# PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs

No Author Given

**Abstract.** This paper presents PAYNT, a tool to automatically synthesise probabilistic programs. PAYNT enables the synthesis of finite-state probabilistic programs from a program sketch representing a finite family of program candidates. A tight interaction between inductive oracle-guided methods with state-of-the-art probabilistic model checking is at the heart of PAYNT. These oracle-guided methods effectively reason about all possible candidates and synthesise programs that meet a given specification formulated as a conjunction of temporal logic constraints and possibly including an optimising objective. We demonstrate the performance and usefulness of PAYNT using several case studies from different application domains; e.g., we find the optimal randomized protocol for network stabilisation among 3M potential programs within minutes, whereas alternative approaches would need days to do so.

## 1 Introduction

Probabilistic programs are a powerful modelling language to describe systems containing probabilistic uncertainty. Their correctness and efficiency can be described as a set of declarative temporal constraints. Various verification tools cater for automating their a posterior verification: does a program satisfy a specification? Here, we focus on finite-state programs and consider specifications given as (conjunction of) temporal logic constraints. The automated verification of such constraints is supported by probabilistic model checkers such as Storm [18], Prism [34] or Modest [26].

These model checkers typically require a fixed program or a fixed model. This is not always in line with their intended usage: To keep development costs manageable and development cycles fast, system designs are preferably be verified as early as possible. However, at early design stages not all system details are known or deliberately left out, and systems or their models are incomplete—they contain holes. A hole may e.g., reflect a partially implemented controller for a complex system or an unspecified component for wireless communication.

A key aspect of the design cycle is to explore these designs, i.e., to do *design space exploration.* The verification challenge now is to analyze all combinations of fixing the hole with a concrete behavior/subsystem and reveal (Pareto-)optimal designs. Alternatively, designs should be robust for engineering choices made downstream, e.g., a system should ideally not depend on the specific characteristics of a single communication interface. Verifying that every combination of options satisfies the specification ensures that changes in available components do not need to trigger a redesign.

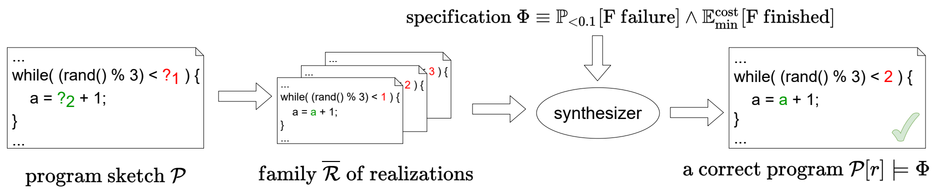
specification $\Phi \equiv \mathbb{P}_{<0.1}[\mathrm{F}\ \text{failure}] \wedge \mathbb{E}^{\text{cost}}_{\min}[\mathrm{F}\ \text{finished}]$
... while( (rand() % 3) < ?1 ) { a = ?2 + 1; } ...
while( (rand() % 3) < 1 ) { a = a + 1; }
synthesizer
... while( (rand() % 3) < 2 ) { a = a + 1; } ...
program sketch $\mathcal{P}$
family $\overline{\mathcal{R}}$ of realizations
a correct program $\mathcal{P}[r] \models \Phi$

**Fig. 1.** The workflow of the synthesis process.

The application areas above require to reason about the presence and absence of designs (aka: realizations) satisfying a specification in a family of designs. To allow for efficient reasoning it is crucial that this family is concisely represented. A convenient way to describe such a family is to use *sketching* [44, 1]. A sketch can be thought of as a program (or model) with holes, naturally fitting the use case outlined above.

Clearly, enumerating single realizations is unfeasible in the light of the combinatorial *design space explosion*. Instead, the prevalent approach connected with sketching is based on inductive synthesis. The idea is to analyze a single realization and generalize the results to a set of realizations, often using the notion of counterexamples. In probabilistic programs, such a notion is challenging, as counterexamples are typically complex objects.

Driven by a range of applications, there has been significant algorithmic progress in the analysis of probabilistic program sketches and temporal logic constraints over the last years. Baier *et al.* [13] explored the use of symbolic model-checking methods so as to consider sets of realizations at once. Češka *et al.* [11] used abstraction-refinement on sets of realizations and complemented this with a counterexample-guided inductive synthesis approach [10]. The latter two approaches have recently been integrated [2] and yield a speed up of multiple orders of magnitude over a baseline that enumerates all realizations.

This paper presents PAYNT (Probabilistic progrAm sYNThesizer) that takes a program sketch, concisely describing a finite family of finite Markov chains (MCs), and a specification, and finds a family member (aka: realization) that (potentially optimally) satisfies the specification, see Fig. 1. The design of PAYNT is rooted in oracle-guided synthesis and enables the flexible combination of a variety of state-of-the-art algorithms. For efficiency purposes, key algorithms are built within the Storm [18] model checker that dominated recent tool comparisons [23]. To deliver flexibility, the tool is built in a modular fashion on top of a python API. To ease the learning curve, the tool takes a conservative extension to the widespread Prism language as input.

