https://doi.org/10.1007/s00165-021-00547-2
BCS © 2021
Formal Aspects of Computing (2021) 33: 637–667

Formal Aspects of Computing

Counterexample-guided inductive synthesis for probabilistic systems

Milan Češka[1], Christian Hensel[2], Sebastian Junges[3], Joost-Pieter Katoen[2]

[1] Brno University of Technology, Brno, Czech Republic
[2] RWTH Aachen University, Aachen, Germany
[3] University of California, Berkeley, CA, USA

Abstract. This paper presents counterexample-guided inductive synthesis (CEGIS) to automatically synthesise probabilistic models. The starting point is a family of finite-state Markov chains with related but distinct topologies. Such families can succinctly be described by a sketch of a probabilistic program. Program sketches are programs containing holes. Every hole has a finite repertoire of possible program snippets by which it can be filled. We study several synthesis problems—feasibility, optimal synthesis, and complete partitioning—for a given quantitative specification φ. Feasibility amounts to determine a family member satisfying φ, optimal synthesis amounts to find a family member that maximises the probability to satisfy φ, and complete partitioning splits the family in satisfying and refuting members. Each of these problems can be considered under the additional constraint of minimising the total cost of instantiations, e.g., what are all possible instantiations for φ that are within a certain budget? The synthesis problems are tackled using a CEGIS approach. The crux is to aggressively prune the search space by using counterexamples provided by a probabilistic model checker. Counterexamples can be viewed as sub-Markov chains that rule out all family members that share this sub-chain. Our CEGIS approach leverages efficient probabilistic model checking, modern SMT solving, and program snippets as counterexamples. Experiments on case studies from a diverse nature—controller synthesis, program sketching, and security—show that synthesis among up to a million candidate designs can be done using a few thousand verification queries.

Keywords: Program Synthesis; Markov Chains; Probabilistic Model Checking; Counterexamples; CEGIS

1. Introduction

Uncertainties and probabilities Computing systems nowadays have to deal with uncertainties of various forms. Tons of data need to be treated, often from unreliable sources such as noisy sensors or the internet. The environment of embedded computing systems is continuously subject to unpredictable changes. Illustrative examples include robots and autonomous vehicles. More and more computing system modules incorporate machine learning mechanisms whose reasoning is inherently probabilistic. This entails that the traditional qualitative notion of system correctness—a system is correct or not—becomes blurred. Quantifying (and minimising) the probability

Correspondence to: Sebastian Junges, E-mail: sebastian.junges@ru.nl of encountering an anomaly or unexpected behaviour becomes crucial. This insight has led to a growing interest in probabilistic models and programs in computing. Henzinger [Hen13] for instance argues that "the Boolean partition of software into correct and incorrect programs falls short of the practical need to assess the behaviour of software in a more nuanced fashion [...]." In his ASE 2016 keynote [Ros16], Rosenblum advocates taking a more probabilistic approach in software engineering. Concrete examples include the quantitative analysis of software product lines [GS13, VK13, RAN+15, CDKB18, VtBLL18, LCA+18], the synthesis of probabilities for adaptive software [CvG+18, CGJ+16], and probabilistic verification at runtime to support verifying dynamic reconfigurations [CGKM12, FTG16].

Synthesis of probabilistic models and programs The development of computing systems under uncertainty is non-trivial. Probabilistic programs are a prominent formalism to deal with uncertainty. Their main random component is a random choice among several statements, e.g., assignments. Unfortunately, designing such programs is rather intricate. A skeleton of the main control flow has to be completed with detailed, often unknown information such as the exact probability distributions over statements or the final bits of the control structure. Their development thus requires quantitative reasoning over, often a huge number of, alternative designs. For finite-state programs, a possible remedy is to take the underlying operational model of a candidate program—typically a *Markov chain*—and check it using an automated technique such as probabilistic model checking [Kat16, BdAFK18]. This amounts to verify each possible program in the design space. Alternatively, one can apply model checking directly on some suitable representation of the entire (finite) design space [CDKB18, ČJJK19].

To synthesise probabilities—the transition labels in the operational model—automated techniques such as parameter synthesis [HHZ11, ČDP+17, QDJ+16] and model repair [BGK+11, CHH+13] have been successful. They however only allow to amend or infer transition probabilities, whereas the control structure—the topology of the underlying probabilistic model—is fixed. Adding or removing transitions is not admitted. That is to say, every possible amendment keeps the topology of the Markov chain invariant. The topic of this paper is to allow for changing the Markov chain's topology. In an automated manner. More precisely, 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, realisations may affect the control structure as well as the probabilistic choices. The aforementioned parameter synthesis techniques are inadequate for this problem, as they restrict symbolic 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 [Cho17].

We describe the possible Markov chains by so-called *families*, finite sets of finite-state Markov chains. Family members, i.e., Markov chains, are defined over the same set of states. The allowed topology changes within a family are described by parameters in the Markov chain family description. A naive approach would be to verify each family member separately so as to find a member (if any) satisfying a given specification. The aim of this paper is to do this is a more efficient manner. At the syntactic level, families correspond to probabilistic programs with "holes". Every hole has a finite repertoire of possible program snippets by which it can be filled. This is described at the model level by the parameters of a family. Program sketching [Sol13] naturally fits within this setting. Program sketches succinctly describe the design space by providing the program-level structure but leaving some parts like command guards or variable assignments unspecified.

Problem statement This paper considers the *automated synthesis of finite-state probabilistic models and programs*. The input is a program sketch—a probabilistic program with holes where each hole can be replaced by finitely many options—, a set of quantitative specifications that the program needs to fulfil, and a budget. Quantitative specifications include imposing thresholds on reachability probabilities as well as on expected cumulative rewards. All possible realisations have a certain cost and the synthesis provides a realisation that fits within the budget. (Costs of realisations should not be confused with rewards in a Markov chain; rewards are earned on runs of a Markov chain while costs of realisations refer to the expenses of a certain instantiation of a program sketch. For instance, an assignment is considered cheaper as a more involved control structure.) We study several synthesis problems—*feasibility, optimal synthesis, and complete partitioning*—for a given quantitative specification φ. Feasibility amounts to determine a family member satisfying φ, optimal synthesis amounts to find a family member that maximises the probability to satisfy φ, and complete partitioning splits the family into satisfying and refuting members. Each of these problems can be considered under the additional constraint of minimising the total cost of instantiation, e.g., what are all possible instantiations for φ that are within a certain budget?

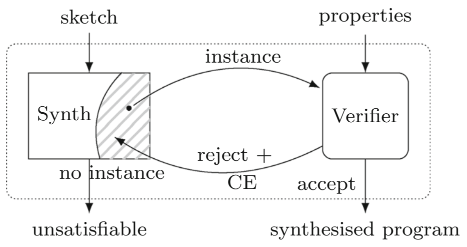


Fig. 1. CEGIS for probabilistic model and program synthesis.

Counter-Example-Guided Inductive Synthesis To tackle the aforementioned synthesis problems, we tailor and generalise the paradigm of *CounterExample-Guided Inductive Synthesis* (CEGIS, cf. Fig. 1) [SLTB+06, ABD+15, SRBE05, ADK+18] to finite-state probabilistic models and programs. An overview of the approach is provided in Fig. 1. Let us explain this process at the family level. Starting from a family, a candidate realisation D is selected and discharged to a verifier. Using off-the-shelf probabilistic model-checking techniques [Kat16, BdAFK18], it verifies whether $D \models \varphi$. If indeed $D \models \varphi$, then a solution is found. If $D \not\models \varphi$, a counterexample (CE) is generated. This CE is a fragment [WJÁ+14] D' of the realisation D such that D' refutes φ. One may consider D' as a core of the family member D that suffices to show that $D \not\models \varphi$.

The key step that the CE D' is exploited by an SMT (satisfiability modulo theory)-based synthesiser to rule out potentially many realisations (indicated by the dashed area in Fig. 1)—rather than just realisation D—at once. Stated somewhat simplified, all realisations that have CE D' as a fragment can be pruned as they all violate the specification φ. This synthesis-verification loop is repeated until either a satisfying realisation is found or the entire family is pruned implying the absence of a realisation $D \models \varphi$. The CEGIS approach iteratively partitions a family into "good", "bad" and inconclusive realisations. It starts with a single candidate realisation, and rules out several realisations by effectively exploiting counterexamples. In the syntax-based setting, CEGIS starts with a program sketch, a program with holes, and iteratively search for good—or even optimal—instantiations of these holes.

Challenges and solutions Tailoring CEGIS to probabilistic programs imposes two fundamental challenges:

- (a) CEs are structurally more complicated than for deterministic programs, and
- (b) verifying a proposed program requires numerical computations.

CEs for deterministic programs are mostly single paths: a finite path ending in a bad state suffices to show the violation of a safety property. In the probabilistic setting, CEs need to carry quantitative information and are no longer single paths but sets of paths [HKD09]. This makes the pruning process computationally more demanding and less effective compared to standard CEGIS. Whereas CEGIS fully relies on SMT techniques, in our setting, SMT encodings of the underlying decision problems require numerical computation which is computationally harder. We tackle these challenges by (a) adopting the notion of *program-level CEs* — a core sub-set of program commands causing a violation of the quantitative specification — for probabilistic programs [DJW+14, WJV+15], and (b) integrating a unique SMT-based pruning strategy with efficient numerical verification techniques [KNP11, DJKV17]. Synthesis requires a generalisation of program-level CEs, viz. sub-programs, such that all CE extensions adhering to the sketch violate the specification and thus can be safely abandoned as candidate program.

Outcomes To summarise, this paper presents a novel synthesis framework for probabilistic programs that adhere to a given set of quantitative specifications and a given budget. Programs are represented in the PRISM modelling language [KNP11], a probabilistic extension of reactive modules. Program sketches are PRISM modules with holes. Specifications are expressed in PCTL (Probabilistic Computational Tree Logic) extended with rewards, a standard temporal logic in probabilistic model checking [KNP11, DJKV17]. We formally define counterexamples (CEs) for PRISM programs and extend a MaxSat approach to obtain such program-level CEs as a by-product of the program verification. Our CEs are similar to program-level CEs [DJW+14, WJV+15], and allow for a flexible sketching language. *To the best of our knowledge, this is the first CEGIS approach for probabilistic programs.* To set the scene, in the first part of the paper we consider the synthesis problem on the underlying operational model of PRISM programs, finite Markov chains. We use families of Markov chains as operational counterpart of program sketches and detail a CEGIS-style algorithm on these families. CEs at this operational level are sub-graphs of

Markov chains. The second part of the paper shows how to generalise this approach on families to reason on probabilistic programs with holes.

The CEGIS approach is *sound and complete*: either an admissible program does exist and it is computed, or no such program exists and the algorithm reports this. We report on a prototypical implementation built on top of the model checker Storm [DJKV17] using the well-known SMT-tool Z3 [dMB08]. Experiments on case studies from a diverse nature—controller synthesis, program sketching, and security—demonstrate that synthesis among up to a million realisations can be achieved by a few thousand verification queries. This results in a significant speed-up compared to a baseline approach in which each individual realisation is verified.

Organisation of the paper The rest of this paper is organised as follows. Section 2 presents the necessary background on PRISM programs, Markov chains, and quantitative specifications. Section 3 defines the concept of program sketches for PRISM programs, describes possible applications thereof, and formalises the various synthesis problems considered in this paper. Section 4 introduces families of Markov chains and describes their relationship to program sketches. Section 5 describes our CEGIS approach applied to families of Markov chains. It in particular describes the use of counterexamples, and presents the verifier and the synthesiser algorithms. Section 6 details how the CEGIS approach can be lifted to the program level. This includes a description of program-level counterexamples, the program verifier and program synthesiser. It includes details on the used SMT encodings and the usage of MaxSat for CE generation. Section 7 reports on our experimental validation using a prototypical implementation. Our evaluation focuses on comparing CEGIS to a baseline approach in which all family members are verified, and on identifying the decisive factors that influence synthesis times. Section 8 wraps up and includes some pointers to future work.

This paper extends our publication in [ČHJK19] by discussing CEGIS in greater detail. In particular, we provide more background on families and program sketches, give additional illustrating examples, and add sections on the necessary adaptions to achieve efficiency, and a comparison to an alternative approach based on counterexample-guided abstraction refinement [ČJJK19]. The extended material is partially based on [Jun20].

Related work *CEGIS approaches.* CE guided inductive synthesis (CEGIS) has been introduced as an SMT-based synthesiser for sketches [SLTB+06]. Due to enormous improvements in SMT solving in the last decade, CEGIS has been successfully applied to find programs for various challenging problems [SRBE05, SLJB08]. Meta-sketches for optimal and quantitative synthesis in a deterministic setting have been considered in, e.g., [ČCH+11, CCS14, BTGC16]. Sketches together with likelihood computation have been used to find probabilistic programs that best match available data [NORV15]. Gerasimou *et al.* [GTC15] presented an approach resembling ours. It synthesises probabilistic systems from specifications and parametric templates. The key difference is that our approach prunes the design space by using counterexamples. Instead, [GTC15] leverages evolutionary optimisation techniques without pruning. The evolutionary algorithms, driven by a fitness function capturing the satisfaction probability, can get stuck in local minima and may thus miss the correct or optimal solution. Completeness is thus only achieved by exploring all designs.

Markov chain families. Modal transition systems [LT88, AHL+08, Kre17] are a prominent representation of sets of alternative designs. Parametric modal transition systems [BKL+15] allow for similar dependencies as in program-level sketches. Synthesis for these modal transition systems has been considered in [BKL+12]. Probabilistic extensions of modal transition systems have been considered in, e.g. [DKL+13]. Their focus was on establishing a specification theory, and not on synthesis.

Verifying model families. An efficient verification procedure of qualitative properties on families of transition systems has recently been considered for LTL specifications by exploiting model-specification dependencies [DR18]. Model checking various related Markov models has been considered in the context of the quantitative verification of software product lines [GS13, VK13, RAN+15, LCA+18]. The typical approach is to analyse all individual designs (aka: product configurations) or build and analyse a single, so-called *all-in-one*, Markov decision process (MDP) describing all the designs simultaneously. These techniques have been integrated into the software tools ProFeat [CDKB18] and QFLan [VtBLL18]. The scalability of these approaches is limited even with using symbolic methods. An incomplete method in [JHTT19] employs abstraction targeting a particular case study. An abstraction-refinement scheme has recently been explored in [ČJJK19]. It iteratively analyses an abstraction of a (sub)set of designs—it is an orthogonal and slightly restricted approach to the inductive method presented here. Detailed differences are discussed later in this paper.

Parameter synthesis. A related approach is parameter synthesis in parametric Markov chains, in which transitions are labeled with functions such as p and $1-p$ over parameters. Successful approaches for the feasibility problem in this setting have been proposed in, e.g., [ČDP+17, QDJ+16]. These works identify transition parameters for which the instantiated models satisfy a quantitative specification. An extension to handle parameters affecting transition probabilities (rates) has been integrated into the evolutionary-driven synthesis of robust systems [CvG+17a, CvG+18] and is available in the software tool RODES [CvG+17b]. Model repair [BGK+11, CHH+13] takes a Markov model that violates a specification and automatically adjusts its transition probabilities to produce a "repaired" model that meets the specification.

SMT and CEs in synthesis. SMT-based encodings for synthesis in Markov models or versions thereof for population models have been used in, e.g. [JJD+16, CČF+17]. These encodings are typically monolithic: they do not prune the search space via CEs. Probabilistic CEs have been recently used to ensure that MDP controllers obtained via learning from positive examples meet a given safety specification [ZL18]. Their CE-guided approach ensures the resulting strategy to be close to the unknown expert strategy.

2. Preliminaries

Programs Probabilistic programs provide a concise modelling formalism in the development of systems under uncertainty. We briefly introduce the PRISM-language as an example of such a language and illustrate our program synthesis approach using PRISM programs. Alternative programming notations include MODEST [HH14], a compositional process-algebraic language that is aimed at describing distributed systems, PGCL [MMS96], an imperative while-language with coin flips as primitives, and JANI [BDH+17], an expressive intermediate language aimed to enable the use of different software tools as back-end for a variety of other domain-specific languages. The synthesis techniques presented in this paper can be easily adapted to these languages[1]. Various models in these languages are available from the quantitative verification benchmark set [HKP+19].

PRISM programs We briefly describe the PRISM guarded-command language. A PRISM program consists of one or more reactive modules that may interact with each other. To ease the presentation, we consider a single *module*. This simplification is not a restriction as every PRISM program can be flattened into this form. Flattening results in programs that are more involved to analyse in an automated manner, but simplifies the technical explanation. A module contains a set of bounded variables, the valuations of these variables span the state space of the underlying model. Transitions between states are described by guarded *commands* of the form:

$$\mathtt{g} \ \rightarrow \ p_1 : \mathtt{up}_1 + \ldots\ldots + p_n : \mathtt{up}_n$$

The *guard* g is a Boolean expression over the module's variables. If the guard g evaluates to `true` (for a given variable valuation), then the module evolves into one of the n successor states by updating its variables. This is accomplished in a program execution by selecting an *update* up according to the probability distribution given by the expressions $p_1, \ldots, p_n$. Every update is an assignment describing how (a subset of) the variables (the left-hand side of the assignment) are reassigned based on an expression (the right-hand side). In every state enabling the guard g, the evaluation of $p_1, \ldots, p_n$ must sum up to one. This ensures a probability distribution over the possible updates. Roughly speaking, a *program* $\mathcal{L}$ thus is a tuple (Var, Cmd) of variables and commands.

Definition 1 (Markov chain) A (*discrete-time*) *Markov chain* (*MC*) is a tuple $D := (S, \iota, P, \mathsf{rew})$ where S is a set of *states*, $\iota \in S$ is an *initial state*, $P : S \rightarrow Distr(S)$ is a *transition probability* function[2], and $\mathsf{rew} : S \rightarrow \mathbb{R}_{\geq 0}$ are finite *state rewards*.

