Model Repair Revamped
— On the Automated Synthesis of Markov Chains —

Milan Češka[1], Christian Dehnert[2], Nils Jansen[3], Sebastian Junges[2], and Joost-Pieter Katoen[2(✉)]

[1] FIT, IT4I Centre of Excellence, Brno University of Technology, Brno, Czech Republic
[2] RWTH Aachen University, Aachen, Germany
katoen@cs.rwth-aachen.de
[3] Radboud University, Nijmegen, The Netherlands

Abstract. This paper outlines two approaches—based on counter-example-guided abstraction refinement (CEGAR) and counterexample-guided inductive synthesis (CEGIS), respectively—to the automated synthesis of finite-state probabilistic models and programs. Our CEGAR approach iteratively partitions the design space starting from an abstraction of this space and refines this by a light-weight analysis of verification results. The CEGIS technique exploits critical subsystems as counterexamples to prune all programs behaving incorrectly on that input. We show the applicability of these synthesis techniques to sketching of probabilistic programs, controller synthesis of POMDPs, and software product lines.

1 Introduction

Model Repair. In 2011, Smolka *et al.* [5] coined the following *model repair* problem [8]: given a finite Markov chain D and a probabilistic specification φ such that $D \not\models \varphi$, find a Markov chain D' that differs from D only in the transition probabilities, such that $D' \models \varphi$. Typical probabilistic specifications impose a threshold on reachability probabilities, such as "is the probability to reach a bad state at most 1/1000?" Model repair thus amounts to tweaking (some of) the probabilities in a given Markov chain in order to obtain a chain satisfying the specification. It can be solved using parameter synthesis [5] techniques as, e.g., supported by the Prophesy tool [21]. An extension of model repair in which repairs are associated a cost and a minimal-total-cost repair is to be found can be solved by non-linear programming [5]. The scalability of model repair can be improved in several ways: by solving a series of convex programs instead of a non-linear program [19], by repairing abstractions of Markov chains [13], or by a greedy approach exploiting monotonicity [32]. Particle swarm optimisation has been used to model repair of Markov decision processes (MDPs) [14].

Topology Changes. In the original setting of model repair, only transition probabilities are subject to change. Adding or removing transitions is not admitted. That is to say, every possible repair keeps the topology of the Markov chain invariant. This notion of repair is thus in fact *transition probability repair*. In this paper, we take this idea a step further, and allow for amending the Markov chain's topology. More precisely, in addition to the possibility of modifying transition probabilities, we consider the possible deletion and/or addition of transitions. Changing the topology results in varying sets of reachable states. Viewing Markov chains as operational model for (discrete) probabilistic programs, repairs may affect the control structure as well as the probabilistic choices. Traditional parameter synthesis techniques are inadequate for this problem, as these techniques restrict parameter expressions like $1/2-\varepsilon$ to range over $(0, 1)$, i.e., excluding zero (no transition) and one (a Dirac transition). Topology changes often come at a price; e.g., for parametric Markov chains the complexity of qualitative (i.e., zero-one) reachability becomes NP-complete whereas in absence of topology changes a polynomial-time algorithm suffices [15].

Synthesis. Model (or: program) repair in the aforementioned sense is a natural instance of model (or: program) synthesis [1,22,23]. Program synthesis amounts to automatically provide an instantiated probabilistic program satisfying all quantitative properties, or returns that such design does not exist. Though the synthesis problem is undecidable in general, there are interesting sub-cases—such as our variant of model repair—that are decidable (but computationally intensive). We consider a family $\mathfrak{D}$ of Markov chains where each family member can be viewed as an admissible repaired version of Markov chain $D \in \mathfrak{D}$. Families are finite and consist of finite chains. As in syntax-guided synthesis, possible repairs are described by some grammar rules, either at the model level or in some high-level description language such as the probabilistic guarded command language [30] or the PRISM modelling language [27]. The successful approach of program sketching [35] naturally fits within this setting. Starting from a program sketch, i.e., a program with "holes", it aims to obtain a program satisfying the specification φ by filling the holes with possible repairs. Holes are the unknown parts of the program and can be replaced by one of finitely many options. All possible program sketch realisations constitute the family $\mathfrak{D}$. The synthesis problems considered in this paper can be used to answer queries like: (a) give a possible repair (if one exists)?, (b) what are all possible repairs?, and (c) which repairs are optimal in the sense of minimising the probability to reach a bad state? Each of these queries can be considered under the additional constraint of minimising the total cost of repairs, e.g., what are all possible repairs for φ that are cost minimal?

Our Synthesis Approaches. A naive enumerative solution is to analyse each individual family member, i.e., each possible amendment, or each possible hole instantiation. This is infeasible for large families. We therefore outline two approaches to the automated synthesis of finite-state probabilistic models: the first one fits within the realm of counterexample-guided abstraction refinement
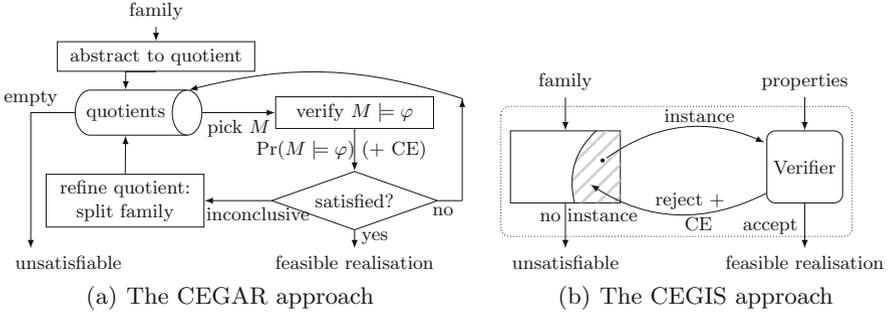
Fig. 1. CEGAR and CEGIS approaches to (feasibility) synthesis.

(CEGAR [17]) while the second approach fits within counterexample-guided inductive synthesis (CEGIS) [36]. Full details of the approaches can be found in [37,38]. We present both techniques at the level of Markov chains for threshold problems on reachability problems and illustrate their usage on simple probabilistic programs.

