Latticed *k*-Induction with an Application to Probabilistic Programs

Kevin Batz[1](✉), Mingshuai Chen[1](✉), Benjamin Lucien Kaminski[2](✉), Joost-Pieter Katoen[1](✉), Christoph Matheja[3](✉), and Philipp Schröer[1]

[1] RWTH Aachen University, Aachen, Germany
{kevin.batz,chenms,katoen}@cs.rwth-aachen.de
[2] University College London, London, UK
b.kaminski@ucl.ac.uk
[3] ETH Zürich, Zürich, Switzerland
cmatheja@inf.ethz.ch

Abstract. We revisit two well-established verification techniques, *k-induction* and *bounded model checking* (BMC), in the more general setting of fixed point theory over complete lattices. Our main theoretical contribution is *latticed k-induction*, which (i) generalizes classical *k*-induction for verifying transition systems, (ii) generalizes Park induction for bounding fixed points of monotonic maps on complete lattices, and (iii) extends from naturals *k* to transfinite ordinals κ, thus yielding *κ-induction*.

The lattice-theoretic understanding of *k*-induction and BMC enables us to apply both techniques to the *fully automatic verification of infinite-state probabilistic programs*. Our prototypical implementation manages to automatically verify non-trivial specifications for probabilistic programs taken from the literature that—using existing techniques—cannot be verified without synthesizing a stronger inductive invariant first.

Keywords: *k*-induction · Bounded model checking · Fixed point theory · Probabilistic programs · Quantitative verification

1 Introduction

Bounded model checking (BMC) [12,17] is a successful method for analyzing models of hardware and software systems. For checking a *finite-state* transition system (TS) against a safety property ("bad states are unreachable"), BMC unrolls the transition relation until it either finds a counterexample and hence refutes the property, or reaches a pre-computed completeness threshold on the unrolling depth and accepts the property as verified. For *infinite-state* systems, however, such completeness thresholds need not exist (cf. [64]), rendering BMC a *refutation-only* technique. To *verify* infinite-state systems, BMC is typically combined with the search for an *inductive invariant*, i.e., a superset of the reachable

This work has been partially funded by the ERC Advanced Project FRAPPANT under grant No. 787914.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 524–549, 2021.
https://doi.org/10.1007/978-3-030-81688-9_25

states which is closed under the transition relation. Proving a—not necessarily inductive—safety property then amounts to *synthesizing* a sufficiently strong, often complicated, inductive invariant that excludes the bad states. A plethora of techniques target computing or approximating inductive invariants, including IC3 [14], induction [13,20], interpolation [50,51], and predicate abstraction [27,36]. However, invariant synthesis may burden full automation, as it either relies on user-supplied annotations or confines push-button technologies to semi-decision or approximate procedures.

k-induction [65] generalizes the principle of simple induction (aka 1-induction) by considering k consecutive transition steps instead of only a single one. It is more powerful: an invariant can be k-inductive for some $k > 1$ but not 1-inductive. Following the seminal work of Sheeran et al. [65] which combines k-induction with SAT solving to check safety properties, k-induction has found a broad spectrum of applications in the realm of hardware [29,37,45,65] and software verification [10,21–23,55,63]. Its success is due to (1) being a foundational yet potent reasoning technique, and (2) integrating well with SAT/SMT solvers, as also pointed out in [45]: "*the simplicity of applying k-induction made it the go-to technique for SMT-based infinite-state model checking*". This paper explores whether k-induction can have a similar impact on the *fully automatic verification* of infinite-state *probabilistic programs*. That is, we aim to verify that the *expected value* of a specified *quantity*—think: "quantitative postcondition"—after the execution of a probabilistic program is bounded by a specified threshold.

Example 1 (Bounded Retransmission Protocol [19,32]*).* The loop

$$\begin{array}{l}\texttt{while}\,(\,\mathit{sent} < \mathit{toSend} \ \wedge\ \mathit{fail} < \mathit{maxFail}\,)\,\{\\ \quad \{\,\mathit{fail} \coloneqq 0\,;\ \mathit{sent} \coloneqq \mathit{sent} + 1\,\}\ [\,0.9\,]\ \{\,\mathit{fail} \coloneqq \mathit{fail} + 1\,;\ \mathit{totalFail} \coloneqq \mathit{totalFail} + 1\,\}\\ \}\end{array}$$

models a simplified version of the bounded retransmission protocol, which attempts to transmit *toSend* packages via an unreliable channel (that fails with probability 0.1) allowing for at most *maxFail* retransmissions per package.