The operational semantics of the program $\mathcal{L}$ is given by a finite discrete-time Markov Chain (MC, for short) denoted $[\![\mathcal{L}]\!]$. The details of this semantics are outside the scope of this paper and can be found in [KNP11]. We explain a few details and then illustrate the semantics by an example. To obtain a direct mapping between program $\mathcal{L}$ and the underlying MC, we have to ensure that every reachable state of the program satisfies exactly one command. The operation fixdl takes a program and adds commands that introduce self-loops for states

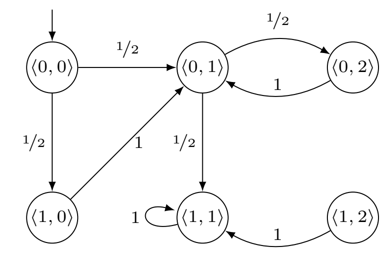
[1]In particular, the implementation of our techniques (cf. Sect. 7.1) is based on JANI, and also supports PRISM.

[2]For Markov chains, we denote $P(s)(s')$ with $P(s, s')$.

```
module rex
x : [0..1] init 0;
y : [0..2] init 0;

x = 0 & y < 2 -> 0.5: x'=1 + 0.5: y'=y+1;
y = 2 -> 1: y'=y-1;
x = 1 & y != 2 -> 1: x'=y & y'=x;
endmodule
```

(a) PRISM program



(b) Underlying model after fixing deadlocks.

Fig. 2. PRISM program $\mathcal{L}$ with underlying model $[\![\, \mathsf{fixdl}(\mathcal{L}) \,]\!]$

without enabled guard: the operation thus *fixes deadlocks*. We disallow *overlapping guards*, that is, every two different guards have to be disjunctive[3]. Assignments may violate resource bounds, contain divisions-by-zero, etc. Programs that contains such assignments are *ill-defined*, as opposed to *well-defined programs* that do not contain such assignments and that do not contain overlapping guards.

Example 1 Figure 2a shows a sample PRISM program. The program contains the variables x and y, where x is either zero or one, and y ranges between zero and two. In total, there are thus 6 different variable valuations, i.e., states. We denote states as tuples with the x- and y-value. The MC (after fixing deadlocks) is illustrated in Fig. 2a. From state $\langle 0, 0\rangle$, (only) the first guard is enabled. Thus there are two transitions, each with probability a half: one transition in which x becomes one and one transition in which y is increased by one. We see a similar pattern for state $\langle 0, 1\rangle$ which also enables the first command. There is no guard enabled in state $\langle 1, 1\rangle$, therefore the fixdl operation adds a command to the program that produces a self-loop.

Specifications Quantitative properties of probabilistic programs can be specified using a probabilistic temporal logic such as PCTL [BK08]. For simplicity, we focus on reachability and expected state reward properties. A reachability property has the following form: $\varphi := \mathbb{P}_{\sim\lambda}(\lozenge T)$ for a set $T \subseteq S$ of target states, threshold $\lambda \in [0, 1] \subseteq \mathbb{R}$, and comparison relation $\sim\ \in \{<, \leq, \geq, >\}$. Let $Pr_D(\lozenge T)$ denote the probability to reach T from the initial state ι in MC D. Then, $D \models \varphi$ if $Pr_D(\lozenge T) \sim \lambda$. A *specification* is a set $\Phi = \{\varphi_i\}_{i \in I}$ of properties, and $D \models \Phi$ if $\forall\, i \in I.\ D \models \varphi_i$. We call upper-bounded properties (with $\sim\ \in \{<, \leq\}$) *safety properties*, and lower-bounded properties *liveness* properties[4]. The expected state reward property $\varphi := \mathbb{E}_{\sim\lambda}(\lozenge T)$ with $\sim \in \{<, \leq\}$ for a set $T \subseteq S$ (where $Pr_D(\lozenge T) = 1$) and $\lambda \in \mathbb{R}_{\geq 0}$ is satisfied in state ι if

$$\sum_{r=0}^{\infty} r \cdot Pr_D(\pi = s_0\, s_1\, \ldots\, s_n \mid s_0 = \iota \wedge s_n \in T \wedge \mathsf{rew}(\pi) = r) \sim \lambda$$

where $Pr_D(\pi)$ denotes the probability of finite path π and $\mathsf{rew}(s_0\, s_1\, \ldots\, s_n) = \sum_{i=0}^{n} \mathsf{rew}(s_i)$. The interpretation of φ on a probabilistic program is naturally carried over to the MC describing its operational semantics. That is, a program $\mathcal{L}$ satisfies a specification Φ if and only if $[\![\, \mathcal{L} \,]\!] \models \Phi$.

3. Synthesis for probabilistic program sketches

In this section, we introduce sketches of probabilistic programs and the syntax-guided synthesis problems considered in this paper.

3.1. What are program sketches?

Intuitively speaking, a program *sketch* [Sol13] is a syntactic template defining a high-level structure of an underlying operational model. It concisely represents a priori knowledge about the system under development.

[3]Some tools permit overlapping guards and resolve the resulting nondeterminism by considering a uniform distribution over the guards.
[4]The correspondence to the standard qualitative, path-based notion of a safety property is that the safety properties can be refuted by a set of finite paths.

```
hole HX either { XA is 1 cost 3, 2}
hole HY either { YA is 1, 3 }
hole HZ either { 1, 2 }
constraint  !(XA && YA);
module rex
s : [0..3] init 0;
s = 0 -> 0.5: s'=HX + 0.5: s'=HY;
s = 1 -> 1: s'=s+HZ;
s >= 2 -> 1: s'=s;
endmodule
```

(a) Program sketch $\mathcal{S}$

```
module rex
s : [0..3] init 0;
s = 0 -> 0.5: s'=1 + 0.5: s'=3;
s = 1 -> 1: s'=s+2;
s >= 2 -> 1: s'=s;
endmodule
```

(b) Instantiation $\mathcal{S}(\{\mathrm{HX} \mapsto 1, \mathrm{HY} \mapsto 3, \mathrm{HZ} \mapsto 2\})$

Fig. 3. Running example

It effectively restricts the size of the design space and also allows for adding constraints and costs to various designs. Program sketching has been successfully applied to e.g., scientific programs and concurrent data structures [ASFS18, GPS17]. We propose to extend sketches to probabilistic programs and in particular, to a language which is supported by model checkers. Therefore, we focus our presentation on PRISM program sketches.

A sketch is a program that contains *holes*. Holes are the program's open, i.e., undefined parts and may be replaced by one of finitely many *options*. They are syntactically declared as:

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

where $k > 0$, h is the hole identifier, and $\texttt{expr}_i$ is an expression over the program variables describing the i-th option. Each option may be given a name and may be associated with a cost:

$$\texttt{hole}\ h\ \texttt{either}\ \{\,n_1\ \texttt{is}\ \texttt{expr}_1\ \texttt{cost}\ c_1, \ldots, n_k\ \texttt{is}\ \texttt{expr}_k\ \texttt{cost}\ c_k\,\}$$

where n_i is the i-th option name and c_i is the cost of the i-th option and is given as expression over natural numbers. A hole h may be used in the updates of a command, in a similar way as a constant, and may occur multiple times within one or multiple commands. Extensions to such occurrences are discussed in [Jun20, Ch. 6]. An realisation of a program sketch is a function that maps every hole onto one of its options. The option names may be used to describe constraints on these realisations. These propositional formulae over option names restrict realisations, e.g.,

$$\texttt{constraint}\quad (n_1 \vee n_2) \implies n_3$$

requires that whenever the options n_1 or n_2 are taken for some (potentially different) holes, option n_3 is also to be taken. Formally, program sketches are defined as follows.

Definition 2 (Program sketch) A *(PRISM) program sketch* $\mathcal{S}$ is a quadruple $(\mathcal{L}_\mathcal{H}, \mathbf{O}_\mathcal{H}, \Gamma, \mathsf{cost})$ where $\mathcal{L}_\mathcal{H}$ is a PRISM program with a set $\mathcal{H}$ of holes, $\mathbf{O}_\mathcal{H} := \bigcup_{h\in\mathcal{H}} \mathbf{O}_h$ is the set of options where $\mathbf{O}_h$ are the options of hole h, Γ is a set of Boolean expressions over $\mathbf{O}_\mathcal{H}$, and $\mathsf{cost}\colon \mathbf{O}_\mathcal{H} \to \mathbb{N}$ are the option-costs.

Example 2 We consider a small running example to illustrate the main concepts. Figure 3a depicts the program sketch $\mathcal{S}$ with holes $\mathcal{H} = \{\mathrm{HX}, \mathrm{HY}, \mathrm{HZ}\}$. The options of HX are $\mathbf{O}_{\mathrm{HX}} = \{1, 2\}$. The constraint forbids XA and YA both being one; it ensures a non-trivial random choice in state s=0.

Definition 3 (Realisations of sketches) Let $\mathcal{S} := (\mathcal{L}_\mathcal{H}, \mathbf{O}_\mathcal{H}, \Gamma, \mathsf{cost})$ be a sketch.

- A *sketch realisation* $\mathbf{v}$ is a function

  $$\mathbf{v}\colon \mathcal{H} \to \mathbf{O}_\mathcal{H} \text{ such that for all } h \in \mathcal{H}:\ \mathbf{v}(h) \in \mathbf{O}_h,$$

  that satisfies all constraints in Γ.
- The *cost* $\mathbf{c}(\mathbf{v})$ *of realisation* $\mathbf{v}$ is the sum of the costs of the selected options, $\mathbf{c}(\mathbf{v}) := \sum_{h\in\mathcal{H}} \mathsf{cost}(\mathbf{v}(h))$.
- The *sketch instantiation* $\mathcal{S}[\mathbf{v}]$ for realisation $\mathbf{v}$ is the program (without holes) $\mathcal{L}_\mathcal{H}[\mathcal{H} \leftarrow \mathbf{v}(\mathcal{H})]$ in which each hole $h \in \mathcal{H}$ in $\mathcal{L}_\mathcal{H}$ is replaced by $\mathbf{v}(h)$.

Let $\mathcal{R}_\mathcal{S}$ denote the set containing all realisations for $\mathcal{S}$.

Example 3 We continue Example 2. The program in Fig. 3b reflects $\mathcal{S}[\mathbf{v}]$ for realisation $\mathbf{v} := \{\mathrm{HX} \mapsto 1, \mathrm{HY} \mapsto 3, \mathrm{HZ} \mapsto 2\}$, with $\mathbf{c}(\mathbf{v}) = 3$ as $\mathsf{cost}(\mathbf{v}(\mathrm{HX})) = 3$ and all other options have cost zero. For realisation $\mathbf{v}' = \{\mathrm{HX} \mapsto 2, \mathrm{HY}, \mathrm{HZ} \mapsto 1\}$, $\mathbf{c}(\mathbf{v}') = 0$. The assignment $\{\mathrm{HX}, \mathrm{HY}, \mathrm{HZ} \mapsto 1\}$ violates the constraint and is not a realisation. In total, $\mathcal{S}$ represents $6 = 2^3{-}2$ programs and their underlying MCs.

```
const int ACTIVE_HIGH = 0; const int ACTIVE_LOW = 1; const int SLEEP = 3, const int OFF = 4;
hole pm11 either { ACTIVE_HIGH, ACTIVE_LOW, SLEEP, OFF }
hole pm12 either { ACTIVE_HIGH, ACTIVE_LOW, SLEEP, OFF }
...
hole pm99 either { ACTIVE_HIGH, ACTIVE_LOW, SLEEP, OFF }
module DPM
pm : [0..1] init 0;
[sched] qH = 1 & qL = 1 -> 1: pm'=pm11;
[sched] qH = 1 & qL = 2 -> 1: pm'=pm12;
..
[sched] qH = 9 & qL = 9 -> 1: pm'=pm99;
endmodule
module SYSTEM
qH : [1..9] init 1;
qL : [1..9] init 1;
...
endmodule
...
```

Fig. 4. Part of a sketch for the DPM (dynamic power manager)

Sketches are well-defined, if all their realisations are well-defined programs.

3.2. Application view on sketches

Program *sketching* [Sol13] starts with a program sketch, a partial program in which certain (e.g., 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, and is left to an automated synthesiser. The synthesised program has to satisfy a given specification Φ. The following examples give some insight into applications and challenges that are within the scope of the CEGIS technique presented in this paper.

3.2.1. Sketching for controller synthesis

Many sketching challenges originate from controller synthesis problems on Markov decision processes (MDPs). MDPs extend Markov chains by having a choice over probability distributions (over states) in each state. Stated differently. an MC is an MDP in which in every state there is a single probability distribution. The choices between the distributions in MDP states are resolved by strategies that describe the controller. There are often additional constraints imposed on the realistic strategies, e.g., a controller may not be able to depend on the internal state of a remote device. Sketching helps: A sketch may describe a partial strategy for partially observable MDPs, decentralised MDPs, or other extensions to MDPs with restricted classes of strategies. Let us give some concrete examples.

Example 4 Consider a *dynamic power manager (DPM)* optimising the energy consumption in a system, see, e.g. [BBPM99, SDM08, GTC15] that can be in different modes such as idle, sleep, on, etc. A DPM controls changing the system's power states at run time by issuing commands (like: go into sleep mode, wake up) to the system depending on its workload (requests queued in buffers) and performance constraints. Example performance constraints are e.g., restricting the maximal buffer occupancy or maximal number of lost requests. A program sketch in this setting describes admissible control strategies. The synthesis goal is to find a DPM program that satisfies the given constraints while minimising the expected energy consumption. Figure 4 shows a snippet of a possible sketch for a DPM including partially specified commands of the form

$$[\mathit{sched}]\ g_H\ \&\ g_L \quad \rightarrow \quad 1:\ \mathtt{pm}' = y$$

where g_H and g_L are (possible unknown) guards concerning the low-priority and high-priority request buffer, respectively, and hole y represents an unknown update of the DPM state. The program synthesis then boils down to finding the right system update (variable pm) for the observed system state (occupancy of low-priority and high-priority request buffers).

Example 5 Consider a *robot navigation task*, in which a floor plan is divided into various rooms. The robot's task is to navigate to a particular room under certain side constraints, e.g., without visiting another room, or within

```
const double pFA;
const double pFB;
module encode
s : [0..1] init 0;
FA : [0..1] init 0;
FB : [0..1] init 0;
s = 0 -> pFA  * pFB: s'=1 & FA'=1 & FB'=1  +
         (1-pFA)  * pFB: s'=1 & FA'=0 & FB'=1 +
         pFA  * (1-pFB): s'=1 & FA'=1 & FB'=0  +
         (1-pFA)  * (1-pFB): s'=1 & FA'=0 & FB'=0;
..
endmodule
```