Using CEGAR. We represent all possible designs, thus the entire family, by a single MDP. A single, initial, non-deterministic choice determines according to which family member the MDP behaves. This technique is adopted from [16] and originated in software product lines [18]. As the MDP can be prohibitively large, we do not solve the synthesis problems directly on this model, but rather on an abstraction of it—similar in spirit as repairing abstract models [13]. Verifying the abstraction, i.e., the quotient MDP M, yields under- and over-approximations of the min and max probability of satisfying φ, respectively. A repair is impossible if e.g., the verification reveals that the min probability exceeds r for φ with threshold $\leq r$. If the model checking is inconclusive, i.e., the abstraction is too coarse, we iteratively refine the quotient MDP by splitting the family into subfamilies, see Fig. 1(a). The refinement is guided using the so-called inconsistent schedulers (aka: counterexamples) that optimise the probability on the MDP.

Using CEGIS. Starting from a family, a candidate realisation D is selected and discharged to a verifier, see Fig. 1(b). Using off-the-shelf probabilistic model-checking techniques [3,26], it verifies whether $D \models \varphi$ in case a solution is found. If $D \not\models \varphi$, a counterexample (CE) is derived which in our setting is a fragment [41] of the realisation D that refutes φ. The key is that this CE is exploited in a clever way by an SMT (satisfiability modulo theory)-based synthesiser to rule out potentially many realisations (the dashed area in Fig. 1(b))—rather than the just refuted realisation D—at once. Thus, in a sense counterexamples are "extended" to a set of refuting realisations. This synthesis-verification loop is repeated until either a satisfying realisation is found or the entire family is pruned implying the non-existence of a realisation $D \models \varphi$.

Design Space Partitioning. Both the CEGAR and CEGIS approach iteratively partition the family into "good", "bad" and inconclusive realisations. Phrased in terms of model repair, they partition the family into repaired, failed, and unknown Markov chains. The two approaches use complementary partitioning strategies. Whereas the CEGAR approach starts from considering all possible realisations, and successively splits the entire family of realisations into sub-families, the CEGIS approach starts with a single candidate realisation, and rules out several realisations by effectively exploiting counterexamples.

2   Preliminaries

We start with basic foundations, for details, see [3,4]. Then, we formalise the notion of families of Markov chains, and define various synthesis problems.

2.1   Probabilistic Models and Specifications

Probabilistic Models. A *probability distribution* over countable set X is a function $\mu\colon X \to [0,1]$ with $\sum_{x\in X}\mu(x) = \mu(X) = 1$. Let $Distr(X)$ denote the set of all distributions on X.

Definition 1 (MC). *A* Markov chain *(MC) $D = (S, s_0, \mathbf{P})$ with finite set S of states, initial state $s_0 \in S$, and transition probability function $\mathbf{P}\colon S \to Distr(S)$.*

MCs have unique distributions over successor states at each state. A sub-Markov chain (sub-MC) is induced by a MC and a subset of its states. For $X \subseteq S$, let the set $\mathsf{Succ}(X)$ denote the successor states of X, i.e., $\mathsf{Succ}(X) = \{t \in S \mid \exists s \in X.\ \mathbf{P}(s,t) > 0\}$.

Definition 2 (sub-MC). *Let MC $D = (S, s_0, \mathbf{P})$ and $C \subseteq S$ a set of* (critical) *states with $s_0 \in C$. The* sub-MC *of D, C is the MC $D' = (S', s_0, \mathbf{P}')$ with $S' = C \cup \mathsf{Succ}(C)$, and*

$$\mathbf{P}'(s,t) = \begin{cases} \mathbf{P}(s,t) & s \in C, t \in S \\ 1 & s \in \mathsf{Succ}(C) \setminus C \wedge t = s \\ 0 & otherwise. \end{cases}$$

Adding non-determinism over distributions leads to Markov decision processes.

Definition 3 (MDP). *A* Markov decision process *(MDP) is a tuple $M = (S, s_0, Act, \mathcal{P})$ where S, s_0 as in Definition 1, Act is a finite set of actions, and $\mathcal{P}\colon S \times Act \nrightarrow Distr(S)$ is a partial transition probability function.*

The *available actions* in $s \in S$ are $Act(s) = \{a \in Act \mid \mathcal{P}(s,a) \neq \bot\}$. An MDP with $|Act(s)| = 1$ for all $s \in S$ is an MC. A *path* of an MDP M is an (in)finite sequence $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$, where $s_i \in S$, $a_i \in Act(s_i)$, and $\mathcal{P}(s_i, a_i)(s_{i+1}) \neq 0$ for all $i \in \mathbb{N}$. For finite π, $last(\pi)$ denotes the last state of π. Let Paths_{fin}^{M} denote the set of finite paths of M. The notions of paths carry over to MCs (actions are omitted).

Definition 4 (Scheduler). *A scheduler for an MDP $M = (S, s_0, Act, \mathcal{P})$ is a function $\sigma \colon \mathsf{Paths}_{fin}^M \to Act$ such that $\sigma(\pi) \in Act(\mathrm{last}(\pi))$ for all $\pi \in \mathsf{Paths}_{fin}^M$. Scheduler σ is* memoryless *if $\mathrm{last}(\pi) = \mathrm{last}(\pi') \implies \sigma(\pi) = \sigma(\pi')$ for all $\pi, \pi' \in \mathsf{Paths}_{fin}^M$. Let Σ^M denote the set of all schedulers of M.*

Schedulers resolve the non-determinism over actions in the MDP. Applying scheduler σ to an MDP M yields the *induced* Markov chain M_σ.

Specifications. For simplicity, we only consider reachability specifications $\varphi = \mathbb{P}_{\sim \lambda}(\Diamond G)$ where $G \subseteq S$ is a set of goal states, $\lambda \in [0,1] \subseteq \mathbb{R}$ is a threshold, and $\sim\ \in \{<, \leq, \geq, >\}$ is a binary comparison operator. Extensions to expected rewards, PCTL* [2], or ω-regular properties are rather straightforward.

The interpretation of φ for MC D is as follows. Let $\mathtt{Prob}(D, \phi)(s)$ denote the probability to satisfy $\phi = \Diamond G$ from state $s \in S$ in MC D. For initial state s_0, we abbreviate $\mathtt{Prob}(D, \phi)(s_0)$ by $\mathtt{Prob}(D, \phi)$. Then, $D \models \varphi$ iff $\mathtt{Prob}(D, \phi) \sim \lambda$. The specification φ holds in MDP M, denoted $M \models \varphi$, iff it holds for the induced MCs under all schedulers. The maximum probability to satisfy ϕ in MDP M is given by a maximising scheduler $\sigma^* \in \Sigma^M$, i.e., there is no scheduler $\sigma' \in \Sigma^M$ with $\mathtt{Prob}(M_{\sigma^*}, \phi) < \mathtt{Prob}(M_{\sigma'}, \phi)$. As we consider finite models, such a maximising scheduler always exists. Minimum probabilities are defined analogously.