PAYNT aims at two user groups: First, it provides a development platform for alternative algorithmic approaches, e.g. exploiting recent neurosymbolic approaches to find good designs. The tool provides the interface to define sketches and all baseline algorithms under one roof. Second, the analysis of sets of realizations is a valuable backend for automatic engines, e.g., when synthesizing finite-state controllers for partially observable MDPs (POMDPs) [32].

*Related work.* The synthesis problems for parametric probabilistic systems can be divided into two categories.

*Topology synthesis,* akin to aim of PAYNT, assumes a finite set of parameters affecting the MC topology. Finding a realization satisfying a given reachability property is NP-complete in the number of parameters [12], and can be naively solved by analysing all individual family members. An alternative [13] is to model the MC family by a Markov decision process (MDP) and use off-the-shelf MDP model-checking algorithms. The ProFeat [13] and QFLan [46] tool take this approach to quantitatively analyze alternative designs of software product lines [22, 35]. These tools are limited to small families. To improve the scalability, inductive methods based on *abstraction-refinement* over the MDP representation [11], and *counter-example guided inductive synthesis* (CEGIS) for MCs [10] have been proposed. As shown by the *Maze* model in Section 5, the topology synthesis is closely linked to controller synthesis for POMDPs, a popular model for planning in AI under uncertainty. Other recent approaches to POMDP controller synthesis include the use of neural network oracles (obtained by reinforcement learning) to guide the search [47] and adaptive learning schemes based on imitation learning [29]. Note that the problem of sketching probabilistic programs that fit given data as studied, e.g., in [38, 43], is different.

*Parameter synthesis* considers models with a fixed topology but with uncertain parameters associated to transition probabilities (or rates). It aims to analyze how the MC (or MDP) behaviour depends on the parameter values. Scalable approximate parameter synthesis techniques treat identical parameters in different transitions independently [9, 41] and have been implemented in Storm [18] and Prism [34]. Exact approaches construct rational functions for symbolic reachability probabilities [15] and were improved in [24, 17, 28]. This approach has been also applied to problems such as model repair [3, 39].

Both synthesis problems can be attacked by *search-based techniques* that do not ensure an exhaustive exploration of the parameter space. These include evolutionary techniques [25, 37] and genetic algorithms [21]. Their combination with parameter synthesis has been pursued in [7] and is implemented in the tool RODES [8] to synthesize robust systems.

## 2 Using PAYNT

We exemplify the usage of PAYNT by the following synthesis problem.

Consider a server for request processing depicted in Figure 2. Requests are generated (externally) in random intervals and upon arrival stored in a request queue of capacity $Q_{\max}$. When the queue is full, the request is lost. The server has three profiles – *sleeping*, *idle* and *active* – that differ in their power consumption. The requests are processed by the server only when it is in the *active* state. Switching from a low-energy state into the active state requires additional energy as well as an additional random latency before the request can be processed. We further assume that the power consumption of request processing depends on the current queue size. The operation time of the server is random but finite.

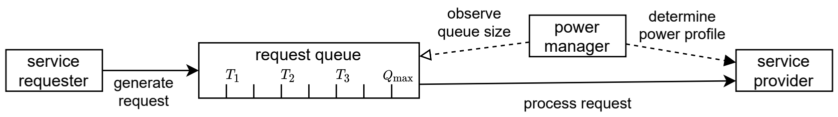

**Fig. 2.** The server for request processing.

The server is controlled by a power manager (PM) that observes the current queue size and then sets the desired power profile. More precisely, the PM distinguishes between four queue occupancy levels determined by the threshold levels $T_1, T_2$, and $T_3$. These values are unknown parameters that denote *which fraction of the queue capacity is occupied*. In other words, the PM observes the queue occupancy of the intervals: $[0, T_1], (T_1, T_2]$ etc. For each occupancy level, the PM changes to the associated power profile $P_1, \ldots, P_4 \in \{0, 1, 2\}$, where numbers 0 through 2 encode the profiles *sleeping*, *idle* and *active*, respectively.