(a) A parametric MC description with 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
```

(b) A sketch depending on features.

Fig. 5. Describing a SPL as parametric Markov chain (pMC) or sketch

a number of turns. The challenge is that the robot cannot distinguish all rooms, i.e., its location is only partially observable. The robot thus has to make its decisions based on the sensor readings and its internal state only. No information from the unobservable environment can be used in decision making. A controller that adheres to this restriction is called admissible. When the internal states (including the memory of a controller) and the possible sensor readings are fixed there are finitely many admissible controllers that can be adequately described by a program sketch.

3.2.2. Sketching for software product lines

A software product line (*SPL*) is (according to Wikipedia[5]) "a set of software-intensive systems that share a common, managed set of features 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 SPL have different features which may be understood as functionalities changing the behaviours of a core software system. SPLs thus provide an elegant way to specify *families* of systems: every member of the family comprises the core system together with a combination of features. Quantitative specifications relevant for SPLs include (minimising) the probability of encountering an anomaly or unexpected behaviour, and limiting the expected energy consumption of battery-based (software) systems, see e.g. [RAN+15]. SPLs and program sketches have strong similarities, in particular they both consider a program with some unspecified behaviour. In sketching, the goal is to concretise the unspecified behaviour such that the completed program satisfies a specification. In SPLs, the goal is to classify which of the available behaviours yield a system (aka: product) that satisfies the specification. It is thus natural to consider a sketch as the description of a SPL. The feasibility problem corresponds then to finding a product of the SPL that satisfies a specification, while partitioning amounts to find all realisations that accept and reject the specification, respectively.

Example 6 The Body Sensor Network (BSN [RAN+15]) SPL describes a network of connected sensors that send measurements to a unit identifying health-critical situations. The BSN includes various configurations of binary features, i.e., whether a sensor is available or not. For each feature, a Boolean parameter f is 1 if the feature is active and 0 otherwise. The synthesis goal is to find all feature combinations for which the induced system meets a certain reliability specification. We consider so-called variation points—states whose future behaviour depends on the features—where depending on the availability of features F_a and F_b the model behaves differently.

Let us outline how to formulate this setting as parametric Markov chain (pMC). 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 the pMC described by the PRISM program in Fig. 5a. A program sketch concisely represents such system as demonstrated in Fig. 5b. Furthermore, in a sketch, additional dependencies between features may naturally be expressed by constraints.

Example 7 Consider the development of a communication system. To that end, we can buy two different types of antennas, and configure both the number of retransmissions of each packet and the time elapse between successive retransmissions. Consequently, large parts of the protocol description are shared among all these instances, but the instances vary. For instance, a different number of retransmissions leads to a different topology of the underlying MC.

[5] https://en.wikipedia.org/wiki/Software_product_line

3.3. Synthesis problems

In general, we are interested in realisations of a given sketch that satisfy a given specification.

Definition 4 Let $\mathcal{S}$ be a sketch and Φ a specification. A realisation $\mathbf{v}$ is *accepting* Φ, if $\mathcal{S}[\mathbf{v}] \models \Phi$, and *rejecting* Φ otherwise[6]. Let $\mathsf{acc}_\Phi^{\mathcal{S}}$ and $\mathsf{rej}_\Phi^{\mathcal{S}}$ denote the set of all accepting and rejecting realisations of Φ and sketch $\mathcal{S}$, respectively.

All accepting and rejecting realisations of $\mathcal{S}$ span the entire spectrum of realisations of $\mathcal{S}$, i.e., $\mathsf{acc}_\Phi^{\mathcal{S}} \cup \mathsf{rej}_\Phi^{\mathcal{S}} = \mathcal{R}_{\mathcal{S}}$. For singleton specifications $\Phi = \{\varphi\}$, we denote $\mathsf{acc}_{\{\varphi\}}^{\mathcal{S}}$ by $\mathsf{acc}_\varphi^{\mathcal{S}}$.

Lemma 1 For a sketch $\mathcal{S}$ and a specification Φ, it holds that

$$\mathsf{acc}_\Phi^{\mathcal{S}} = \bigcap_{\varphi \in \Phi} \mathsf{acc}_\varphi^{\mathcal{S}} \quad \text{and} \quad \mathsf{rej}_\Phi^{\mathcal{S}} = \bigcup_{\varphi \in \Phi} \mathsf{rej}_\varphi^{\mathcal{S}}. \tag{1}$$

We are now in a position to formulate the synthesis problems of interest in this paper. Let $\mathcal{S}$ be a sketch and Φ a specification.

1. Feasibility problem: Is $\mathsf{acc}_\Phi^{\mathcal{S}}$ non-empty? In words, is there a realisation of $\mathcal{S}$ satisfying Φ?

This problem is a typical instance of synthesis problems aimed to obtain an instantiation $\mathcal{S}[\mathbf{v}]$ of $\mathcal{S}$ accepting Φ (if it exists). The dual problem is *validity*: is $\mathsf{rej}_\Phi^{\mathcal{S}}$ empty? The validity problem can be considered a verification problem in which we aim to certify that certain system behaviour (characterised by $\neg\Phi$) is irrespective of the precise topology.

The *optimal* feasibility problems aims at determining optimal instantiation rather than some arbitrary instantiation. We define this problem for specifications concerning the maximal reachability problem for a given target set T. This formulation can be straightforwardly adapted to expected reward specifications or minimisation objectives.

2. Max synthesis problem:

Find $\max_{\mathbf{v} \in \mathcal{R}_{\mathcal{S}}} Pr_{\mathcal{S}[\mathbf{v}]}(\lozenge T)$**.** In words, find a realisation of $\mathcal{S}$ maximising the probability to reach T.

The maximal synthesis problem can be naturally extended by considering a specification Φ that restricts the set of feasible solutions. In this case, we search for $\max_{\mathbf{v} \in \mathsf{acc}_\Phi^{\mathcal{S}}} Pr_{\mathcal{S}[\mathbf{v}]}(\lozenge T)$. A relaxed variant of the problem searches for an almost optimal realisation. Formally, for a tolerance value $\epsilon \in (0, 1]$, find $\mathbf{v} \in \mathcal{R}_{\mathcal{S}}$ such that $\forall\, \mathbf{v}' \in \mathcal{R}_{\mathcal{S}} : Pr_{\mathcal{S}[\mathbf{v}]}(\lozenge T) \geq \epsilon \cdot Pr_{\mathcal{S}[\mathbf{v}']}(\lozenge T)$. Typically, this variant is computationally less demanding.

Ultimately, it is useful to effectively analyse all instantiations and find the set $\mathsf{acc}_\Phi^{\mathcal{S}}$. This problem is known as threshold synthesis [ČDP+17] and includes finding a concise representation of $\mathsf{acc}_\Phi^{\mathcal{S}}$ and $\mathsf{rej}_\Phi^{\mathcal{S}}$.

3. Complete partitioning problem: Find $\mathsf{acc}_\Phi^{\mathcal{S}}$, i.e., determine all realisations of $\mathcal{S}$ satisfying Φ.

Finally, the synthesis problems may impose cost constraints on the realisations, see the cost $\mathbf{c}(\mathbf{v})$ in Definition 3. In particular, we can either require that the cost of the resulting realisations is within a given budget or we can search for $\mathbf{v} \in \mathsf{acc}_\Phi^{\mathcal{S}}$ that minimises the cost.

We illustrate the problem statements using two examples of before.

Example 8 Consider the communication protocol from Example 7. Potential specifications impose e.g., upper bounds on the expected energy consumption and on the probability of successfully dropped packets. Feasibility then amounts to finding an instance that satisfies both requirements, while max feasibility amounts to, e.g., finding the instance with the lowest expected energy consumption that bounds the probability of successfully dropped packets. Complete partitioning allows a designer to ensure that all released instances satisfy the specification.

Example 9 Consider the robot navigation task from Example 5. A safety property is that the probability of crashing into static or dynamic obstacles is within a given threshold. A typical liveness property is that the

[6]Recall that $\mathcal{S}[\mathbf{v}] \models \Phi$ if $D \models \Phi$, where $D = [\![\mathcal{S}[\mathbf{v}]]\!]$ is the MC describing the operational semantics of instantiation $\mathcal{S}[\mathbf{v}]$.

probability of reaching a particular room is above some threshold. The conjunction of these properties is a sample specification. Solving feasibility with such a specification can, for fixed thresholds, result in an admissible (fixed-memory) strategy. The outcome of max feasibility is a strategy that (almost, i.e., up to ε) maximises the probability of reaching a particular room while meeting the threshold on the probability of crashing. Partitioning yields all strategies that meet the specification and may allow a designer to make a selection among them, based on secondary objectives.

4. Families

This section presents our main conceptual ideas on a simple, low-level, representation of sets of MCs. These MCs have a set of states in common, but their topology may vary. MCs with varying topology are a natural model when aggregating various alternative designs, and closely related to sketches. In fact, they can be considered as operational model of sketches as we will clarify in Sect. 4.2.

4.1. Families of Markov chains

We introduce a dedicated formalism to capture MCs with an uncertain topology.

Definition 5 (Family MC) A *family MC* (*fMC*) is a tuple $\mathfrak{D} = (S, \iota, Y, \mathsf{Dom}_Y, \mathbf{P}, \mathsf{rew})$ with a finite set of states $S := S_1 \times S_2$, an initial state $\iota \in S$, a finite set Y of parameters that range over $\mathsf{Dom}_Y \colon Y \to 2^{S_1}$, a parameterised transition probability function $\mathbf{P} \colon S \to \mathit{Distr}\big((Y \cup S_1) \times S_2\big)$, and rewards $\mathsf{rew} \colon S \to \mathbb{R}_{\geq 0}$.

We omit the rewards if $\mathsf{rew}(s) := 0$ for all $s \in S$.

Remark 1 States in fMCs are tuples of the form $\langle s_1, s_2 \rangle$. This state structure may look peculiar at first sight but corresponds to the operational semantics of a probabilistic program where states are tuples that encode program-variable valuations; this is illustrated further in Sect. 4.2.

Whereas transition probability functions of MCs map states to distributions over states, in fMCs these functions map states to distributions over states whose first element is given by a parameter.

Remark 2 The signature of the transition function can be simplified to $\mathbf{P} \colon S \to \mathit{Distr}\big(Y \times S_2\big)$ when a parameter for every state in S_1 is used.

By varying these parameters, the topology varies. Assigning every parameter a concrete state yields an MC.

Definition 6 (Assignment) A (parameter) *assignment* of an fMC $\mathfrak{D} = (S_1 \times S_2, \iota, Y, \mathsf{Dom}_Y, \mathbf{P}, \mathsf{rew})$ is a function $\nu \colon Y \to S_1$ where for all $y \in Y$ it holds that $\nu(y) \in \mathsf{Dom}_Y(y)$.

Let $\mathcal{F}_{\mathfrak{D}}$ denote the set of all assignments for $\mathfrak{D}$. This set is finite, but exponential in $\mid Y \mid$.

Definition 7 (Instantiation) Let $\mathfrak{D} = (S := S_1 \times S_2, \iota, Y, \mathsf{Dom}_Y, \mathbf{P}, \mathsf{rew})$ be an fMC and ν an assignment for $\mathfrak{D}$. The *instantiation* $\mathfrak{D}[\nu]$ is the MC $\mathfrak{D}[\nu] := (S, \iota, P, \mathsf{rew})$, where for all $s, s' \in S$, with $s' := \langle s'_1, s'_2 \rangle$:

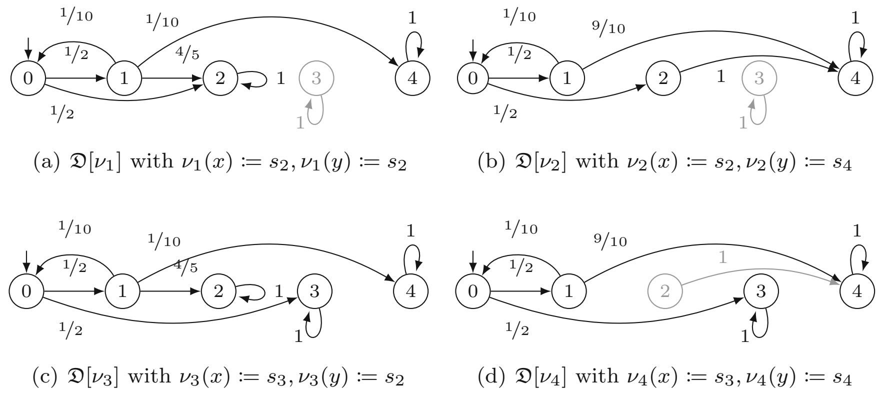
$$P(s, s') := \mathbf{P}[\nu] := \sum_{\substack{y \in Y \\ \nu(y) = s'_1}} \mathbf{P}(s)(\langle y, s'_2 \rangle) + \mathbf{P}(s)(\langle s'_1, s'_2 \rangle).$$

We refer to sets of assignments as (sub)*families*. An fMC and a family may be seen as a generator for the set of all instantiations:

Definition 8 (Generator for fMCs) The *generator* $\langle \mathfrak{D} \mid F \rangle$ for fMC $\mathfrak{D}$ and family $F \subseteq \mathcal{F}_{\mathfrak{D}}$ is:

$$\langle \mathfrak{D} \mid F \rangle := \{\mathfrak{D}[\nu] \mid \nu \in F\}.$$

We write $\langle \mathfrak{D} \rangle$ to denote $\langle \mathfrak{D} \mid \mathcal{F}_{\mathfrak{D}} \rangle$.



(a) $\mathfrak{D}[\nu_1]$ with $\nu_1(x) \coloneqq s_2, \nu_1(y) \coloneqq s_2$ (b) $\mathfrak{D}[\nu_2]$ with $\nu_2(x) \coloneqq s_2, \nu_2(y) \coloneqq s_4$

(c) $\mathfrak{D}[\nu_3]$ with $\nu_3(x) \coloneqq s_3, \nu_3(y) \coloneqq s_2$ (d) $\mathfrak{D}[\nu_4]$ with $\nu_4(x) \coloneqq s_3, \nu_4(y) \coloneqq s_4$

Fig. 6. The four different instantiations of family $\mathfrak{D}$. States s_i are denoted with i

Example 10 Consider the fMC $\mathfrak{D} = (S\times\{t_0\}, \iota, Y, \mathsf{Dom}_Y, \mathbf{P})$ where $S := \{s_0, \ldots, s_4\}$, $\iota := \langle s_0, t_0\rangle$, and $Y := \{x, y\}$ with $\mathsf{Dom}_Y(x) := \{s_2, s_3\}$ and $\mathsf{Dom}_Y(y) := \{s_2, s_4\}$, and $\mathbf{P}$ given by:

$$\begin{aligned}
&\mathbf{P}(\langle s_0, t_0\rangle) := 1/2\colon \langle s_1, t_0\rangle + 1/2\colon \langle x, t_0\rangle, && \mathbf{P}(\langle s_1, t_0\rangle) := 1/10\colon \langle s_0, t_0\rangle + 4/5\colon \langle y, t_0\rangle + 1/10\colon \langle s_4, t_0\rangle,\\
&\mathbf{P}(\langle s_2, t_0\rangle) := 1\colon \langle y, t_0\rangle, &&\\
&\mathbf{P}(\langle s_3, t_0\rangle) := 1\colon \langle s_3, t_0\rangle, && \mathbf{P}(\langle s_4, t_0\rangle) := 1\colon \langle s_4, t_0\rangle.
\end{aligned}$$

Figure 6 depicts the generator $\langle\mathfrak{D}\rangle$. The captions clarify the assignments $\nu_1, \ldots, \nu_4$. Consider the specification

$$\Phi := \{\varphi := \mathbb{P}_{\leq 2/5}(\Diamond\{s_2\})\}.$$

The instantiations ν_1, ν_2, ν_3 in Fig. 6a–c reject Φ and the instantiation ν_4 in Fig. 6d accepts Φ.

Remark 3 The example above corresponds to fMCs with $|\ S_2\ |= 1$. Their main rationale is their simplicity. Observe that whenever S_2 is a singleton, we may omit explicitly denoting all tuples. We illustrate this based on the example from above. We write s_i to denote $\langle s_i, s_j\rangle$ and define $\mathbf{P} : S_1 \to \mathit{Distr}(Y)$ by:

$$\begin{aligned}
&\mathbf{P}(s_0) := 1/2\colon s_1 + 1/2\colon x, && \mathbf{P}(s_1) := 1/10\colon s_0 + 4/5\colon y + 1/10\colon s_4, && \mathbf{P}(s_2) := 1\colon y,\\
&\mathbf{P}(s_3) := 1\colon s_3, && \mathbf{P}(s_4) := 1\colon s_4. &&
\end{aligned}$$

The class with $|\ S_2\ | = 1$ is not expressive enough for most practical purposes, as shown in [Jun20, Ch. 6], but is useful to explain some concepts. Because all parametric transitions lead to a single state, taking such a transition amounts to forgetting everything but this transition, which severely limits the applicability. The next example shows a family with $|\ S_2\ |> 1$.

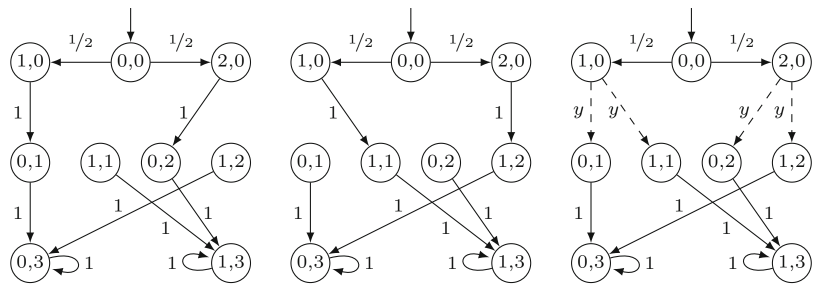
Example 11 Consider the fMC $\mathfrak{D} = (S_1 \times S_2, \iota, Y, \mathsf{Dom}_Y, \mathbf{P})$ where

$$S_1 := \{s_0, s_1, s_2\},\ S_2 := \{t_0, t_1, t_2, t_3\},\ \iota := \langle s_0, t_0\rangle, \text{ and } Y := \{y\} \text{ with } \mathsf{Dom}_Y(y) := \{s_0, s_1\}.$$

The parametric transition function $\mathbf{P}$ is defined by:

$$\begin{aligned}
&\mathbf{P}(\langle s_0, t_0\rangle) := 1/2\colon \langle s_1, t_0\rangle + 1/2\colon \langle s_2, t_0\rangle, &&\\
&\mathbf{P}(\langle s_1, t_0\rangle) := 1\colon \langle y, t_1\rangle, && \mathbf{P}(\langle s_2, t_0\rangle) := 1\colon \langle y, t_2\rangle,\\
&\mathbf{P}(\langle s_0, t_1\rangle) := 1\colon \langle s_0, t_3\rangle, && \mathbf{P}(\langle s_1, t_1\rangle) := 1\colon \langle s_1, t_3\rangle,\\
&\mathbf{P}(\langle s_0, t_2\rangle) := 1\colon \langle s_1, t_3\rangle, && \mathbf{P}(\langle s_1, t_2\rangle) := 1\colon \langle s_0, t_3\rangle,
\end{aligned}$$

and for all other pairs $\langle s, t\rangle \in S_1 \times S_2$: $\mathbf{P}(\langle s, t\rangle) := 1\colon \langle s, t\rangle$. Figure 7 shows the two MCs corresponding to $\mathcal{F}_{\mathfrak{D}}$ and presents a graphical representation of the fMC. In particular, a dashed transition in Fig. 7c refers to a parametric transition, i.e., to a transition that only exists for some parameter assignments. The label y clarifies on which parameter the existence of the transition depends. By considering the first component of the target state, one can determine for which assignments this transition exists.



(a) MC $\mathfrak{D}_2[\{y \mapsto s_0\}]$ (b) MC $\mathfrak{D}_2[\{y \mapsto s_1\}]$ (c) fMC $\mathfrak{D}_2$

Fig. 7. Instantiations of the fMC from Example 11. States $\langle s_i, s_j \rangle$ are labelled as i,j

Sketches		Families	
Sketch $\mathcal{S}$	Definition 2	Family $\mathfrak{D}$	Definition 5
Holes $\mathcal{H}$	Definition 2	Parameters Y	Definition 5
Instantiation $\mathcal{S}[\mathbf{v}]$ is a program		Instantiation $\mathfrak{D}[\nu]$ is an MC	
Realisations $\mathbf{v} \in \mathcal{R}$	Definition 3	Assignments $\nu \in \mathcal{F}$	Definition 6
Counterexamples are sets of commands.	Definition 14	Counterexamples are sets of states.	Definition 12
Conflicts are partial realisations.	Footnote 12	Conflicts are partial assignments.	Definition 9

Table 1. The table clarifies similar concepts in families and sketches. Some of these notions only occur later in the paper.

Synthesis problems for families of MCs

The synthesis problems introduced in Sect. 3.3 for sketches carry over straightforwardly to families of MCs. For example, the feasibility problem is formalised as follows.

Feasibility problem for fMCs:

Given fMC $\mathfrak{D}$, family $F \subseteq \mathcal{F}_{\mathfrak{D}}$, and specification Φ, does $D \models \Phi$ for some MC $D \in \langle \mathfrak{D} \mid F \rangle$?

This is equivalent to: is $\mathrm{acc}_{\Phi}^{\mathfrak{D}} \cap F \neq \emptyset$? The following result characterises the complexity of this feasibility problem.

Lemma 2 [Jun20] For $\mid S_2 \mid \geq 2$, the feasibility problem for fMCs is NP-hard[7].

4.2. Families versus sketches

The following example together with Table 1 clarify how sketches are natural descriptions of fMCs.

Example 12 The sketch in Fig. 8a reflects the fMC in Fig. 7c. Although the construction here is slightly compressed, the idea is simple. The program contains two variables to reflect S_1 and S_2, and one hole to reflect the parameter y.