Using our generalization of k-induction, we can fully automatically verify that the *expected total number of failed transmissions* is at most 1, if the number of packages we want to (successfully) send is at most 3. In terms of weakest preexpectations [38,44,49], this quantitative property reads

$$\mathsf{wp}[\![C]\!]\,(\mathit{totalFail}) \quad \preceq \quad [\mathit{toSend} \leq 3] \cdot (\mathit{totalFail} + 1) + [\mathit{toSend} > 3] \cdot \infty .$$

The bound on the right-hand-side of the inequality is 4-inductive, but *not* 1-inductive; verifying the same bound using 1-induction requires finding a non-trivial—and far less perspicuous—inductive invariant. Moreover, if we consider an arbitrary number of packages to send, i.e., we drop $[\mathit{toSend} \leq 3]$, this bound becomes invalid. In this case, our BMC procedure produces a counterexample, i.e., values for *toSend* and *maxFail*, proving that the bound does not hold. ◁

Lifting the classical formalization (and SAT encoding) of k-induction over TSs to the probabilistic setting is non-trivial. We encounter the following challenges:

(A) *Quantitative reachability.* In a TS, a state reachable within k steps remains reachable on increasing k. In contrast, reachability *probabilities* in Markov chains—a common operational model for probabilistic programs [28]—may increase on increasing k. Hence, proving that the probability of reaching a bad state remains below a given threshold is more intricate than reasoning about qualitative reachability.

(B) *Counterexamples are subsystems.* In a TS, an acyclic path from an initial to a bad state suffices as a witness for refuting safety, i.e., non-reachability. SAT encodings of k-induction rely on this by expressing the absence of witnesses up to a certain path-length. In the probabilistic setting, however, witnesses are no longer single paths [30]. Rather, a witness for the probability of reaching a bad state to exceed a threshold is a *subsystem* [15], i.e., a set of possibly cyclic paths.

(C) *Symbolic encodings.* To enable fully automated verification, we need a suitable encoding such that our lifting integrates well into SMT solvers. Verifying probabilistic programs involves reasoning about execution *trees*, where each (weighted) branch corresponds to a probabilistic choice. A suitable encoding needs to capture such trees which requires more involved theories than encoding paths in classical k-induction.

We address challenges (A) and (B) by developing *latticed k-induction*, which is a proof technique in the rather general setting of fixed point theory over complete lattices. Latticed k-induction generalizes classical k-induction in three aspects: (1) it works with any monotonic map on a complete lattice instead of being confined to the transition relation of a transition system, (2) it generalizes the Park induction principle for bounding fixed points of such monotonic maps, and (3) it extends from natural numbers k to (possibly transfinite) ordinals κ, hence its short name: *κ-induction*.

It is this lattice-theoretic understanding that enables us to lift both k-induction and BMC to reasoning about quantitative properties of probabilistic programs. To enable *automated* reasoning, we address challenge (C) by an incremental SMT encoding based on the theory of quantifier-free mixed integer and real arithmetic with uninterpreted functions ($\mathsf{QF\_UFLIRA}$). We show how to effectively compute all needed operations for κ-induction using the SMT encoding and, in particular, how to decide *quantitative entailments*.

A prototypical implementation of our method demonstrates that κ-induction for (linear) probabilistic programs manages to automatically verify non-trivial specifications for programs taken from the literature which—using existing techniques—cannot be verified without synthesizing a stronger inductive invariant.

Due to space restrictions, most proofs and details about individual benchmarks have been omitted; they are found in an extended version of this paper [8].

Related Work. Besides the aforementioned related work on k-induction, we briefly discuss other automated analysis techniques for probabilistic systems and other approaches for bounding fixed points. Symbolic engines exist for exact inference [26] and sensitivity analysis [33]. Other automated approaches focus on bounding expected costs [56], termination analysis [2,16], and static analysis [3,67]. BMC has been applied in a rather rudimentary form to the on-the-fly verification of finite unfoldings of probabilistic programs [35], and the enumerative generation of counterexamples in finite Markov chains [68]. (Semi-)automated invariant-synthesis techniques can be found in [6,24,41]. A recent variant of IC3 for probabilistic programs called PrIC3 [7] is restricted to finite-state systems. When applied to finite-state Markov chains, our κ-induction operator is related to other operators that have been employed for determining reachabilitiy probabilities through value iteration [4,31,61]. In particular, when iterated on the candidate upper bound, the κ-induction operator coincides with the (upper value iteration) operator in interval iteration [4]; the latter operator can be used together with the up-to techniques (cf. [53,58,59]) to prove our κ-induction rule sound (in contrast, we give an elementary proof). However, the κ-induction operator avoids comparing current and previous iterations. It is thus easier to implement and more amenable to SMT solvers. Finally, the proof rules for bounding fixed points recently developed in [5] are restricted to finite-state systems.

2 Verification as a Fixed Point Problem

We start by recapping some fundamentals on fixed points of monotonic operators on complete lattices before we state our target verification problem.

Fundamentals. For the next three sections, we fix a *complete lattice* $(E, \sqsubseteq)$, i.e. a carrier set E together with a partial order $\sqsubseteq$, such that every subset $S \subseteq E$ has a *greatest lower bound* $\bigsqcap S$ (also called the *meet* of S) and a *least upper bound* $\bigsqcup S$ (also called the *join* of S). For just two elements $\{g, h\} \subseteq E$, we denote their meet by $g \sqcap h$ and their join by $g \sqcup h$. Every complete lattice has a *least* and a *greatest* element, which we denote by $\bot$ and $\top$, respectively.