2.2   Families of Markov Chains

We present our approaches on an explicit representation of a *family of MCs* using a parametric transition probability function. Such an explicit representation allows to reason about practically interesting synthesis problems, see Sect. 4.

Definition 5 (Family of MCs). *A family of MCs is a tuple $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ where S and s_0 are as before, K is a finite set of discrete parameters such that the domain of each parameter $k \in K$ is $T_k \subseteq S$, and $\mathfrak{P} \colon S \to Distr(K)$.*

The transition probability function of MCs maps states to distributions over successor states. For families of MCs, this function maps states to distributions over parameters. Instantiating each of these parameters with a value from its domain yields a "concrete" MC, called a *realisation*.

Definition 6 (Realisation). *A realisation of a family $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ is a function $r \colon K \to S$ where $\forall k \in K \colon r(k) \in T_k$. A realisation r yields an MC $D_r = (S, s_0, \mathfrak{P}(r))$, where $\mathfrak{P}(r)$ is the transition probability matrix in which each $k \in K$ in $\mathfrak{P}$ is replaced by $r(k)$. Let $\mathcal{R}^{\mathfrak{D}}$ denote the set of all realisations for $\mathfrak{D}$.*

As a family $\mathfrak{D}$ of MCs is defined over finite parameter domains, the number of family members (i.e. realisations from $\mathcal{R}^{\mathfrak{D}}$) of $\mathfrak{D}$ is finite, viz. $|\mathfrak{D}| := |\mathcal{R}^{\mathfrak{D}}| = \prod_{k \in K} |T_k|$, but exponential in $|K|$. Subsets of $\mathcal{R}^{\mathfrak{D}}$ induce so-called *subfamilies* of $\mathfrak{D}$. While all these MCs share the same state space, their *reachable* states may differ, as demonstrated by the following example.
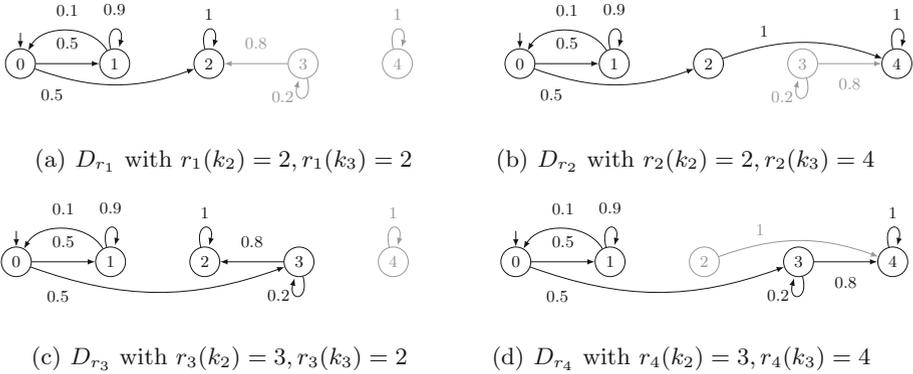
(a) D_{r_1} with $r_1(k_2) = 2, r_1(k_3) = 2$

(b) D_{r_2} with $r_2(k_2) = 2, r_2(k_3) = 4$

(c) D_{r_3} with $r_3(k_2) = 3, r_3(k_3) = 2$

(d) D_{r_4} with $r_4(k_2) = 3, r_4(k_3) = 4$

Fig. 2. The four different realisations of family $\mathfrak{D}$.

Example 1 (Family of MCs). Consider the family of MCs $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ where $S = \{0, \ldots, 4\}$, $s_0 = 0$, and $K = \{k_0, \ldots, k_5\}$ with domains $T_{k_0} = \{0\}$, $T_{k_1} = \{1\}$, $T_{k_2} = \{2, 3\}$, $T_{k_3} = \{2, 4\}$, $T_{k_4} = \{3\}$ and $T_{k_5} = \{4\}$. The parametric transition function is defined by:

$$\mathfrak{P}(0) = 0.5\colon k_1 + 0.5\colon k_2 \qquad \mathfrak{P}(1) = 0.1\colon k_0 + 0.9\colon k_1 \qquad \mathfrak{P}(2) = 1\colon k_3$$
$$\mathfrak{P}(3) = 0.8\colon k_3 + 0.2\colon k_4 \qquad \mathfrak{P}(4) = 1\colon k_5$$

We can simplify the representation by substituting the constants:

$$\mathfrak{P}(0) = 0.5\colon 1 + 0.5\colon k_2 \qquad \mathfrak{P}(1) = 0.1\colon 0 + 0.9\colon 1 \qquad \mathfrak{P}(2) = 1\colon k_3$$
$$\mathfrak{P}(3) = 0.8\colon k_3 + 0.2\colon 3 \qquad \mathfrak{P}(4) = 1\colon 4$$

Figure 2 shows the four MCs that result from the realisations $\{r_1, r_2, r_3, r_4\} = \mathcal{R}^{\mathfrak{D}}$ of $\mathfrak{D}$. States that are unreachable from the initial state are greyed out. The family has five states, each of which are reachable in one of the realisations. Yet, every realisation has at most four reachable states.

2.3   Synthesis Problems

Problem 1 (Synthesis). Let $\mathfrak{D}$ be a family of MCs and $\varphi = \mathbb{P}_{\sim\lambda}(\phi)$ with $\phi = \Diamond G$ for $G \subseteq S$. We consider the following synthesis problems:

1. Find a realisation $r \in \mathcal{R}^{\mathfrak{D}}$ with $D_r \models \varphi$.
2. Partition $\mathcal{R}^{\mathfrak{D}}$ into T and F with $r \in T$ iff $D_r \models \varphi$ and $r \in F$ otherwise.
3. Find a realisation $r^* \in \mathcal{R}^{\mathfrak{D}}$ with $r^* = \underset{r \in \mathcal{R}^{\mathfrak{D}}}{\arg\max} \{\texttt{Prob}(D_r, \phi)\}$.