```
hole HY either { 0, 1 }
module fam
s : [0..2] init 0;
t : [0..3] init 0;
s = 0 & t = 0 -> 0.5: s'=1 + 0.5: s'=2;
s > 0 & t = 0 -> 1: s'=HY & t'=s;
t = 1 | t = 3 -> 1: t'=3;
t = 2 -> 1: s'=1-s & t'=3;
endmodule
```

(a) Sketching an fMC

```
hole HX either { 1, 2, 4 }
hole HY either { 1, 3, 5 }
module rex
s : [0..10] init 0;
t : [0..10] init 0;
s > t -> 0.5: s'=HX+HY + 0.5: s'=HY & t'=HX;
s < t -> 1: s'=s+HX & t'=HX;
s = t -> 1: s'=s;
endmodule
```

(b) Sketching beyond an fMC

Fig. 8. Sketches versus fMCs

[7]This statement is a correction of [ČJJK19] which omitted the restriction on the cardinality of S_2.

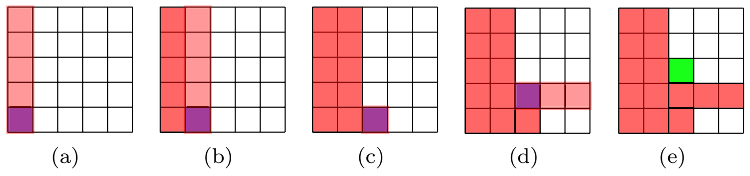


Fig. 9. Schematic example run of CEGIS. The grid depicts a family with two parameters, each with 5 possible values. Thus, each cell corresponds to a (parameter) assignment. Purple cells correspond to the currently considered assignment (rejecting), with light red indicating assignments pruned in this iteration. The green cell corresponds to an accepting assignment

The constraints on sketches enable to concisely describe a subfamily in the modelling language. However, giving semantics to sketches in terms of fMCs is not straightforward, as illustrated by the following example.

Example 13 The sketch in Fig. 8b goes beyond the sketch in Fig. 8a. First, both program variables s and t are affected by the holes. Second, the hole HX appears in updates for different variables, thereby encoding implicit constraints on the possible combinations of commands in different instantiations of the sketch, and between the assignments to the different program variables.

Additional dependencies between transitions may be expressed by some combination of constraints. Having more program variables may be reflected by fMCs with state tuples with an entry for every program variable. Families could be extended to formally support such constructions, but this would clutter their definition[8]. In our experiments, sketches additionally contain the parallel composition of modules and further extensions to ease modelling. Providing a formal fMC semantics to such sketches goes beyond the scope of this paper. Further details can be found in [Ch. 6+7][Jun20].

Finally, we emphasise that the connection between the sketches and families together with Lemma 2 implies *the NP-hardness of the feasibility problem for probabilistic program sketches.*

5. Counterexample-Guided Inductive Synthesis for Families

This section introduces *counterexample-guided inductive synthesis* (*CEGIS*) for families of MCs. (Syntax-Guided) CEGIS [SLTB⁺06, ASFS18, JS17, ADK⁺18] is a successful technique to synthesise programs. The aim of this section is to provide intuition on our adaption of CEGIS. We only consider feasibility. Conceptually, CEGIS for probabilistic models is easier to understand on an explicit model of fMCs than on sketches.

The idea is simple: let $\mathfrak{D}$ be a fMC and φ some property. We tackle the feasibility problem by an enumerative search for an accepting (parameter) assignment for φ, among all instantiations $\mathcal{F}_{\mathfrak{D}}$. After considering some assignment ν, we *prune* the search space of all assignments with additional unconsidered assignments: if ν rejects φ, the verifier provides a counterexample indicating why the instantiation $\mathfrak{D}[\nu]$ rejects φ. This counterexample may be thought of as a “core” of $\mathfrak{D}[\nu]$ that suffices to show that $\mathfrak{D}[\nu]$ rejects φ. CEGIS then reasons that other instantiations having the same core also reject φ.

Example 14 Figure 9 illustrates the reasoning used by CEGIS. Consider a family with two parameters y_1 (horizontal axis) and y_2 (vertical axis), where each parameter has five possible values. Let some property of interest be given. We start considering the assignment represented in the left lower corner. The assignment is rejecting, and by the counterexample from the verifier we obtain that all assignments that assign y_1 to the first value are rejecting. Thus, we prune the full column, i.e., we prune all five assignments that assign y_1 to its first value. Next, we consider a yet unpruned assignment, and again obtain that the value for parameter y_2 is irrelevant if y_1 is assigned to the second value. In the third iteration, we do not get any non-trivial counterexample. In the fourth iteration, we obtain that if y_2 is assigned to the fourth value, the value for y_1 is irrelevant, thereby pruning the fourth row. Finally, in the fifth iteration, we find a feasible instantiation and are done. Within five iterations, this process covered 15 instantiations.

The CEGIS approach is illustrated in Fig. 10: a *synthesiser* (on the left) selects an assignment ν, and the verifier checks whether $\mathfrak{D}[\nu]$ accepts the specification. If it does accept the specification, we are done. Otherwise, the

[8]Tool support for fMCs actually covers these more complex models implicitly.

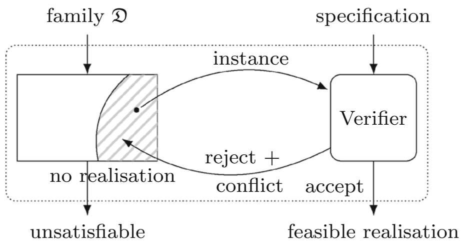


Fig. 10. Schematic view on CEGIS for fMC feasibility (Repeated from Fig. 1)

verifier returns a *conflict* representing the holes that are relevant to form the core (the counterexample) of the MC $\mathfrak{D}[\nu]$. This conflict allows the synthesiser to prune some instantiations rejecting φ.

Example 15 Consider the fMC from Example 10. We may determine its complete partitioning by four model-checking invocations. Our goal is to use less. To that end, we start by verifying $\mathfrak{D}[\nu_1]$. Assume the verifier (for now, magically) reports that the value for parameter y is irrelevant for rejecting the specification. We may then conclude that ν_2 also rejects the specification. Thereby, we save one model-checking call.

In this section, we focus on feasibility, i.e., answering whether there exists an instantiation $D \in \langle\mathfrak{D}\rangle$ such that $D \models \Phi$. These ideas can be straightforwardly extended to max synthesis and partitioning as shown in Sect. 6.

5.1. Conflicts and the synthesiser

Before we focus on the conflict generation by the verifier, we introduce a naive synthesiser which interacts with the verifier. The main goal is to formalise the type of conflicts that we expect the verifier to return. Intuitively, a conflict may be thought of as encoding a particular set of (parameter) assignments. Therefore, we define the following. A *partial assignment* $\bar{\nu}$ for an fMC $\mathfrak{D}$ is a function

$$\bar{\nu} \colon Y \to S_1 \cup \{\bot\} \quad \text{such that} \quad \bar{\nu}(y) \in \mathsf{Dom}_Y(y) \cup \{\bot\} \text{ for all } y \in Y.$$

Notice that this definition adapts Definition 6. For partial assignments $\bar{\nu}_1, \bar{\nu}_2$, let

$$\bar{\nu}_1 \subseteq \bar{\nu}_2 \quad \text{iff} \quad \bar{\nu}_1(y) \in \{\bar{\nu}_2(y), \bot\} \text{ for all } y \in Y.$$

Definition 9 (Conflict) Let $\mathfrak{D}$ be an fMC, φ a property, and $\nu \in \mathcal{F}_{\mathfrak{D}}$ an assignment such that $\mathfrak{D}[\nu] \not\models \varphi$. The partial assignment $\bar{\nu}_\varphi \subseteq \nu$ is a *conflict in ν for φ* if

$$\mathfrak{D}[\nu'] \not\models \varphi \text{ for each } \bar{\nu}_\varphi \subseteq \nu'.$$

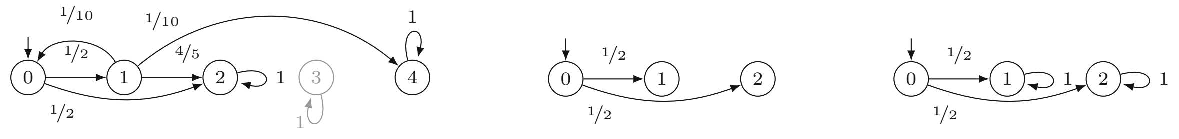
A set of conflicts is called a *conflict set* and an assignment $\nu \in \mathcal{F}_{\mathfrak{D}}$ with $\mathfrak{D}[\nu] \not\models \varphi$ is called a *trivial conflict*.

To explore all instantiations, the synthesiser starts with a set $Q := \mathcal{F}_{\mathfrak{D}}$ and picks some assignment $\nu \in Q$[9]. If $\mathfrak{D}[\nu] \models \Phi$, we return ν; otherwise the verifier returns a conflict set $\{\nu_\varphi\}$ and the Q is pruned by removing all $\nu \supseteq \bar{\nu}_\varphi$ for *each* $\bar{\nu}_\varphi$ in the conflict set. If Q is empty, we are done and conclude that each assignment rejects some property $\varphi \in \Phi$.

Example 16 Recall Example 10, with the instantiations defined in Fig. 6. After considering $\nu_1 = \{x \mapsto s_2, y \mapsto s_2\}$, the verifier magically returns the partial assignment $\bar{\nu} := \{x \mapsto s_2, y \mapsto \bot\}$ as a conflict for φ. Assignment $\nu_2 = \{x \mapsto s_2, y \mapsto s_4\} \supseteq \bar{\nu}$ only differs from ν_1 in the value of y. As $\bar{\nu}(y) = \bot$, i.e., y is not included in the conflict, we conclude $\mathfrak{D}[\nu_2] \not\models \Phi$. The partial assignments $\{x \mapsto s_3, y \mapsto \bot\}$ and $\{x \mapsto \bot, y \mapsto s_3\}$ are not conflicts, as their x-value differs from $\bar{\nu}$.

To summarise, the synthesiser merely iterates over all assignments while discarding any assignment that is covered by some conflict. The essential step thus is to find conflicts, which is the topic of the next section.

[9] We defer discussing details—like how to represent the set Q—to program-level CEGIS.



(a) $\mathfrak{D}[\nu_1]$, repeated from Figure 6(a) (b) Fragment of $\mathfrak{D}[\nu_1]$ (c) subMC of $\mathfrak{D}[\nu_1]$ with $C = \{0\}$

Fig. 11. Fragment and corresponding sub-MC that suffices to refute $\mathbb{P}_{\leq 2/5}(\Diamond\{s_2\})$

5.2. Counterexample-based verifier

The key task of a verifier is to check whether an instantiation accepts the specification. We require the verifier to be sound and complete. Formally,

Definition 10 (Sound and complete verifier) Let $\mathfrak{D}$ be an fMC, ν an assignment, and Φ a specification.

A verifier is *sound and complete*, if on termination (a) the returned conflict set is empty iff ν is accepting Φ, and (b) if it is not empty, it contains a conflict $\bar{\nu}_\varphi \subseteq \nu$ for some $\varphi \in \Phi$.

An example sound and complete verifier is a model-checking procedure that checks $\mathfrak{D}[\nu]$. If $\mathfrak{D}[\nu] \models \Phi$, then it returns an empty conflict set, and otherwise it returns a conflict set containing the trivial conflict ν. While this verifier is sound and complete, the synthesiser still has to iterate over all assignments.

State-level counterexamples The verification procedure that we use to generate conflicts is built on top of the concept of fragments of MCs [WJÁ+12]. We use the following two auxiliary concepts: The successors $\mathsf{succ}(S')$ of a set of states S' is defined as

$$\mathsf{succ}(S') = \{s \in S \mid \exists\, \nu \in \mathcal{F}_{\mathfrak{D}}, \exists\, s' \in S' \text{ s.t. } \mathbf{P}[\nu](s')(s) > 0\}.$$

The set $\mathsf{occurs}(s)$ of parameters occurring at a state s is defined by

$$\mathsf{occurs}(s) := \{y \in Y \mid \exists\, s_2' \in S_2 \text{ s.t. } \mathbf{P}(s)(\langle y, s_2' \rangle) > 0\}.$$

Definition 11 (sub-fMC) Let $\mathfrak{D} = (S, \iota, Y, \mathsf{Dom}_Y, \mathbf{P}, \mathsf{rew})$ be an fMC with $C \subseteq S, \iota \in C$. States in C are referred to as *critical* states. The *sub-fMC* of $\mathfrak{D}$ with respect to C is the fMC

$$\mathfrak{D}\downarrow C := \left(C \cup \mathsf{succ}(C), \iota, \bigcup_{s \in C} \mathsf{occurs}(s), \mathbf{P}', \mathsf{rew}'\right)$$

where $\mathbf{P}'$ is defined by

$$\mathbf{P}'(s) := \begin{cases} \mathbf{P}(s) & \text{if } s \in C, \\ \{s \mapsto 1\} & \text{if } s \in \mathsf{succ}(C) \setminus C, \end{cases}$$

and rew'(s) = rew(s) for all $s \in C$ and $\mathsf{rew}(s) = 0$ for all $s \in \mathsf{succ}(C) \setminus C$.

If the fMC is an MC, we refer to this construction as a *sub-MC*.
Example 17 Figure 11c shows the sub-MC $\mathfrak{D}[\nu_1]\downarrow C$ of $\mathfrak{D}[\nu_1]$ depicted in Fig. 11a with critical states $C = \{s_0\}$.

Recall that upper-bounded reachability properties are safety properties whereas lower-bounded reachability properties are liveness properties. Let us first consider safety properties, and then discuss liveness. The essential attribute of sub-MCs is the following monotonicity which motivates the definition of a counterexample.

Proposition 1 ([WJÁ+12]) Let D be an MC and $\varphi := \mathbb{P}_{\leq\lambda}(\Diamond T)$ a safety property. For any set of critical states $C \subseteq S$ it holds:

$$D\downarrow C \not\models \varphi \quad \text{implies} \quad D \not\models \varphi.$$

Definition 12 (Counterexample) Let D be an MC and $\varphi := \mathbb{P}_{\leq\lambda}(\Diamond T)$ a property. Any set of critical states $C \subseteq S$ with $D\downarrow C \not\models \varphi$ is a *counterexample* (CE) for D and φ.

Remark 4 We remark that conceptually, a counterexample is best described by a set of paths. The set of states induces a subgraph, all paths in that subgraph are the paths that together yield a such a path-based counterexample.

Algorithm 1 Verifier (for safety properties on a family of MCs)

```
function VERIFY(fMC $\mathfrak{D}$, assignment $\nu$, safety specification $\Phi$)
    Violated ← $\emptyset$; Conflict ← $\emptyset$;
    $D$ ← GENERATEMC($\mathfrak{D}$, $\nu$)                                  ▷ Construct $D := \mathfrak{D}[\nu]$ once
    for all $\varphi \in \Phi$ do                                          ▷ Consider every property $\varphi$ in specification $\Phi$
        if not CHECK($D$, $\varphi$) then                                  ▷ Model checking: $D \models \varphi$?
            Violated ← Violated $\cup \{\varphi\}$                         ▷ store all violated properties
    for all $\varphi \in$ Violated do                                      ▷ Generate counterexamples for each violated property
        $C_\varphi$ ← COMPUTECOUNTEREXAMPLE($D$, $\varphi$);               ▷ Take a MC and a spec, return a counterexample (Definition 12)
        Conflict ← Conflict $\cup$ GENERATECONFLICT($\mathfrak{D}$, $C_\varphi$)   ▷ Take a family and a counterexample, return a conflict (Definition 9)
    return Conflict
```

However, in the scope of this paper and the implemented algorithm, we identify counterexample solely via the aforementioned set of states.

Example 18 Recall Example 10. We have to check whether $\mathfrak{D}[\nu_1] \models \mathbb{P}_{\leq 2/5}(\lozenge\{s_2\})$. In fact, the paths in the fragment of $\mathfrak{D}[\nu_1]$ in Fig. 11b (ignoring the outgoing transitions of states s_1 and s_2) suffice to show that the probability to reach state s_2 exceeds $2/5$. These paths are contained in the sub-MC $\mathfrak{D}[\nu_1] \downarrow C$ with critical states $C = \{0\}$ as depicted in Fig. 11c. The fragment from Fig. 11b is part of $\mathfrak{D}[\nu_2]$ too. Formally, the sub-MC of $\mathfrak{D}[\nu_2] \downarrow C$ is isomorphic to $\mathfrak{D}[\nu_1] \downarrow C$ and therefore rejects Φ too. Thus, $\mathfrak{D}[\nu_2] \not\models \Phi$.

Next, we embed finding sub-MCs and generating conflicts from them into a verifier.

Verifier Algorithm 1 outlines a basic verifier. First, we construct $D := \mathfrak{D}[\nu]$ once (line 3). We then use an off-the-shelf probabilistic model-checking procedure CHECK(D, φ) to determine all rejected $\varphi \in \Phi$ (line 5). The algorithm then iterates over the violated φ and computes critical sets C of D that induce sub-MCs with $D \downarrow C \not\models \varphi$ (line 8). These critical sets for safety properties are obtained via existing methods [HKD09, ÁBD+14]. Adaptations for liveness properties (in fact, lower bounded reachability properties) are discussed below. Next, the procedure GENERATECONFLICT$(\mathfrak{D}, C)$ identifies the parameters $\bigcup_{s \in C}$ occurs$(()s) \subseteq Y$ that occur in the sub-MCs $\mathfrak{D} \downarrow C$. It returns the corresponding conflict *Conflict*(C, ν), i.e., the partial assignment with *Conflict*$(C, \nu)(y) = \nu(y)$ if $y \in \bigcup_{s \in C}$ occurs$(()s)$ and $\perp$ otherwise. Ultimately, it stores the conflict in the conflict set Conflict (line 9). The proposition below clarifies the relation between critical sets and conflicts.

Proposition 2 If C is a counterexample for $\mathfrak{D}[\nu]$ and φ, with $\mathfrak{D}[\nu] \not\models \varphi$, then C is also a counterexample for each $\mathfrak{D}[\nu']$ with $\nu' \supseteq$ *Conflict*(C, ν).

Example 19 Recall from Example 18 that ν_2 rejects Φ, and that we may deduce this by inspecting the sub-MC from Fig. 11c. In particular, we may deduce that ν_2 rejects Φ *without constructing* $\mathfrak{D}[\nu_2]$. Just considering ν_2, $\mathfrak{D}$ and C suffices. First, take all parameters occurring in $\mathfrak{D} \downarrow C$. This operation yields $\{x\}$ and may be implemented by inspection of $\mathfrak{D}$. The partial realisation $\bar{\nu} := \{x \mapsto s_2, y \mapsto \perp\}$ is a conflict. The assigned values for the other parameters do not affect the shape of the sub-MC induced by C.

Conflicts for liveness properties To support liveness properties such as $\varphi := \mathbb{P}_{>\lambda}(\lozenge T)$, we first consider a (standard) dual safety property $\varphi' := \mathbb{P}_{<1-\lambda}(\lozenge B)$. The straightforward idea is to fix an instantiation and set $B := S^T_{=0}$—i.e., let B be the states that reach the target with probability zero—based on [BK08, Thm. 10.122 and Thm. 10.127]. For any MC, the set B of states may be efficiently computed using graph algorithms. We have to be careful, however.

Example 20 We adapt Example 10. Consider $\mathfrak{D}[\nu_1]$ and let $\varphi := \mathbb{P}_{>3/5}(\lozenge\{s_4\})$. Assignment ν_1 rejects φ. Let $\varphi' = \mathbb{P}_{<2/5}(\lozenge\{s_2\})$ which is refuted with critical set $C = \{s_0\}$ (as before). Although $\mathfrak{D}[\nu_2] \downarrow C$ is (again) isomorphic to $\mathfrak{D}[\nu_1] \downarrow C$, assignment ν_2 accepts φ. The problem here is that the zero states depend on the assignment: state s_2 is a zero-state for $\mathfrak{D}[\nu_1]$ as $\nu_1(y) = s_2$, but not a zero-state for $\mathfrak{D}[\nu_2]$, as $\nu_2(y) = s_4$.

To prevent these situations, we ensure that the states in B are zero states in *all* instantiations. We enforce this by including the zero states for the current instantiation in the counterexample of φ:

Definition 13 For any MC D and property $\varphi := \mathbb{P}_{>\lambda}(\lozenge T)$ and C a counterexample for D and $\mathbb{P}_{\leq 1-\lambda}(\lozenge S^T_{=0})$. Then, $C \cup S^T_{=0}$ is a counterexample for D and φ.

Algorithm 2 Synthesiser (feasibility synthesis for probabilistic programs)