In addition to $(E, \sqsubseteq)$, we also fix a *monotonic operator* $\Phi\colon E \to E$. By the Knaster-Tarski theorem [43,47,66], every monotonic operator Φ admits a *complete lattice of (potentially infinitely many) fixed points*. The least fixed point $\mathsf{lfp}\ \Phi$ and the greatest fixed point $\mathsf{gfp}\ \Phi$ are moreover constructible by (possibly transfinite) *fixed point iteration* from $\bot$ and $\top$, respectively: Cousot & Cousot [18] showed that there exist ordinals α and β, such that[1]

$$\mathsf{lfp}\ \Phi \;=\; \Phi^{\lceil \alpha \rceil}(\bot) \quad \text{and} \quad \mathsf{gfp}\ \Phi \;=\; \Phi^{\lfloor \beta \rfloor}(\top)\,, \tag{†}$$

where $\Phi^{\lceil \delta \rceil}(g)$ denotes the *upper δ-fold iteration* and $\Phi^{\lfloor \delta \rfloor}(g)$ denotes the *lower δ-fold iteration* of Φ on g, respectively. Formally, $\Phi^{\lceil \delta \rceil}(g)$ is given by[2]

[1] We use lowercase greek letters α, β, γ, δ, etc. to denote arbitrary (possibly transfinite) ordinals and i, j, k, m, n, etc. to denote natural (finite) numbers in $\mathbb{N}$.

[2] To ensure well-definedness of transfinite iterations, we fix an *ambient ordinal* ν and *tacitly assume $\delta < \nu$ for all ordinals δ considered throughout this paper.* Formally, ν is the smallest ordinal such that $|\nu| > |E|$. Intuitively, ν then upper-bounds the length of any repetition-free sequence over elements of E.

$$\Phi^{\lceil\delta\rceil}(g) \quad = \quad \begin{cases} g & \text{if } \delta = 0, \\ \Phi\left(\Phi^{\lceil\gamma\rceil}(g)\right) & \text{if } \delta = \gamma + 1 \text{ is a successor ordinal,} \\ \bigsqcup\left\{\Phi^{\lceil\gamma\rceil}(g) \mid \gamma < \delta\right\} & \text{if } \delta \text{ is a limit ordinal.} \end{cases}$$

Intuitively, if δ is the successor of γ, then we simply do another iteration of Φ. If δ is a limit ordinal, then $\Phi^{\lceil\delta\rceil}(g)$ can also be thought of as a limit, namely of iterating Φ on g. However, simply iterating Φ on g need not always converge, especially if the iteration does not yield an ascending chain. To remedy this, we take as limit the join over the whole (possibly transfinite) iteration sequence, i.e., the least upper bound over all elements that occur along the iteration. The lower δ-fold iteration $\Phi^{\lfloor\delta\rfloor}(g)$ is defined analogously to $\Phi^{\lceil\delta\rceil}(g)$, except that we take a meet instead of a join whenever δ is a limit ordinal.

An important special case for fixed point iteration (see (†)) is when the operator Φ is *Scott-continuous* (or simply *continuous*), i.e., if $\Phi\left(\bigsqcup\{g_1 \sqsubseteq g_2 \sqsubseteq \ldots\}\right) = \bigsqcup\Phi\left(\{g_1 \sqsubseteq g_2 \sqsubseteq \ldots\}\right)$. In this case, α in (†) coincides with the first infinite limit ordinal ω (which can be identified with the set $\mathbb{N}$ of natural numbers). This fact is also known as the Kleene fixed point theorem [1].

Problem Statement. Fixed points are ubiquitous in computer science. Prime examples of properties that can be conveniently characterized as least fixed points include both the set of reachable states in a transition system and the function mapping each state in a Markov chain to the probability of reaching some goal state (cf. [60]). However, least and greatest fixed points are often difficult or even impossible [39] to compute; it is thus desirable to *bound* them.

For example, it may be sufficient to prove that a system modeled as a Markov chain reaches a bad state from its initial state with probability *at most* 10^{-6}, instead of computing *precise* reachability probabilities for each state. Moreover, if said probability is *not* bounded by 10^{-6}, we would like to witness that as well.

In general lattice-theoretic terms, our problem statement reads as follows:

> Given a complete lattice $(E, \sqsubseteq)$, a monotonic operator $\Phi\colon E \to E$, and a candidate upper bound $f \in E$ on $\mathsf{lfp}\ \Phi$,
>
> *prove* or *refute* that $\mathsf{lfp}\ \Phi \ \sqsubseteq\ f$.

For *proving*, we will present *latticed k-induction*; for *refuting*, we will present *latticed bounded model checking*. Running both in parallel may (and under certain conditions: *will*) lead to a decision of the above problem.