PAYNT takes as an input a *sketch* – program description in PRISM (or JANI) language containing some undefined parameters (holes) with associated options from domains. A PRISM program consists of one or more reactive modules that may interact with each other using synchronisation. A module has a set of (bounded) variables that span its state space. Possible transitions between states of a module are described by a set of guarded commands of the form:

$$\texttt{[action]} \quad \texttt{guard} \quad \to \quad p_1 : \texttt{update}_1 \ldots\ldots + p_n : \texttt{update}_n$$

If the guard evaluates to true, an update of the variables is chosen according to the probability distribution given by expressions $p_1$ through $p_n$. The actions are used to force two or more modules to make the command simultaneously (i.e. to synchronise). The holes can appear in guards and updates. Replacing each hole with one of its options yields a complete program with the semantics given by a finite-state Markov chain. The following sketch describes the PM (the modules implementing the other components of the server are omitted for brevity).

```
module PM
    pm  :  [0..2] init 0; // 0 - sleep, 1 - idle, 2 - active
    [sync0] q <= T1*QMAX -> (pm'=P1);
    [sync0] q > T1*QMAX & q <= T2*QMAX -> (pm'=P2);
    [sync0] q > T2*QMAX & q <= T3*QMAX -> (pm'=P3);
    [sync0] q > T3*QMAX -> (pm'=P4);
endmodule
```

In our example, we consider the following holes and domains describing: the thresholds $T_1 \in \{0, 0.1, 0.2, 0.3, 0.4\}, T_2 \in \{0.5\}, T_3 \in \{0.6, 0.7, 0.8, 0.9\}$[1], the corresponding power profiles $P_1, \ldots, P_4 \in \{0, 1, 2\}$, and the queue capacity $Q_{\max} \in \{1, \ldots, 10\}$. The resulting sketch describes a *design space* of $10 \cdot 5 \cdot 4 \cdot 3^4 = 16,200$ different power managers where the average size of the underlying MC (of the complete system) is around 900 states.

[1] Note that this simply ensures that $T_1 < T_2 < T_3$. PAYNT further supports *restrictions*—additional constraints on parameter combinations.

The goal is to find the concrete power manager, i.e., the instantiation of the holes, that minimizes power consumption while the expected number of lost requests during the operation time of the server is below 1. Such specification $\Phi$ is formalized as a list of temporal logic formulae in the PRISM syntax:

```
R{"lost"}<= 1 [ F "finished" ]  R{"power"}min=? [ F "finished" ]
```

Using the sketch and the specification $\Phi$, PAYNT effectively explores the design space and finds a hole assignment inducing a program that satisfies $\Phi$, provided such assignment exists. Otherwise, it reports that such design does not exist. For the example, PAYNT produces the following output containing the hole assignment and the quality wrt. $\Phi$ of the corresponding program:

```
hole assignment: QMAX=5,T1=0,T3=0.7,P1=1,P2=2,P3=2,P4=2
R[exp]{"lost"}=0.6822759696 [F "finished"]
R[exp]{"power"}min=9100.064246 [F "finished"]
```

The obtained optimal power manager has queue capacity 5 with thresholds (after rounding) at 0, $2 = \lfloor 5 \cdot 0.5 \rfloor$ and $3 = \lfloor 5 \cdot 0.7 \rfloor$. In addition, the power manager always maintains an active profile unless the request queue is empty, in which case the device is put into an idle state. This solution leads to the expected number of lost requests of $\approx 0.68 < 1$ and the power consumption of 9,100 units. PAYNT computes this *optimal* solution in one minute. This is three times faster than a naive enumeration of all solutions.

Let us consider a more complex variant of the synthesis problem inspired by the well-studied model of a dynamical power manger for complex electronic systems [4, 20]. The corresponding sketch describes around 43M available solutions with an the average MC size of 3.6k states. While enumeration needs more than 1 month to find the optimal power manager, PAYNT solves it within 10 hours.

## 3 Synthesis of Probabilistic Programs

We formalize the synthesis problems supported by PAYNT and briefly present state-of-the-art synthesis algorithms; more details can be found in [10, 11, 2].

### Problem statement

*Sketch.* PAYNT uses sketches to define the set of designs. Let $\mathcal{P}$ be a sketch containing holes from the set $\mathcal{H} = \{H_k\}_k$ with $R_k$ being the set of options available for hole $H_k$. Let $\overline{\mathcal{R}} = \prod_k R_k$ denote the set of all hole assignments (realizations), $\mathcal{P}[r]$ denote the program induced by a substitution $r \in \overline{\mathcal{R}}$ and $\mathcal{D}_r$ denote the underlying MC. Note that the set $\overline{\mathcal{R}}$ is exponential in $|\mathcal{H}|$.

*Specification.* PAYNT supports conjunctions of specifications with reachability and expected rewards. For a set $T$ of target states, *reachability* properties $\varphi \equiv \mathbb{P}_{\bowtie\lambda}[\mathrm{F}\ T]$ with $\lambda \in [0, 1]$ and $\bowtie \in \{<, \leq, >, \geq\}$ express that the probability to reach $T$ relates to $\lambda \in [0, 1]$ according to $\bowtie$. *Expected reward* properties

$\varphi \equiv \mathbb{E}_{\bowtie\lambda}[\mathrm{F}\ T]$ express that the expected reward accumulated before $T$ is reached relates to $\lambda \in \mathbb{R}^+$ according to $\bowtie \in \{<, \leq\}$. Let $\mathcal{P}[r] \models \varphi$ denote that the program $\mathcal{P}[r]$ induced by the realisation $r$ satisfies $\varphi$. For a specification $\Phi = \{\varphi_i\}_{i \in I}$ given by a finite set of properties, we write $\mathcal{P}[r] \models \Phi$ to denote $\forall i \in I : \mathcal{P}[r] \models \varphi_i$.