The first synthesis problem (referred to as feasibility synthesis) is to determine a realisation satisfying φ, provided some exists. The second problem (referred to as threshold synthesis) is to identify the set of realisations satisfying and violating

a given specification, respectively. The feasibility synthesis problem is in a sense just a simple instance of threshold synthesis to find one realisation $r \in T$. The last problem (referred to as max synthesis) is to find a realisation that maximises the reachability probability. It can be defined for minimising such probabilities in a similar way. As our families are finite, such optimal realisations r^* always exist. Phrased in terms of model repair, the first problem is concerned with the question whether a possible repair (under all admissible repairs) does exist, the second problem partitions the realisations into those that are repaired and those that cannot, while the last problem is about finding the repair that maximises (or, dually, minimises) the objective. The simplest synthesis problem, feasibility, is NP-complete [38] (for a minor extension to families).

Example 2 (Synthesis problems). Recall the family of MCs $\mathfrak{D}$ from Example 1. For the specification $\varphi = \mathbb{P}_{\geq 1/10}(\Diamond\{4\})$, the solution to the threshold synthesis problem is $T = \{r_2, r_4\}$ and $F = \{r_1, r_3\}$, as the goal state 4 is not reachable for D_{r_1} and D_{r_3}. For $\phi = \Diamond\{4\}$, the solution to the max synthesis problem on $\mathfrak{D}$ is r_2 or r_4, as D_{r_2} and D_{r_4} almost surely reach state 1.

Remark 1. It is sometimes beneficial to consider a mild variant of the max-synthesis problem in which the realisation r^* is not required to achieve the maximal reachability probability, but it suffices to be sufficiently close to it. This notion of *ε-optimal synthesis* for a given $0 < \varepsilon \leq 1$ amounts to find a realisation r^* with $\mathtt{Prob}(D_{r^*}, \phi) \geq (1-\varepsilon) \cdot \max_{r \in \mathcal{R}^{\mathfrak{D}}} \{\mathtt{Prob}(D_r, \phi)\}$.

2.4   Synthesis Costs

As in model repair [5], it is quite natural to associate non-negative integer costs to the various realisation options. This enables distinguishing cheap and expensive repairs. The realisation (aka: repair) costs should not be confused with the concept of rewards in MCs; the latter impose a cost structure on the MC while realisation costs impose costs on the realisation at hand.

Definition 7 (Realisation costs). *For family $\mathfrak{D}$, the function $c : \mathcal{R}^{\mathfrak{D}} \to \mathbb{N}$ assigns to each realisation r of $\mathfrak{D}$ a realisation cost $c(r)$.*

The realisation costs are deliberately defined in a rather abstract manner. Concrete instances may depend on the probability distribution over K, the number of options, weighted combinations thereof, and so forth. By imposing an available budget on the possible realisations, we obtain the following cost-dependent variants of the earlier synthesis problems.

Problem 2 (Cost-constrained synthesis). Let $\mathfrak{D}$ be a family of MCs, φ and ϕ as before, and $B \in \mathbb{N}$ a budget. Consider the synthesis problems:

1. Find a realisation $r \in \mathcal{R}^{\mathfrak{D}}$ with $D_r \models \varphi$ and $c(r) \leq B$.
2. Partition $\mathcal{R}^{\mathfrak{D}}$ with $r \in T$ iff $(D_r \models \varphi$ and $c(r) \leq B)$, and $r \in F$ otherwise.
3. Find $r^* \in \mathcal{R}^{\mathfrak{D}}$ with $r^* = \operatorname*{argmax}_{r \in \mathcal{R}_{\mathfrak{D}}} \{\mathtt{Prob}(D_r, \phi) \mid c(r) \leq B\}$.

```
hole k2 either { 2, 3 }
hole k3 either { 2, 4 }
module encode
s : [0..4] init 0;
s = 0 -> 0.5: s'=1 + 0.5: s'=k2;
s = 1 -> 0.1: s'=0 + 0.9: s'=1;
s = 2 -> 1: s'=k3
s = 3 -> 0.2: s'=3 + 0.8: s'=k3;
s = 4 -> 1: s'=s
endmodule
```

Fig. 3. Toy-encoding of the family in Example 1.

Cost-constrained maximal synthesis does not need to have a solution; therefore $\arg\max \varnothing$ equals undefined. Cost-optimal versions of the synthesis problems are:

Problem 3 (Cost-optimal synthesis). Let $\mathfrak{D}$ be a family of MCs, φ and ϕ as before. We consider the following cost-optimal synthesis problems:

1. Find a realisation $r^* \in T = \{r \in \mathcal{R}^{\mathfrak{D}} \mid D_r \models \varphi\}$ with $c(r^*) = \min_{r \in T}\{c(r)\}$.
2. Find a minimal-cost realisation r^* for the max/min-synthesis problem.

Example 3 (Cost-constrained and cost-optimal synthesis). Consider our running example, and let the cost of a realisation r be the sum of its number of reachable states and their outgoing transitions. That is, $c(r_1) = 8, c(r_2) = 10, c(r_3) = 11$, and $c(r_4) = 11$. For $\phi = \Diamond\{4\}$, and budget $B=10$, cost-constrained max synthesis yields r_2. Lowering the budget B to 9, yields r_1, while for B less than 8, no realisation is found.

2.5   A Program Sketching Language

Probabilistic models are typically specified by means of a high-level modelling language, such as PRISM [27], PIOA [42], JANI [9], or MODEST [7]. Let us briefly describe how the model-based concepts translate to language concepts in the PRISM guarded-command language. The aim is to describe families of MCs, possible constraints on its members, and repair costs in a succinct manner. A (basic) encoding for the family of Example 1 is given in Fig. 3.

A PRISM program consists of one or more reactive modules that may interact with each other. Consider a single module. This is not a restriction as every PRISM program can be flattened into this form. A module has a set of (bounded) variables that span its state space. The possible transitions between states of a module are described by a set of guarded commands of the form:

$$\texttt{guard} \quad \rightarrow \quad p_1 : \texttt{update}_1 \dots \dots + p_n : \texttt{update}_n$$

The guard is a boolean expression over the variables of the module. If the guard evaluates to true, the module can evolve into a successor state by updating its variables. An update is chosen according to the probability distribution given by expressions p_1 through p_n. In every state enabling the guard, the evaluation of these expressions must sum up to one.