3 Latticed *k*-Induction

In this section, we generalize the well-established k-induction verification technique [23,29,37,45,55,65] to *latticed k-induction* (for short: *κ-induction*; reads:

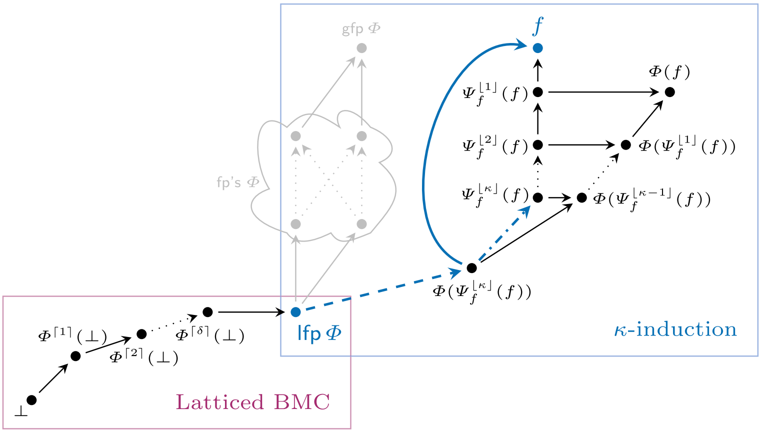
gfp Φ
fp's Φ
f
$\Psi_f^{\lfloor 1 \rfloor}(f)$
$\Phi(f)$
$\Psi_f^{\lfloor 2 \rfloor}(f)$
$\Phi(\Psi_f^{\lfloor 1 \rfloor}(f))$
$\Psi_f^{\lfloor \kappa \rfloor}(f)$
$\Phi(\Psi_f^{\lfloor \kappa-1 \rfloor}(f))$
$\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f))$
$\Phi^{\lceil 1 \rceil}(\bot)$
$\Phi^{\lceil \delta \rceil}(\bot)$
$\Phi^{\lceil 2 \rceil}(\bot)$
lfp Φ
κ-induction
$\bot$
Latticed BMC

Fig. 1. κ-induction and latticed BMC in case that $\mathsf{lfp}\ \Phi \sqsubseteq f$. An arrow from g to h indicates $g \sqsubseteq h$. The solid blue arrow from $\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f))$ to f is the premise of κ-induction, i.e., the LHS of Lemma 2, which implies the dash-dotted blue arrow from $\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f))$ to $\Psi_f^{\lfloor \kappa \rfloor}(f)$, i.e., the RHS of Lemma 2. The dashed blue arrow from $\mathsf{lfp}\ \Phi$ to $\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f))$ is a consequence of the dash-dotted arrow (by Park induction, Theorem 1) and ultimately proves that $\mathsf{lfp}\ \Phi \sqsubseteq f$.

"kappa induction"). With κ-induction, our aim is to *prove* that $\mathsf{lfp}\ \Phi \sqsubseteq f$. To this end, we attempt "ordinary" induction, also known as *Park induction*:

Theorem 1 (Park Induction [57]). *Let $f \in E$. Then*

$$\Phi(f) \ \sqsubseteq\ f \qquad \text{implies} \qquad \mathsf{lfp}\ \Phi \ \sqsubseteq\ f.$$

Intuitively, this principle says: if pushing our candidate upper bound f through Φ takes us *down* in the partial order $\sqsubseteq$, we have verified that f is indeed an upper bound on $\mathsf{lfp}\ \Phi$. The true power of Park induction is that applying Φ *once* tells us something about iterating Φ possibly *transfinitely often* (see (†) in Sect. 2).

Park induction, unfortunately, does *not* work in the reverse direction: If we are unlucky, $f \sqsupseteq \mathsf{lfp}\ \Phi$ *is* an upper bound on $\mathsf{lfp}\ \Phi$, but nevertheless $\Phi(f) \not\sqsubseteq f$. In this case, we say that f is *not inductive*. But how can we verify that f is indeed an upper bound in such a non-inductive scenario? We search *below* f for a *different, but inductive*, upper bound on $\mathsf{lfp}\ \Phi$, that is, we

$$\text{search for an } h \in E \qquad \text{such that} \qquad \mathsf{lfp}\ \Phi \ \sqsubseteq\ \Phi(h) \ \sqsubseteq\ h \ \sqsubseteq\ f.$$

In order to perform a *guided* search for such an h, we introduce the κ-induction operator—a modified version of Φ that is parameterized by our candidate f:

Definition 1 (κ-Induction Operator). *For $f \in E$, we call*

$$\Psi_f\colon \quad E \ \to\ E, \qquad g \ \mapsto\ \Phi(g) \sqcap f$$

the κ-induction operator *(with respect to f and Φ).*

What does Ψ_f do? As illustrated in Fig. 1, if $\Phi(f) \not\sqsubseteq f$ (i.e. f is non-inductive) then "*at least some part of* $\Phi(f)$ *is greater than* f". If the whole of $\Phi(f)$ is greater than f, then $f \sqsubset \Phi(f)$; if only some part of $\Phi(f)$ is greater and some is smaller than f, then f and $\Phi(f)$ are incomparable. The κ-induction operator Ψ_f now *rectifies* $\Phi(f)$ being (partly) greater than f by *pulling* $\Phi(f)$ *down* via the meet with f (i.e., via $\ldots \sqcap f$), so that the result is in no part greater than f. Applying Ψ_f to f hence always yields something below or equal to f.