*Synthesis problems.* PAYNT is able to answer two types of synthesis questions for a PRISM sketch $\mathcal{P}$ with a set $\overline{\mathcal{R}}$ of realizations and a specification $\Phi$:

**Feasibility**: Find a realization $r \in \overline{\mathcal{R}}$ such that $\mathcal{P}[r] \models \Phi$.

**Maximality**: For property $\varphi_{\max}$, find a realization $r^* \in \overline{\mathcal{R}}$ such that

$$r^* \in \arg\max_{r \in \overline{\mathcal{R}}} \{\mathbb{P}[\mathcal{P}[r] \models \varphi_{\max}] \mid \mathcal{P}[r] \models \Phi\} .$$

Variants of the maximal synthesis problem for expected rewards and minimization are defined analogously. PAYNT also supports a relaxed variant of maximal synthesis, *$\varepsilon$-maximal synthesis*: find a realization $r^*$ such that $\mathcal{P}[r^*] \models \Phi$ and $\mathbb{P}[\mathcal{P}[r^*] \models \varphi_{\max}] \geq (1-\varepsilon) \cdot \max_{r \in \overline{\mathcal{R}}} \{\mathbb{P}[\mathcal{P}[r] \models \varphi_{\max}] \mid \mathcal{P}[r] \models \Phi\}$ for a given value $\varepsilon \in (0, 1]$.

## Existing synthesis methods

Synthesis methods can be classified into two orthogonal groups: i) *complete* methods allowing to prove non-existence or optimally of the given problem, and ii) *incomplete* methods leveraging various smart search strategies and evolutionary algorithms [25, 37, 21]. While its architecture is flexible, the current release of PAYNT is built around state-of-the-art *complete* methods. As a baseline and reference algorithm, the tool implements the so-called *one-by-one approach* [14] which simply enumerates through each realization $r \in \overline{\mathcal{R}}$. The design-space explosion renders this approach unusable for large problems, necessitating the usage of advanced techniques that exploit any structure of the family of MCs.

*Oracle-guided synthesis.* At the heart of PAYNT is an oracle-guided inductive synthesis approach [30, 45, 31]. A *learner* selects a realization $r$ and passes it to an *oracle*. The oracle answers whether $r$ satisfies $\Phi$ and, crucially, gives additional information, usually a counter-example (CE), whenever this is not the case. PAYNT implements two orthogonal different oracles: (a) an *inductive* oracle CE examines single realizations to infer statements about other realizations [10]. (b) a *deductive* oracle AR (Abstraction Refinement) argues about sets of realizations by considering (an aggregation of) these realizations at once [11]. PAYNT supports the combined use of these two oracles as a hybrid synthesis method [2].

Figure 3 [2] illustrates the communication between the learner and the two oracles. The Abstr-Oracle analyzes a sub-family $\mathcal{R}$ with 3 possible outcomes: 1) it proves that all its realizations satisfy $\Phi$, i.e., that the synthesis problem is feasible, or 2) it proves that all its realizations violate $\Phi$, i.e., the learner can

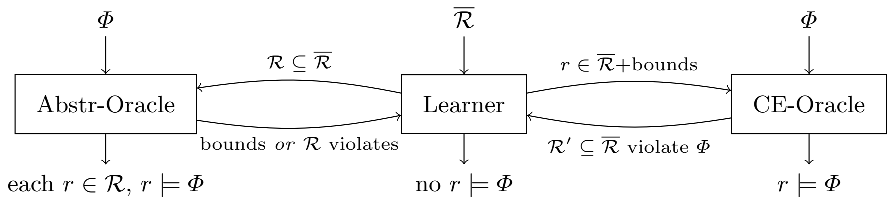
$\Phi$
$\overline{\mathcal{R}}$
$\Phi$
Abstr-Oracle
Learner
CE-Oracle
$\mathcal{R} \subseteq \overline{\mathcal{R}}$
bounds *or* $\mathcal{R}$ violates
$r \in \overline{\mathcal{R}}$+bounds
$\mathcal{R}' \subseteq \overline{\mathcal{R}}$ violate $\Phi$
each $r \in \mathcal{R}$, $r \models \Phi$
no $r \models \Phi$
$r \models \Phi$

**Fig. 3.** Oracle-guided synthesis (adapted from [2]).
.

safely discard $\mathcal{R}$, or 3) the analysis is inconclusive and it returns safe bounds on the best- and worst-case behavior of all realizations in $\mathcal{R}$ wrt. $\Phi$. The CE-Oracle analyzes a realization $r$ and either proves that $r$ satisfies $\Phi$ or it generalizes $r$ into a subfamily $\mathcal{R}'$. The learner can discard $\mathcal{R}'$ since it is guaranteed that all its realizations violate $\Phi$. In the hybrid approach, the CE-Oracle exploits the bounds in order to compute smaller CEs allowing a better generalization. The learner maintains a queue of subfamilies $\mathcal{R}' \subseteq \overline{\mathcal{R}}$ that has to be further processed and also controls which oracle is used based on their previous performance.