```
1: function SYNTHESIS(program sketch $\mathcal{S}$, specification $\Phi$, budget $\mathcal{B}$)
2:     $\psi \leftarrow$ INITIALISE($\mathcal{S}, \mathcal{B}$)                    ▷ Construct propositional formula for set of realisations
3:     while sat($\psi$) $\neq \emptyset$ do                                  ▷ As long as not all realisations are pruned
4:         $\mathbf{v} \leftarrow$ GETREALISATION($a_{\mathbf{v}} \in$ sat($\psi$))                          ▷ Pick a realisation
5:         Conf $\leftarrow$ VERIFY($\mathcal{S}, \mathbf{v}, \Phi$)                        ▷ Verify and return conflict sets
6:         if Conf $= \emptyset$ then return $\mathbf{v}$           ▷ No conflict: This realisation satisfies the specification
7:         $\psi \leftarrow \psi \wedge \left( \bigwedge_{\bar{\mathbf{v}} \in \text{Conf}} \text{LEARNFROMCONFLICT}(\mathcal{S}, \bar{\mathbf{v}}) \right)$     ▷ Prune the set of realisations using the conflicts
8:     return Unsat
```

Together, this ensures that—in any instantiation with an isomorphic sub-MC—we reach states $B = S^{T}_{=0}$ with a critical probability mass[10] and never leave B. Indeed, this ensures that Proposition 2 holds also for liveness properties.

Example 21 Continue the example above with liveness property $\varphi := \mathbb{P}_{>3/5}(\lozenge\{s_4\})$. Recall that all instantiations ν_i are given in Fig. 6. In $\mathfrak{D}[\nu_1]$, we obtain critical states $\{s_0, s_2\}$, different from the reasoning from Example 10. For $\mathfrak{D}[\nu_4]$, we obtain $C' := \{s_0\} \cup \{s_3\}$ as critical states, and as $\mathfrak{D}[\nu_4] \downarrow C'$ is isomorphic to $\mathfrak{D}[\nu_3] \downarrow C'$, we obtain that ν_3 also rejects (liveness) property φ.

Conflicts for expected rewards Performance criteria such as the expected time to completion or the expected energy consumption are widespread and typically expressed as upper bounds on expected rewards. Algorithms to compute counterexamples for upper bounded constraints on rewards have been considered in [QJD+15]. Put in a nutshell, for MCs with finite expected rewards, the sub-MCs are changed such that all states in succ$(C) \setminus C$ are made target states and get zero reward assigned. If the reward collected in the counterexample already rejects the upper bound, then any extension does. The results as presented for safety properties carry over immediately. We omit a technical treatment of this construction. For lower bounds on expected rewards, there is no analogue to the construction above.

6. Counterexample-guided Inductive Synthesis for Sketches

So far, we explained the underlying principles of CEGIS at the Markov chain level. Below we adapt CEGIS to probabilistic programs described in the PRISM modelling language. In particular, we employ so-called *program-level counterexamples* [DJW+14, WJV+15], rather than counterexamples that are sub-MCs. This adaption is motivated by a number of observations. First, the program sketch contains additional information that may be essential for quickly finding feasible solutions. Second, CEGIS has originally been formulated at the program level, making the adaption to probabilistic program sketches natural. Third, methods for state-level CEs concentrate on generating small CEs, where small is measured in the size (number of states) of the CE. For generating conflicts, we would like a more flexible notion of small CEs, as we are interested in finding small conflicts. The notion of size in program-level CEs is more flexible and thus easier to adapt.

This section presents a synthesiser working at the level of program sketches and an adapted verifier that uses program-level counterexamples. The section finishes with some technical details about the verifier and a brief outline of the treatment of ill-formed sketches by the synthesiser. Our method support sketches with holes in guards. We reuse the running example from Fig. 3.

6.1. Program-level synthesiser

We present a synthesiser for the feasibility synthesis problem that acts on probabilistic programs and discuss the adaptations for max synthesis and partitioning.

6.1.1. *Feasibility*

[10]A good implementation may find smaller counterexamples by taking an $B' \subseteq B$ such that $Pr_D(\lozenge B')$ already exceeds $1 - \lambda$.

Overview The synthesiser is outlined in Algorithm 2. We give a brief overview before detailing its various steps. The set of realisations not yet pruned is kept track of during the synthesis process. These realisations are represented by (the satisfying assignments of) the formula ψ over hole-assignments. Iteratively strengthening ψ by adding conjunctions to thus prunes the set of realisations. The method INITIALISE($\mathcal{S}, \mathcal{B}$) constructs ψ such that ψ represents *all* realisations that satisfy the constraints in the sketch $\mathcal{S}$ within the budget $\mathcal{B}$. We exploit an SMT-solver for linear-bounded integer arithmetic to obtain a realisation $\mathbf{v}$ consistent with ψ, or to return `Unsat` if no such realisation exists. As long as new realisations are found, the verifier analyses them and returns a conflict set Conf. If $\mathsf{Conf} = \emptyset$, then $\mathbf{v}$ is accepting, and the search terminates. Otherwise, the synthesiser updates ψ based on the conflicts. The realisation $\mathbf{v}$ is always pruned, i.e., after updating, $\mathbf{v}$ is not compliant with ψ.

Initialisation INITIALISE($\mathcal{S}, \mathcal{B}$): Let hole $h \in \mathcal{H}$ have (ordered) options $\mathbf{O}_h = \{o_h^1, \ldots, o_h^n\}$. To encode realisation $\mathbf{v}$ as SMT-formula, we introduce integer-valued meta-variables $K_H := \{\kappa_h \mid h \in \mathcal{H}\}$ with the semantics that $\kappa_h = i$ whenever hole h has value o_h^i, i.e., $\mathbf{v}(h) = o_h^i$. We set

$$\psi := \psi_{\text{opti}} \wedge \psi_\Gamma \wedge \psi_{\text{cost}},$$

where ψ_{opti} ensures that each hole is assigned to some option, ψ_Γ ensures that the sketch's constraints Γ are satisfied, and ψ_{cost} ensures that the budget $\mathcal{B}$ is respected. These sub-formulae[11] are:

$$\psi_{\text{opti}} := \bigwedge_{h\in\mathcal{H}} 1 \leqslant \kappa_h \leqslant \mid \mathbf{O}_h \mid, \qquad \psi_\Gamma := \bigwedge_{\gamma\in\Gamma} \gamma[N_h^i/\kappa_h \doteq i],$$

$$\psi_{\text{cost}} := \sum_{h\in\mathcal{H}} \omega_h \leqslant \mathcal{B} \wedge \left(\bigwedge_{h\in\mathcal{H}} \bigwedge_{i=1}^{|\mathbf{O}_h|} \kappa_h \doteq i \to \omega_h \doteq \mathsf{cost}(o_h^i) \right),$$

where

- $\gamma[N_h^i/\kappa_h \doteq i]$ denotes that in constraint $\gamma \in \Gamma$ each option name N_h^i is replaced by an option o_h^i with $\kappa_h \doteq i$,
- ω_h are fresh variables storing the cost for the selected option at hole h.

Let us clarify the encoding with an example.

Example 22 For the sketch in Fig. 3a, we obtain (slightly simplified):

$$\begin{aligned} \psi := {} & 1 \leq \kappa_{\text{HX}} \leqslant 2 \wedge 1 \leqslant \kappa_{\text{HY}} \leqslant 2 \wedge 1 \leqslant \kappa_{\text{HZ}} \leqslant 2 \\ & \wedge \neg(\kappa_{\text{HX}} \doteq 1 \wedge \kappa_{\text{HY}} \doteq 1) \wedge \omega_{\text{HX}} + \omega_{\text{HY}} + \omega_{\text{HZ}} \leqslant \mathcal{B} \\ & \wedge \big(\kappa_{\text{HX}} \doteq 1 \to \omega_{\text{HX}} \doteq 3\big) \wedge \big(\kappa_{\text{HX}} \doteq 2 \to \omega_{\text{HX}} \doteq 0\big) \wedge \omega_{\text{HY}} \doteq 0 \doteq \omega_{\text{HZ}}, \end{aligned}$$

where $\kappa_{\text{HY}} = 2$ encodes $\text{HY} \mapsto 3$. The formula ψ encodes that there are two options for each hole and HX and HY are not allowed to be both 1 by the sketch constraint. Finally, the cost for the first option for HX is 3, all other costs are zero.

Finding a realisation GETREALISATION(ψ): To obtain a realisation $\mathbf{v}$, we check the satisfiability of ψ. The solver either returns `Unsat` indicating that the synthesiser is finished, or `Sat`, together with a satisfying assignment $a_\mathbf{v} \colon K_H \to \mathbb{N}$. The assignment $a_\mathbf{v}$ uniquely identifies a realisation $\mathbf{v}$ by $\mathbf{v}(h) := o_h^{a_\mathbf{v}(\kappa_h)}$. The sum over all ω_H gives $\mathbf{c}(\mathbf{v})$, i.e., the cost of the realisation.

Example 23 Consider ψ from Example 22. The satisfying assignment (for $\mathcal{B} \geq 3$) $a_\mathbf{v} := \{\kappa_{\text{HX}} \mapsto 1, \kappa_{\text{HY}}, \kappa_{\text{HZ}} \mapsto 2, \omega_{\text{HX}} \mapsto 3, \omega_{\text{HY}}, \omega_{\text{HZ}} \mapsto 0\}$ represents $\mathbf{v}$—with $c(\mathbf{v}) = 3$—from Fig. 3b. Note that $\kappa_{\text{HY}} = 2$ encodes $\text{HY} \mapsto 3$.

Verifying the realisation VERIFY($\mathcal{S}[\mathbf{v}], \Phi$): invokes any sound and complete verifier, e.g., the verifier from Sect. 5.2, or an adaption as presented in Sect. 6.2. Either $\mathbf{v}$ is accepting, or it returns a set of conflicts[12].

[11] We additionally have to ensure that the integer variables are assigned integers.

[12] As in Sect. 5.1: A *partial realisation* for $\mathcal{S}$ is a function $\bar{\mathbf{v}} \colon \mathcal{H} \to \mathbf{O}_\mathcal{H} \cup \{\bot\}$ such that $\forall\, h \in \mathcal{H}.\ \bar{\mathbf{v}}(h) \in \mathbf{O}_h \cup \{\bot\}$. For partial realisations $\bar{\mathbf{v}}_1, \bar{\mathbf{v}}_2$, let $\bar{\mathbf{v}}_1 \subseteq \bar{\mathbf{v}}_2$ iff $\bar{\mathbf{v}}_1(h) \in \{\bar{\mathbf{v}}_2(h), \bot\}$ for all $h \in \mathcal{H}$. Let $\mathbf{v}$ be a realisation such that $\mathcal{S}[\mathbf{v}] \not\models \varphi$ for $\varphi \in \Phi$. Partial realisation $\bar{\mathbf{v}}_\varphi \subseteq \mathbf{v}$ is a *conflict* for φ iff $\forall\, \mathbf{v}' \supseteq \bar{\mathbf{v}}_\varphi\ \mathcal{S}[\mathbf{v}'] \not\models \varphi$.

Algorithm 3 Synthesiser (max synthesis for probabilistic programs)

```
1:  function SYNTHESIS($\mathcal{S}$, $\Phi$, $\mathcal{B}$, target predicate $\mathbf{T}$, tolerance $\varepsilon$)
2:      $\lambda^* \leftarrow \infty$, $\mathbf{v}^* \leftarrow$ Unsat                    ▷ Max value and associated realisation
3:      $\psi \leftarrow$ INITIALISE($\mathcal{S}$, $\mathcal{B}$)                    ▷ Construct propositional formula for set of realisations
4:      while sat($\psi$) $\neq \emptyset$ do                    ▷ As long as not all realiations are pruned
5:          $\mathbf{v} \leftarrow$ GETREALISATION($a_\mathbf{v} \in$ sat($\psi$))                    ▷ Select realisation
6:          Conf, $\lambda_{\text{new}} \leftarrow$ OPTIMISEVERIFY($\mathcal{S}$, $\mathbf{v}$, $\Phi$, $\mathbf{T}$, $\lambda^*$, $\varepsilon$)    ▷ Either $\lambda_{\text{new}} > \lambda^*$ or verification of $\Phi \cup \{\mathbb{P}_{\geq(1-\varepsilon)\cdot\lambda^*}(\lozenge T)\}$
7:          if Conf $= \emptyset$ then $\lambda^* \leftarrow \lambda_{\text{new}}$, $\mathbf{v}^* \leftarrow \mathbf{v}$                    ▷ Improved best candidate
8:              $\psi \leftarrow \psi \wedge$ LEARNFROMCONFLICT($\mathcal{S}$, $\mathbf{v}$)                    ▷ Do not consider this candidate again
9:          $\psi \leftarrow \psi \wedge \left(\bigwedge_{\bar{\mathbf{v}} \in \text{Conf}}\right.$ LEARNFROMCONFLICT($\mathcal{S}$, $\bar{\mathbf{v}}$)$\left.\right)$                    ▷ Prune design space
10:     return $\mathbf{v}^*$
```

Example 24 We continue with $\mathbf{v}$ from Example 23. Consider $\Phi = \{\mathbb{P}_{\leq 2/5}[\lozenge\ \texttt{s=3}]\}$. The verifier (for now, magically) constructs a conflict set $\{\bar{\mathbf{v}}\}$ with $\bar{\mathbf{v}} = \{\text{HY} \mapsto 3\}$.

Learning from a conflict LEARNFROMCONFLICT($\mathcal{S}$, $\bar{\mathbf{v}}$): For a conflict $\bar{\mathbf{v}} \in$ Conf, we conjoin ψ with

$$\neg\Big(\bigwedge_{h \in \mathcal{H}, \bar{\mathbf{v}}(h) \neq \bot} \kappa_h \doteq a_\mathbf{v}(\kappa_h) \Big). \tag{2}$$

This formula excludes realisations $\mathbf{v}' \supseteq \bar{\mathbf{v}}$. Intuitively, it asserts that the potentially accepting realisations (encoded by the updated ψ) must pick a different option for at least one of the holes h assigned by $\bar{\mathbf{v}}$.

Example 25 We proceed with $\bar{\mathbf{v}} = \{\text{HY} \mapsto 3\}$ from Example 24. The synthesiser updates $\psi \leftarrow \psi \wedge \kappa_{\text{HY}} \neq 2$. A satisfying assignment $\{\kappa_{\text{HX}}, \kappa_{\text{HY}}, \kappa_{\text{HZ}} \mapsto 1\}$ for ψ encodes $\mathbf{v}'$ from Example 3. As $\mathfrak{D}[\mathbf{v}'] \models \Phi$, the verifier reports no conflict.

6.1.2. Max synthesis

Let us now adapt the synthesiser to support optimal synthesis, cf. Algorithm 3. We focus on maximising the probability of reaching states T w.r.t. a tolerance $\varepsilon \in (0, 1]$. The target states are described by a predicate $\mathbf{T}$, i.e., a propositional formula over variables occurring in the sketch. The states satisfying this formula are the target states. Algorithm 3 stores in λ^* the maximal probability $Pr_{\mathcal{S}[\mathbf{v}]}(\lozenge T)$ over all $\mathbf{v}$, and this maximising $\mathbf{v}$ as $\mathbf{v}^*$. In each iteration, an optimising verifier is invoked (line 6) on realisation $\mathbf{v}$. An optimising verifier wraps standard model-checking procedures to ensure the following behaviour. If $\mathcal{S}[\mathbf{v}] \models \Phi$ and $Pr_{\mathcal{S}[\mathbf{v}]}(\lozenge T) > \lambda^*$, it returns an empty conflict set *and* $\lambda_{\text{new}} := Pr_{\mathcal{S}[\mathbf{v}]}(\lozenge T)$. Otherwise, it reports a conflict set Conf for $\Phi \cup \{\mathbb{P}_{\geq(1-\varepsilon)\cdot\lambda^*}(\lozenge T)\}$.

Example 26 Consider some sketch with a target predicate $\mathbf{T}$. We select a first realisation, which refutes the specification Φ. We apply pruning as in feasibility analysis. Assume that in the second iteration, we pick a realisation which accepts Φ. Furthermore, we obtain by model checking that the probability to reach the target is $1/3$. In the third iteration, we pick a realisation which accepts Φ, but the probability to the target is only $1/4$.

We use $\mathbb{P}_{\geq 1/3}(\lozenge T)$ as a property for counterexample generation (we already know that it is rejected by the current realisation). We may then prune additional realisations that do not improve our optimisation criterium. In the fourth iteration, we then pick a realisation which satisfies the specification, *and* improves the reachability probability to $3/4$. We continue with these iterations, until we have used all realisations to either improve our current optimum, or to prune (potentially other) realisations.

6.1.3. Complete partitioning

Recall that the goal of complete partitioning is to effectively find and represent all realisations $\text{acc}^{\mathcal{S}}_{\Phi}$ of sketch $\mathcal{S}$ that satisfy Φ and all realisations $\text{rej}^{\mathcal{S}}_{\Phi}$ that violate Φ. The key idea is to use the CE-pruning to build a formula over hole-assignments representing $\text{acc}^{\mathcal{S}}_{\Phi}$ and $\text{rej}^{\mathcal{S}}_{\Phi}$, respectively. As before, we start with a formula ψ representing all realisations that satisfy the constraints in the sketch $\mathcal{S}$ within the budget $\mathcal{B}$. Algorithm 4 for complete partitioning then iteratively builds formulae $\psi_\top$ and $\psi_\bot$ containing hole-assignments representing realisations satisfying Φ and $\neg\Phi$, respectively. Building $\psi_\bot$ (line 13) is a straightforward adaptation of the feasibility algorithm. If a realisation $\mathbf{v}$ violating Φ (i.e., satisfying $\neg\Phi$) is found, the algorithm extends $\psi_\bot$ with the hole-assignments

Algorithm 4 Synthesiser (complete partitioning for probabilistic programs)