Together with the observation that Ψ_f is monotonic, iterating Ψ_f on f necessarily *descends* from f downwards in the direction of $\mathsf{lfp}\ \Phi$ (and never below):

Lemma 1 (Properties of the $\boldsymbol{\kappa}$-Induction Operator). *Let $f \in E$ and let Ψ_f be the κ-induction operator with respect to f and Φ. Then*

(a) Ψ_f is monotonic, i.e., $\forall\, g_1, g_2 \in E\colon\ g_1 \sqsubseteq g_2$ implies $\Psi_f(g_1) \sqsubseteq \Psi_f(g_2)$.
(b) Iterations of Ψ_f starting from f are descending, i.e., for all ordinals γ, δ,

$$\gamma < \delta \quad \text{implies} \quad \Psi_f^{\lfloor\delta\rfloor}(f) \sqsubseteq \Psi_f^{\lfloor\gamma\rfloor}(f).$$

(c) Ψ_f is dominated by Φ, i.e., $\forall\, g \in E\colon\ \Psi_f(g) \sqsubseteq \Phi(g)$.

(d) If $\mathsf{lfp}\ \Phi \sqsubseteq f$*, then for any ordinal δ,*

$$\mathsf{lfp}\ \Phi \sqsubseteq \ldots \sqsubseteq \Psi_f^{\lfloor\delta\rfloor}(f) \sqsubseteq \ldots \sqsubseteq \Psi_f^{\lfloor 2\rfloor}(f) \sqsubseteq \Psi_f(f) \sqsubseteq f.$$

The descending sequence $f \sqsupseteq \Psi_f(f) \sqsupseteq \Psi_f^{\lfloor 2\rfloor}(f) \sqsupseteq \ldots$ constitutes our guided search for an inductive upper bound on $\mathsf{lfp}\ \Phi$. For each ordinal κ (hence the short name: κ-induction), $\Psi_f^{\lfloor\kappa\rfloor}(f)$ is a potential candidate for Park induction:

$$\Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \overset{\text{potentially}}{\sqsubseteq} \Psi_f^{\lfloor\kappa\rfloor}(f). \tag{‡}$$

For efficiency reasons, e.g., when offloading the above inequality check to an SMT solver, we will not check the inequality (‡) directly but a property equivalent to (‡), namely whether $\Phi(\Psi_f^{\lfloor\kappa\rfloor}(f))$ is below f instead of $\Psi_f^{\lfloor\kappa\rfloor}(f)$:

Lemma 2 (Park Induction from $\boldsymbol{\kappa}$-Induction). *Let $f \in E$. Then*

$$\Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \sqsubseteq f \quad \text{iff} \quad \Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \sqsubseteq \Psi_f^{\lfloor\kappa\rfloor}(f).$$

Proof. The if-direction is trivial, as $\Psi_f^{\lfloor\kappa\rfloor}(f) \sqsubseteq f$ (Lemma 1(d)). For only-if:

$$\begin{aligned}
\Psi_f^{\lfloor\kappa\rfloor}(f) &\sqsupseteq \Psi_f^{\lfloor\kappa+1\rfloor}(f) && \text{(by Lemma 1(b))}\\
&= \Psi_f\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) && \text{(by definition of } \Psi_f^{\lfloor\kappa+1\rfloor}(f))\\
&= \Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \sqcap f && \text{(by definition of } \Psi_f)\\
&\sqsupseteq \Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right). && \text{(by the premise)} \quad \square
\end{aligned}$$

```
Algorithm 1: Latticed k-induction
  input: $\Phi\colon E \to E$ and $f \in E$.
  output: “verify” if $f$ is a $k$-inductive
          invariant, diverge otherwise.

1 $g \leftarrow f$ ;
2 while $\Phi(g) \not\sqsubseteq f$ do
3     $g \leftarrow \Psi_f(g)$ ;
      // recall: $\Psi_f(g) = \Phi(g) \sqcap f$
4 return verify ;
```

```
Algorithm 2: Latticed BMC
  input: $\Phi\colon E \to E$ and $f \in E$.
  output: “refute” if there exists
          $k \in \mathbb{N}$ with $\Phi^{\lceil k \rceil}(\bot) \not\sqsubseteq f$,
          diverge otherwise.
1 $g \leftarrow \bot$ ;
2 repeat
3     $g \leftarrow \Phi(g)$ ;
4 until $g \not\sqsubseteq f$ ;
5 return refute ;
```

If $\Phi\big(\Psi_f^{\lfloor \kappa \rfloor}(f)\big) \sqsubseteq f$, then Lemma 2 tells us that $\Psi_f^{\lfloor \kappa \rfloor}(f)$ is Park inductive and thereby an upper bound on $\mathsf{lfp}\ \Phi$. Since iterating Ψ_f on f yields a descending iteration sequence (see Lemma 1(b)), $\Psi_f^{\lfloor k \rfloor}(f)$ is below f and therefore f is also an upper bound on $\mathsf{lfp}\ \Phi$. Put in more traditional terms, we have shown that $\Psi_f^{\lfloor \kappa \rfloor}(f)$ is an inductive invariant stronger than f. Formulated as a proof rule, we obtain the following induction principle:

