=3) + 1/5 : (lastSeen
  '=1) & (phase'=3) + 1/5 : (lastSeen'=2) & (phase'=3) + 1/5 : (lastSeen
  '=3) & (phase'=3) + 1/5 : (lastSeen'=4) & (phase'=3);

// if the current member is a bad member, record the most recently seen
  index
[] phase=2 & !good & lastSeen=0 & observe0 < TotalRuns -> 1: (observe0'=
  observe0+1) & (phase'=4);
[] phase=2 & !good & lastSeen=1 & observe1 < TotalRuns -> 1: (observe1'=
  observe1+1) & (phase'=4);
[] phase=2 & !good & lastSeen=2 & observe2 < TotalRuns -> 1: (observe2'=
  observe2+1) & (phase'=4);
[] phase=2 & !good & lastSeen=3 & observe3 < TotalRuns -> 1: (observe3'=
  observe3+1) & (phase'=4);
[] phase=2 & !good & lastSeen=4 & observe4 < TotalRuns -> 1: (observe4'=
  observe4+1) & (phase'=4);

// good crowd members forward with probability PF and deliver otherwise
[] phase=3 -> PF : (phase'=1) + notPF : (phase'=4);

// deliver the message and start over
[] phase=4 -> 1: (phase'=0);

endmodule

label "observe0Greater1" = observe0>1;
```

Listing A.2: crowds10_5 in the input language of StoRM

```
mdp

// probability of forwarding
const double PF = 0.8;
const double notPF = .2; // must be 1-PF
// probability that a crowd member is bad
const double badC = .167;
// probability that a crowd member is good
const double goodC = 0.833;
// Total number of protocol runs to analyze
const int TotalRuns = 5;
// size of the crowd
const int CrowdSize = 10;

module crowds
  // protocol phase
  phase: [0..4] init 0;

  // crowd member good (or bad)
  good: bool init false;

  // number of protocol runs
  runCount: [0..TotalRuns] init 0;

  // observe_i is the number of times the attacker observed crowd member i
  observe0: [0..TotalRuns] init 0;

  observe1: [0..TotalRuns] init 0;

  observe2: [0..TotalRuns] init 0;
```

```

observe3: [0..TotalRuns] init 0;
observe4: [0..TotalRuns] init 0;
observe5: [0..TotalRuns] init 0;
observe6: [0..TotalRuns] init 0;
observe7: [0..TotalRuns] init 0;
observe8: [0..TotalRuns] init 0;
observe9: [0..TotalRuns] init 0;

// the last seen crowd member
lastSeen: [0..CrowdSize - 1] init 0;

// get the protocol started
[] phase=0 & runCount<TotalRuns -> (phase'=1) & (runCount'=runCount+1) & (
  lastSeen'=0);
[] phase=0 & runCount=TotalRuns -> 1: (phase'=0);

// decide whether crowd member is good or bad according to given
probabilities
[] phase=1 -> goodC : (phase'=2) & (good'=true) + badC : (phase'=2) & (
  good'=false);

// if the current member is a good member, update the last seen index (
chosen uniformly)
[] phase=2 & good -> 1/10 : (lastSeen'=0) & (phase'=3) + 1/10 : (lastSeen
'=1) & (phase'=3) + 1/10 : (lastSeen'=2) & (phase'=3) + 1/10 : (
  lastSeen'=3) & (phase'=3) + 1/10 : (lastSeen'=4) & (phase'=3) + 1/10 :
  (lastSeen'=5) & (phase'=3) + 1/10 : (lastSeen'=6) & (phase'=3) + 1/10
  : (lastSeen'=7) & (phase'=3) + 1/10 : (lastSeen'=8) & (phase'=3) +
  1/10 : (lastSeen'=9) & (phase'=3);

// if the current member is a bad member, record the most recently seen
index
[] phase=2 & !good & lastSeen=0 & observe0 < TotalRuns -> (observe0'=
  observe0+1) & (phase'=4);
[] phase=2 & !good & lastSeen=1 & observe1 < TotalRuns -> (observe1'=
  observe1+1) & (phase'=4);
[] phase=2 & !good & lastSeen=2 & observe2 < TotalRuns -> (observe2'=
  observe2+1) & (phase'=4);
[] phase=2 & !good & lastSeen=3 & observe3 < TotalRuns -> (observe3'=
  observe3+1) & (phase'=4);
[] phase=2 & !good & lastSeen=4 & observe4 < TotalRuns -> (observe4'=
  observe4+1) & (phase'=4);
[] phase=2 & !good & lastSeen=5 & observe5 < TotalRuns -> (observe5'=
  observe5+1) & (phase'=4);
[] phase=2 & !good & lastSeen=6 & observe6 < TotalRuns -> (observe6'=
  observe6+1) & (phase'=4);
[] phase=2 & !good & lastSeen=7 & observe7 < TotalRuns -> (observe7'=
  observe7+1) & (phase'=4);
[] phase=2 & !good & lastSeen=8 & observe8 < TotalRuns -> (observe8'=
  observe8+1) & (phase'=4);
[] phase=2 & !good & lastSeen=9 & observe9 < TotalRuns -> (observe9'=
  observe9+1) & (phase'=4);

// good crowd members forward with probability PF and deliver otherwise
[] phase=3 -> PF : (phase'=1) + notPF : (phase'=4);

```

A. Appendix

```
// deliver the message and start over
[] phase=4 -> (phase'=0);

endmodule

label "observe0Greater1" = observe0 > 1;
```