A PRISM sketch is a program that may contain "holes". Holes are the unknown parts of the program and can be replaced by one of finitely many options. A hole is of the form:

$$\texttt{hole}\ h\ \texttt{either}\{\,\texttt{expr}_1, \ldots, \texttt{expr}_k\,\}$$

where h is the hole identifier and $\texttt{expr}_i$ is an expression over the program variables. A hole h can be used in commands in a similar way as a constant, and may occur multiple times within a command. To distinguish cheap and expensive options, options within a hole can have a cost:

$$\texttt{hole}\ h\ \texttt{either}\{\,x_1\ \texttt{is}\ \texttt{expr}_1\ \texttt{cost}\ c_1, \ldots, x_k\ \texttt{is}\ \texttt{expr}_k\ \texttt{cost}\ c_k\,\}$$

where option i is named x_i and has associated cost c_i. Costs can be constants or expressions that evaluate to natural numbers. The option names x_1 through x_n can be used to describe constraint on realisations. These propositional formulae over option names restrict hole instantiations, e.g.,

$$\texttt{constraint}(x_1 \vee x_2) \implies x_3$$

requires that whenever the options x_1 or x_2 are taken for some (potentially different) holes, option x_3 (for some hole) is also to be taken.

The family of realisations of a given PRISM program sketch is now obtained by all possible substitutions of holes h by their options x_1 through x_n that satisfy all specified constraints.

3   Counterexample-Guided Synthesis

Enumeration. A straightforward approach to the synthesis problems for finite families of MCs is to just enumerate all realisations and analyse each of them individually. This naive method is practically applicable to small- to medium-sized families only. For more realistic settings, alternative approaches to this baseline are needed. We present two different counterexample-guided approaches: one based on CEGAR [17] and one on CEGIS [36].

A Bird's Eye View on Our Two Approaches. Let us explain the intuition behind the CEGAR and CEGIS approaches towards synthesis. Both approaches successively partition the family $\mathfrak{D}$ into MCs satisfying φ and those refuting φ. Figure 4 illustrates this for a two-dimensional parameter space, each parameter having five possible values. Each cell thus corresponds to a realisation. CEGAR first checks if all realisations satisfy φ on a sound abstraction. Figure 4(a) shows the situation when the verification fails, i.e. it gives an inconclusive result. This can happen either due to the subfamily consisting of both satisfying and refuting realisations, or because the abstraction is too coarse. In the next step, CEGAR refines the family into two subfamilies and establishes e.g., that all members in the subfamily represented by the first two columns refute φ (indicated in
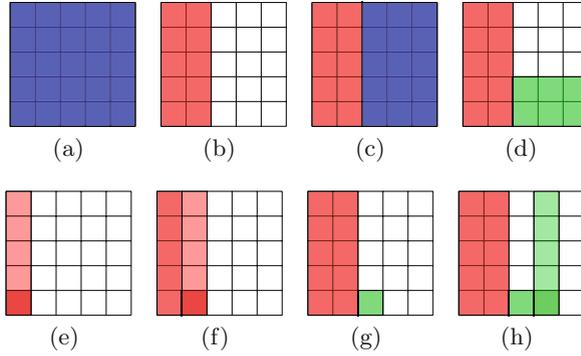
Fig. 4. CEGAR (a)–(d) vs. CEGIS (e)–(h) illustrated. The grid depicts a family with two parameters, each with 5 possible values. Thus, each cell corresponds to a realisation. Blue indicates a verification call that fails, green/red a covered region satisfying/refuting φ. The lighter shaded cells indicate realisations ruled out by counterexample analysis. (Color figure online)

red in Fig. 4(b)), while verifying the remaining subfamily is again inconclusive (Fig. 4(c)). Partitioning that subfamily reveals that the six realisations in the lower two rows fulfil φ. In contrast, CEGIS starts to select a realisation r, e.g., the one in the lower left corner. As $D_r \not\models \varphi$ (indicated dark red in Fig. 4(e)), the counterexample provided by the verifier rules out all realisations in the leftmost column (indicated in lighter shade). This scheme is repeated. In Fig. 4(f), a realisation is selected (second column, lowest row), and similar to the first case, its counterexample rules out all realisations in that column. In Fig. 4(g), a satisfying realisation for φ is selected. In contrast to Fig. 4(g), (h) shows that the analysis of counterexamples to $\neg\varphi$ gives rise to more satisfying realisations. Besides the selected candidate (lowest row), the counterexamples to $\neg\varphi$ cover the entire column.

3.1 CEGAR

We first represent the family $\mathfrak{D}$ by a single *all-in-one* MDP. Selecting action a_r in the (fresh) initial state $s_0^{\mathfrak{D}}$ of the MDP corresponds to choosing the realisation $r \in \mathcal{R}^{\mathfrak{D}}$ and entering the concrete MC D_r. Let us illustrate this with our example.

Figure 5 shows the MDP $M^{\mathfrak{D}}$ for the family $\mathfrak{D}$ from Example 1, where for the sake of readability, only the transitions and states corresponding to realisations r_1 and r_2 are included. Transitions to states (s, r_i) are labeled with action a_{r_i}; these action labels are omitted here. Unreachable states from the initial MDP state $s_0^{\mathfrak{D}}$ are marked grey. There is a one-to-one relationship between a deterministic memoryless scheduler of MDP $M^{\mathfrak{D}}$ and a realisation of $\mathfrak{D}$. Thus, model checking $M^{\mathfrak{D}}$ yields extremal probabilities for all realisations of the family $\mathfrak{D}$.

The MDP model grows linearly with the number of family members. To mitigate the complexity, we apply a simple abstraction where the realisation of

a state in the MDP $M^{\mathfrak{D}}$ is abstracted away, i.e. the item r is ignored in state (s, r). Applying this to our running example amounts to a column-wise grouping of states in Fig. 5. This results in the quotient MDP $M_\sim^{\mathfrak{D}}$ in Fig. 6. By the over-approximation in the quotient MDP $M_\sim^{\mathfrak{D}}$, a scheduler may first choose actions a_r and then $a_{r'}$. This corresponds to switching from realisation r to r'. Such inconsistent regimes result in an MC outside the family $\mathfrak{D}$. There is one-to-one relationship between consistent schedulers—those that globally stick to a single realisation r—and the realisation of $\mathfrak{D}$.
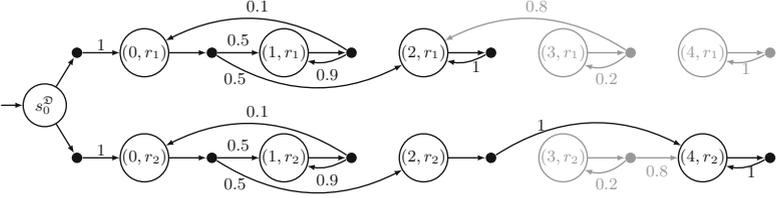

Fig. 5. Reachable fragment of the all-in-one MDP $M^{\mathfrak{D}}$ for realisations r_1 and r_2.
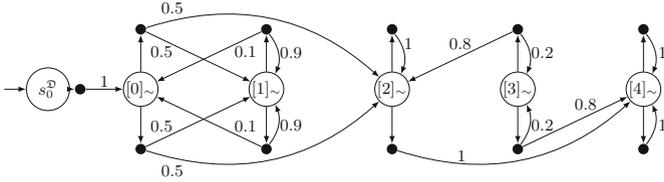

Fig. 6. The quotient MDP $M_\sim^{\mathfrak{D}}$ for realisations r_1 (top actions) and r_2 (bottom actions)

Example 4. Consider the quotient MDP $M_\sim^{\mathfrak{D}}$ in Fig. 6. Transitions from previously unreachable states, marked grey before, are reachable in the quotient. The scheduler σ on the quotient $M_\sim^{\mathfrak{D}}$ that picks a_{r_1} in $[2]_\sim$ and a_{r_2} in state $[3]_\sim$ is inconsistent, as it behaves according to two different realisations r_1 and r_2.