Theorem 2 (*κ*-Induction). *Let $f \in E$ and let κ be an ordinal. Then*

$$\Phi\left(\Psi_f^{\lfloor \kappa \rfloor}(f)\right) \sqsubseteq f \quad \text{implies} \quad \mathsf{lfp}\ \Phi \sqsubseteq f.$$

Proof. Following the argument above, for details see [8, Appx. A.2]. □

An illustration of κ-induction is shown in (the right frame of) Fig. 1. For every ordinal κ, if $\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f)) \sqsubseteq f$, then we call f $(\kappa{+}1)$-*inductive* (for Φ). In particular, κ-induction generalizes Park induction, in the sense that 1-induction *is* Park induction and, $(\kappa > 1)$-induction is a *more general principle of induction.*

Algorithm 1 depicts a (semi-)algorithm that performs *latticed k-induction* (for $k < \omega$) in order to prove $\mathsf{lfp}\ \Phi \sqsubseteq f$ by iteratively increasing k. For implementing this algorithm, we require, of course, that both Φ and Ψ_f are computable and that $\sqsubseteq$ is decidable. Notice that the loop (lines 2–3) never terminates if $f \sqsubset \Phi(f)$—a condition that can easily be checked before entering the loop. Even with this optimization, however, Algorithm 1 is a *proper* semi-algorithm: even if $\mathsf{lfp}\ \Phi \sqsubseteq f$, then f is still not guaranteed to be k-inductive for some $k < \omega$. And even if an algorithm *could* somehow perform transfinitely many iterations, then f is still not guaranteed to be κ-inductive for some ordinal κ:

Counterexample 1 (Incompleteness of *κ*-Induction). *Consider the carrier set $\{0, 1, 2\}$, partial order $0 \sqsubset 1 \sqsubset 2$, and the monotonic operator Φ with $\Phi(0) = 0 = \mathsf{lfp}\ \Phi$, and $\Phi(1) = 2$, and $\Phi(2) = 2 = \mathsf{gfp}\ \Phi$. Then $\mathsf{lfp}\ \Phi \sqsubseteq 1$, but for any ordinal κ, $\Psi_1^{\lfloor \kappa \rfloor}(1) = 1$ and $\Phi(1) = 2 \not\sqsubseteq 1$. Hence 1 is not κ-inductive.* ◁

Despite its incompleteness, we now provide a *sufficient* criterion which ensures that *every* upper bound on $\mathsf{lfp}\ \Phi$ is κ-inductive for some ordinal κ.

Theorem 3 (Completeness of κ-Induction for Unique Fixed Point). *If* $\mathsf{lfp}\ \Phi = \mathsf{gfp}\ \Phi$ *(i.e. Φ has* exactly one *fixed point), then, for every $f \in E$,*

$$\mathsf{lfp}\ \Phi \ \sqsubseteq\ f \quad \text{implies} \quad f \textit{ is } \kappa\textit{-inductive for some ordinal } \kappa.$$

Proof. By the Knaster-Tarski theorem, we have $\Phi^{\lfloor\beta\rfloor}(\top) = \mathsf{gfp}\ \Phi$ for some ordinal β. We then show that f is (β+1)-inductive; see [8, Appx A.3] for details. □

The proof of the above theorem immediately yields that, if the unique fixed point can be reached through *finite* fixed point iterations starting at $\top$, then f is k-inductive for some *natural* number k; Algorithm 1 thus eventually terminates.

Corollary 1. *If $\Phi^{\lfloor n\rfloor}(\top) = \mathsf{lfp}\ \Phi$ for some $n \in \mathbb{N}$, then, for every $f \in E$,*

$$\mathsf{lfp}\ \Phi \ \sqsubseteq\ f \quad \text{implies} \quad f \textit{ is } n\textit{-inductive for some } n \in \mathbb{N}.$$

4 Latticed vs. Classical k-Induction

We show that our purely lattice-theoretic κ-induction from Sect. 3 generalizes classical k-induction for hardware- and software verification. To this end, we first recap how k-induction is typically formalized in the literature [10,23,29,37]: Let TS $= (S, I, T)$ be a transition system, where S is a (countable) set of *states*, $I \subseteq S$ is a non-empty set of *initial states*, and $T \subseteq S \times S$ is a *transition relation*. As in the seminal work on k-induction [65], we require that T is a *total* relation, i.e., every state has at least one successor. This requirement is sometimes overlooked in the literature, which renders the classical SAT-based formulation of k-induction ((1a) and (1b) below) unsound in general.