## 4 Tool Architecture of PAYNT

PAYNT is implemented on top of the probabilistic model checker Storm [18]. While the high-performance parts were implemented in C++, we use a python API to flexibly construct the overall synthesis loop. For SMT-solving, we use Z3 [36]. PAYNT takes a PRISM [34] or JANI [5] sketch and a set of temporal properties, and returns a satisfying realization, if such exists. Otherwise, it reports that no such realization exists.

Figure 4 depicts a high-level view on the tool architecture, which primarily consists of components for family handling (**purple**), chain building (**green**) and model checkers (**red**).

The *family handlers* are used to store information about the previously covered design space: Member enumeration simply iterates over all realizations. The SAT representation stores a SAT-formula describing unexplored realizations and uses the SMT solver Z3 to retrieve the next candidate realization. The subfamily queue stores a collection of unexplored subfamilies and refines these subfamilies as hyper-rectangles. The *chain builders* take as input a single assignment $r \in \mathcal{R}$ or a set $\mathcal{R}' \subseteq \mathcal{R}$ of realizations, and produce an representation of the MC or a quotient MDP, respectively in the internal memory model of the model checkers. The *model checkers* are then used to verify these chains. They either output yes/no or, in the case of MDPs, provide lower and upper bounds on satisfiability probabilities. PAYNT includes a module for counterexample generation by using either a MaxSat [16, 48] or a greedy state-expansion [2] approach.

Figure 4 also illustrates three analysis loops that mirror the behaviour of 1-by-1 enumeration (the baseline), CEGIS and AR. The 1-by-1 approach simply iterates over all possible realizations until a satisfying one is obtained. The CEGIS

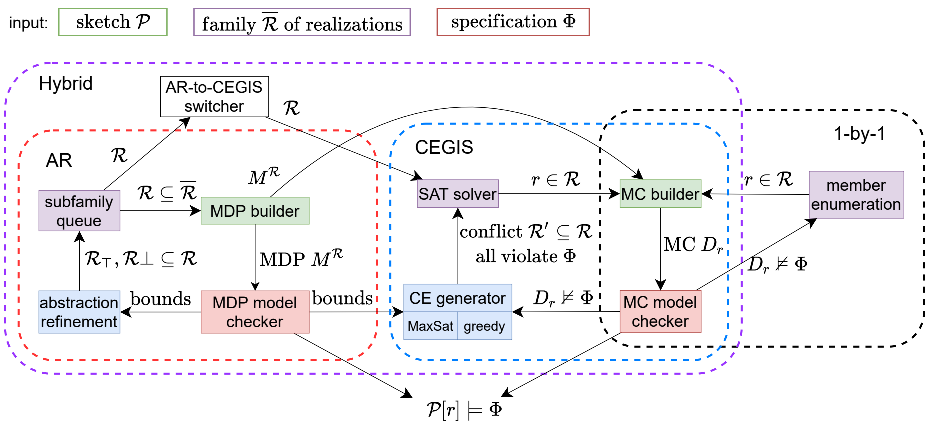

**Fig. 4.** The tool architecture

loop additionally constructs counterexamples to each unsatisfying realization $r \in \mathcal{R}$, yielding a whole subset $\mathcal{R}' \subseteq \mathcal{R}$ of realizations that are pruned from the family. In contrast to this enumeration, the AR loop constructs and model checks MDPs from the subfamily queue and subsequently refines these subfamilies if the obtained bounds on satisfiability yield inconclusive results.

The hybrid approach combines both AR and CEGIS approaches and switches between the two loops mid-execution. In particular, the integrated method executes the abstraction-refinement loop and, whenever it encounters an undecidable family that needs to be split, CEGIS takes a chance at analyzing it for a limited time period. If some family members are excluded based on a counterexample, the CEGIS engine updates the corresponding SAT representation to ensure it does not analyze the same member twice. There are two additional links that couple the AR and CEGIS loops and enable efficient integrated analysis. First is the use of bounds from MDP model checking during the greedy CE generation to allow the construction of larger family-aware conflicts. Since these bounds are associated with the states of the quotient MDP $M^{\mathcal{R}}$ for the (sub-)family and counterexamples are constructed as sub-MCs of the MC $\mathcal{D}_r$, $r \in \mathcal{R}$, in the integrated setting we construct $\mathcal{D}_r$ directly from $M^{\mathcal{R}}$, to save time on converting bound values between the two chains.

