Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants

Abstract. Essential tasks for the verification of probabilistic programs include bounding expected outcomes and proving termination in finite expected runtime. We contribute a simple yet effective *inductive synthesis* approach for proving such *quantitative reachability properties* by generating *inductive invariants* on *source-code level*. Our implementation shows promise: It finds invariants for (in)finite-state programs, can beat state-of-the-art probabilistic model checkers, and is competitive with modern tools dedicated to invariant synthesis and expected runtime reasoning.

1 Introduction

Reasoning about reachability probabilities is a foundational task in the analysis of randomized systems. Such systems are (possibly infinite-state) *Markov chains*, which are typically described as *probabilistic programs* – imperative programs that may sample from probability distributions. We contribute a method for proving bounds on *quantitative properties* of probabilistic programs, which finds *inductive invariants* on *source-code level* by *inductive synthesis*. We discuss each of these ingredients below, present our approach with a running example in Sect. 2, and defer a detailed discussion of related work to Sect. 8.

1) *Quantitative Reachability Properties.* We aim to verify properties such as *"is the probability of reaching an error at most 1%?"* More generally, our technique proves bounds on the expected value of a probabilistic program terminating in designated states (see Sect. 2.1). Various verification problems are ultimately solved by bounding quantitative reachability properties (cf. [7,46]). Further examples of such problems include *"does a program terminate with finite expected runtime?"* and *"is the expected sum of program variables x and y at least one?"*

2) *Inductive Invariants.* An inductive invariant is a *certificate* that witnesses a certain quantitative reachability property. Quantitative (and qualitative) reachability are typically captured as least fixed points (cf. [51,46,7]). For upper bounds, this characterization makes it natural to search for a prefixed point – the inductive invariant – that, by standard fixed point theory [55], is greater than or equal to the least fixed point. Our invariants assign every state a quantity. If the initial state is assigned a quantity below the desired threshold, then the invariant certifies that the property in question holds. We detail quantitative inductive invariants in Sect. 2.2; we adapt our method to lower bound reasoning in Sect. 6.

3) *Source-Code Level.* We consider probabilistic programs over (potentially unbounded) integer variables that conceptually extend while-programs with coin

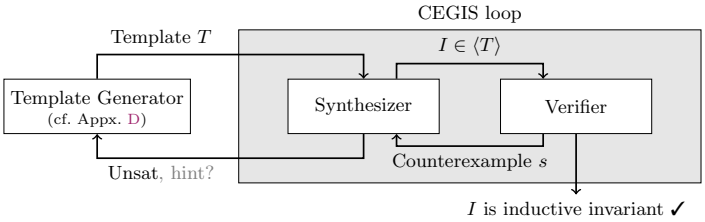


Fig. 1: Our CEGIS framework for synthesizing quantitative inductive invariants.

flips, see e.g. Fig. 2.[1] We exploit the program structure to reason about infinite-state (and large finite-state) programs: We *never* construct a Markov chain but find *symbolic* inductive invariants (mapping from program states to nonnegative reals) on *source-code* level. We particularly discover inductive invariants that are piecewise linear, as they can often be verified efficiently.

4) *Inductive Synthesis.* Our approach to finding invariants, as sketched in Fig. 1, is inspired by *inductive synthesis* [4]: The inner loop (shaded box) is provided with a *template* T which may generate an infinite set $\langle T \rangle$ of instances. We then synthesize a template instance I that is an inductive invariant witnessing quantitative reachability, or determine that no such instance exists. We search for such instances in a *counterexample-guided inductive synthesis* (CEGIS) loop: The synthesizer constructs a candidate. (A tailored variant of) an off-the-shelf verifier either (i) decides that the candidate is a suitable inductive invariant or (ii) reports a counterexample state s back to the synthesizer. Upon termination (guaranteed for finite-state programs), the inner loop has either found an inductive invariant or the solver reports that the template T does not admit an inductive invariant.

Contributions. We show that inductive synthesis for verifying *quantitative reachability properties* by finding *inductive invariants* on *source-code level* is feasible: Our approach is sound for arbitrary probabilistic programs, and complete for finite-state programs. We implemented our simple yet powerful technique. The results are promising: Our CEGIS loop is sufficiently fast to support large templates and finds inductive invariants for various probabilistic programs and properties. It can prove, amongst others, upper and lower bounds on reachability probabilities and universal positive almost-termination [41]. Our implementation is competitive with three state-of-the-art tools – Storm [38], Absynth [49], and Exist [9] – on subsets of their benchmarks fitting our framework.

Applicability and Limitations. We consider programs with possibly unbounded nonnegative integer-valued variables and arbitrary affine expressions in quantitative specifications. As for other synthesis-based approaches, there are unrealizable cases – loops for which no piecewise linear invariant exists. But, if there is an invariant, our CEGIS loop often finds it within a few iterations.

[1] Prism programs can be interpreted as an implicit `while(not error-state) {...}` program – see [39] for an explicit translation.

```
1:  fail ≔ 0 ; sent ≔ 0 ;
2:  while ( sent < 8 000 000  ∧  fail < 10 ) {
3:        { fail ≔ 0 ; sent ≔ sent + 1 } [0.999] { fail ≔ fail + 1 }    }
```

Fig. 2: Model for the bounded retransmission protocol (BRP).

2 Overview

We illustrate our approach using the bounded retransmission protocol (BRP) – a standard probabilistic model checking benchmark [37,26] – modeled by the probabilistic program in Fig. 2. The model attempts to transmit 8 million packets[2] over a lossy channel, where each packet is lost with probability 0.1%; if a packet is lost, we retry sending it; if any packet is lost in 10 consecutive sending attempts ($\mathit{fail} = 10$), the *entire* transmission fails; if all packets have been transmitted successfully ($\mathit{sent} = 8\,000\,000$), the transmission succeeds.

2.1 Reachability Probabilities and Loops

We aim to reason about the transmission-failure probability of BRP, i.e. the probability that the loop terminates in a target state t with $t(\mathit{fail}) = 10$ when started in initial program state s_0 with $s_0(\mathit{fail}) = s_0(\mathit{sent}) = 0$. One approach to determine this probability is to (i) construct an explicit-state Markov chain (MC) per Fig. 2, (ii) derive its Bellmann operator Φ [51], (iii) compute its least fixed point $\mathsf{lfp}\ \Phi$ (a vector containing for *each* state the probability to reach t), e.g. using value iteration (cf. [7, Thm 10.15]), and finally (iv) evaluate $\mathsf{lfp}\ \Phi$ at s_0.

The explicit-state MC of BRP has ca. 80 million states. We *avoid* building such large state spaces by computing a symbolic representation of Φ from the program. More formally, let S be the set of all states, `loop` the entire loop (ll. 2–3 in Fig. 2), `body` the `loop`'s body (l. 3), and $[\![\texttt{body}]\!](s)(s')$ the probability of reaching state s' by executing `body` once on state s. Then the least fixed point of the `loop`'s Bellmann operator $\Phi\colon \big(S \to \mathbb{R}^{\infty}_{\geq 0}\big) \to \big(S \to \mathbb{R}^{\infty}_{\geq 0}\big)$, defined by

$$\Phi(I) \;=\; \lambda s.\; \begin{cases} 1, & \text{if } s(\mathit{fail}) = 10\ , \\ \displaystyle\sum_{s' \in S} [\![\texttt{body}]\!](s)(s') \cdot I(s'), & \text{if } s(\mathit{sent}) < 8\,000\,000 \\ & \quad \text{and } s(\mathit{fail}) < 10\ , \\ 0, & \text{otherwise}\ , \end{cases}$$

captures the transmission-failure probability for the *entire* execution of `loop` and for *any* initial state, that is, $(\mathsf{lfp}\ \Phi)(s)$ is the probability of terminating in a target state when executing `loop` on s (even if `loop` would not terminate almost-surely). Intuitively, $\Phi(I)(s)$ maps to 1 if `loop` has terminated meeting the target condition

[2] Large constants like the number of packets appear naturally in quantitative models of protocols and have a non-trivial impact on probabilities.

(transmission failure); and to 0 if `loop` has terminated otherwise (transmission success). If `loop` is still running (i.e. it has neither failed nor succeeded yet), then $\Phi(I)(s)$ maps to the expected value of I after executing `body` on state s.

2.2 Quantitative Inductive Invariants

Reachability probabilities are generally not computable for infinite-state probabilistic programs [42]. Even for finite-state programs the state-space explosion may prevent us from computing reachability probabilities exactly. However, it often suffices to know that the reachability probability is bounded from above by some threshold λ. For BRP, we hence aim to prove that $(\mathsf{lfp}\ \Phi)(s_0) \leq \lambda$.

We attack the above task by means of *(quantitative) inductive invariants*: a candidate for an inductive invariant is a mapping $I \colon S \to \mathbb{R}^{\infty}_{\geq 0}$ that one can plug into Φ; in our running example, I will map every state to a probability.

Intuitively, such a candidate I is *inductive* if the following holds: when assuming that $I(s)$ is (an over-approximation of) the probability to reach a target state upon termination of `loop` on s, then the probability to reach a target state after performing one more guarded loop iteration, i.e. executing `if` $(\, \mathit{sent} < \dots \,) \,\{\, \texttt{body} \,;\, \texttt{loop} \,\}$ on s, must be *at most* $I(s)$.

Formally, I is an inductive invariant (i.e. a prefixed point of Φ)[3] if

$$\forall s \colon \quad \Phi(I)(s) \;\leq\; I(s) \qquad \text{which implies} \qquad \forall s \colon \quad \big(\mathsf{lfp}\ \Phi\big)(s) \;\leq\; I(s)$$

by Park induction [50]. Hence, $I(s)$ bounds for each initial state s the exact reachability probability from above. If we are able to choose an inductive I that is below λ for the initial state s_0 with $\mathit{fail} = \mathit{sent} = 0$, i.e. $I(s_0) \leq \lambda$, then we have indeed proven the upper bound λ on the transmission-failure probability of our BRP model. In a nutshell, our goal can be phrased as follows:

Goal: Find an inductive invariant I, i.e. an I with $\Phi(I) \leq I$, s.t. $I(s_0) \leq \lambda$.

2.3 Our CEGIS Framework for Synthesizing Inductive Invariants

While *finding* a safe inductive invariant I is challenging, *checking* whether a given candidate I is indeed inductive is easier: it is decidable for certain infinite-state programs (cf. [12, Sect. 7.2]), it may not require an explicit exploration of the whole state space, and it can be done efficiently for piecewise linear I. Hence, techniques that generate decent candidate expressions fast and then check their inductivity could enable the automatic verification of probabilistic programs with gigantic and even infinite state spaces.

In this paper, we test this hypothesis by developing the CEGIS framework depicted in Fig. 1 for incrementally synthesizing inductive invariants. A template generator generates parametrized templates for inductive invariants. The inner

[3] For an exposition of why it makes sense to speak of *invariants* even in a quantitative setting, [41, Sect. 5.1] relates quantitative invariants to invariants in Hoare logic.

loop (shaded box in Fig. 1) then tries to solve for appropriate template-parameter instantiations. If it succeeds, an inductive invariant has been synthesized. Otherwise, the template provably cannot be instantiated into an inductive invariant. The inner loop then reports that back to the template generator (possibly with some hint on why it failed, see Appx. D) and asks for a refined template.