```
function SYNTHESIS(program sketch $\mathcal{S}$, specification $\Phi$, budget $\mathcal{B}$)
    $\psi \leftarrow$ INITIALISE($\mathcal{S}, \mathcal{B}$)    ▷ Construct propositional formula for set of realisations
    $\psi_\perp \leftarrow$ false, $\psi_\top \leftarrow$ false
    while sat($\psi \wedge \neg\psi^\top \wedge \neg\psi^\perp$) $\neq \emptyset$ do    ▷ As long as not all realisations are partitioned
        $\mathbf{v} \leftarrow$ GETREALISATION($a_\mathbf{v} \in$ sat($\psi \wedge \neg\psi^\top \wedge \neg\psi^\perp$))    ▷ Pick a realisation
        Conf $\leftarrow$ VERIFY($\mathcal{S}, \mathbf{v}, \Phi$)    ▷ Verify and return conflict sets for $\Phi$
        if Conf $= \emptyset$ then    ▷ If the realisation $\mathbf{v}$ satisfies $\Phi$
            $\psi_* \leftarrow$ true
            for all $\varphi_i \in \Phi$ do
                Conf$_i \leftarrow$ VERIFY($\mathcal{S}, \mathbf{v}, \neg\varphi_i$)    ▷ Return conflict sets for each $\neg\varphi_i$
                $\psi_* \leftarrow \psi_* \wedge \left(\bigvee_{\bar{\mathbf{v}} \in \mathsf{Conf}_i} \neg\,\text{LEARNFROMCONFLICT}(\mathcal{S}, \bar{\mathbf{v}})\right)$    ▷ Learn from $\mathbf{v}$ realisations satisfying $\varphi_i$
            $\psi^\top \leftarrow \psi^\top \vee \psi_*$    ▷ Add realisations satisfying $\Phi$ obtained from $\mathbf{v}$
        else $\psi^\perp \leftarrow \psi^\perp \vee \left(\bigvee_{\bar{\mathbf{v}} \in \mathsf{Conf}} \neg\,\text{LEARNFROMCONFLICT}(\mathcal{S}, \bar{\mathbf{v}})\right)$    ▷ Add realisations satisfying $\neg\Phi$ learned from $\mathbf{v}$
    acc$^\mathcal{S}_\Phi \leftarrow \psi \wedge \psi^\top$
    rej$^\mathcal{S}_\Phi \leftarrow \psi \wedge \psi^\perp$
    return acc$^\mathcal{S}_\Phi$, rej$^\mathcal{S}_\Phi$
```

Algorithm 5 Verifier

```
function VERIFY(Sketch $\mathcal{S}$, realisation $\mathbf{v}$, specification $\Phi$)
    Violated $\leftarrow \emptyset$; Conflict $\leftarrow \emptyset$;
    $D \leftarrow$ GENERATEMC($\mathcal{S}, \mathbf{v}$)    ▷ construct $[\![\mathcal{S}[\mathbf{v}]]\!]$ once
    for all $\varphi \in \Phi$ do    ▷ Run model checking for each property seperately.
        if not CHECK($D, \varphi$) then
            Violated $\leftarrow$ Violated $\cup \{\varphi\}$
    for all $\varphi \in$ Violated do    ▷ Generate conflicts for properties that are violated
        $C_\varphi \leftarrow$ COMPUTECOUNTEREXAMPLE($\mathcal{S}[\mathbf{v}], D, \varphi$)    ▷ For a program and a property, return a CE as in Definition 14
        Conflict $\leftarrow$ Conflict $\cup$ GENERATECONFLICT($\mathcal{S}, C_\varphi$)    ▷ Generate a conflict on the level of the sketch, see Proposition 3
    return Conflict
```

representing realisations learned from all conflicts $\bar{\mathbf{v}}$ obtained from $\mathbf{v}$. Recall that LEARNFROMCONFLICT($\mathcal{S}, \bar{\mathbf{v}}$) returns formulae describing realisations that must assign a different option to at least one of the holes that are assigned by $\bar{\mathbf{v}}$. Therefore, the formula $\bigvee_{\bar{\mathbf{v}} \in \mathsf{Conf}} \neg$ LEARNFROMCONFLICT($\mathcal{S}, \bar{\mathbf{v}}$) represents all realisations, learnt from the realisation $\mathbf{v}$, violating Φ. Constructing $\psi_\top$ is more complicated due to the fact that the specification Φ includes a set of properties $\{\varphi_i\}_{i \in I}$ (i.e. $\Phi = \bigwedge_{i \in I} \varphi_i$) and thus a satisfying realisation has to satisfy *all* φ_i. For each Conf$_i$ (obtained from the verification of $\neg\varphi_i$), we learn (as above) formulae describing realisations satisfying φ_i. Realisations satisfying Φ is then obtained as a conjunction of these formulae stored in ψ^* (line 11) that is used to iteratively built $\psi^\top$. The algorithm terminates if all realisations satisfying ψ are described in $\psi^\top$ or $\psi^\perp$ and thus $\psi \wedge \neg\psi^\top \wedge \neg\psi^\perp$ is not a satisfying formula. It returns the two propositional formulae – $\psi^\top$ and $\psi^\perp$ – as a compact representation of the partitioning.

6.2. Program-level verifier

We now adapt the state-level verifier from Sect. 5.2 in Algorithm 1 to use program-level counterexamples [WJV+15] for generating conflicts. We show the adapted algorithm in Algorithm 5. The important change is that we now work at the program level and thus use a different counterexample generation. All other changes are just to match the syntax of this section.

Constructing the Markov chain GENERATEMC($\mathcal{S}, \mathbf{v}$): This procedure first constructs the instantiation $\mathcal{S}[\mathbf{v}]$, i.e., a program without holes, from $\mathcal{S}$ and $\mathbf{v}$, as in Fig. 3b: We replace each hole declaration in a sketch by a declaration of integers in the PRISM program: Thus

```
hole h either ...;    becomes    const int h;
```

Constraints in the sketch are removed, as they are handled by the synthesiser. From the perspective of the implementation, the passed program is a standard PRISM program with open constants. The sketch is parsed *once* and appropriately instantiated with the realisations selected by the synthesiser. The underlying MC $[\![\mathcal{S}[\mathbf{v}]]\!]$

```
module rex
s : [0..3] init 0;
s = 0 -> 0.5: s'=1 + 0.5: s'=3;
s = 1 -> 1: s'=s+2;
s >= 2 -> 1: s'=s;
endmodule
```

(a) Program (repeated)

```
const int X = 1, Y = 3;
...
module rex
s : [0..3] init 0;
s=0 -> 0.5: s'=X + 0.5: s'=Y;
endmodule
```

(b) CE for upper bound