The implementation of PAYNT is composed of *30* Python modules containing *7k* source lines of code. These metrics consider only our implementation and do not include the extensions contributed to STORM and its Python API, invoked by PAYNT. All modules adhere to coding conventions for the Python code *PEP 8* [42, 40] and are documented with *Sphinx* for automatic generation of documentation. The specific logic components are tested with unit tests to maintain their correct functionality. Regression tests verify the accuracy and correctness of the synthesis results. Our tests currently cover more than *90%* of the source code lines.

| model | number of parameters | family size | average MC size | 1-by-1 enumeration | tool performance hard | tool performance easy |
|---|---|---|---|---|---|---|
| *DPM* | 16 | 43M | 3.6k | 35 days * | 9.3 h | 1.1 h |
| *Maze* | 22 | 9.4M | 0.2k | 1.8 days * | 1 h | 54 min |
| *Herman* | 7 | 3.1M | 1.1k | 1.5 days * | 17 min | 1.1 min |
| *Pole* | 17 | 1.3M | 5.6k | 1 day * | 8.5 min | 5 s |
| *Grid* | 8 | 65k | 1.2k | 32 min | 37 s | 21 s |

**Table 1.** Case study statistics and PAYNT synthesis times versus the naive 1-by-1 enumeration. Two problems per model are considered: an optimal synthesis problem (hard) and a feasibility problem (easy). In both cases, all realizations need to be explored to prove optimality and unsatisfiability, resp. Values indicated with $^*$ are estimates.

## 5 Performance Evaluation and Applicability

Table 1 lists the results of PAYNT on two variants (hard and easy) of five different case studies from various domains taken from [10, 11]. Further on, we demonstrate the applicability of PAYNT and interpret the synthesis results for two of these case studies. All experiments are run on an Ubuntu 19.04 machine with Intel i5-8300H (4 cores at 2.3 GHz) and using up to 8 GB RAM, with all the algorithms being executed single-threaded.

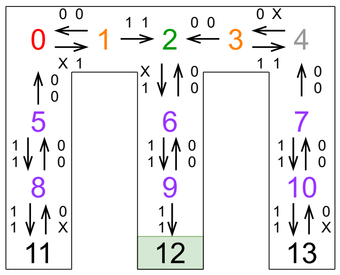

**Fig. 5.** The spatial structure of *Maze*. Cells with identical sets of surrounding walls are depicted with similar colors. The arrows depict the synthesized controller.

*Maze.* This synthesis problem can be seen as an instance of POMDP controller synthesis. A robot is deployed at a random location inside a known maze, see Fig. 5. The robot is only equipped with a simple wall sensor, and cannot distinguish maze cells with identical sets of surrounding walls such as cells 1 and 3, and cells 11 through 13. Observation-equivalent cells are indicated by the same color in Fig. 5. Possible actions are movements in the four cardinal directions. Movements are subject to a random error: e.g., upon moving east, with a small probability the robot actually moves west. We sketch a robot controller that helps it to reach the exit of the maze (cell 12). The controller may use a single bit of memory initially having the value 0. The holes in this sketch are taken actions (where to steer, how to change the memory bit) based on the current observation (detected walls, current memory state). This sketch describes a family of 9.4M candidate programs. Our goal is to find a realization that minimizes the expected number of steps to reach the exit.

Using the inductive synthesis techniques, PAYNT explores the set of candidate realizations in an hour (1-by-1 enumeration takes more than one day) and synthesizes the controller depicted in Fig. 5. Here arrows represent the steering direction based on the current memory value (number at the base of an arrow), as well as the corresponding memory update (number at the tip of an arrow).

For instance, a robot in cell 1 goes west if the memory value is 0 and goes east otherwise, without changing the memory in either case. A robot at cell 0 always goes east and sets its memory bit to 1. The synthesized controller is optimal. If a robot reaches a cell with a unique set of enclosing walls (cells 0, 2 and 4), then it knows its precise position within the maze and can navigate to the exit. Similarly, navigating north from cells 11 or 13 ensures to eventually reach cells 0 or 4. If the robot is deployed in an orange or purple cell, then it has to 'try' one possible direction in order to recognize its position within the maze. For example, a robot deployed at cells 5-10 will first go north (recall that the initial memory value is 0), from where it can determine its cell. Note that in this observation group it is indeed more beneficial to first explore north since the robot is twice as likely to be initially deployed at locations 5/7/8/10, as compared to locations 6 and 9. The expected time to reach the exit for this policy is $\approx$9.8 steps. Note that this cannot be improved by adding more memory to the controller.