For our running example, we start with the template

$$T \;=\; [\mathit{fail} < 10 \wedge \mathit{sent} < 8\,000\,000] \cdot (\alpha \cdot \mathit{sent} + \beta \cdot \mathit{fail} + \gamma) \;+\; [\mathit{fail} = 10]\,, \quad (1)$$

where we use *Iverson brackets* for indicators, i.e. $[\varphi]\,(s) = 1$ if $s \models \varphi$ and 0 otherwise. T contains two kinds of variables: integer program variables $\mathit{fail}, \mathit{sent}$ and real-valued parameters α, β, γ. While the template is nonlinear, substituting α, β, γ with concrete values yields piecewise linear candidate invariants I. We ensure that those I are piecewise linear to render the repeated inductivity checks efficient. We construct only so-called *natural* templates T with Φ in mind, e.g. we want to construct only I such that $I(s) = 1$ when $s(\mathit{fail}) = 10$.

Our inner CEGIS loop checks whether there exists an assignment from these template variables to concrete values such that the resulting piecewise linear expression is an inductive invariant. Concretely, we try to determine whether there exist values for α, β, γ such that $T(\alpha, \beta, \gamma)$ is inductive. For that, we first guess values for α, β, γ, say all 0's, and ask a verifier whether the instantiated (and now piecewise linear) template $I = T(0, 0, 0)$ is indeed inductive. In our example, the verifier determines that I is *not* inductive: a counterexample is $s(\mathit{fail}) = 9$, $s(\mathit{sent}) = 7999999$. Intuitively, the probability to reach the target after one more loop iteration exceeds the value in I for this state, that is, $\Phi(I)(s) = 0.001 > 0 = I(s)$. From this counterexample, our synthesizer learns

$$\Phi(T)(s) \;=\; 0.001 \;\overset{!}{\leq}\; \alpha \cdot 7999999 + \beta \cdot 9 + \gamma \;=\; T(s)\ .$$

Observe that this learned lemma is linear in α, β, γ. The synthesizer will now keep "guessing" assignments to the parameters which are consistent with the learned lemmas until either no such parameter assignment exists anymore, or until it produces an *inductive* invariant $I = T(\ldots)$. In our running example, assuming $\lambda = 0.9$, after 6 lemmas, our synthesizer finds the inductive invariant I

$$\left[\mathit{fail} < 10 \wedge \mathit{sent} < 8 \cdot 10^6\right] \cdot \left(-\tfrac{9}{8 \cdot 10^7} \cdot \mathit{sent} + \tfrac{79\,991}{72 \cdot 10^7} \cdot \mathit{fail} + \tfrac{9}{10}\right) + [\mathit{fail} = 10] \quad (2)$$

where indeed $I(s_0) \leq \lambda$ holds. For a tighter threshold λ, such simple templates do not suffice. For example, it is impossible to instantiate this template to an inductive invariant for $\lambda = 0.8$, even though 0.8 is an upper bound on the actual reachability probability. We therefore support *more general templates* of the form

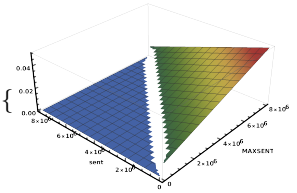
$$T \;=\; \sum_i [B_i] \cdot (\alpha_i \cdot \mathit{sent} + \beta_i \cdot \mathit{fail} + \gamma_i) \;+\; [\mathit{fail} = 10]\ ,$$

where the B_i are (restricted) predicates over program and template variables which partition the state space. In particular, we allow for a template such as

$$\begin{aligned} T \;=\;& [\mathit{fail} < 10 \wedge \mathit{sent} < \delta] \cdot (\alpha_1 \cdot \mathit{sent} + \beta_1 \cdot \mathit{fail} + \gamma_1) \;+ \\ & [\mathit{fail} < 10 \wedge \mathit{sent} \geq \delta] \cdot (\alpha_2 \cdot \mathit{sent} + \beta_2 \cdot \mathit{fail} + \gamma_2) \;+\; [\mathit{fail} = 10] \end{aligned} \quad (3)$$

$$
\begin{array}{l}
\mathit{fail} \coloneqq 0\,;\ \mathit{sent} \coloneqq 0\,; \\
\texttt{while}\,(\ \mathit{sent} < P\ \wedge\ \mathit{fail} < R\ \wedge\ P \leq 8\,000\,000\ \wedge\ R \geq 5\,)\,\{ \\
\quad \{\,\mathit{fail} \coloneqq 0\,;\ \mathit{sent} \coloneqq \mathit{sent} + 1\,\}\ [0.99]\ \{\,\mathit{fail} \coloneqq \mathit{fail} + 1\,\} \\
\}
\end{array}
$$

(a) A family of retransmission protocols



(b) Inductive invariant for $\mathit{fail} = 0$ and $R \geq 5$

Fig. 3: A bounded retransmission protocol family and piece of a matching invariant.

However, such templates are challenging for the CEGIS loop. Thus, we additionally consider templates where the B_i's range only over program variables, e.g.

$$[\mathit{fail} < 10 \wedge \mathit{sent} < 4\,000\,000] \cdot (\ldots)\ +\ [\mathit{fail} < 10 \wedge \mathit{sent} \geq 4\,000\,000] \cdot (\ldots)\ +\ \ldots$$

Our partition refinement algorithms automatically produce these templates, without the need for user interaction.

Finally, we highlight that we may use our approach for more general questions. For BRP, suppose we want to verify an upper bound $\lambda = 0.05$ on the probability of failing to transmit *all* packages for an *infinite set of models* (also called a *family*) with varying upper bounds on packets $1 \leq P \leq 8000000$ and retransmissions $R \geq 5$. This infinite set of models is described by the loop shown in Fig. 3a. Our approach fully automatically synthesizes the following inductive invariant I:

$$\left[\begin{array}{l} \mathit{fail} < R\ \wedge\ \mathit{sent} < P\ \wedge\ P < 8\,000\,000\ \wedge\ R \geq 5 \\ \quad \wedge\ R > 1 + \mathit{fail}\ \wedge\ \frac{13067990199}{5280132671650} \cdot \mathit{fail} \leq \frac{5278689867}{211205306866000} \end{array}\right] \cdot \left(\begin{array}{l} \frac{-19}{3820000040} \cdot \mathit{sent} \\ +\ \frac{19}{3820000040} \cdot P \\ +\ \frac{19500001}{1910000020} \end{array}\right)$$
$$+\ \ldots \text{ (7 additional summands omitted)}$$

The first summand of I is plotted in Fig. 3b. Since I overapproximates the probability of failing to transmit all packages for every state, I may be used to infer additional information about the reachability probabilities.

3 Formal Problem Statement

Before we state the precise invariant synthesis problem that we aim to solve, we summarize the essential concepts underlying our formalization.

Probabilistic Loops. We consider *single probabilistic loops* $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$ whose *loop guard* φ and (loop-free) *body* C adhere to the grammar

$$
\begin{array}{rcl}
C & \longrightarrow & \texttt{skip}\ \mid\ x \coloneqq e\ \mid\ C\,;\,C\ \mid\ \{\,C\,\}\,[p]\,\{\,C\,\}\ \mid\ \texttt{if}\,(\,\varphi\,)\,\{\,C\,\}\,\texttt{else}\,\{\,C\,\} \\
\varphi & \longrightarrow & e < e\ \mid\ \neg\varphi\ \mid\ \varphi \wedge \varphi \qquad e\ \longrightarrow\ q\ \mid\ x\ \mid\ q \cdot e\ \mid\ e + e\,,
\end{array}
$$

where $q \in \mathbb{Q}$ is a constant and x is from an arbitrary *finite* set Vars of $\mathbb{N}$-valued program variables. Program states in $S = \{\,s \mid s\colon \mathsf{Vars} \to \mathbb{N}\,\}$ map variables to

natural numbers.[4] All statements are standard (cf. [46]). $\{\,C_1\,\}\ [p]\ \{\,C_2\,\}$ is a probabilistic choice which executes C_1 with probability $p \in [0,1] \cap \mathbb{Q}$ and C_2 with probability $1-p$. Fig. 2 (ll. 2–3) is an example of a probabilistic loop.

Expectations. In Sect. 2, we considered whether final states meet some target condition by assigning 0 or 1 to each final state. The assignment can be generalized to more general quantities in $\mathbb{R}^{\infty}_{\geq 0}$. We call such assignments f *expectations* [46] (think: random variable) and collect them in the set $\mathbb{E}$, i.e.

$$\mathbb{E} \;=\; \left\{\, f \;\middle|\; f\colon S \to \mathbb{R}^{\infty}_{\geq 0} \,\right\}\,, \qquad \text{where} \qquad f \preceq g \quad \text{iff} \quad \forall\, s \in S\colon f(s) \leq g(s)\,.$$

$\preceq$ is a partial order on $\mathbb{E}$ – necessary to sensibly speak about least fixed points.

Characteristic Functions. The expected behavior of a probabilistic loop for an expectation f is captured by an expectation transformer (namely the $\Phi\colon \mathbb{E} \to \mathbb{E}$ of Sect. 2), called the loop's *characteristic function*. To focus on invariant synthesis, we abstract from the details[5] of constructing characteristic functions from probabilistic loops; our framework only requires the following key property:

Proposition 1 (Characteristic Functions). *For every loop* $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$ *and expectation* f*, there exists a monotone function* $\Phi_f\colon \mathbb{E} \to \mathbb{E}$ *such that*

$$\Phi_f(I)(s) \;=\; \begin{cases} f(s), & \text{if } s \not\models \varphi\,, \\ \text{“expected value of } I \text{ after executing } C \textit{ once} \text{ on } s\text{”}, & \text{if } s \models \varphi\,, \end{cases}$$

and the least fixed point of Φ_f*, denoted* $\mathsf{lfp}\ \Phi_f$*, satisfies*

$$\big(\mathsf{lfp}\ \Phi_f\big)(s) \;=\; \text{“expected value of } f \text{ after executing } \texttt{while}\,(\,\varphi\,)\,\{\,C\,\} \text{ on } s\text{”}\,.$$

Example 1. In our running example from Sect. 2.1, we chose as f the expression $[\mathit{fail} = 10]$, which evaluates to 1 in every state s where $\mathit{fail} = 10$ and to 0 otherwise. The characteristic function $\Phi_f(I)$ of the loop in Fig. 2 is
$[\neg\varphi] \cdot [\mathit{fail}{=}10] \;+\; [\varphi] \cdot \big(0.999 \cdot I\,[\mathit{sent}/\mathit{sent}{+}1]\,[\mathit{fail}/0] + 0.001 \cdot I\,[\mathit{fail}/\mathit{fail}{+}1]\big)$,
where $\varphi = \mathit{sent} < 8\,000\,000 \wedge \mathit{fail} < 10$ is the loop guard and $I\,[x/e]$ denotes the (syntactic) substitution of variable x by expression e in expectation I – the latter is used to model the effect of assignments as in standard Hoare logic. ◁

Inductive Invariants. For a probabilistic loop $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$, and *pre-* and *post*expectations $g, f \in \mathbb{E}$, we aim to verify $\mathsf{lfp}\ \Phi_f \preceq g$, i.e. that the expected value of f after termination of the loop is bounded from above by g. We discuss how to adapt our approach to expected runtimes and lower bounds in Sect. 6. Intuitively,

[4] Considering only unsigned integers does not decrease expressive power but simplifies the technical presentation (cf. [14, Sect. 11.2] for a detailed discussion). We statically ensure that for every assignment $x \coloneqq e$, e always evaluates to some value in $\mathbb{N}$.