```
module rex
s : [0..3] init 0;
s=0 -> 0.5:s'=X + 0.5:s'=Y;
s=3 -> s'=3
endmodule
```

(c) CE for lower bound

Fig. 12. Program **(a)** with CEs for **(b)** $\mathbb{P}_{\leq 2/5}(\Diamond \texttt{s=3})$ and **(c)** $\mathbb{P}_{> 3/5}(\Diamond \texttt{s=2})$

of the program instantiation is obtained via standard procedures, where transitions in the MC are annotated with the commands that generated them.

Computing program-level CEs The method COMPUTECOUNTEREXAMPLE($\mathcal{S}[\mathbf{v}]$, D, φ) computes program-level CE as analogues of critical sets. These CEs are defined on commands rather than on states. Let $\mathcal{L} = (\mathsf{Var}, \mathsf{Cmd})$ be a program and $\mathsf{Cmd}' \subseteq \mathsf{Cmd}$. Let $\mathcal{L}_{|\mathsf{Cmd}'} := (\mathsf{Var}, \mathsf{Cmd}')$ denote the restriction of $\mathcal{L}$ to Cmd', with variables (and initial states) as in $\mathcal{L}$. Building $\mathcal{L}_{|\mathsf{Cmd}'}$ preserves non-overlapping guards, but may introduce deadlocks in $[\![\mathcal{L}_{|\mathsf{Cmd}'}]\!]$ (just like a critical set introduces deadlocks). To remedy this, we use the operation fixdl introduced for programs, see page 641.

Definition 14 (Program-level CEs) For program $\mathcal{L} = (\mathsf{Var}, \mathsf{Cmd})$ and specification Φ with $\mathcal{L} \not\models \Phi$, a *program-level counterexample* $\mathsf{Cmd}' \subseteq \mathsf{Cmd}$ is a set of commands, such that for all (non-overlapping) programs $\mathcal{L}' := (\mathsf{Var}, \mathsf{Cmd}'')$ with $\mathsf{Cmd}'' \supseteq \mathsf{Cmd}'$ it holds that $\mathsf{fixdl}(\mathcal{L}') \not\models \Phi$.

Example 27 We consider the setting from Example 24 in more detail. Recall that we consider $\mathcal{S}[\mathbf{v}]$ as in Fig. 3b (repeated in Fig. 12a) with $\Phi = \{\mathbb{P}_{\leq 2/5}[\Diamond\ \texttt{s=3}]\}$. Figure 12b shows a CE. The probability to reach s=3 in the underlying MC is $1/2 > 2/5$.

From counterexample to conflict The method GENERATECONFLICT($\mathcal{S}$, $\mathbf{v}$, Cmd) generates conflicts from counterexamples, i.e., from the set of commands Cmd. We map these commands from realisation $\mathcal{S}[\mathbf{v}]_{|\mathsf{Cmd}}$ to commands Cmd' in sketch $\mathcal{S}$ such that $\{\mathsf{cmd}[\mathbf{v}] \mid \mathsf{cmd} \in \mathsf{Cmd}'\} = \mathsf{Cmd}$. That is, we restore which holes appear in the part of the sketch leading to the CE Cmd for $\mathcal{S}[\mathbf{v}]$. The conflict $\mathit{Conflict}(\mathsf{Cmd}, \mathbf{v})(h) := \mathbf{v}(h)$ contains all $h \in \mathcal{H}$ appearing in Cmd'.

Proposition 3 If Cmd is a CE for $\mathcal{S}[\mathbf{v}]$, then for any $\mathbf{v}' \supseteq \mathit{Conflict}(\mathsf{Cmd}, \mathbf{v})$, Cmd is also a CE for $\mathcal{S}[\mathbf{v}']$.

Example 28 The CEs in Fig. 12b contain commands which depend on the realisations for holes X and Y. For these fixed values, the program rejects the specification *independent of the value for Z*, so Z is not in the conflict $\{\mathrm{X} \mapsto 1, \mathrm{Y} \mapsto 3\}$.

6.3. Program-level CEs for CEGIS

We report how to compute program-level CEs in the context of CEGIS. Therefore, we first recapture a MaxSat-based approach to computing counterexamples, show how we extend and improve that algorithm, and how to integrate it into the conflict computation.

6.3.1. A MaxSat approach for safety properties

Our approach is inspired by (and re-uses parts of) an MaxSat [BHvMW09] approach from [DJW+14] to compute program-level CEs which were proposed in [WJV+15].

Computing counterexamples The MaxSat technique computes a *minimal* program-level CE[13] violating a safety property, i.e., a reachability property with an upper bound λ.

[13] Differences with program-level CEs are discussed below.

Definition 15 (High-level CEs) ([DJW+14]) Let program $\mathcal{L} = (\mathsf{Var}, \mathsf{Cmd})$ and $\varphi = \mathbb{P}_{\leq\lambda}(\lozenge T)$ s.t. $\mathcal{L} \not\models \varphi$. The set $\mathsf{Cmd}' \subseteq \mathsf{Cmd}$ is a *high-level counterexample* if $\mathsf{fixdl}(\mathcal{L}_{|\mathsf{Cmd}'}) \not\models \varphi$. A high-level CE is *minimal*, if for every $\mathsf{Cmd}'' \subset \mathsf{Cmd}$, $\mathsf{fixdl}(\mathcal{L}_{|\mathsf{Cmd}''}) \models \varphi$.

The set Cmd' is not unique in general. The approach repeatedly solves `MaxSat` instances over the conjunction of two propositional formulae, Ξ and Υ. The formula Ξ encodes a selection of commands and (via a negation) the `MaxSat` solver minimises the size of the command set. The formula Υ encodes necessary constraints on valid CEs, e.g, it states that a command enabled in the initial state must be selected. In this way, the `MaxSat` solver returns a candidate set Cmd of commands. Using model-checking procedures it is then checked whether Cmd is sufficient to exceed the probability bound λ, i.e., whether or not it is a CE. If affirmative, Cmd by construction is a minimal CE. Otherwise, the formula Υ is strengthened to exclude Cmd and (possibly) other candidate sets. Further details about this approach are given in [Hen18].

Program-level CEs versus traditional high-level CEs We aim to reuse the high-level CE computation. We observe the following connection between high-level and program-level CEs.

Proposition 4 For safety properties, high-level and program-level CEs coincide.

Each program-level CE is trivially a high-level CE. A high-level CE is a program-level CE, as any additional command may only be enabled in unreachable or deadlock states. Otherwise, the program would contain overlapping commands, violating Definition 14. Furthermore, unreachable states are irrelevant to any property and, intuitively speaking, adding transitions to deadlocks cannot decrease the probability to reach the target states.

Remark 5 We do *not* adopt the CE notion of [WJV+15] as it is not straightforward to adapt to non-safety properties, and our notion straightforwardly yields the soundness of the verifier.

Recall that we consider PRISM programs whose guards are disjunct. This assumption is more strict than the PRISM semantics for MCs: There overlapping guards are resolved by *rescaling* (i.e., taking a uniform distribution over enabled commands) instead of non-determinism (which is the standard semantics for MDPs). Assuming non-overlapping programs is thus the same as assuming the MDP semantics for PRISM programs. This assumption is crucial, as illustrated in the example below.

Example 29 Suppose we add the command `s=0 -> s'=2` to the program in Fig. 12b. The program now has overlapping guards (in state `s=0`). Applying the alternative rescaling semantics, the probability to reach `s=3` in this program, i.e., in the underlying MC, is *reduced* to $1/4$. Thus, the extended program no longer rejects Φ.

6.3.2. *Beyond safety properties*

Liveness properties Proposition 4 is crucial for the correctness of our approach, but does not hold for liveness properties. This can be seen as follows. Suppose Cmd is a (high-level) CE for $\mathcal{L}$ and liveness property φ. Adding commands may increase the probability mass of reaching the target, as discussed in Example 20 and may thus yield a program $\mathcal{L}'$ with $\mathcal{L}' \models \varphi$. However, a high-level CE is a program-level CE only if there are no deadlocks reachable in the MC $[\![\mathcal{L}_{|\mathsf{Cmd}}]\!]$.

To obtain CEs for liveness properties, the idea, analogue to Sect. 5.2, is to trap more probability mass than $1-\lambda$ in the bad states $B := S^{T}_{=0}$ from which T is unreachable. If there is less than λ probability mass reaching T, then more than $1-\lambda$ probability mass reaches B. We thus first compute a high-level CE Cmd for reaching states in B with probability at most $1-\lambda$. This is done using the approach explained above. We then extend Cmd with commands ensuring that the states in B cannot reach T. This done by adding commands that “lock” the behaviour of states in B. Thereby, the program cannot be extended to a program with transitions from B to T. The resulting set of commands is a program-level CE as given in Definition 14. Note that these CEs are not necessarily the smallest (in terms of the number of commands) for any given $\mathcal{L}$ and φ.

Example 30 The property $\varphi = \mathbb{P}_{>3/5}(\lozenge \texttt{s=2})$ is violated by the program in Fig. 12a. Figure 12c constitutes a CE since with probability $1/2$ state `s=3` (in $S_{=0}T$) is reached. Hence, the probability to ever reach `s=2` is guaranteed to be at most $1/2$. Dropping the command with guard `s=3` is insufficient: an additional command could add transitions from `s=3` and `s=1` to `s=2`, thereby increasing the probability mass and exceeding the (lower) bound of φ.

Expected rewards For upper bounds on expected rewards, we adapt ideas of state-level CEs for rewards from [QJD+15] to the program level. Intuitively, in the underlying MC of the program, we replace the self-loops introduced by the fixdl operation by transitions to a target state. To be sound, the method has to assume that for each realisation, the probability to reach the target is one. Otherwise, the expected reward is by definition infinite, and the instantiation may be rejected. We establish soundness by extending the specification to explicitly check that the expected reward is finite. This extension amounts to specifying that the target states are reached with probability one, and this part of the specification is then also amenable to counterexample generation. Support for lower bounds on rewards requires additional assumptions and is not considered here.

6.3.3. Tightly integrating CE generation and CEGIS

So far, we considered the generation of CEs independent of their usage within CEGIS. In this section we describe the most important changes we made to improve the integration of CE generation and CEGIS. The following adaptions enable deriving conflicts that typically are smaller than the conflicts derived from the minimal CEs described above. Smaller conflicts have more potential for effective pruning and thus often improve the performance of the CEGIS framework. The conflicts we obtain using the techniques discussed below are at least as small as before. We discuss the following aspects: how to focus on commands in CEs that are "relevant"?, how to achieve more aggressive pruning by using several CEs rather than one?, and obtaining multiple CEs for various properties in a simultaneous way?

Relevant commands The `MaxSat` approach in [DJW+14] minimises over all commands. However, the size of the conflict only depends on the holes contained in the commands. Therefore, we amend the propositional formula Ξ to minimise the number of relevant commands. A command is *relevant*, if it contains a hole. Any non-relevant command can be added to a CE without negatively affecting the size of the generated conflicts. This optimisation significantly reduces the number of candidates considered during the CE generation. Formula Υ still considers all commands and enables a further restriction of the candidate command sets.

Example 31 In the sketch of Fig. 3a only the commands with guards `s=0` and `s=1` are relevant. Instead of considering up to 2^4 candidate command sets (all four commands), we only consider up to 2^2 CEs (the two relevant commands).

Multiple CEs CEs (even minimal ones) are not unique. CEs for the same realisation may lead to different conflicts. Each such CE may be useful to prune realisations. Fortunately, our approach is not restricted to using specific CEs. But minimality of CEs is crucial: if a proper subset of a CE Cmd is a CE, conflicts generated by Cmd do not prune additional realisations. We thus extend the `MaxSat` loop as follows. If we encounter a CE Cmd, we extend the propositional formula Υ by blocking all CEs which include the set of relevant commands in Cmd. We may then continue searching for further minimal CEs. Note that some generated conflicts may not prune any further realisations, as these realisations might have been considered before. Interesting future work is to investigate using additional knowledge from the synthesiser focusing on CEs that definitively prune unexplored parts of the family.

CEs for specifications with multiple properties We so far considered CE generation for each property separately. Computed CEs for a property might all be subsumed by CEs for other properties. It is thus beneficial to consider all properties at once, thereby reducing the candidate CEs. We integrated support for CE generation for multiple properties with the same target states in our prototypical implementation. For other property combinations, a naive implementation induces a severe performance hit to the `MaxSat` solver. Proper support for such property combinations is left for future work.

6.4. Support for ill-formed programs

Program sketches can induce ill-formed programs as sketch instantiations do not always induce a PRISM program. Ill-formed programs do contain, e.g.:

- deadlocks, i.e., programs in which in a (reachable) state all guards evaluate to false,
- variables that go out of bounds, or

- guards that overlap.

These phenomena are typically considered modelling errors and most model checkers may not even properly check for such errors (as these checks may be computationally expensive). During analysis, such phenomena are assumed to be absent, i.e., the analysis result of such programs is mostly unspecified. While it is convenient to adopt such a policy for sketches, the existence of ill-formed programs is more likely as a sketch describes various (possibly ill-formed) programs. Furthermore, while modelling concise sketches with small underlying MCs, it may be hard to avoid including realisations that describe ill-formed programs. Imposing constraints to avoid ill-formed programs is not helpful, especially when the sketch is used to search for a feasible realisation. Thus, it is better to assume that sketches may contain ill-formed programs. The characteristics of ill-formed programs mean that we can apply the same pruning techniques that we discussed above. Below, we briefly describe how we support sketches with invalid realisations. Observe that we assume that such programs do not satisfy the specification and should be considered as rejecting.

Out-of-bounds Variable bounds in programs should be respected. However, constraining a sketch to prevent the generation of realisations that violate these bounds might be complex. Therefore, we extended the semantics such that the underlying MC contains a sink state that reflects a variable being out-of-bounds (by checking the variable values during the MC construction from the program). The specification is extended with a property asserting that such sink state is reached with zero probability. This extension ensures that accepting assignments have no variables that go out of bounds. CEs for this property can also be generated.

Overlapping guards As pointed out earlier, overlapping guards lead to non-determinism—and thus the operational models of programs are no longer MCs. The assumption of uniform resolution of this non-determinism (as optional in PRISM) may prevent effective CE generalisation. Therefore, we reject sketch realisations with overlapping guards. In order to mitigate the performance degradation caused by returning trivial conflicts, we implement a similar extension as for out-of-bounds, in which we label states that satisfy guards of multiple commands accordingly, and extend the specification to prevent reaching these states.

Deadlocks The existence of deadlocks prevents verifying unbounded reachability properties. Therefore, most model checkers automatically apply the fixdl operation. While this operation is sensible for a single program, it is not for sketches with holes in guards, as deadlocks are problematic in CE generalisation for liveness properties. Thus, we treat them analogously to overlapping guards by labelling states accordingly.

7. Empirical Evaluation

This section contains information on the empirical evaluation of the proposed approach.

7.1. Set-up

Implementation We evaluate the synthesis framework within a prototype of Dynasty[14], a Python tool for the synthesis in sketches. As an input, it takes a (constraint-free) program sketch in the PRISM language, together with a list of options for each hole, a list of constraints, and a list of properties. Internally, sketches are translated into the JANI modelling language. Dynasty contains an implementation of CEGIS as explained in this paper, with the restriction that complete partitioning is only supported for single properties, but not for conjunctions thereof. We use (the Python bindings of) the SMT-solver Z3 [dMB08] in the synthesiser, and (the Python bindings of) the model checker STORM [DJKV17] for the verifier. In particular, we have extended STORM based on the description in Sect. 6.3.

Research questions We compare the performance of the CEGIS approach with an enumerative approach as baseline. The synthesis time of the baseline linearly depends on the number of realisations, and on the underlying MCs' size. We focus on sketches where all realisations are explored, as relevant for optimal synthesis. For concise

[14]Available open-source on https://github.com/moves-rwth/dynasty/

presentation we use `Unsat` variants of feasibility synthesis. Enumerative methods perform mostly independent of the order of enumerating realisations.

Environment All results are obtained on a Macbook MF839LL/A, within three hours and using less than 8 GB RAM.

7.2. Case studies

We consider the following three case studies:

Dynamic power management (DPM) The goal of this adapted DPM problem [BBPM99] is to trade-off power consumption of complex electronic systems for performance in a controlled fashion. We sketch a controller that decides based on the current workload expressed by the occupancy of low-priority and high-priority request buffers, inspired by [GTC15]. On buffer overflow, incoming jobs are lost. The fixed environment contains no holes. The goal is to synthesise the guards and updates to satisfy a specification with properties such as φ_1: the expected number of lost requests is below λ_1, and φ_2: the expected energy consumption is below λ_2, and φ_3: the expected number of lost low-priority requests is below λ_3.

Network intrusion This model (adapted from [KNPV09]) describes a network, in which the controller tries to infect a target node via intermediate nodes. The node has to be attacked by intruding intermediate nodes. If an attack fails, the node is temporarily on guard and more difficult to intrude. We sketch a partial strategy aiming to minimise the expected time to intrusion based on intruded nodes and the last action. Constraints encode domain-specific knowledge, e.g., we never target nodes whose neighbours are already infected.

Grid This model is based on a classical benchmark for solving partially observable MDPs (POMDPs) [KLC98]. Solving POMDPs amounts to find an observation-based strategy, i.e. a strategy that only on the basis of the observations seen so far takes a decision. This problem is undecidable for the properties we consider. Therefore, we resort to finding a deterministic k-memory strategy [MKKC99], a strategy that decides on the basis of the last k observations, such that in expectation, the strategy requires less than λ steps to reach the target. This task is still hard: We create a family describing all k-memory strategies (for some fixed k) for the POMDP. Like in [JJW+18] actions are reflected by parameters, while parameter dependencies ensure that the strategy is observation-based.

7.3. Evaluation

We evaluate the results for *DPM* and summarise further results we obtained using our implementation.

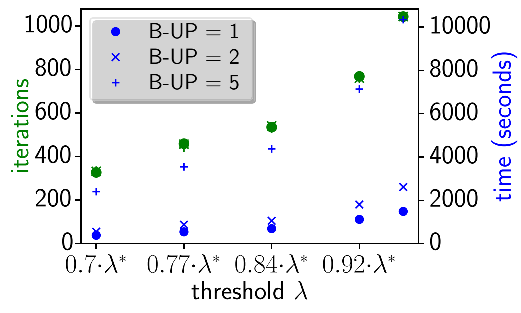
DPM *DPM* has 9 holes with 260K realisations, and realisations have 5K (reachable) states on average, ranging from 2K to 8K states. The baseline—enumerating all realisations—needs approximately 11 hours. We use this sketch to illustrate the important performance characteristics of our approach.

The performance of CEGIS significantly depends on the specification,
namely, on the thresholds appearing in the properties.

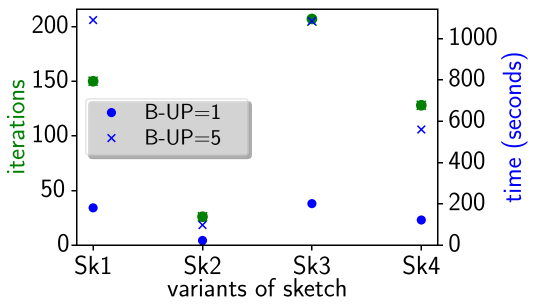
Figure 13a shows how the number of iterations (left axis, green circle) in Algorithm 2 and the runtime in seconds (right axis, blue) change for varying λ_1 for property φ_1 (plusses and crosses are explained later). We obtain a speedup of 100× over the baseline for $\lambda_1 = 0.7 \cdot \lambda^*$, dropping to 23× for $\lambda = 0.95 \cdot \lambda^*$, where λ^* is the minimal probability over all realisations. The strong dependency between performance and "unsatisfiability" is not surprising. The more unsatisfiable, the smaller the conflicts (as in [DJW+14]). Small conflicts have a double beneficial effect. First, the prototype uses an optimistic verifier searching for minimal conflicts; small conflicts are found faster than large ones. Second, small conflicts prune more realisations. A slightly higher number of small conflicts yields a severe decrease in the number of iterations. Thus

the further the threshold from the optimum, the better the performance.

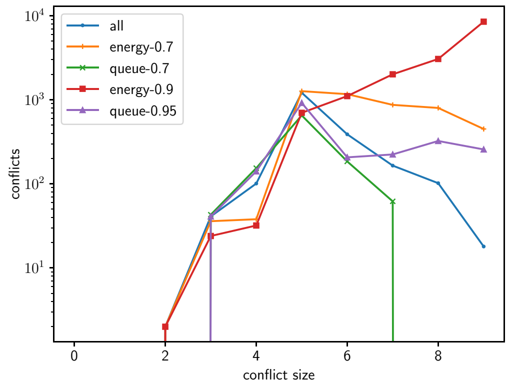
Figure 13c indicates how often (x-axis) the verifier returns conflicts of certain sizes (y-axis) for different properties (a bound on either the expected *energy* consumption or on the dropped messages in the *queue*), and



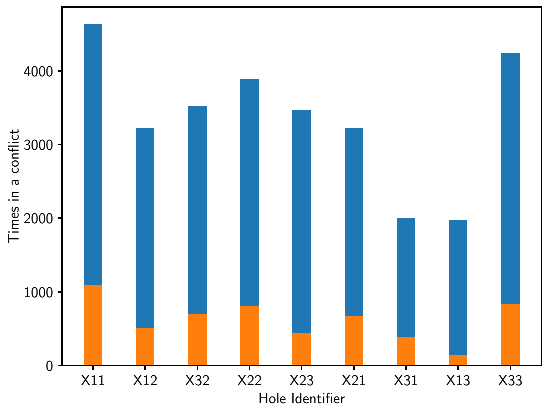
(a) Performance for varying λ



(b) Performance for varying $\mathcal{S}$



(c)



(d)

Fig. 13. Performance (runtime and iterations) on DPM

different thresholds (indicated as fractions of λ^*). The line (all) is explained later. In general, we observe that the sum over all conflicts is larger if more conflicts are large. This is natural: the less we prune, the more realisations have to be considered (yielding more conflicts). For more relaxed bounds ($0.7 \cdot \lambda^*$) more small conflicts result, because it is easier to prune with such bounds.

Reconsider now Fig. 13a. Crosses and plusses correspond to a variant of the sketch in which the state space[15] of the underlying MCs is blown up by a factor B-UP. Observe that performance degrades similarly for the baseline and our algorithm, which means that