3.2   CEGIS

We follow the typical separation of concerns as in oracle-guided inductive synthesis [1, 22, 23]: a *synthesiser* selects single realisations that have not been considered before, and a *verifier* checks the selected realisation. Let us first focus on the *verifier*. Consider our running example with $\varphi = \mathbb{P}_{\leq 2/5}(\Diamond\{2\})$. Assume the synthesiser picks realisation r_1. The verifier then builds D_{r_1} and determines $D_{r_1} \not\models \varphi$. Observe that the verifier does not need the full realisation D_{r_1} to refute φ. In fact, the paths in the fragment of D_{r_1} in Fig. 7(a) suffice to show that the probability to reach state 2 exceeds $2/5$. Formally, the fragment in Fig. 7(b) is a sub-MC with critical states $C = \{0\}$. The essential property is [41]:

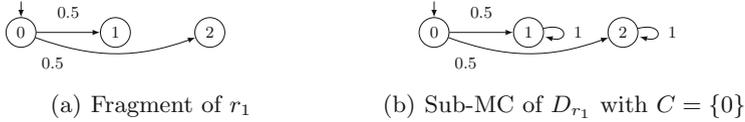*If a sub-MC of a MCD refutes the safety property φ, then D refutes φ too.*

(a) Fragment of r_1 (b) Sub-MC of D_{r_1} with $C = \{0\}$

Fig. 7. Fragment and corresponding sub-MC that suffices to refute φ

Now observe that the considered sub-MC is part of realisation r_2 too. Thus, $D_{r_2} \not\models \varphi$. This can be concluded by considering r_2, $\mathfrak{D}$ and C *without constructing* D_{r_2}. First, take the parameters occurring in $\mathfrak{P}(c)$ for any $c \in C$. This yields k_1 and k_2. The values for the other parameters thus do not affect the shape of the sub-MC induced by C. Realisation r_2 only varies from r_1 in the value of k_3. Therefore, the sub-MC of D_{r_2} induced by C is isomorphic to the sub-MC of D_{r_1} induced by C. This results in concluding $D_{r_2} \not\models \varphi$.

Let us generalise our observations: The verifier gets a realisation r, builds D_r and checks whether $D_r \models \varphi$. If yes, then $\neg\varphi$ is considered to seek for other realisations satisfying φ[1]. Otherwise, some sub-MC of D_r refutes φ. (These counterexamples can be constructed from MC models using techniques from [41], or as snippets of PRISM programs using [20].) The verifier constructs the conflict set C and checks which parameters K' occur on the outgoing transitions of states in C. Each r' with the same parameter values for K' can be immediately refuted, without building the realisation $D_{r'}$. Put in a nutshell, counterexamples are exploited to rule out several realisations in one shot.

The main task that remains for the *synthesiser* is to book-keep the considered and excluded realisations, and, heuristically select realisations that lead to small[2] counterexamples, as these exclude potentially many realisations.

4   Applications

This section illustrates the potential wide applicability of probabilistic model synthesis by providing examples from three different areas: program sketching, software product lines, and controller synthesis for partially observable models.

4.1   Program Sketching

Background. The idea of program sketching [35] is to start with a program sketch, a partial program in which difficult expressions, guards, and statements are left unspecified. The hypothesis of program sketching is that programmers often have an idea about the main control flow of the program but filling in all low-level details is laborious and error prone. Completing these low-level details

---

[1] Note that $\neg\varphi$ is not a safety property, but the presented idea can be extended to liveness properties too.

[2] For some suitable measure of size.

is left to an automated synthesiser. Syntax generators are used to describe a space of the possible code fragments that can be used to complete the program. The synthesised program has to satisfy the specification φ. Program sketching has been successfully applied to e.g., scientific programs and concurrent data structures [1]. We show how our approaches can be applied to sketching of probabilistic programs.

Concrete Challenge. Our program sketch is describing a *dynamic power manager (DPM)*, a key component in dynamically optimising energy consumption [6]. A DPM controls changing the system's power states at run time. Depending on workload and performance constraints, it issues commands (e.g., go into sleep mode, wake up) to the system. We consider a DPM system with two request priorities, low and high; these priorities typically depend on the time-criticality. Requests are placed in finite buffers (based on their priority), provided the buffers are not full. Otherwise, the requests are lost. A similar DPM has been analysed by probabilistic model checking [34].

Problem Statement. Our goal is to *synthesise a DPM program* that decides to switch power state based on the current workload expressed in terms of the occupancy of the low-priority and high-priority request buffers.

Approach. The starting point is a program sketch that includes partially specified commands of the form:
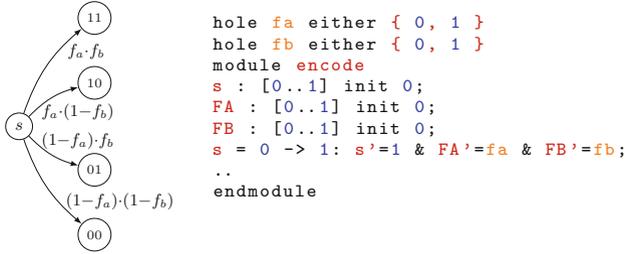
$$g_H \ \& \ g_L \quad \longrightarrow \quad 1 : state' = X$$

where g_H and g_L are partially specified guards concerning the low-priority and high-priority request buffer, respectively, and hole X represents an unknown update of the DPM state. Possible code fragments to complete the commands are e.g., in $state = 1$, the DPM sends a control signal to switch to an active state, while g_H (and similarly g_L) indicates that the occupancy is within a given interval, e.g., at most 50%. We synthesise the guards and updates such that the resulting DPM control program meets a conjunction of objectives (inspired by [22]) that constrain the expected number of lost low- and high-priority requests and the expected energy consumption, for different thresholds λ imposed on these expected values.