[5] We can (and our tool does) derive a symbolic representation of a loop's characteristic function from the program structure using a weakest-precondition-style calculus (cf. [46]); see Appx. A for details. If f maps only to 0 or 1, Φ_f corresponds to the least fixed point characterization of reachability probabilities [7, Thm. 10.15].

f assigns a quantity to all *target* states reached upon termination. g assigns to all *initial states* a desired bound on the expected value of f after termination of the loop. By choosing $g(s) = \infty$ for certain s, we can make s so-to-speak "irrelevant". An $I \in \mathbb{E}$ is an *inductive invariant* proving $\mathsf{lfp}\ \Phi_f \preceq g$ iff $\Phi_f(I) \preceq I$ and $I \preceq g$. Continuing our example, Eq. (2) on p. 5 shows an inductive invariant proving that $\mathsf{lfp}\ \Phi_f \preceq g \coloneqq [\mathit{fail} = 0 \land \mathit{sent} = 0] \cdot 0.9 + [\neg(\mathit{fail} = 0 \land \mathit{sent} = 0)] \cdot \infty$.

Our framework employs syntactic fragments of expectations on which the check $\Phi_f(I) \preceq I$ can be done symbolically by an SMT solver. As illustrated in Fig. 1, we use *templates* to further narrow down the invariant search space.

Templates. Let $\mathsf{TVars} = \{\alpha, \beta, \ldots\}$ be a countably infinite set of $\mathbb{Q}$-valued *template variables*. A *template valuation* is a function $\mathfrak{I} \colon \mathsf{TVars} \to \mathbb{Q}$ that assigns to each template variable a rational number. We will use the same expressions as in our programs except that we admit both rationals and template variables as coefficients. Formally, arithmetic and Boolean expressions E and B adhere to

$$E \quad \longrightarrow \quad r \mid x \mid r \cdot x \mid E + E \qquad\qquad B \quad \longrightarrow \quad E < E \mid \neg B \mid B \land B\ ,$$

where $x \in \mathsf{Vars}$ and $r \in \mathbb{Q} \cup \mathsf{TVars}$. The set TExp of templates then consists of all

$$T \ = \ [B_1] \cdot E_1 + \ldots + [B_n] \cdot E_n\ ,$$

for $n \geq 1$, where *the Boolean expressions B_i partition the state space*, i.e. for all template valuations $\mathfrak{I}$ and all states s, there is *exactly one* B_i such that $\mathfrak{I}, s \models B_i$. T is a *fixed-partition template* if additionally no B_i contains a template variable.

Notice that templates are generally *not* linear (over $\mathsf{Vars} \cup \mathsf{TVars}$). Sect. 2 gives several examples of templates, e.g. Eq. (1).

Template Instances. We denote by $T\,[\mathfrak{I}]$ the *instance* of template T under $\mathfrak{I}$, i.e. the expression obtained from substituting every template variable α in T by its valuation $\mathfrak{I}(\alpha)$. For example, the expression in Eq. (2) on p. 5 is an instance of the template in Eq. (1) on p. 5. The set of all instances of template T is defined as $\langle T \rangle = \{\, T\,[\mathfrak{I}] \mid \mathfrak{I} \colon \mathsf{TVars} \to \mathbb{Q} \,\}$. We chose the shape of templates on purpose: To evaluate an instance $T\,[\mathfrak{I}]$ of a template T in a state s, it suffices to find the *unique* Boolean expression B_i with $\mathfrak{I}, s \models B_i$ and then evaluate the *single* linear arithmetic expression $E_i\,[\mathfrak{I}]$ in s. For fixed-partition templates, the selection of the right B_i does not even depend on the template evaluation $\mathfrak{I}$.

Piecewise Linear Expectations. Some template instances $T\,[\mathfrak{I}]$ do *not* represent expectations, i.e. they are not of type $S \to \mathbb{R}^{\infty}_{\geq 0}$, as they may evaluate to *negative numbers*. Template instances $T\,[\mathfrak{I}]$ that *do* represent expectations are *piecewise linear*; we collect such *well-defined* instances in the set LinExp. Formally,

Definition 1 (LinExp). *The set* LinExp *of* (piecewise) linear expectations *is* $\mathsf{LinExp} \ = \ \{T\,[\mathfrak{I}] \mid T \in \mathsf{TExp} \ \text{ and } \ \mathfrak{I} \colon \mathsf{TVars} \to \mathbb{Q} \ \text{ and } \ \forall s \in S \colon T\,[\mathfrak{I}]\,(s) \geq 0\}$.

We identify well-defined instances of templates in LinExp with the expectation in $\mathbb{E}$ that they represent, e.g. when writing the inductivity check $\Phi_f(T\,[\mathfrak{I}]) \stackrel{?}{\preceq} (T\,[\mathfrak{I}])$.

Natural Templates. As suggested in Sect. 2.3, it makes sense to focus only on so-called *natural* templates. Those are templates that even have a chance of

becoming inductive, as they take the loop guard φ and postexpectation f into account. Formally, a template T is *natural* (wrt. to φ and f) if T is of the form

$$T \;=\; \underbrace{[\neg\varphi \wedge B_1]\cdot E_1 + \ldots + [\neg\varphi \wedge B_n]\cdot E_n}_{\text{must be equivalent to } [\neg\varphi]\cdot f} \;+\; [B'_1]\cdot E'_1 + \ldots + [B'_m]\cdot E'_m \;.$$

We collect all natural templates in the set TnExp.

Formal Problem Statement. Throughout this paper, we fix an ambient single loop $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$, a postexpectation $f \in \mathsf{LinExp}$, and a preexpectation $g \in \mathsf{LinExp}$[6] such that $\mathsf{lfp}\;\Phi_f(I) \preceq g$[7]. The set AdmInv of *admissible invariants* (i.e. those expectations that are both *inductive* and *safe*) is then given by

$$\mathsf{AdmInv} \;=\; \{\; \underbrace{I \;\in\; \mathsf{LinExp}}_{\text{well-definedness: } I \succeq 0} \;\mid\; \underbrace{\Phi_f(I) \;\preceq\; I}_{\text{inductivity}} \quad\text{and}\quad \underbrace{I \;\preceq\; g}_{\text{safety}} \;\}\;,$$

where the underbraces summarize the tasks for a verifier to decide whether a template instance I is an admissible inductive invariant. We require $\mathsf{lfp}\;\Phi_f \preceq g$, so that AdmInv is not vacuously empty due to an unsafe bound g.

> **Formal problem statement**: Given a natural template T, find an instantiation $I \in \langle T\rangle \cap \mathsf{AdmInv}$ or determine that there is no such I.

Notice that AdmInv might be empty, even for safe g's, because generally one might need more complex invariants than piecewise linear ones [14]. However, there always exists an inductive invariant in LinExp if a loop can reach only finitely many states.[8] We call a loop $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$ *finite-state*, if only finitely many states satisfy the loop guard φ, i.e. if $S_\varphi = \{\, s \in S \mid s \models \varphi \,\}$ is finite.

Syntactic Characteristic Functions. We work with $I, f \in \mathsf{LinExp}$, i.e. linear expectations, so that we can perform inductivity checks $\Phi_f(I) \preceq I$ symbolically (via SMT) without constructing the state space first. In particular, we can construct *syntactic counterpart* Ψ_f to Φ_f that operates on templates. Intuitively, we can either evaluate Ψ_f on a (syntactic) template T and then instantiate the resulting template with some valuation $\mathfrak{I}$, or we can evaluate Φ_f on the (semantic) expectation $T\,[\mathfrak{I}]$ that emerges from instantiating template T with $\mathfrak{I}$ – the results will coincide if the instance $T\,[\mathfrak{I}]$ is well-defined. Formally:

Proposition 2. *Given* $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$ *and* $f \in \mathsf{LinExp}$*, one can effectively compute a mapping* $\Psi_f \colon \mathsf{TExp} \to \mathsf{TExp}$*, such that for all* T *and* $\mathfrak{I}$

$$T\,[\mathfrak{I}] \;\in\; \mathsf{LinExp} \quad\text{implies}\quad \Psi_f(T)\,[\mathfrak{I}] \;=\; \Phi_f\big(T\,[\mathfrak{I}]\big)\;.$$

Moreover, Ψ_f *maps fixed-partition templates to fixed-partition templates.*

In Ex. 1, we have already constructed such a Ψ_f to represent Φ_f. The general construction is inspired by [12], but treats template variables as constants.

[6] To enable declaring certain states as irrelevant, we additionally allow $E_i = \infty$ in the linear preexpectation $g = [B_1]\cdot E_1 + \ldots + [B_n]\cdot E_n$.

[7] We discuss in Sect. 6 how to reason about lower bounds $g \preceq \mathsf{lfp}\;\Phi_f(I)$.

[8] Bluntly just choose as many pieces as there are states.

4 One-Shot Solver

One could address the template instantiation problem from Sect. 3 in one shot: encode it as an SMT query, ask a solver for a model, and infer from the model an admissible invariant. While this approach is infeasible in practice (as it involves quantification over S_φ), it inspires the CEGIS loop in Fig. 1.

Regarding the encoding, given a template T, we need a formula over TVars that is satisfiable if and only if there exists a template valuation $\mathfrak{I}$ such that $T[\mathfrak{I}]$ is an admissible invariant, i.e. $T[\mathfrak{I}] \in \mathsf{AdmInv}$. To get rid of program variables in templates, we denote by $T(s)$ the expression over TVars in which all *program* variables $x \in \mathsf{Vars}$ have been substituted by $s(x)$.

Intuitively, we then encode that, for every state s, the expression $T(s)$ satisfies the three conditions of admissible invariants, i.e. well-definedness, inductivity, and safety. In particular, we use Prop. 2 to compute a template $\Psi_f(T)$ that represents the application of the characteristic function Φ_f to a candidate invariant, i.e. $\Phi_f(T[\mathfrak{I}])$ – a necessity for encoding inductivity.