the speedup w.r.t. the baseline is not considerably affected by the size of the underlying MCs.

We observed the same trend for various other models and specifications.

Figure 13d indicates how often (y-axis) the holes (x-axis) are included in a conflict. The orange bar is for φ_1 with threshold $0.7 \cdot \lambda^*$, whereas the blue bar is for φ_2 with threshold $0.4 \cdot \lambda^*$ and also goes from 0. The selected thresholds are irrelevant for the message: How often a hole appears varies wildly. Roughly, the higher the bar the harder it is to prune options in that hole. The hardness of a hole is (informally) correlated to the probability mass of relevant paths touching commands containing the hole. More precisely, the hardness is partially affected by the MC topology—hole X11 is in the command that is enabled in the initial state and thus, it is touched by all probability mass, and always in the conflict—and partially as a consequence of the property—for φ_2, holes X13 and X31 are equally hard as the paths through them are equally relevant for the property φ_2, but not for φ_1.

Varying the sketch (slightly) tremendously affects performance. This is indicated in Fig. 13b that illustrates the performance on variants of the original sketch with a single hole substituted by one of its options. The framework performs significantly better on sketches with holes that lie in local regions of the MC. It is easier to find CEs

[15]This blow-up introduces counters to count events. We thereby change the structure slightly.

threshold	#iters	time (minutes)	speedup
$0.7\cdot\lambda^*$	116	<1	5230
$0.8\cdot\lambda^*$	1105	27	127
$0.9\cdot\lambda^*$	9524	570	6
0.2-optimal (opt = 25.57)	2056	52	65
0.1-optimal (opt = 25.42)	12495	630	5

Table 2. Results for *Intrusion* model. The baseline algorithm would require more than 56 hours to explore the entire family.

that do not include such local regions of the MC and thus the framework can efficiently prune the corresponding part of the family. On the other hand, holes relating to states that are spread all-over the MC are harder to prune as typically very small CEs are required.

Finally, we explore the effect of a more complicated specification that has multiple (conflicting) properties: In particular, we set $\lambda_1 = 0.7 \cdot \lambda_1^*$, $\lambda_2 = 1.2 \cdot \lambda_2^*$ and $\lambda_3 = 0.7 \cdot \lambda_3^*$. Notice that each property is satisfiable on its own, but their conjunction is unsatisfiable. Figure 13c (all) indicates that we indeed obtain a smaller number of large conflicts: Some realisations may be effectively pruned by conflicts w.r.t. φ_1, whereas other realisations are easily pruned by conflicts w.r.t., e.g., φ_2.

Intrusion This sketch has 26 holes and 6800K realisations. The underlying MCs have only 500 states on average. We observe an even more significant effect of the property thresholds on the performance than for DPM, as the number of holes is larger (recall the optimistic verifier). Table 2 reports the performance for different thresholds.

For threshold $0.7\cdot\lambda^*$, the conflicts typically cover just 8 holes, which leads to a speedup of several orders of magnitude. Blowing up the MC does not affect the obtained speedups. Differences among variants of the sketches are again significant, albeit less extreme than for DPM.

Furthermore, we investigate the performance of the *almost optimal synthesis* for the same sketch and property. The last two rows of Table 2 show the results for the relaxed variant of the minimal synthesis with the tolerance value $\epsilon = 0.2$ and $\epsilon = 0.1$, respectively. These values should be compared with $\lambda = 0.8\cdot\lambda^*$ and $0.9\cdot\lambda^*$, respectively. The required precision ϵ, similarly as the threshold λ, significantly affects the performance of the synthesis process. First, we remark that the solution that we find when searching for an 0.2-optimal solution is indeed 0.022-optimal and the 0.1-optimal solution is indeed 0.016-optimal. Comparing the run-times of the almost optimal and feasibility synthesis with comparable thresholds (i.e., $0.8\cdot\lambda^*$ and $0.9\cdot\lambda^*$) demonstrates the extra time the optimal synthesis algorithm needs to find the ϵ-optimal solution before the non-existence of another improving solution is proved. The main reason for this overhead is that for optimal synthesis, we prune with respect with a factor of the λ found so far, whereas for feasibility checking λ is fixed from the start.

Grid This sketch is structurally different: only 6 holes in 3 commands and 1800 realisations, but MCs having 100K states on average. Observe that reaching the targets with some minimal expected value below implies that the goal is almost surely reached. The MCs' topology and the few commands in the sketch make pruning hard: our algorithm needs about 480 seconds and 453 iterations for $\lambda = 0.98 \cdot \lambda^*$. This is a 3.7× speedup w.r.t. the baseline algorithm. Pruning mostly takes place by considering realisations that do not reach the target almost surely. Therefore, the speedup is mostly independent of the relation between λ and λ^*. Additionally, we considered a more complicated variant of this benchmark, with 8 holes in 4 commands, yielding a family of 65K realizations. The considered specification and the average size of MCs in the family remain the same. Our algorithm shows unsatisfiability in about 1.5 hours after more than 7K iterations. For this family, the speedup is 7.9× for $\lambda = 0.98 \cdot \lambda^*$. The improved speedup for larger families indicates the potential scalability of the method.

Comparison with CEGAR We experimentally compared the approach with a CEGAR-prototype implementing the method of [ČJJK19], which applies an abstraction-refinement loop towards, e.g., feasibility synthesis on families of Markov chains. In particular, the abstraction aggregates multiple assignments in a single model to effectively reason about a family. This approach does not support multiple property specifications (as of now), and, more importantly, the algorithm is conceptually not capable of handling constraints. For *DPM* with a single property, the CEGAR-prototype is drastically faster (for unsatisfiable feasibility problems), while on *Grid*, the CEGAR-prototype is typically (often significantly) slower. Comparing prototypes is intricate, but there is a strength and weakness of the CEGIS prototype that explains the different characteristics (for unsatisfiable instances). We conclude with a remark on satisfiable instances.

Weakness: Upon invocation, the CEGIS verifier gets exactly one assignment and is (as of now) *unaware* of other possible assignments. The verifier constructs a CE which is valid for all possible extensions (cf. Definition 14), even for extensions which do not correspond to any realisation. It would be better if we compute CEs that are (only) valid for the family. The following example (exaggerating *DPM*) illustrates that considering multiple realisations at once may be helpful: Consider a family with a parametric transition (hole) from the initial state and specification that requires reaching the failure state with probability smaller than $1/10$. Assume that all 100 options lead to a failure states with probability 1. CEGIS never prunes this parameter as it is contained in the CE for every assignment, and thereby considered relevant. However, knowing that all parameter assignments lead to the failure state, makes the hole trivially not relevant. Thus, all corresponding assignments may be pruned.

Strength: The weakness is related to its strength: the verifier works with *one* concrete MC. An extreme example (exaggerating *Grid*) is a sketch with holes $h_0, \ldots, h_m$, where hole h_0 has options $1, \ldots, m$, and option i makes holes h_j with $j \neq i$ irrelevant (by the model topology). CEGIS considers a realisation, say $\{h_0 \mapsto i, \ldots, h_i \mapsto x, \ldots\}$, that rejects the specification. As holes h_j with $j \neq i$ are not relevant, CEGIS finds a conflict $\{h_0 \mapsto i, h_i \mapsto x\}$. Indeed, for every realisation, CEGIS is able to prune all but two holes. However, if the verifier would consider many realisations for h_0, it may (without advanced reasoning) generate much larger conflicts. Thus, considering a single realisation naturally fixes the context of the selected options and makes it clearer which holes are not relevant.

Satisfiable instances: For satisfiable instances, CEGIS may start with an accepting realisation and immediately terminate. The probability of this happening depends on the heuristic of the synthesiser and the fraction of satisfiable instances. Further research is required for a proper comparison.

8. Conclusion

This paper presented a counterexample-guided inductive synthesis (CEGIS) approach to synthesise finite-state probabilistic programs in a fully automated manner. The key idea is to reason about rejecting instantiations and conclude (on the basis of diagnostic feedback provided by a verifier) that other instantiations must be rejecting without analysing these instantiations separately. We have detailed this procedure for the operational model of families of Markov chains, using sub-Markov chains as counterexamples, and then lifted this approach to PRISM program sketches and program-level counterexamples. The empirical evaluation is encouraging, providing results for program sketching, controller synthesis and security.

Future work include further improving the approach: We want to develop a synthesiser that selects better instantiations. The use of evolutionary algorithms seems to be an interesting direction. We want to further improve the verifier: First, the verifier is unaware of the notion of holes and generates counterexamples that consist of a minimal number of commands containing any hole. Second, our counterexamples are on the level of states, when smaller counterexamples could be obtained on the level of transitions. Third, the focus on minimal counterexamples may create too much overhead, and a more greedy approach may be worthwhile. Finally, we want to combine the verifier with the CEGAR prototype from [ČJJK19]. With these improvements in place, an extended empirical evaluation would be worthwhile.

Acknowledgements

We thank the reviewers for their detailed and constructive comments. We thank Roman Andriushchenko for help with some experiments. M. Češka is funded by the Czech Science Foundation grant GJ20-02328Y. S. Junges was supported in part by NSF grants 1545126 (VeHICaL) and 1646208, by the DARPA Assured Autonomy program and the DARPA SDCPS program (Contract: FA8750-20-C-0156), by Berkeley Deep Drive, and by Toyota under the iCyPhy center. J.P. Katoen is funded by the ERC Advanced Research Grant 787914 FRAPPANT.

References

[ÁBD$^+$14] Ábrahám E, Becker B, Dehnert C, Jansen N, Katoen J-P, Wimmer R (2014) Counterexample generation for discrete-time Markov models: An introductory survey, Springer, vol 8483 of LNCS, pp 65–121

[ABD$^+$15] Alur R, Bodík R, Dallal E, Fisman D, Garg P, Juniwal G, Kress-Gazit H, Madhusudan P, Martin MMK, Raghothaman M, Saha S, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2015) Syntax-guided synthesis. In: Dependable software systems engineering, IOS Press, vol 40 of NATO Science for Peace and Security Series, pp 1–25

[ADK$^+$18] Abate A, David C, Kesseli P, Kroening D, Polgreen E (2018) Counterexample guided inductive synthesis modulo theories. In: CAV (1), Springer, vol 10981 of LNCS, pp 270–288

[AHL$^+$08] Antonik A, Huth M, Larsen KG, Nyman U, Wasowski A (2008) 20 years of modal and mixed specifications. Bulletin of the EATCS, 95:94–129

[ASFS18] Alur R, Singh R, Fisman D, Solar-Lezama A (2018) Search-based program synthesis. Commun ACM, 61(12):84–93

[BBPM99] Benini L, Bogliolo A, Paleologo GA, De Micheli G (1999) Policy optimization for dynamic power management. IEEE Trans CAD Integr Circuits Syst, 18(6):813–833

[BdAFK18] Baier C, de Alfaro L, Forejt V, Kwiatkowska M (2018) Model checking probabilistic systems. In: Handbook of model checking, Springer, pp 963–999

[BDH$^+$17] Budde CE, Dehnert C, Hahn EM, Hartmanns A, Junges S, Turrini A (2017) JANI: quantitative model and tool interaction. In: TACAS, vol 10206 of LNCS, pp 151–168

[BGK$^+$11] Bartocci E, Grosu R, Katsaros P, Ramakrishnan CR, Smolka SA (2011) Model repair for probabilistic systems. In: TACAS, Springer, vol 6605 of LNCS, pp 326–340

[BHvMW09] Biere A, Heule M, van Maaren H, Walsh T (eds) (2009) Handbook of Satisfiability, IOS Press, vol 185 of Frontiers in artificial intelligence and applications

[BK08] Baier C, Katoen J-P (2008) Principles of model checking MIT Press

[BKL$^+$12] Benes N, Křetínský J, Larsen KG, Møller MH, Srba J (2012) Dual-priced modal transition systems with time durations. In: LPAR, Springer, vol 7180 of LNCS, pp 122–137

[BKL$^+$15] Benes N, Kretínský J, Larsen KG, Møller MH, Sickert S, Srba J (2015) Refinement checking on parametric modal transition systems. Acta Inf 52(2-3):269–297

[BTGC16] Bornholt J, Torlak E, Grossman D, Ceze L (2016) Optimizing synthesis with metasketches. In: POPL, ACM, pp 775–788

[CČF$^+$17] Cardelli L, Češka M, Fränzle M, Kwiatkowska M, Laurenti L, Paoletti N, Whitby M (2017) Syntax-guided optimal synthesis for chemical reaction networks. In: CAV, Springer, vol 10427 of LNCS, pp 375–395

[ČCH$^+$11] Černý P, Chatterjee K, Henzinger TA, Radhakrishna A, Singh R (2011) Quantitative synthesis for concurrent programs. In: CAV, Springer, vol 6806 of LNCS, pp 243–259

[CCS14] Chaudhuri S, Clochard M, Solar-Lezama A (2014) Bridging boolean and quantitative synthesis using smoothed proof search. In: POPL, ACM, pp 207–220

[CDKB18] Chrszon P, Dubslaff C, Klüppelholz S, Baier C (2018) ProFeat: feature-oriented engineering for family-based probabilistic model checking. Formal Asp Comput 30(1):45–75

[ČDP$^+$17] Češka M, Dannenberg F, Paoletti N, Kwiatkowska M, Brim L (2017) Precise parameter synthesis for stochastic biochemical systems. Acta Inf 54(6):589–623

[CGJ$^+$16] Calinescu R, Ghezzi C, Johnson K, Pezzé M, Rafiq Y, Tamburrelli G (2016) Formal verification with confidence intervals to establish quality of service properties of software systems. IEEE Trans Rel 65(1):107–125

[CGKM12] Calinescu R, Ghezzi C, Kwiatkowska MZ, Mirandola R (2012) Self-adaptive software needs quantitative verification at runtime. Commun ACM 55(9):69–77

[CHH$^+$13] Chen T, Hahn EM, Han T, Kwiatkowska MZ, Qu H, Zhang L (2013) Model repair for Markov decision processes. In: TASE, IEEE, pp 85–92

[ČHJK19] Češka M, Hensel C, Junges S, Katoen J-P (2019) Counterexample-driven synthesis for probabilistic program sketches. In: Formal methods – the next 30 years, Springer International Publishing, vol 11800 of LNCS, pp 101–120

[Cho17] Chonev V (2017) Reachability in augmented interval Markov chains. CoRR abs/1701.02996

[ČJJK19] Češka M, Jansen N, Junges S, Katoen J-P (2019) Shepherding hordes of Markov chains. In: TACAS, Springer, vol 11428 of LNCS

[CvG$^+$17a] Calinescu R, Češka M, Gerasimou S, Kwiatkowska M, Paoletti N (2017) Designing robust software systems through parametric Markov chain synthesis. In: ICSA, IEEE, pp 131–140

[CvG$^+$17b] Calinescu R, Češka M, Gerasimou S, Kwiatkowska M, Paoletti N (2017) RODES: A robust-design synthesis tool for probabilistic systems. In: QEST, Springer, pp 304–308

[CvG$^+$18] Calinescu R, Češka M, Gerasimou S, Kwiatkowska M, Paoletti N (2018) Efficient synthesis of robust models for stochastic systems. J Syst Softw 143:140–158

[DJKV17] Dehnert C, Junges S, Katoen J-P, Volk M (2017) A storm is coming: A modern probabilistic model checker. In: CAV, Springer, vol 10427 of LNCS, pp 592–600

[DJW$^+$14] Dehnert C, Jansen N, Wimmer R, Ábrahám E, Katoen J-P (2014) Fast debugging of PRISM models. In ATVA, Springer, vol 8837 of LNCS, pp 146–162

[DKL$^+$13] Delahaye B, Katoen J-P, Larsen KG, Legay A, Pedersen ML, Sher F, Wasowski A (2013) Abstract probabilistic automata. Inf Comput 232:66–116

[dMB08] de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: TACAS, Springer, vol 4963 of LNCS, pp 337–340

[DR18] Dureja R, Rozier KY (2018) More scalable LTL model checking via discovering design-space dependencies. In: TACAS (1), Springer, vol 10805 of LNCS, pp 309–327

[FTG16] Filieri A, Tamburrelli G, Ghezzi C (2016) Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. IEEE Trans Software Eng 42(1):75–99

[GPS17] Gulwani S, Polozov O, Singh R (2017) Program synthesis. Found Trends Program Lang 4(1-2):1–119

[GS13] Ghezzi C, Sharifloo AM (2013) Model-based verification of quantitative non-functional properties for software product lines. Inf Softw Technol 55(3):508–524

[GTC15] Gerasimou S, Tamburrelli G, Calinescu R (2015) Search-based synthesis of probabilistic models for quality-of-service software engineering. In: ASE, IEEE Computer Society, pp 319–330

[Hen13] Henzinger TA (2013) Quantitative reactive modeling and verification. Comput Sci - R&D 28(4):331–344

[Hen18] Hensel C (2018) The probabilistic model checker storm: Symbolic methods for probabilistic model checking. PhD thesis, RWTH Aachen University, Germany

[HH14] Hartmanns A, Hermanns H (2014) The modest toolset: An integrated environment for quantitative modelling and verification. In: TACAS, Springer, pp 593–598

[HHZ11] Hahn EM, Hermanns H, Zhang L (2011) Probabilistic reachability for parametric Markov models. Softw Tools Technol Transf 13(1):3–19

[HKD09] Han T, Katoen J-P, Damman B (2009) Counterexample generation in probabilistic model checking. IEEE Trans Software Eng 35(2):241–257

[HKP$^+$19] Hartmanns A, Klauck M, Parker D, Quatmann T, Ruijters E (2019) The quantitative verification benchmark set. In: TACAS (1), Springer, vol 11427 of Lecture Notes in Computer Science, pp 344–350

[JHTT19] Jansen N, Humphrey L, Tumova J, Topcu U (2019) Structured synthesis for probabilistic systems. In: NFM, Springer, vol 11460 of LNCS, pp 237–254

[JJD$^+$16] Junges S, Jansen N, Dehnert C, Topcu U, Katoen J-P (2016) Safety-constrained reinforcement learning for MDPs. In: TACAS, Springer, vol 9636 of LNCS, pp 130–146

[JJW$^+$18] Junges S, Jansen N, Wimmer R, Quatmann T, Winterer L, Katoen J-P, Becker B (2018) Finite-state controllers of POMDPs using parameter synthesis. In: UAI, AUAI Press, pp 519–529

[JS17] Jha S, Seshia SA (2017) A theory of formal synthesis via inductive learning. Acta Inf 54(7):693–726

[Jun20] Junges S (2020) Parameter synthesis in Markov models. PhD thesis, RWTH Aachen University, Germany, to appear

[Kat16] Katoen J-P (2016) The probabilistic model checking landscape. In: LICS, ACM, pp 31–45

[KLC98] Kaelbling LP, Littman ML, Cassandra AR (1998) Planning and acting in partially observable stochastic domains. Artif Intell 101(1-2):99–134

[KNP11] Kwiatkowska M, Norman G, Parker D (2011) Prism 4.0: Verification of probabilistic real-time systems. In: CAV, vol 6806 of LNCS, Springer, pp 585–591

[KNPV09] Kwiatkowska MZ, Norman G, Parker D, Vigliotti MG (2009) Probabilistic mobile ambients. Theor Comput Sci 410(12-13):1272–1303

[Kre17] Kretínský J (2017) 30 years of modal transition systems: Survey of extensions and analysis. In: Models, algorithms, logics and tools, Springer, vol 10460 of LNCS, pp 36–74

[LCA$^+$18] Lanna A, Castro T, Alves V, Rodrigues G, Schobbens P-Y, Apel S (2018) Feature-family-based reliability analysis of software product lines. Inform Softw Technol 94:59–81

[LT88] Larsen KG, Thomsen B (1988) A modal process logic. In: LICS, IEEE Computer Society, pp 203–210

[MKKC99] Meuleau N, Kim K-E, Kaelbling LP, Cassandra AR (1999) Solving POMDPs by searching the space of finite policies. In: UAI, Morgan Kaufmann Publishers Inc., pp 417–426

[MMS96] Morgan C, McIver A, Seidel K (1996) Probabilistic predicate transformers. ACM Trans Program Lang Syst 18(3):325–353

[NORV15] Nori AV, Ozair S, Rajamani SK, Vijaykeerthy D (2015) Efficient synthesis of probabilistic programs. In: PLDI, ACM, pp 208–217

[QDJ$^+$16] Quatmann T, Dehnert C, Jansen N, Junges S, Katoen J-P (2016) Parameter synthesis for Markov models: Faster than ever. In: ATVA, vol 9938 of LNCS, pp 50–67

[QJD$^+$15] Quatmann T, Jansen N, Dehnert C, Wimmer R, Ábrahám E, Katoen J-P, Becker B (2015) Counterexamples for expected rewards. In: FM, Springer, vol 9109 of LNCS, pp 435–452

[RAN$^+$15] Rodrigues GN, Alves V, Nunes V, Lanna A, Cordy M, Schobbens P-Y, Sharifloo AM, Legay A (2015) Modeling and verification for probabilistic properties in software product lines. In: HASE, IEEE pp 173–180

[Ros16] Rosenblum DS (2016) The power of probabilistic thinking. In: ASE, ACM, p 3

[SDM08] Sesic A, Dautovic S, Malbasa V (2008) 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

[SLJB08] Solar-Lezama A, Jones CG, Bodik R (2008) Sketching concurrent data structures. In: PLDI, ACM, pp 136–148

[SLTB$^+$06] Solar-Lezama A, Tancau L, Bodik R, Seshia S, Saraswat V (2006) Combinatorial sketching for finite programs. In: ASPLOS, ACM, pp 404–415

[Sol13] Solar-Lezama A (2013) Program sketching. STTT, 15(5-6):475–495

[SRBE05] Solar-Lezama A, Rabbah RM, Bodík R, Ebcioglu K (2005) Programming by sketching for bit-streaming programs. In: PLDI, ACM, pp 281–294

[VK13] Varshosaz M, Khosravi R (2013) Discrete time Markov chain families: modeling and verification of probabilistic software product lines. In: SPLC Workshops, ACM, pp 34–41

[VtBLL18] Vandin A, ter Beek MH, Legay A, Lluch-Lafuente A (2018) Qflan: A tool for the quantitative analysis of highly reconfigurable systems. In: FM, Springer, vol 10951 of LNCS, pp 329–337

[WJÁ$^+$12] Wimmer R, Jansen N, Ábrahám E, Becker B, Katoen J-P (2012) Minimal critical subsystems for discrete-time Markov models. In TACAS, Springer, vol 7214 of LNCS, pp 299–314

[WJÁ$^+$14] Wimmer R, Jansen N, Ábrahám E, Katoen J-P, Becker B (2014) Minimal counterexamples for linear-time probabilistic verification. Theor Comput Sci 549:61–100

[WJV$^+$15] Wimmer R, Jansen N, Vorpahl A, Ábrahám E, Katoen J-P, Becker B (2015) High-level counterexamples for probabilistic automata. Log Methods Comput Sci 11(1)

[ZL18] Zhou W, Li W (2018) Safety-aware apprenticeship learning. In CAV'18, Springer, vol 10981 of LNCS, pp 662–680

Received 3 March 2020

Accepted in revised form 24 March 2021 by Annabelle McIver, Maurice ter Beek and Cliff Jones

Published online 9 May 2021