Results. The MC family has over $3 \cdot 10^5$ realisations, i.e., control programs. The average realisation has more than 5000 states. We consider an unsatisfiable conjunction of 3 properties describing a possible DPM specification. Within 20 min, the conjunction is shown to be unsatisfiable (although each property alone is satisfiable). An enumerative approach takes more than 20 h to show this. For a satisfiable conjunction, we find a realisation within minutes.

4.2   Software Product Lines

Background. A software product line is (according to wikipedia) "a set of software-intensive systems that share a common, managed set of features

```
hole fa either { 0, 1 }
hole fb either { 0, 1 }
module encode
s  : [0..1] init 0;
FA : [0..1] init 0;
FB : [0..1] init 0;
s = 0 -> 1: s'=1 & FA'=fa & FB'=fb;
..
endmodule
```

(a) Parametric MC (b) The PRISM encoding for the family

Fig. 8. Translating a parametric MC formulation to an encoding of a family of MCs.

satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". Products in a software product line have different features which can be understood as functionalities changing the behaviours of a core software system. They thus provide an elegant way to specify families of systems: every member of the family comprises the core system and a combination of features. Randomness appears when modelling energy consumption or failure probabilities.

Concrete Challenge. We consider the BSN (Body Sensor Network) software product line benchmark from [33], the largest benchmark analysed by probabilistic model checking with the ProFeat tool [16]. BSN describes a network of connected sensors that send measurements to a unit identifying health-critical situations. The family contains the various configurations of 10 binary features, that is, whether a sensor is available or not. We are interested in the reliability of the system, that is, in the probability that the system behaves as described. The system is described as a parametric Markov chain:

Example 5. We consider a variation point (a state whose future behaviour depends on the features) where depending on the availability of features F_a, F_b the model behaves differently. For each feature, a Boolean parameter f is 1 if the feature is active and 0 otherwise. At a variation point, the probability of every transition is scaled by factor p, which equals f if the feature enables the transition and $1-f$ otherwise. This results in parametric MC in Fig. 8(a).

Problem Statement. Find all features combinations where the induced system does not meet a certain reliability.

Approach. The formulation in [33] is a parametric MC, and therefore seems amenable to standard parameter synthesis in which probabilities have to be synthesised. However, in absence of certain features, transitions are taken with probability zero. Traditional parameter synthesis techniques do not allow for such assignments. We translate the parametric MC into a PRISM-description of a family, as illustrated in the following example:

Example 6. We adapt the encoding in Fig. 8(a) to an encoding in Fig. 8(b).

Results. Though this is the largest product line example used in [16], verifying a family with just 1024 family members, with an average size of the realisation of roughly 100 states is mostly trivial. Within seconds, we can categorise the different realisations based on their reliability, either by our approaches or by enumeration.

4.3 Controller Synthesis in Partially Observable Systems

Background. As a next application, we consider controller synthesis (aka: scheduler synthesis) in partially-observable MDPs (POMDPs, for short). A POMDP [25] is an MDP in which an observation $o(s)$ is associated with every state s. POMDP controllers do not have access to the current state of the POMDP; instead, they can only use the observations of the visited states. Thus, whereas an MDP scheduler bases its decisions on finite paths of the form $\pi = s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} s_n$, a POMDP controller does so using the observation sequence $o(\pi) = o(s_0) \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} o(s_n)$. Several paths in the underlying MDP M may give rise to the same observation sequence. Controllers have to take this restricted observability into account: They cannot distinguish paths with the same observation sequence. Controller synthesis for POMDPs is notoriously hard: Finding an optimal strategy is undecidable [12] and finding an optimal memoryless strategy is already NP- and SQRT-SUM hard [39]. The complexity of the problem makes the possibility to guide the search for a strategy by means of synthesis very interesting.

Concrete Challenge. We consider Maze, a classical motion planning problem considered as POMDP, see e.g., [29]. A robot is put in a maze with paths surrounded by walls, and its aim is to go to a goal position in the maze. The problem is partially observable because the robot cannot perceive its true location, but only the presence or the absence of a wall on either side of its current position. There is a non-zero probability of slipping, so that the robot does not always know if its last attempt to make a move had any consequence on its actual position in the maze.

Problem Statement. The objective is to *synthesise a deterministic finite-state controller* [11,29] for a Maze (of different size) with a bounded number of states that minimises the expected time for the robot to reach the goal.

Approach. To cast this POMDP problem in our framework of families of MCs, we adapt a recent result [24] that established a one-to-one correspondence between finding finite memory randomised controllers in POMDPs and satisfying parameter valuations in parametric MCs. We sketch a controller by restricting the memory to a fixed bound. Costs are used to penalise the complexity of the controllers such that simple, i.e., easy implementable, finite-memory controllers

result. The family describes all MCs induced by small-memory observation-based deterministic strategies with a fixed upper bound on their amount of memory. We are interested in the expected time to the goal. (This problem can be formalised by adding rewards to MDPs in the usual way.)

Results. The MC family has a bit more than 10^6 realisations, i.e., observation-based strategies. The average realisation has 134 states. Among other results, within seconds, we find the 4 strategies that were at most 2% off the maximum. In comparison, an enumeration-based approach takes several hours, and enumerating all consistent strategies of the quotient (see Sect. 3.1) takes more than an hour.

5   Epilogue

Summary. This paper outlined two techniques for the automated synthesis of finite-state probabilistic models or programs. The CEGAR approach takes as a starting point an abstract representation of a family of Markov chains and exploits inconsistent policies—policies that switch between different realisations—to iteratively refine the design space. The CEGIS approach exploits critical subsystems as counterexamples and uses SMT techniques to prune the design space by analysing the counterexamples. We foresee a wide applicability of these kind of synthesis techniques; we illustrated this by examples from program sketching, controller synthesis, and software product lines. Both techniques significantly outperform a naive enumerative approach and differ substantially from the few existing approaches to synthesising probabilistic programs [10,31]. CEGAR works particularly well if the quotient MDP is succinct while CEGIS excels the "more unsatisfiable" the synthesis problem is. CEGAR has difficulties treating constraints on family members (which are straightforward with CEGIS), whereas the performance of CEGIS significantly drops for synthesis problems for which the threshold is close to the true reachability probability.