Formally, we denote by $\mathsf{Sat}(\phi)$ the set of all models of a first-order formula ϕ (with a fixed underlying structure), i.e. $\mathsf{Sat}(\phi) = \{\mathfrak{I} \mid \mathfrak{I} \models \phi\}$. Then:

Theorem 1. *For every natural template $T \in \mathsf{TnExp}$ and $f, g \in \mathsf{LinExp}$, we have*

$$\begin{aligned} & \langle T \rangle \cap \mathsf{AdmInv} \neq \emptyset \\ \text{iff} \quad & \mathsf{Sat}\big(\, \forall s \in S_\varphi \colon \underbrace{0 \leq T(s)}_{\textit{well-definedness}} \;\wedge\; \underbrace{\Psi_f(T)(s) \leq T(s)}_{\textit{inductivity}} \;\wedge\; \underbrace{T(s) \leq g(s)}_{\textit{safety}} \,\big) \;\neq\; \emptyset \;. \end{aligned}$$

Notice that, for fixed-partition templates, the above encoding is particularly simple: $T(s)$ and $\Psi_f(T)(s)$ are equivalent to single linear arithmetic expressions over TVars; $g(s)$ is either a single expression or ∞ – in the latter case, we get an equisatisfiable formula by dropping the always-satisfied constraint $T(s) \leq g(s)$.

For general templates, one can exploit the partitioning to break it down into multiple inequalities, i.e. every inequality becomes a conjunction over implications of linear inequalities over the template variables TVars.

Example 2. Reconsider template T in Eq. (3) on p. 5 and assume a state s with $s(\mathit{fail}) = 5$ and $s(\mathit{sent}) = 2$. Then, we encode the well-definedness, $T(s) \geq 0$, as

$$\big(5 < 10 \wedge 2 < \delta \Rightarrow \alpha_1 \cdot 2 + \beta_1 \cdot 5 + \gamma_1 \geq 0\big) \wedge \big(5 < 10 \wedge 2 \geq \delta \Rightarrow \alpha_2 \cdot 2 + \beta_2 \cdot 5 + \gamma_2 \geq 0\big)$$

where the trivially satisfiable conjunct $5 = 10 \Rightarrow \mathsf{true}$ encoding the last summand, i.e. $[\mathit{fail} = 10]$, has been dropped. ◁

Since using an SMT solver for the query in Thm. 1 involves mixed real and integer arithmetic with quantifiers, the one-shot approach is, unsurprisingly, incomplete in general. However, for finite-state loops and natural templates, our instantiation problem becomes decidable: one can replace the universal quantifier $\forall s$ by a finite conjunction $\bigwedge_{s \in S_\varphi}$ to obtain a (decidable) `QF_LRA` formula.

Theorem 2. *The problem $\langle T \rangle \cap \mathsf{AdmInv} \stackrel{?}{\neq} \emptyset$ is decidable for finite-state loops and $T \in \mathsf{TnExp}$. If T is fixed-partition, it is decidable via linear programming.*

5 Constructing an Efficient CEGIS Loop

We now present a CEGIS loop (see inner loop of Fig. 1) in which a *synthesizer* and a *verifier* attempt to incrementally solve our problem statement (cf. p. 9).

5.1 The Verifier

We assume a verifier for checking $I \stackrel{?}{\in} \mathsf{AdmInv}$. For CEGIS, it is important to get some feedback whenever $I \notin \mathsf{AdmInv}$. To this end, we define:

Definition 2. *For a state $s \in S$, the set* $\mathsf{AdmInv}(s)$ *of* s-admissible invariants *is*

$$\mathsf{AdmInv}(s) \;=\; \{\, I \mid \underbrace{I(s) \geq 0}_{s\text{-well-defined}} \quad \text{and} \quad \underbrace{\Phi_f(I)(s) \leq I(s)}_{s\text{-inductive}} \quad \text{and} \quad \underbrace{I(s) \leq g(s)}_{s\text{-safe}} \,\} \;.$$

For a subset $S' \subseteq S$ of states, we define $\mathsf{AdmInv}(S') = \bigcap_{s \in S'} \mathsf{AdmInv}(s)$.

Clearly, if $I \notin \mathsf{AdmInv}$, then $I \notin \mathsf{AdmInv}(s)$ for some $s \in S$, i.e. state s is a *counterexample* to well-definedness, inductivity, or safety of I. We denote the set of all such counterexamples (to the claim $I \in \mathsf{AdmInv}$) by $\mathsf{CounterEx}_I$. We assume an effective (baseline) verifier for detecting counterexamples:

Definition 3. *A verifier is* any *function* $\mathsf{Verify} \colon \mathsf{LinExp} \to \{\mathsf{true}\} \cup S$ *such that*

1. $\mathsf{Verify}(I) = \mathsf{true}$ *if and only if* $I \in \mathsf{AdmInv}$, *and*
2. $\mathsf{Verify}(I) = s$ *implies* $s \in \mathsf{CounterEx}_I$.

Proposition 3 ([12]). *There exist effective verifiers.*

For example, one can implement an SMT-backed verifier using an encoding analogous to Thm. 1, where every model is a counterexample $s \in \mathsf{CounterEx}_I$:

$$I \notin \mathsf{AdmInv} \quad \text{iff} \quad \underbrace{\mathsf{Sat}\Big(\neg\big(0 \leq I \;\wedge\; \Phi_f(I) \leq I \;\wedge\; I \leq g\big)\Big) \;\neq\; \emptyset}_{\exists s \in S\colon\; I \notin \mathsf{AdmInv}(s)} \;.$$

5.2 The Counterexample-Guided Inductive Synthesizer

A synthesizer must generate from a given template T instances $I \in \langle T \rangle$ which can be passed to a verifier for checking admissibility. To make an informed guess, our synthesizers can take a finite set of witnesses $S' \subseteq S$ into account:

Definition 4. *Let* $\mathsf{FinStates}$ *be the set of finite sets of states. A* synthesizer *for template $T \in \mathsf{TnExp}$ is* any *function* $\mathsf{Synt}_T \colon \mathsf{FinStates} \to \langle T \rangle \cup \{\mathsf{false}\}$ *such that*

1. *if* $\mathsf{Synt}_T(S') = I$, *then* $I \in \langle T \rangle \cap \mathsf{AdmInv}(S')$, *and*
2. $\mathsf{Synt}_T(S') = \mathsf{false}$ *if and only if* $\langle T \rangle \cap \mathsf{AdmInv}(S') = \emptyset$.

Algorithm 1: Template-Instance Synthesizer for template T

```
S' ← ∅ ;
while Synt_T(S') ≠ false do
    I ← Synt_T(S') ;
    result ← Verify(I) ;
    if result = true then
        return I ;                    /* Verifier returns true, we have I ∈ AdmInv */
    S' ← S' ∪ {result} ;              /* result is a counterexample */
return false ;                        /* ⟨T⟩ ∩ AdmInv = ∅ */
```

To build a synthesizer $\mathsf{Synt}_T(S')$ for finite sets of states $S' \subseteq S$, we proceed analogously to one-shot solving for finite-state loops (Thm. 2), i.e. we exploit

$$T\,[\mathfrak{I}] \in \mathsf{AdmInv}(S') \;\text{ iff }\; \mathfrak{I} \models \bigwedge_{s \in S'} \underbrace{0 \leq T(s) \wedge \Psi_f(T)(s) \leq T(s) \wedge T(s) \leq g(s)}_{T\,[\mathfrak{I}] \in \mathsf{AdmInv}(s)} \,.$$

That is, our synthesizer may return any model $\mathfrak{I}$ of the above constraint system; it can be implemented as one SMT query. In particular, one can efficiently find such an $\mathfrak{I}$ for fixed-partition templates via linear programming.

Theorem 3 (Synthesizer Completeness). *For finite-state loops and natural templates $T \in \mathsf{TnExp}$, we have $\mathsf{Synt}_T(S_\varphi) \in \mathsf{AdmInv}$ or $\langle T \rangle \cap \mathsf{AdmInv} = \emptyset$.*

Using the synthesizer and verifier in concert is then intuitive as in Alg. 1. We incrementally ask our synthesizer to provide a candidate invariant I that is s-admissible for all states $s \in S'$. Unless the synthesizer returns false, we ask the verifier whether I is admissible. If yes, we return I; otherwise, we get a counterexample s and add it to S' before synthesizing the next candidate.

Remark 1. The verifier from Def. 3 may return any counterexample. In two benchmarks, we observed that without further restrictions, our verifier started enumerating these counterexamples. In Appx. C, we therefore discuss additional constraints that make these verifiers act more cooperatively. ◁

6 Generalization to Termination and Lower Bounds

We extend our approach to (i) proving *universal positive almost-sure termination* (UPAST) – termination in finite expected runtime on all inputs, see [41, Sect. 6] – by synthesizing piecewise linear upper bounds on expected runtimes, and to (ii) verifying *lower bounds* on possibly unbounded expected values.

UPAST. We leverage Kaminski et al.'s weakest-precondition-style calculus for reasoning about expected runtimes [43,44]:

Proposition 4. *For every loop* $\texttt{while}\,(\varphi)\,\{C\}$, *the monotone function*

$$\Theta\colon \quad \mathbb{E} \to \mathbb{E}, \qquad \Theta(I)(s) \;=\; 1 + \Phi_0(I)(s)\;,$$

obtained from Φ_0 *(cf.* Prop. 1*) satisfies*

$$\big(\mathsf{lfp}\ \Theta\big)(s) \;=\; \begin{array}{l}\text{“expected number of loop guard evaluations}\\ \qquad\text{when executing } \texttt{while}\,(\,\varphi\,)\,\{\,C\,\} \text{ on } s\text{”}\ .\end{array}$$

All properties of Φ_0 relevant to our approach carry over to Θ, thus enabling the synthesis of inductive invariants $I \in \mathsf{LinExp}$ satisfying $0 \preceq I$ and $\Theta(I) \preceq I$. Such I *upper-bound the expected number of loop iterations* [43] and, since expectations in LinExp never evaluate to infinity, I witnesses UPAST of the `while`-loop.