Our goal is to verify that a given *invariant property* $P \subseteq S$ covers all states reachable in TS from some initial state. Suppose that I, T and P are characterized by logical formulae $I(s)$, $T(s, s')$ and $P(s)$ (over the free variables s and s'), respectively. Then, achieving the above goal with classical k-induction amounts to proving the validity of

$$I(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \implies P(s_1) \wedge \ldots \wedge P(s_k), \quad \text{and} \tag{1a}$$

$$P(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge P(s_k) \wedge T(s_k, s_{k+1}) \implies P(s_{k+1}). \tag{1b}$$

Here, the *base case* (1a) asserts that P holds for *all states reachable within k transition steps from some initial state*; the *induction step* (1b) formalizes that P is *closed under taking up to k transition steps*, i.e., if we start in P and stay in P for up to k steps, then we also end up in P after taking the (k+1)-st step. If both (1a) and (1b) are valid, then classical k-induction tells us that the property P holds for *all* reachable states of TS. How is the above principle reflected in *latticed* k-induction (cf. Sect. 3)? For that, we choose the complete lattice $(2^S, \subseteq)$, where 2^S denotes the powerset of S; the least element is $\bot = \emptyset$ and the meet operation is standard intersection $\cap$.

Moreover, we define a monotonic operator Φ whose least fixed point precisely characterizes the set of reachable states of the transition system TS:

$$\Phi\colon \quad 2^S \;\to\; 2^S, \qquad F \;\mapsto\; I \cup \mathsf{Succs}(F),$$

That is, Φ maps any given set of states $F \subseteq S$ to the union of the initial states I and of those states $\mathsf{Succs}(F)$ that are reachable from F using a single transition.[3]

Using the κ-induction operator Ψ_P constructed from Φ and P according to Definition 1, the principle of κ-induction (cf. Theorem 2) then tells us that

$$\Phi\left(\Psi_P^{\lfloor\kappa\rfloor}(P)\right) \;\subseteq\; P \qquad \text{implies} \qquad \underbrace{\mathsf{lfp}\,\Phi}_{\text{reachable states of TS}} \;\subseteq\; P.$$

For our above choices, the premise of κ-induction equals the classical formalization of k-induction—formulae (1a) and (1b)—because the set of initial states I is "baked into" the operator Φ. More concretely, for the base case (1a), we have

$$\underbrace{\underbrace{\underbrace{\underbrace{I(s_1)}_{\Phi(\emptyset)} \wedge T(s_1, s_2)}_{\Phi^{\lceil 2\rceil}(\emptyset)} \wedge \ldots \wedge T(s_{k-1}, s_k)}_{\Phi^{\lceil k\rceil}(\emptyset)} \;\Longrightarrow\; P(s_1) \wedge \ldots \wedge P(s_k)}_{\text{meaning} \quad \Phi^{\lceil k\rceil}(\emptyset) \;\subseteq\; P}\,.$$

In other words, formula (1a) captures those states that are reachable from I via at most k transitions. If we assume that (1a) is valid, then P contains all initial states and formula (1b) coincides with the premise of κ-induction:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{P(s_1) \wedge T(s_1, s_2)}_{\Phi(P)} \wedge P(s_2)}_{\Psi_P(P) \;=\; \Phi(P) \cap P} \wedge T(s_2, s_3) \wedge \ldots \wedge P(s_k)}_{\Psi_P^{\lfloor k-1\rfloor}(P)} \wedge T(s_k, s_{k+1})}_{\Phi\left(\Psi_P^{\lfloor k-1\rfloor}(P)\right)} \;\Longrightarrow\; P(s_{k+1})}_{\text{meaning} \quad \Phi\left(\Psi_P^{\lfloor k-1\rfloor}(P)\right) \;\subseteq\; P}\,.$$

It follows that, when considering transition systems, our (latticed) κ-induction is equivalent to the classical notion of k-induction for $\kappa < \omega$:

Theorem 4. *For every natural number $k \geq 1$,*

$$\Phi\left(\Psi_P^{\lfloor k-1\rfloor}(P)\right) \;\subseteq\; P \qquad \text{iff} \qquad \text{formulae (1a) and (1b) are valid.}$$

[3] Formally, $\mathsf{Succs}(F) \;\triangleq\; \{\, t' \mid t \in F,\, (t, t') \in T \,\}$.

$$\begin{array}{rcl} C & ::= & \texttt{skip} \\ & \mid & x \coloneqq e \\ & \mid & C\,;C \\ & \mid & \{\,C\,\}\,[p]\,\{\,C\,\} \\ & \mid & \texttt{if}\ (\,\varphi\,)\ \{\,C\,\}\ \texttt{else}\ \{\,C\,\} \\ & \mid & \texttt{while}\,(\,\varphi\,)\,\{\,C\,\} \end{array}$$

(a) pGCL programs

$$\begin{array}{rcl} e & ::= & n \\ & \mid & x \\ & \mid & n \cdot e \\ & \mid & e + e \\ & \mid & e \dot{-} e \ \ (\text{monus } \max\{0, e - e\}) \end{array}$$

(b) Linear expressions

$$\begin{array}{rcl} \varphi & ::= & e < e \\ & \mid & \varphi \wedge \varphi \\ & \mid & \neg\varphi \end{array}$$

(c) Linear guards

Fig. 2. Syntax of pGCL programs, linear expressions, and guards, where x is a variable taken from a countable set Vars of program variables (evaluating to natural numbers), $p \in [0,1] \cap \mathbb{Q}$ is a rational probability, and $n \in \mathbb{N}$ is a constant.