Future Work. The approaches in this paper are first stepping stones towards the automated synthesis of probabilistic models. This topic has plenty of interesting directions for future work. This includes synthesising infinite-state probabilistic programs, integrating efficient parameter synthesis and model synthesis, developing adequate modeling formalisms for families of probabilistic models, and learning algorithms for probabilistic models [28,40].

methods to new applications such as cardiac devices and, more recently, bird flocking. His work has been (and still is) an enormous source of inspiration. This paper celebrates his (and his co-authors') work on model repair for probabilistic models and illustrates how tweaking probabilities (as in model repair) can be generalised towards synthesising model structures. Happy birthday, Scott!

References

1. Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Commun. ACM **61**(12), 84–93 (2018)
2. Aziz, A., Singhal, V., Balarin, F., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: It usually works: the temporal logic of stochastic systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 155–165. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60045-0_48
3. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. Handbook of Model Checking, pp. 963–999. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Bartocci, E., Grosu, R., Katsaros, P., Ramakrishnan, C.R., Smolka, S.A.: Model repair for probabilistic systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 326–340. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_30
6. Benini, L., Bogliolo, A., Paleologo, G.A., Micheli, G.D.: Policy optimization for dynamic power management. IEEE Trans. CAD Integr. Circuits Syst. **18**(6), 813–833 (1999)
7. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MODEST: a compositional modeling formalism for hard and softly timed systems. IEEE Trans. Softw. Eng. **32**(10), 812–830 (2006)
8. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing model checking in verification by AI techniques. Artif. Intell. **112**(1–2), 57–104 (1999)
9. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 151–168. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_9
10. Chasins, S., Phothilimthana, P.M.: Data-driven synthesis of full probabilistic programs. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, Part I. LNCS, vol. 10426, pp. 279–304. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_14
11. Chatterjee, K., Chmelik, M., Davies, J.: A symbolic SAT-based algorithm for almost-sure reachability with small strategies in POMDPs. In: AAAI, pp. 3225–3232. AAAI Press (2016)
12. Chatterjee, K., Chmelik, M., Tracol, M.: What is decidable about partially observable Markov decision processes with ω-regular objectives. J. Comput. Syst. Sci. **82**(5), 878–911 (2016)
13. Chatzieleftheriou, G., Bonakdarpour, B., Katsaros, P., Smolka, S.A.: Abstract model repair. Log. Methods Comput. Sci. **11**(3) (2015)
14. Chen, T., Hahn, E.M., Han, T., Kwiatkowska, M.Z., Qu, H., Zhang, L.: Model repair for Markov decision processes. In: TASE, pp. 85–92. IEEE (2013)

15. Chonev, V.: Reachability in augmented interval Markov chains. CoRR arXiv:1701.02996 (2017)

16. Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: ProFeat: feature-oriented engineering for family-based probabilistic model checking. Formal Asp. Comput. **30**(1), 45–75 (2018)

17. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15

18. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.: Model checking software product lines with SNIP. STTT **14**(5), 589–612 (2012)

19. Cubuktepe, M., Jansen, N., Junges, S., Katoen, J.-P., Topcu, U.: Synthesis in pMDPs: a tale of 1001 parameters. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 160–176. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_10

20. Dehnert, C., Jansen, N., Wimmer, R., Ábrahám, E., Katoen, J.-P.: Fast debugging of PRISM models. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 146–162. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_11

21. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruintjes, H., Katoen, J.-P., Ábrahám, E.: PROPhESY: a PRObabilistic ParamEter SYnthesis tool. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 214–231. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_13

22. Gerasimou, S., Tamburrelli, G., Calinescu, R.: Search-based synthesis of probabilistic models for quality-of-service software engineering (T). In: ASE, pp. 319–330. IEEE Computer Society (2015)

23. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1–2), 1–119 (2017)

24. Junges, S., et al.: Finite-state controllers of POMDPs using parameter synthesis. In: UAI, pp. 519–529. AUAI Press (2018)

25. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1–2), 99–134 (1998)

26. Katoen, J.P.: The probabilistic model checking landscape. In: LICS, pp. 31–45. ACM (2016)

27. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47

28. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning deterministic probabilistic automata from a model checking perspective. Mach. Learn. **105**(2), 255–299 (2016)

29. Meuleau, N., Kim, K., Kaelbling, L.P., Cassandra, A.R.: Solving POMDPs by searching the space of finite policies. In: UAI, pp. 417–426. Morgan Kaufmann (1999)

30. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Program. Lang. Syst. **18**(3), 325–353 (1996)

31. Nori, A.V., Ozair, S., Rajamani, S.K., Vijaykeerthy, D.: Efficient synthesis of probabilistic programs. In: PLDI, pp. 208–217. ACM (2015)

32. Pathak, S., Ábrahám, E., Jansen, N., Tacchella, A., Katoen, J.-P.: A greedy approach for the efficient repair of stochastic models. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 295–309. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_21

33. Rodrigues, G.N., et al.: Modeling and verification for probabilistic properties in software product lines. In: HASE, pp. 173–180. IEEE (2015)

34. Sesic, A., Dautovic, S., Malbasa, V.: Dynamic power management of a system with a two-priority request queue using probabilistic-model checking. IEEE Trans. CAD Integr. Circuits Syst. **27**(2), 403–407 (2008)

35. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013)

36. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: PLDI, pp. 281–294. ACM (2005)

37. Češka, M., Hensel, C., Junges, S., Katoen, J.P.: Counterexample-driven synthesis for probabilistic program sketches. CoRR abs/1904.12371 (2019)

38. Češka, M., Jansen, N., Junges, S., Katoen, J.-P.: Shepherding hordes of Markov chains. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 172–190. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_10

39. Vlassis, N., Littman, M.L., Barber, D.: On the computational complexity of stochastic controller optimization in POMDPs. TOCT **4**(4), 12:1–12:8 (2012)

40. Wang, J., Sun, J., Yuan, Q., Pang, J.: Learning probabilistic models for model checking: an evolutionary approach and an empirical study. STTT **20**(6), 689–704 (2018)

41. Wimmer, R., Jansen, N., Ábrahám, E., Katoen, J., Becker, B.: Minimal counterexamples for linear-time probabilistic verification. Theor. Comput. Sci. **549**, 61–100 (2014)

42. Wu, S., Smolka, S.A., Stark, E.W.: Composition and behaviors of probabilistic I/O automata. Theor. Comput. Sci. **176**(1–2), 1–38 (1997)