Lower Bounds. Consider the problem of verifying a lower bound $g \preceq \mathsf{lfp}\ \Phi_f$ for some loop $C' = \texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$. It is straightforward to modify our CEGIS approach for synthesizing *sub-invariants*, i.e. $I \in \mathsf{LinExp}$ with $I \preceq \Phi_f(I)$. However, Hark et al. [35] showed that sub-invariants *do not necessarily lower-bound* $\mathsf{lfp}\ \Phi_f$; they hence proposed a more involved yet sound induction rule for lower bounds:

Theorem 4 (Adapted from Hark et al. [35]). *Let* T *be a natural template and* $I \in \langle T \rangle$*. If* $0 \preceq I$*,* $I \preceq \Phi_f(I)$*, and* C' *is UPAST, then*

$$\underbrace{\exists\, c \in \mathbb{R}_{\geq 0}\ \forall\, s \in S_\varphi\colon \quad \Phi_f\big(|I - I(s)|\big)(s) \;\leq\; c}_{I \text{ is conditionally difference bounded (c.d.b.)}} \qquad \text{implies} \qquad I \;\preceq\; \mathsf{lfp}\ \Phi_f\ .$$

Akin to Prop. 2, given $T \in \mathsf{TnExp}$, we can *compute* $T' \in \mathsf{TnExp}$ s.t. for all $\mathfrak{I}$,

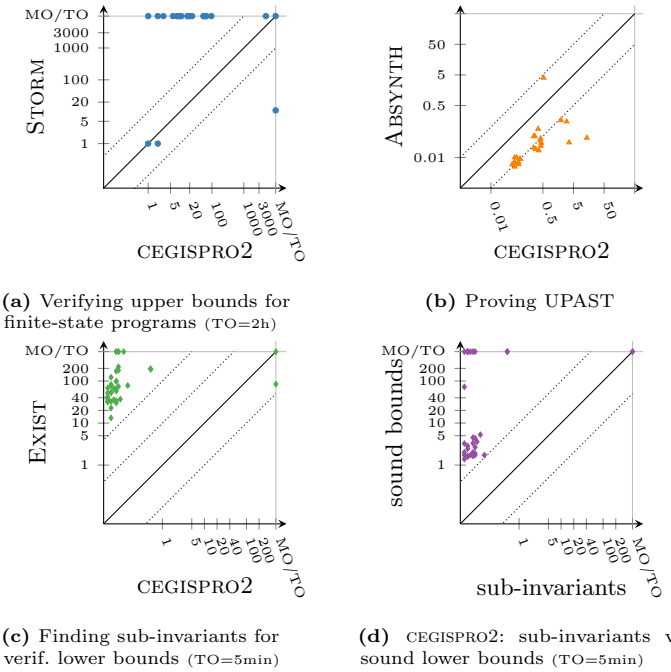
$$T\,[\mathfrak{I}] \;\in\; \mathsf{LinExp} \quad \text{implies} \quad T'\,[\mathfrak{I}] \;=\; \lambda s\mathbf{.}\ \Phi_f\big(|T\,[\mathfrak{I}] - T\,[\mathfrak{I}]\,(s)|\big)(s)\ ,$$

which facilitates the extension of our verifier and synthesizer (see Sect. 5) for encoding and checking conditional difference boundedness. Hence, we can employ our CEGIS framework for verifying $g \preceq \mathsf{lfp}\ \Phi_f$ by (i) proving UPAST of C' as demonstrated above and (ii) synthesizing a c.d.b. sub-invariant I with $g \preceq I$.

7 Empirical Evaluation

We have implemented a prototype of our techniques called CEGISPRO2: CEGIS for PRObabilistic PROgrams. The tool is written in Python using pySMT [32] with Z3 [48] as the backend for SMT solving. CEGISPRO2 proves upper- or lower bounds on expected outcomes of a probabilistic program by synthesizing quantitative inductive invariants. We investigate the applicability and scalability of our approach with a focus on the expressiveness of piecewise linear invariants. Moreover, we compare with three state-of-the-art tools – STORM [38], ABSYNTH [49], and EXIST [9] – on subsets of their benchmarks fitting into our framework.

Template Refinement. We start with a fixed-partition template T_1 constructed automatically from the syntactic structure of the given loop (i.e. the loop guard and branches in the loop body, see e.g. Eq. (1)). If we learn that T_1 admits no admissible invariant, we generate a refined template T_2, and so on, until we find a template T_i with $\langle T_i \rangle \cap \mathsf{AdmInv} \neq \emptyset$ or realize that no further refinement is possible. We implemented three strategies for template refinement (including one producing non-fixed-partition templates); see Appx. D for details.



(a) Verifying upper bounds for finite-state programs (TO=2h)

(b) Proving UPAST

(c) Finding sub-invariants for verif. lower bounds (TO=5min)

(d) CEGISPRO2: sub-invariants vs. sound lower bounds (TO=5min)

Fig. 4: Performance of CEGISPRO2 vs. state-of-the-art tools on three verification tasks (time in seconds, log-scaled; MO=8GB). Markers above the solid line depict benchmarks where CEGISPRO2 is faster (in different orders of magnitude marked by the dashed lines).

Finite-State Programs. Fig. 4a depicts experiments on verifying upper bounds on expected values of finite-state programs. For each benchmark, i.e. program and property with increasingly sharper bounds, we evaluate CEGISPRO2 on all template-refinement strategies (cf. Appx. D). We compare explicit- and symbolic-state engines of the probabilistic model checker STORM 1.6.3 [38] with exact arithmetic. STORM implements LP-based model checking (as in Sect. 4) but employs more efficient methods in its default configuration. Fig. 4a depicts the runtime of the best configuration. See detailed configurations in Appx. E.1 (Tab. 2).

Results. (i) Our CEGIS approach synthesizes inductive invariants for a variety of programs. We mostly find *syntactically small* invariants with a *small number of counterexamples* compared to the state-space size (cf. Tab. 2, except gridsmall). This indicates that piecewise linear inductive invariants can be sufficiently expressive for the verification of finite-state programs. The overall performance of CEGISPRO2 depends highly on the sharpness of the given thresholds. (ii) Our approach can outperform state-of-the-art *explicit- and symbolic-state* model checking techniques and can scale to huge state spaces. There are also simple programs where our method fails to find an inductive invariant (gridbig) or finds inductive invariants only for rather simple properties while requiring many counterexamples (gridsmall). Whether we need more sophisticated template refinements or whether these programs are not amenable to piecewise linear expectations is left for future

work. (iii) There is no clear winner between the two fixed-partition template-refinement strategies (cf. Tab. 2). We further observe that the non-fixed-partition refinement is not competitive as significantly more time is spent in the synthesizer to solve formulae with Boolean structures. We thus conclude that searching for good fixed-partition templates in a separate outer loop (cf. Fig. 1) pays off.

Proving UPAST. Fig. 4b depicts experiments on proving UPAST of (possibly infinite-state) programs taken from [49] (restricted to $\mathbb{N}$-valued, linear programs with flattened nested loops). We compare to the LP-based tool Absynth [49] for computing upper bounds on expected runtimes. These benchmarks do not require template refinements. Details on the comparison are given in Appx. E.2.

Results. cegispro2 can prove UPAST of various infnite-state programs from the literature using very few counterexamples. Absynth mostly outperforms cegispro2[9], which is to be expected as Absynth is tailored to the computation of expected runtimes. Remarkably, the runtime bounds synthesized by cegispro2 are often as tight as the bounds synthesized by Absynth (cf. Tab. 3).

Verifying Lower Bounds. Fig. 4c depicts experiments aiming to verify lower bounds on expected values of (possibly infinite-state) programs taken from [9]. We compare to Exist [9][10], which combines CEGIS with sampling- and ML-based techniques. However, Exist synthesizes sub-invariants only, which might be unsound for proving lower bounds (cf. Sect. 6). Thus, for a fair comparison, Fig. 4c depicts experiments where *both* Exist and cegispro2 synthesize sub-invariants only, whereas in Fig. 4d, we compare cegispro2 that finds sub-invariants only with cegispro2 that *additionally* proves UPAST and c.d.b., thus obtaining sound lower bounds as per Thm. 4. No benchmark requires template refinements.

Results. cegispro2 is capable of verifying quantitative lower bounds and outperforms Exist (on 30/32 benchmarks) for synthesizing sub-invariants. Additionally proving UPAST and c.d.b. naturally requires more time. A manual inspection reveals that, for most TO/MO cases in Fig. 4d, there is no c.d.b. sub-invariant. We are thus not aware of an inductive proof rule for soundly concluding that those synthesized sub-invariants are indeed lower bounds. One soundness check times out, since we could not prove UPAST for that benchmark.

8 Related Work

Besides the comparisons in Sect. 7, we discuss related works in invariant synthesis, probabilistic model checking, and symbolic inference. In the *qualitative* setting, ICE [31] is a template-based, counterexample-guided technique for learning invariants. More inductive synthesis approaches are surveyed in [4,27].

Quantitative Invariant Synthesis. Apart from the extensively discussed data-driven method [9] for synthesizing sub-invariants, *constraint solving-based approaches*

[9] Absynth uses floating-point arithmetic whereas cegispro2 uses exact arithmetic.

[10] Exist supports parametric probabilities, which are not supported by our tool. We have instantiated these parameters with varying probabilities to enable a comparison.

[28,24,45] aim to synthesize quantitative invariants for proving lower bounds over $\mathbb{R}$-valued program variables – arguably a simplification as it allows solvers to use (decidable) real arithmetic. In particular, [24] also obtains linear constraints from counterexamples ensuring certain validity conditions on candidate invariants. Apart from various technical differences, we identify three conceptual differences: (i) we support piecewise expectations which have been shown sufficiently expressive for reasoning about quantitative reachability properties; (ii) we focus on the integration of fast verifiers over efficiently decidable theories; and (iii) we do not need assumptions on termination or boundedness of expectations.

Various *martingale-based approaches*, such as [17,21,22,30,29,2,47], aim to synthesize quantitative invariants over $\mathbb{R}$-valued variables, see [54] for a recent survey. Most of these approaches yield invariants for proving almost-sure termination or bounding expected runtimes. *ε-decreasing supermartingales* [17,18] and *nonnegative repulsing supermartingales* [54] can upper-bound arbitrary reachability probabilities. In contrast, we synthesize invariants for proving upper- or lower bounds for more general quantities, i.e. expectations. [10] can prove bounds on expected values via symbolic reasoning and *Doob's decomposition*, which, however, requires user-supplied invariants and hints. [1] employs a CEGIS loop to train a neural network dedicated to learning a ranking supermartingale witnessing UPAST of (possibly continuous) probabilistic programs. Similar to our validity checks, they also use SMT solvers to check the supermartingale condition and use the provided counterexamples to guide the learning process.

The *recurrence solving-based approach* in [11] synthesizes nonlinear invariants encoding (higher-order) moments of program variables. However, the underlying algebraic techniques are confined to the sub-class of *prob-solvable loops*.

Probabilistic Model Checking. Symbolic probabilistic model checking focusses mostly on algebraic decision diagrams [6,3], representing the transition relation symbolically and using equation solving or value iteration [8,36,52] on that representation. PrIC3 [13] finds quantitative invariants by iteratively overapproximating k-step reachability. Alternative CEGIS approaches synthesize Markov chains [16] and probabilistic programs [5] that satisfy reachability properties.

Symbolic Inference. Probabilistic inference – in the finite-horizon case – employs weighted model counting via either decision diagrams annotated with probabilities as in Dice [40,39] or approximate versions by SAT/SMT-solvers [19,20,25,53,15]. PSI [33] determines symbolic representations of exact distributions. Prodigy [23] decides whether a probabilistic loop agrees with an (invariant) specification.

9 Conclusion

We have developed an inductive invariant synthesis method for proving bounds on quantitative properties of probabilistic programs on source-code level. Our prototype implementation shows promise: It can find invariants for a variety of verification problems and is competitive with state-of-the-art tools. Future work includes incorporating nondeterminism and exploring advanced template shapes.