*Herman.* This case study considers a token ring with an odd number of stations that are connected by a unidirectional ring. Each station has a Boolean flag, observable by itself and by its successor in the ring. A station has a token when the two flags it observes are identical. A good configuration is a situation in which only one station has a token. All other configurations are faulty. A token protocol is self-stabilizing, if the ring gets from a faulty configuration into a good configuration. The performance can be measured as stabilization time, i.e., the expected number of rounds to reach a good configuration.

We sketch a variant of *Herman's randomized self-stabilization protocol* [27, 33, 6]. In this protocol, all stations behave the same[2]. The protocol is synchronized, and in every round a station without token flips its flag. Every station that has a token must *choose* whether to pass a token (by setting its flag accordingly). In the original protocol this choice is the resolved on a single (biased) coin flip. We are interested in the synthesis of alternatives. We give each station an additional single bit of memory and the choice between 25 different coin biases. The parameters in the sketch are the choice of a coin based on the memory value as well as the memory updates. By resolving the choices, we obtain the same protocol for each station. The parameter combinations yield a family of 3.1M programs and the goal of the synthesizer is to identify the one that minimizes stabilization time from an initial configuration (all flags true). For a sketch describing a system with 5 stations, PAYNT finds the optimal protocol in around 18 minutes, while the 1-by-1 enumeration takes more than a day. The obtained optimal strategy relies on initially using the most fair coins available (bias $\approx 0.25$) and keeping the memory bit at 1. Whenever a process eventually decides to keep the token, the memory is reset to 0 and the process starts using highly unfair coins (bias $\approx 0.07$), implying that the process is more likely to keep its token for a long time until it is eventually passed further. Using this strategy, the system can on average stabilize in four rounds.

[2] In such anonymous networks, stabilization cannot be solved in a deterministic way [19].

## References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD). pp. 1–17 (October 2013)
2. Andriushchenko, R., Češka, M., Junges, S., Katoen, J.P.: Inductive synthesis for probabilistic programs reaches new horizons. In: TACAS. Lecture Notes in Computer Science, Springer (2021), (to appear, see https://arxiv.org/abs/2101.12683)
3. Bartocci, E., Grosu, R., Katsaros, P., Ramakrishnan, C.R., Smolka, S.A.: Model repair for probabilistic systems. In: TACAS'11. LNCS, vol. 6605, pp. 326–340 (2011)
4. Benini, L., Bogliolo, A., Paleologo, G.A., Micheli, G.D.: Policy optimization for dynamic power management. IEEE Trans. on CAD of Integrated Circuits and Systems **18**(6), 813–833 (1999)
5. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: POPL'16. p. 775–788. Association for Computing Machinery (2016)
6. Bruna, M., Grigore, R., Kiefer, S., Ouaknine, J., Worrell, J.: Proving the Herman-protocol conjecture. In: ICALP. LIPIcs, vol. 55, pp. 104:1–104:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
7. Calinescu, R., Češka, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N.: Efficient synthesis of robust models for stochastic systems. J. of Systems and Softw. **143**, 140–158 (2018)
8. Calinescu, R., Češka, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N.: RODES: A robust-design synthesis tool for probabilistic systems. In: QEST. pp. 304–308. Springer (2017)
9. Češka, M., Dannenberg, F., Paoletti, N., Kwiatkowska, M., Brim, L.: Precise parameter synthesis for stochastic biochemical systems. Acta Inf. **54**(6), 589–623 (2017)
10. Češka, M., Hensel, C., Junges, S., Katoen, J.P.: Counterexample-driven synthesis for probabilistic program sketches. In: FM. LNCS, vol. 11800, pp. 101–120. Springer (2019)
11. Češka, M., Jansen, N., Junges, S., Katoen, J.P.: Shepherding hordes of markov chains. In: TACAS. LNCS, vol. 11428, pp. 172–190. Springer (2019)
12. Chonev, V.: Reachability in augmented interval Markov chains. In: RP'2019. LNCS, vol. 11674, pp. 79–92. Springer (2019)
13. Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: ProFeat: feature-oriented engineering for family-based probabilistic model checking. Formal Asp. Comput. **30**(1), 45–75 (2018)
14. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. Int. J. on Softw. Tools for Technol. Transf. **14**, 589–612 (2012)
15. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: ICTAC. LNCS, vol. 3407, pp. 280–294. Springer (2004)
16. Dehnert, C., Jansen, N., Wimmer, R., Ábrahám, E., Katoen, J.P.: Fast debugging of PRISM models. In: ATVA. LNCS, vol. 8837, pp. 146–162. Springer (2014)
17. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruintjes, H., Katoen, J.P., Ábrahám, E.: PROPhESY: A PRObabilistic ParamEter SYNnthesis Tool. In: CAV'15. LNCS, vol. 9206, pp. 214–231. Springer (2015)
18. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is coming: A modern probabilistic model checker. In: CAV. LNCS, vol. 10427, pp. 592–600. Springer (2017)

19. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
20. Gerasimou, S., Tamburrelli, G., Calinescu, R.: Search-based synthesis of probabilistic models for quality-of-service software engineering (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 319–330 (Nov 2015)
21. Gerasimou, S., Calinescu, R., Tamburrelli, G.: Synthesis of probabilistic models for quality-of-service software engineering. Autom. Softw. Eng. **25**(4), 785–831 (2018)
22. Ghezzi, C., Sharifloo, A.M.: Model-based verification of quantitative non-functional properties for software product lines. Inf. & Softw. Technol. **55**(3), 508–524 (2013)
23. Hahn, E.M., Hartmanns, A., Hensel, C., Klauck, M., Klein, J., Kretínský, J., Parker, D., Quatmann, T., Ruijters, E., Steinmetz, M.: The 2019 comparison of tools for the analysis of quantitative formal models - (qcomp 2019 competition report). In: TACAS (3). Lecture Notes in Computer Science, vol. 11429, pp. 69–92. Springer (2019)
24. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. Int. J. on Softw. Tools for Technol. Transf. **13**(1), 3–19 (2011)
25. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Comp. Surveys **45**(1), 11:1–11:61 (2012)
26. Hartmanns, A., Hermanns, H.: The modest toolset: An integrated environment for quantitative modelling and verification. In: TACAS. pp. 593–598. Springer (2014)
27. Herman, T.: Probabilistic self-stabilization. Inf. Process. Lett. **35**(2), 63–67 (1990)
28. Hutschenreiter, L., Baier, C., Klein, J.: Parametric markov chains: PCTL complexity and fraction-free gaussian elimination. In: GandALF. EPTCS, vol. 256, pp. 16–30 (2017)
29. Inala, J.P., Bastani, O., Tavares, Z., Solar-Lezama, A.: Synthesizing programmatic policies that inductively generalize. In: International Conference on Learning Representations (2020)
30. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE. p. 215–224. ACM (2010)
31. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Informatica **54**(7), 693–726 (2017)
32. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artificial intelligence **101**(1-2), 99–134 (1998)
33. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic verification of Herman's self-stabilisation algorithm. Formal Aspects of Computing **24**(4), 661–670 (2012)
34. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. LNCS, vol. 6806, pp. 585–591. Springer (2011)
35. Lanna, A., Castro, T., Alves, V., Rodrigues, G., Schobbens, P.Y., Apel, S.: Feature-family-based reliability analysis of software product lines. Inf. and Softw. Technol. **94**, 59–81 (2018)
36. Lindemann, C.: Performance modelling with deterministic and stochastic Petri nets. SIGMETRICS Perform. Eval. Rev. **26**(2), 3 (1998)
37. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: WOSP/SIPEW. pp. 105–116. ACM (2010)
38. Nori, A.V., Ozair, S., Rajamani, S.K., Vijaykeerthy, D.: Efficient synthesis of probabilistic programs. In: PLDI'14. pp. 208–217. ACM (2015)
39. Pathak, S., Ábrahám, E., Jansen, N., Tacchella, A., Katoen, J.P.: A greedy approach for the efficient repair of stochastic models. In: NFM'15. LNCS, vol. 9058, pp. 295–309. Springer (2015)

40. Peters, T.: The Zen of Python. PEP 20 (2004), https://www.python.org/dev/peps/pep-0020/
41. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.P.: Parameter synthesis for Markov models: Faster than ever. In: ATVA'16. LNCS, vol. 9938, pp. 50–67 (2016)
42. van Rossum, G., Warsaw, B., Coghlan, N.: Style guide for Python code. PEP 8 (2001), https://www.python.org/dev/peps/pep-0008/
43. Saad, F.A., Cusumano-Towner, M.F., Schaechtle, U., Rinard, M.C., Mansinghka, V.K.: Bayesian synthesis of probabilistic programs for automatic data modeling. Proceedings of the ACM on Programming Languages **3**(POPL), 1–32 (2019)
44. Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, USA (2008)
45. Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioğlu, K.: Programming by sketching for bit-streaming programs. In: PLDI'05. pp. 281–294. ACM (2005)
46. Vandin, A., ter Beek, M.H., Legay, A., Lluch-Lafuente, A.: Qflan: A tool for the quantitative analysis of highly reconfigurable systems. In: FM. LNCS, vol. 10951, pp. 329–337. Springer (2018)
47. Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. In: International Conference on Machine Learning. pp. 5045–5054. PMLR (2018)
48. Wimmer, R., Jansen, N., Vorpahl, A., Ábrahám, E., Katoen, J.P., Becker, B.: High-level counterexamples for probabilistic automata. Logical Methods in Computer Science **11**(1) (2015)