References

1. Abate, A., Giacobbe, M., Roy, D.: Learning probabilistic termination proofs. In: CAV (2). Lecture Notes in Computer Science, vol. 12760, pp. 3–26. Springer (2021)
2. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales. PACMPL **2**(POPL), 34:1–34:32 (2018)
3. de Alfaro, L., Kwiatkowska, M.Z., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: TACAS. Lecture Notes in Computer Science, vol. 1785, pp. 395–410. Springer (2000)
4. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, vol. 40, pp. 1–25. IOS Press (2015)
5. Andriushchenko, R., Ceska, M., Junges, S., Katoen, J.: Inductive synthesis for probabilistic programs reaches new horizons. In: TACAS (1). Lecture Notes in Computer Science, vol. 12651, pp. 191–209. Springer (2021)
6. Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: ICALP. Lecture Notes in Computer Science, vol. 1256, pp. 430–440. Springer (1997)
7. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
8. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: CAV (1). Lecture Notes in Computer Science, vol. 10426, pp. 160–180. Springer (2017)
9. Bao, J., Trivedi, N., Pathak, D., Hsu, J., Roy, S.: Data-driven invariant learning for probabilistic programs. In: CAV (1). Lecture Notes in Computer Science, vol. 13371, pp. 33–54. Springer (2022)
10. Barthe, G., Espitau, T., Fioriti, L.M.F., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: CAV (1). Lecture Notes in Computer Science, vol. 9779, pp. 43–61. Springer (2016)
11. Bartocci, E., Kovács, L., Stankovic, M.: Automatic generation of moment-based invariants for prob-solvable loops. In: ATVA. Lecture Notes in Computer Science, vol. 11781, pp. 255–276. Springer (2019)
12. Batz, K., Chen, M., Kaminski, B.L., Katoen, J., Matheja, C., Schröer, P.: Latticed k-induction with an application to probabilistic programs. In: CAV (2). Lecture Notes in Computer Science, vol. 12760, pp. 524–549. Springer (2021)
13. Batz, K., Junges, S., Kaminski, B.L., Katoen, J., Matheja, C., Schröer, P.: Pric3: Property directed reachability for MDPs. In: CAV (2). Lecture Notes in Computer Science, vol. 12225, pp. 512–538. Springer (2020)
14. Batz, K., Kaminski, B.L., Katoen, J., Matheja, C.: Relatively complete verification of probabilistic programs: An expressive language for expectation-based reasoning. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021)
15. Belle, V., Passerini, A., van den Broeck, G.: Probabilistic inference in hybrid domains by weighted model integration. In: IJCAI. pp. 2770–2776. AAAI Press (2015)
16. Ceska, M., Hensel, C., Junges, S., Katoen, J.: Counterexample-guided inductive synthesis for probabilistic systems. Formal Aspects Comput. **33**(4-5), 637–667 (2021)
17. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 511–526. Springer (2013)

18. Chakarov, A., Voronin, Y., Sankaranarayanan, S.: Deductive proofs of almost sure persistence and recurrence properties. In: TACAS. Lecture Notes in Computer Science, vol. 9636, pp. 260–279. Springer (2016)
19. Chakraborty, S., Fried, D., Meel, K.S., Vardi, M.Y.: From weighted to unweighted model counting. In: IJCAI. pp. 689–695. AAAI Press (2015)
20. Chakraborty, S., Meel, K.S., Mistry, R., Vardi, M.Y.: Approximate probabilistic inference via word-level counting. In: AAAI. pp. 3218–3224. AAAI Press (2016)
21. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: CAV (1). Lecture Notes in Computer Science, vol. 9779, pp. 3–22. Springer (2016)
22. Chatterjee, K., Novotný, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: POPL. pp. 145–160. ACM (2017)
23. Chen, M., Katoen, J., Klinkenberg, L., Winkler, T.: Does a program yield the right distribution? Verifying probabilistic programs via generating functions. In: CAV (1). Lecture Notes in Computer Science, vol. 13371, pp. 79–101. Springer (2022)
24. Chen, Y., Hong, C., Wang, B., Zhang, L.: Counterexample-guided polynomial loop invariant generation by Lagrange interpolation. In: CAV (1). Lecture Notes in Computer Science, vol. 9206, pp. 658–674. Springer (2015)
25. Chistikov, D., Dimitrova, R., Majumdar, R.: Approximate counting in SMT and value estimation for probabilistic programs. Acta Informatica **54**(8), 729–764 (2017)
26. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: PAPM-PROBMIV. Lecture Notes in Computer Science, vol. 2165, pp. 39–56. Springer (2001)
27. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 251–269. Springer (2018)
28. Feng, Y., Zhang, L., Jansen, D.N., Zhan, N., Xia, B.: Finding polynomial loop invariants for probabilistic programs. In: ATVA. Lecture Notes in Computer Science, vol. 10482, pp. 400–416. Springer (2017)
29. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: Soundness, completeness, and compositionality. In: POPL. pp. 489–501. ACM (2015)
30. Fu, H., Chatterjee, K.: Termination of nondeterministic probabilistic programs. In: VMCAI. Lecture Notes in Computer Science, vol. 11388, pp. 468–490. Springer (2019)
31. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 69–87. Springer (2014)
32. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: SMT Workshop (2015)
33. Gehr, T., Misailovic, S., Vechev, M.T.: PSI: Exact symbolic inference for probabilistic programs. In: CAV (1). Lecture Notes in Computer Science, vol. 9779, pp. 62–83. Springer (2016)
34. Gretz, F., Katoen, J., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Evaluation **73**, 110–132 (2014)
35. Hark, M., Kaminski, B.L., Giesl, J., Katoen, J.: Aiming low is harder: Induction for lower bounds in probabilistic program verification. Proc. ACM Program. Lang. **4**(POPL), 37:1–37:28 (2020)
36. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: CAV (2). Lecture Notes in Computer Science, vol. 12225, pp. 488–511. Springer (2020)

37. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: TYPES. Lecture Notes in Computer Science, vol. 806, pp. 127–165. Springer (1993)
38. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022)
39. Holtzen, S., Junges, S., Vazquez-Chanlatte, M., Millstein, T.D., Seshia, S.A., van den Broeck, G.: Model checking finite-horizon Markov chains with probabilistic inference. In: CAV (2). Lecture Notes in Computer Science, vol. 12760, pp. 577–601. Springer (2021)
40. Holtzen, S., van den Broeck, G., Millstein, T.D.: Scaling exact inference for discrete probabilistic programs. Proc. ACM Program. Lang. **4**(OOPSLA), 140:1–140:31 (2020)
41. Kaminski, B.L.: Advanced Weakest Precondition Calculi for Probabilistic Programs. Ph.D. thesis, RWTH Aachen University, Germany (2019)
42. Kaminski, B.L., Katoen, J., Matheja, C.: On the hardness of analyzing probabilistic programs. Acta Inform. **56**(3), 255–285 (2019)
43. Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run-times of probabilistic programs. In: ESOP. Lecture Notes in Computer Science, vol. 9632, pp. 364–389. Springer (2016)
44. Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected runtimes of randomized algorithms. J. ACM **65**(5), 30:1–30:68 (2018)
45. Katoen, J., McIver, A., Meinicke, L., Morgan, C.: Linear-invariant generation for probabilistic programs: Automated support for proof-based methods. In: SAS. Lecture Notes in Computer Science, vol. 6337, pp. 390–406. Springer (2010)
46. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science, Springer (2005)
47. Moosbrugger, M., Bartocci, E., Katoen, J., Kovács, L.: Automated termination analysis of polynomial probabilistic programs. In: ESOP. Lecture Notes in Computer Science, vol. 12648, pp. 491–518. Springer (2021)
48. de Moura, L.M., Bjørner, N.S.: Z3: An efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
49. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: Resource analysis for probabilistic programs. In: PLDI. pp. 496–512. ACM (2018)
50. Park, D.: Fixpoint induction and proofs of program properties. Mach. Intell. **5** (1969)
51. Puterman, M.L.: Markov Decision Processes. Wiley Series in Probability and Statistics, Wiley (1994)
52. Quatmann, T., Katoen, J.: Sound value iteration. In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 643–661. Springer (2018)
53. Rabe, M.N., Wintersteiger, C.M., Kugler, H., Yordanov, B., Hamadi, Y.: Symbolic approximation of the bounded reachability probability in large Markov chains. In: QEST. Lecture Notes in Computer Science, vol. 8657, pp. 388–403. Springer (2014)
54. Takisaka, T., Oyabu, Y., Urabe, N., Hasuo, I.: Ranking and repulsing supermartingales for reachability in randomized programs. ACM Trans. Program. Lang. Syst. **43**(2), 5:1–5:46 (2021)
55. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2), 285–309 (1955)

A Symbolically Constructing Characteristic Functions

Fix a probabilistic loop $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$ and a postexpectation $f \in \mathbb{E}$. We employ a *weakest preexpectation* à la McIver and Morgan [46] to symbolically compute the expectation $\mathsf{wp}[\![C]\!]\,(f) \in \mathbb{E}$ that maps every state $s \in S$ to the expected value of postexpectation f evaluated in the final states reached after executing loop body C when started in state s. Formally, the weakest preexpectation transformer $\mathsf{wp}\colon C \to (\mathbb{E} \to \mathbb{E})$ is given by the rules in Tab. 1. The characteristic function Φ_f of a probabilistic loop $\texttt{while}\,(\,\varphi\,)\,\{\,C\,\}$ with respect to postexpectation f is

$$\Phi_f(X) \;=\; [\neg\varphi] \cdot f \;+\; [\varphi] \cdot \mathsf{wp}[\![C]\!]\,(X) \;\;.$$

The symbolic template transformer Ψ_f from Prop. 2 is defined analogously for templates instead of arbitrary expectations (notice that all transformations of wp are syntactic). Every piecewise linear expectation in LinExp (or template in TExp) can then be converted into a template using the transformation into guarded normal form in [12], where template variables are treated as constants.

$\boldsymbol{C}$	$\mathsf{wp}\;[\![\boldsymbol{C}]\!]\,(\boldsymbol{f})$
$\texttt{skip}$	f
$x \coloneqq e$	$f\,[x/e]$
$C_1\,;\,C_2$	$\mathsf{wp}[\![C_1]\!]\,\big(\mathsf{wp}[\![C_2]\!]\,(f)\big)$
$\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}$	$p \cdot \mathsf{wp}[\![C_1]\!]\,(f) + (1-p) \cdot \mathsf{wp}[\![C_2]\!]\,(f)$
$\texttt{if}\,(\,\varphi\,)\,\{\,C_1\,\}\,\texttt{else}\,\{\,C_2\,\}$	$[\varphi] \cdot \mathsf{wp}[\![C_1]\!]\,(f) + [\neg\varphi] \cdot \mathsf{wp}[\![C_2]\!]\,(f)$

Table 1: Rules defining the weakest preexpectation transformer for loop-free programs. $[\varphi]$ denotes an indicator function, i.e. $[\varphi]\,(s) = 1$ if $s \models \varphi$ and 0 otherwise. Moreover, $f\,[x/e] = \lambda s\mathbf{.}\; f(s[x \mapsto e(s)])$ denotes the expectation f in which every free occurrences of variable x are substituted by expression e.

Notice that, as long as only finitely many states are reachable, one can alternatively construct the characteristic function by first building the Markov chain underlying a probabilistic loop [34], and then deriving the well-known Bellman operator (cf. [51]). Furthermore, if f is a predicate (i.e. maps states to 0 or 1), the characteristic function corresponds to the least fixed point characterization of reachability probabilities [7, Thm. 10.15].

B Proof of Thm. 1

Claim. For every template $T \in \mathsf{TnExp}$ and $f \in \mathsf{LinExp}$, we have

$$\langle T \rangle \cap \mathsf{AdmInv} \neq \emptyset$$

$$\text{iff}\quad \mathsf{Sat}\big(\,\forall s \in S_\varphi\colon\quad \underbrace{0 \le T(s)}_{\text{well-definedness}}\ \wedge\ \underbrace{\Psi_f(T)(s) \le T(s)}_{\text{inductivity}}\ \wedge\ \underbrace{T(s) \le g(s)}_{\text{safety}}\,\big)\ \neq\ \emptyset.$$

Proof. Recall the definition of AdmInv from our problem statement. Then:

$$\begin{aligned}
&\langle T\rangle \cap \mathsf{AdmInv} \neq \emptyset\\
\text{iff}\quad &\exists I \in \langle T\rangle\colon I \in \mathsf{AdmInv}\\
\text{iff}\quad &\exists I \in \langle T\rangle\colon 0 \preceq I \wedge \Phi_f(I) \preceq I \wedge I \preceq g \qquad \text{(see formal problem statement)}\\
\text{iff}\quad &\exists I \in \langle T\rangle\colon \forall s \in S\colon 0 \le I(s) \wedge \Phi_f(I)(s) \le I(s) \wedge I(s) \le g(s)\\
\text{iff}\quad &\exists \mathfrak{I}\colon \forall s \in S\colon 0 \le T\,[\mathfrak{I}]\,(s) \wedge \Phi_f(T\,[\mathfrak{I}])(s) \le T\,[\mathfrak{I}]\,(s) \wedge T\,[\mathfrak{I}]\,(s) \le g(s)\\
\text{iff}\quad &\exists \mathfrak{I}\colon \forall s \in S_\varphi\colon 0 \le T\,[\mathfrak{I}]\,(s) \wedge \Phi_f(T\,[\mathfrak{I}])(s) \le T\,[\mathfrak{I}]\,(s) \wedge T\,[\mathfrak{I}]\,(s) \le g(s)\\
&\qquad\qquad \text{(since } T \text{ is natural and } \mathsf{lfp}\ \Phi_f \preceq g \text{ by assumption)}\\
\text{iff}\quad &\exists \mathfrak{I}\colon \forall s \in S_\varphi\colon 0 \le T\,[\mathfrak{I}]\,(s) \wedge \Psi_f(T)\,[\mathfrak{I}]\,(s) \le T\,[\mathfrak{I}]\,(s) \wedge T\,[\mathfrak{I}]\,(s) \le g(s)\\
&\qquad\qquad \text{(by Prop. 2)}\\
\text{iff}\quad &\mathsf{Sat}\,(\forall s \in S_\varphi\colon 0 \le T(s) \wedge \Psi_f(T)(s) \le T(s) \wedge T(s) \le g(s)) \neq \emptyset\ .
\end{aligned}$$

□

C Cooperative Verifiers

The baseline verifier from Def. 3 may return any counterexample. Without further restrictions, a verifier may start enumerating these counterexamples, which may yield a bad performance. For instance, let $f = [c = 1]$,

$$\begin{aligned}
&\texttt{while}\,(\,c = 0 \wedge x < 100\,)\,\{\,\{\,x \coloneqq x + 1\,\}\,[\,0.99\,]\,\{\,c \coloneqq 1\,\}\,\}\,,\ \text{and}\\
&T\ =\ [c = 0 \wedge x < 100] \cdot (\alpha \cdot c + \beta \cdot x + \gamma) + [c = 1] + [c = 0 \wedge x \ge 100] \cdot 0\ .
\end{aligned}$$

Assume that the verifier returns two consecutive states s_1, s_2 with $s_i(c) = 0$ and $s_i(x) = i$ for $i \in \{1, 2\}$. The constraints ensuring inductivity at s_1 and s_2 are

$$\Psi_f(T)(s_i)\quad =\quad 0.99 \cdot ((i+1) \cdot \beta + \gamma) + 0.01\ \le\ i \cdot \beta + \gamma\quad =\quad T(s_i)\ .$$

The two constraints are very similar and there is only little information gain from the constraint obtained from s_2. Constraints obtained from more diverse states such as s' with, e.g. $s'(x) = 98$, prune many more undesired template instances.

We are therefore interested in cooperative verifiers – *teachers* – that return sufficiently diverse counterexamples. We tackle diversity by defining a distance measure $\mu\colon S \times S \to \mathbb{R}$ on states[11] and provide the verifier *additionally* to I with the last counterexample s' and a lower bound m. The cooperative verifier then preferably returns a new counterexample s such that $\mu(s, s') \ge m$.

Definition 5. *A* cooperative verifier $\mathsf{CVerify}_\mu\colon S \times \mathbb{R} \to \mathsf{LinExp} \to \{\mathsf{true}\} \cup S$ *is a function such that (i)* $\mathsf{CVerify}_\mu(s_{last}, m)$ *is a verifier for all* $s_{last} \in S$ *and* $m \in \mathbb{R}$*, and (ii)* $\mathsf{CVerify}_\mu(s_{last}, m)(I) = s$ *implies either (a)* $\mu(s, s_{last}) \ge m$ *or (b) for all* $\hat{s} \in \mathsf{CounterEx}_I$*,* $\mu(s_{last}, \hat{s}) \le m$*.*

[11] In our implementation, we use the Manhattan distance on the program variables.

Since every cooperative verifier refines the contract of a verifier, our correctness and completenes results carry over. Furthermore, we can easily generalize the cooperative verifier beyond taking into account only the last counterexample. In our implementation, rather than a fixed value m, we adapt m during runtime: If we succeed in finding two counterexamples that were m apart, we update $m \leftarrow m \cdot d$, otherwise, $m \leftarrow m \cdot {}^1\!/_d$ for suitable values of d.

D Additional Details on Template Refinement

Our problem statement requires a *given* template T and selecting templates is hard. If T is too restrictive, it excludes admissible invariants, i.e. $\langle T \rangle \cap \mathsf{AdmInv} = \emptyset$. If T is too liberal, the synthesizer has to search a high-dimensional space.

We take an optimistic approach as depicted in Fig. 1: we start with a small restrictive template T_1. If we learn that T_1 contains no admissible invariant, we ask the template generator for a new template T_2, and so on, until we eventually (and hopefully) find a small template T_i with $\langle T_i \rangle \cap \mathsf{AdmInv} \neq \emptyset$. Below, we investigate three heuristics. While the approaches do *not* yield step-wise refinements, i.e. $\langle T_i \rangle \subseteq \langle T_{i+1} \rangle$, all approaches ensure *progress*, i.e. $\langle T_i \rangle \subsetneq \langle T_{i+j} \rangle$ for some $j \geq 1$. For finite-state programs, progress ensures completeness: we eventually reach a maximally-partitioned template T in which every state has its own piece; $\langle T \rangle$ then contains at least one admissible invariant, namely $\mathsf{lfp}\ \Phi_f$.

Static Hyperrectangle Refinement for Finite-State Programs. For *finite-state* programs, we obtain lower- and upper bounds on each variable. Hence, the state space is (a subset of) a bounded hyperrectangle. We obtain template T_i for $i > 1$ by splitting each dimension of this hyperrectangle into i equally-sized parts[12], thus obtaining (at most) $i^{|\mathsf{Vars}|}$ new hyperrectangles. Let the hyperrectangles be described by expressions $R_1, \ldots, R_m$, and assume that the initial template is $T_1 = \sum_{j=1}^{n} [B_j] \cdot E_j$. Then, we obtain T_i as $T_i = \sum_{j=1}^{n} \sum_{k}^{m} [B_j \wedge R_k] \cdot E_{k,j}$.

Dynamic Hyperrectangle Refinement. We proceed as for static refinement but do not fix *where* we split the hyperrectangle, i.e. given a possibly unbounded hyperrectangle capturing the (now possibly infinite) state space, we introduce template variables encoding the boundaries of the hyperrectangles, as in Eq. (3).

Inductivity-Guided Refinement. We refine templates using a hint, particularly the last partially admissible instance I in the synthesizer concerning a fixed-partition template $T = \sum_{j=1}^{n} [B_j] \cdot E_j$. We split every B_j into those parts where I is partially inductive, i.e. $\Psi_f(I) \preceq I$, and where it is not. This partitioning can be computed symbolically using the construction from [12, Thm. 7].

E Additional Details on Experiments

The finite-state (Fig. 4a) benchmarks were conducted on a 2.1GHz Intel Xeon processor (one core per benchmark). All other benchmarks were conducted on a 2.3 GHz Dual-Core Intel Core i5.

[12] if possible, i.e. if i does not exceed the size of the dimension.

E.1 Details on the Comparison of cegispro2 and Storm

Table 2 depicts the results for finite-state programs; the corresponding programs are depicted in Appx. E.4. We employ a *cooperative verifier* with $d = 2$. Column Prog depicts the name of the program and S is the program's state-space size. sp and dd are the runtimes of Storm's sparse and decision diagram-based engines, respectively. best is the best runtime under all CEGIS configurations, which are then listed. We show the number $|S'|$ of counterexamples, the size $|I|$ of the inductive invariant (in terms of the number of linear piecewise), the fraction $t_s\%$ of time spent in the synthesizer, and the total time t.

Table 2: Storm vs. cegispro2 on finite-state programs (TO=2h, MO=8GB. Time in seconds).

		Storm			cegispro2											
					induct.-guided				static				dynamic			
Prog	$\|S\|$	sp	dd	best	$\|S'\|$	$\|I\|$	$t_s\%$	t	$\|S'\|$	$\|I\|$	$t_s\%$	t	$\|S'\|$	$\|I\|$	$t_s\%$	t
boundedrwmultistep	$1 \cdot 10^5$	MO	TO	3	33	10	40	3	–	–	–	TO	–	–	–	TO
		MO	TO	10	55	16	36	10	–	–	–	TO	–	–	–	TO
		MO	TO	–	–	–	–	TO	–	–	–	TO	–	–	–	TO
brp	$1 \cdot 10^{10}$	MO	TO	11	56	23	40	11	70	10	35	18	–	–	–	TO
		MO	TO	54	138	42	63	253	125	17	27	54	–	–	–	TO
		MO	TO	56	104	41	54	111	122	17	30	56	–	–	–	TO
brpfinitefamily	$16 \cdot 10^{13}$	TO	TO	8	53	7	72	10	67	7	44	8	54	9	52	29
		TO	TO	17	64	13	74	17	215	19	66	373	–	–	–	TO
		TO	TO	18	68	12	68	18	231	19	80	731	–	–	–	TO
chain	$1 \cdot 10^{12}$	MO	TO	1	97	6	68	10	66	3	50	1	–	–	–	TO
		MO	TO	24	–	–	–	TO	116	5	86	24	–	–	–	TO
		MO	TO	4933	–	–	–	TO	503	23	81	4933	–	–	–	TO
chainselectstepsize	$3 \cdot 10^7$	MO	TO	9	156	7	71	29	156	7	71	29	81	7	49	9
		MO	TO	96	–	–	–	TO	179	15	70	96	–	–	–	TO
		MO	TO	66	–	–	–	TO	164	15	58	66	–	–	–	TO
gridbig	$1 \cdot 10^6$	11	–	–	–	–	–	TO	–	–	–	TO	–	–	–	TO
gridsmall	$1 \cdot 10^2$	<1	32	1	15	7	36	1	46	10	37	3	20	5	39	2
		<1	32	2	26	11	35	2	77	17	32	10	71	10	82	59
zeroconf	$1 \cdot 10^8$	MO	TO	<1	7	3	22	<1	7	3	26	<1	7	3	31	<1
		MO	TO	<1	105	22	60	32	9	5	39	<1	156	7	92	215
		MO	TO	63	–	–	–	TO	173	23	59	63	–	–	–	TO
zeroconffamily	$1 \cdot 10^{16}$	TO	TO	2	48	3	49	2	48	3	49	2	59	3	52	3
		TO	TO	6	–	–	–	TO	77	9	56	6	252	13	81	854
		TO	TO	20	–	–	–	TO	99	9	70	20	164	13	73	265

E.2 Details on the Comparison of Absynth and cegispro2

Tab. 3 depicts the results on the UPAST benchmarks. Prog is the name of the benchmark, t denotes the runtime required by the tools and bound indicates their respective synthesized bounds. $|S'|$ depicts the number of counterexamples required by cegispro2.

Table 3: ABSYNTH vs. CEGISPRO2 on expected runtimes (TO=20min, MO=8GB. Time in seconds).

	ABSYNTH			CEGISPRO2	
PROG	t	bound	$\|S'\|$	t	bound
bayesiannetwork	0.15	1 + 2 max(0, n)	1	2.99	(n * 2.0)
ber	≤0.01	1 + 2 max(0, n - x)	3	0.08	((x * -2.0) + (n * 3.0))
cowboyduel	≤0.01	1 + 1.2 max(0, flag)	1	0.06	11/5
C4Bt09	0.02	1 + max(0, -j + x)	15	0.37	max[((j * -1.0) + x + 1.0), ((j * -1.0) + x + 1.0)]
C4Bt13	0.02	1 + 2 max(0, x) + max(0, y)	8	0.25	((x * 2.0) + y)
C4Bt19	0.04	3 + max(0, 51 + i + k) + 2 max(0, i)	16	0.42	((i * 2.0) + k + -46.0)
C4Bt61	0.02	2 + max(0, l)	15	0.43	(l + -3.0)
condand	≤0.01	1 + 2 max(0, m)	4	0.09	((m * 2.0) + 1.0)
coupon	0.05	10 max(0, 5 - i)	5	0.25	max[149/12, 137/12, 61/6, 17/2, (i * 3/2)]
fcall	≤0.01	1 + 4 max(0, n - x)	3	0.09	((x * -3.0) + (n * 3.0))
fillingvol	0.09	1 + 0.666667 max(0, 10 - vM + vTF)	10	0.35	max[((vM * -1/36) + (vTF * 1/36) + 2.0), ((vM * -2/3) + (vTF * 2/3) + -583/180)]
geo	≤0.01	3	1	0.06	3.0
linear01	≤0.01	1 + 0.6 max(0, x)	1	0.06	x
trappedminer	0.03	1 + 11.5 max(0, -i + n)	72	3.58	((i * -3.0) + (n * 3.0) + 1.0)
noloop	≤0.01	2	1	0.06	2.0
prdwalk	0.05	1 + 0.571429 max(0, 4 + n - x)	6	0.27	((x * -4/7) + (n * 4/7) + 37/21)
prseq	0.04	0.65 max(0, x - y) + 0.35 max(0, y)	266	13.63	((x * 13/20) + (y * -1/2) + 47/20)
prspeed	0.04	1 + 2 max(0, m - y) + 0.666667 max(0, n - x)	18	0.41	max[((y * -2.0) + (m * 2.0) + (n * 2/3) + 1/3), ((x * -2/3) + (n * 2/3) + 1/3)]
race	0.17	1 + 0.666667 max(0, 9 - h + t)	32	1.95	((h * -2/3) + (t * 2/3) + 13/3)
rejectionsampling	4.03	1 + 5 max(0, n)	20	0.53	((n * 5.0) + 1.0)
rfindmc	≤0.01	1 + max(0, -i + k)	1	0.07	(k * 2.0)
rfindlv	≤0.01	1 + 2 max(0, flag)	1	0.05	3.0
rdseql	0.02	1 + 2.25 max(0, x) + max(0, y)	13	0.29	((x * 9/4) + y + -1/4)
rdspeed	0.03	1 + 2 max(0, m - y) + 0.666667 max(0, n - x)	18	0.44	max[((y * -2.0) + (m * 2.0) + (n * 2/3) + 1/3), ((x * -2/3) + (n * 2/3) + 1/3)]
sprdwalk	≤0.01	1 + 2 max(0, n - x)	3	0.08	((x * -3.0) + (n * 3.0))

E.3 Details on the Comparison of Exist and cegispro2

Tab. 4 depicts the results on the sub-invariant benchmarks. Prog is the name of the benchmark, t denotes the runtime required by the tools, and $|S'|$ depicts the number of counterexamples required by cegispro2.

Table 4: Exist vs. cegispro2 (TO=5min, MO=8GB. Time in seconds).

	Exist	cegispro2	
Prog	t	$\|S'\|$	t
BiasDir1_0	78.31	0	0.15
BiasDir1_1	97.29	1	0.08
BiasDir2_0	66.42	0	0.07
BiasDir2_1	171.83	1	0.08
BiasDir3_0	66.55	0	0.08
BiasDir3_1	99.23	1	0.08
Bin01_0	53.19	1	0.06
Bin02_0	83.04	1	0.06
Bin03_0	53.13	1	0.06
Bin11_0	32.52	1	0.06
Bin11_1	TO	3	0.08
Bin12_0	78.19	1	0.06
Bin12_1	218.17	3	0.09
Bin13_0	32.95	1	0.05
Bin13_1	TO	3	0.08
Bin21_0	54.12	2	0.06
Detm1_0	22.8	3	0.06
Detm1_1	57.72	3	0.08
Duel1_0	TO	5	0.12
Duel2_0	TO	1	0.09
Fair1_0	30.29	2	0.08
Fair1_1	37.05	2	0.1
Gambler01_0	35.81	2	0.07
Geo01_0	31.71	1	0.05
Geo01_1	68.53	1	0.05
Geo01_2	73.85	2	0.09
Geo11_0	42.13	1	0.05
Geo21_0	37.1	1	0.05
GeoAr01_0	122.83	2	0.06
GeoAr01_1	33.81	2	0.08
LinExp1_0	189.11	7	0.53
LinExp1_1	196.32	3	0.52
PrinSys1_0	13.27	0	0.06
RevBin1_0	179.97	2	0.09
RevBin1_1	52.03	1	0.05
Sum01_0	TO	4	0.09
Mart1_0	84.39	–	TO
Mart1_1	TO	–	TO

E.4 Programs from Tab. 2

```
nat x [0,20000];
nat s [1,5];

while(0<x & x<20000 & 1<=s & s <= 5){
   {x:=x-1}
   [0.5]
   {
   if (x=1){
      s := 1 : 1/5 + 2 : 1/5 + 3 : 1/5 + 4: 1/5 + 5 : 1/5;
   }else{
      skip
   };
    x := x + s;
  }
}
```

Fig. 5: boundedrwmultistep

```
nat sent [0,8000000000];
nat failed [0,10];

while(failed<10 & sent<8000000000){
   {failed:=0;sent:=sent+1}[0.99]{failed:=failed+1}
}
```

Fig. 6: brp

```
nat sent [0,8000000];
nat failed [0,5];
nat MAXSENT [0,8000000];
nat MINFAILED;

while(failed<MINFAILED & sent<MAXSENT & MAXSENT <= 8000000 & 5 <= MINFAILED){
   {failed:=0;sent:=sent+1}[0.99]{failed:=failed+1}
}
```

Fig. 7: brpfinitefamily

```
nat c [0,1];
nat x [0,1000000000000];

while(c<=0 & x<1000000000000){
   {c:=1}[0.000000000001]{x:=x+1}
}
```

Fig. 8: chain

```
nat c [0,1];
nat x [0,10000000];
nat step [0,10];

while(c<=0 & x<10000000 & 0<=step & step <=10){
   if(step=0){
      step:= (1) : 1/10 + (2) : 1/10 + (3) : 1/10 + (4) : 1/10
                  + (5) : 1/10+ (6) : 1/10+ (7) : 1/10+ (8) : 1/10
                  + (9) : 1/10 + (10) : 1/10;
   }else{
      if(step <= 2){
         {c:=1}[0.0000001]{x:=x+step}
      }else{
         if(step <= 4){
            {c:=1}[0.0000002]{x:=x+step}
         }else{
            if(step <=6){
               {c:=1}[0.0000003]{x:=x+step}
            }else{
               if(step <=8){
                  {c:=1}[0.0000004]{x:=x+step}
               }else{
                  {c:=1}[0.0000005]{x:=x+step}
               }
            }
         }
      }
    }
}
```

Fig. 9: chainselectstepsize

```
nat a [0,1000];
nat b [0,1000];

while(a<1000 & b<1000){
   {a:=a+1}[0.5]{b:=b+1}
}
```

Fig. 10: gridbig

```
nat a [0,10];
nat b [0,10];

while(a<10 & b<10){
   {a:=a+1}[0.5]{b:=b+1}
}
```

Fig. 11: gridsmall

```
nat start [0,1];
nat established [0,1];
nat curprobe [0,100000000];

while(curprobe < 100000000 & established <=0 & start <= 1){
   if(start = 1){
      {start:=0} [0.5] {start:=0; established:=1}
   }else{
      {curprobe := curprobe + 1}
          [0.999999999] {start:=1;curprobe:=0}
   }
}
```

Fig. 12: zeroconf

```
nat start [0,1];
nat established [0,1];
nat curprobe [0,200000000];
nat N [100000000,200000000];

while(curprobe < N & established <=0 & start <= 1 & 100000000 <= N & N <= 200000000){

   if(start = 1){
      {start:=0} [0.5] {start:=0; established:=1}
   }else{
      {curprobe := curprobe + 1}
         [0.999999999] {start:=1;curprobe:=0} }
   }
```

Fig. 13: zeroconffamily