5 Latticed Bounded Model Checking

We complement κ-induction with a latticed analog of bounded model checking [11,12] for *refuting* that $\mathsf{lfp}\ \Phi \sqsubseteq f$. In lattice-theoretic terms, bounded model checking amounts to a *fixed point iteration* of Φ on $\bot$ while continually checking whether the iteration exceeds our candidate upper bound f. If so, then we have indeed refuted $\mathsf{lfp}\ \Phi \sqsubseteq f$:

Theorem 5 (Soundness of Latticed BMC). *Let $f \in E$. Then*

$$\exists \,\text{ordinal}\ \delta\colon \quad \Phi^{\lceil \delta \rceil}(\bot) \ \not\sqsubseteq\ f \qquad \text{implies} \qquad \mathsf{lfp}\ \Phi \ \not\sqsubseteq\ f.$$

Furthermore, if we were actually able to perform transfinite iterations of Φ on $\bot$, then latticed bounded model checking is also complete: If f is in fact *not* an upper bound on $\mathsf{lfp}\ \Phi$, this *will* be witnessed at some ordinal:

Theorem 6 (Completeness of Latticed BMC). *Let $f \in E$. Then*

$$\mathsf{lfp}\ \Phi \ \not\sqsubseteq\ f \qquad \text{implies} \qquad \exists \,\text{ordinal}\ \delta\colon \quad \Phi^{\lceil \delta \rceil}(\bot) \ \not\sqsubseteq\ f.$$

More practically relevant, if Φ is continuous (which is the case for Bellman operators characterizing reachability probabilities in Markov chains), then a simple *finite* fixed point iteration, see Algorithm 2, is sound and complete for refutation:

Corollary 2 (Latticed BMC for Continuous Operators). *Let $f \in E$ and let Φ be continuous. Then*

$$\exists\, n \in \mathbb{N}\colon \quad \Phi^{n}(\bot) \ \not\sqsubseteq\ f \qquad \text{iff} \qquad \mathsf{lfp}\ \Phi \ \not\sqsubseteq\ f.$$

6 Probabilistic Programs

In the remainder of this article, we employ latticed k-induction and BMC to verify imperative programs with access to discrete probabilistic choices—branching on the outcomes of coin flips. In this section, we briefly recap the necessary background on formal reasoning about probabilistic programs (cf. [44,49] for details).

6.1 The Probabilistic Guarded Command Language

Syntax. Programs in the *probabilistic guarded command language* pGCL adhere to the grammar in Fig. 2a. The semantics of most statements is standard. In particular, the *probabilistic choice* $\{\, C_1 \,\}\ [\,p\,]\ \{\, C_2 \,\}$ flips a coin with bias $p \in [0,1] \cap \mathbb{Q}$. If the coin yields heads, it executes C_1; otherwise, C_2. In addition to the syntax in Fig. 2, we admit standard expressions that are definable as syntactic sugar, e.g., true, false, $\varphi_1 \vee \varphi_2$, $e_1 = e_2$, $e_1 \leq e_2$, etc.

Program States. A *program state* σ maps every variable in Vars to its value, i.e., a natural number in $\mathbb{N}$.[4] To ensure that the set of program states Σ remains countable[5], we restrict ourselves to states in which only finitely many variables—those that appear in a given program—evaluate to non-zero values. Formally,

$$\Sigma \;\triangleq\; \left\{ \sigma\colon \mathsf{Vars} \to \mathbb{N} \;\middle|\; \left|\{\, x \in \mathsf{Vars} \mid \sigma(x) \neq 0 \,\}\right| < \infty \right\}.$$

The evaluation of expressions e and guards φ under a state σ, denoted by $e(\sigma)$ and $\varphi(\sigma)$, is standard. For example, we define the evaluation of "monus" as

$$(e_1 \dot{-} e_2)(\sigma) \;\triangleq\; \max\{\, 0,\; e_1(\sigma) - e_2(\sigma) \,\}\,.$$

6.2 Weakest Preexpectations

Expectations. An *expectation* $f\colon \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$ is a map from program states to the non-negative reals extended by infinity. We denote by $\mathbb{E}$ the set of all expectations. Moreover, $(\mathbb{E},\, \preceq)$ forms a complete lattice, where the partial order $\preceq$ is given by the pointwise application of the canonical ordering $\leq$ on $\mathbb{R}_{\geq 0}^{\infty}$, i.e.,

$$f \;\preceq\; g \qquad \text{iff} \qquad \forall\, \sigma \in \Sigma\colon \quad f(\sigma) \;\leq\; g(\sigma).$$

To conveniently describe expectations evaluating to some $r \in \mathbb{R}_{\geq 0}^{\infty}$ for every state, we slightly abuse notation and denote by r the constant expectation $\lambda\sigma\text{.}\; r$. Similarly, given an arithmetic expression e, we denote by e the expectation $\lambda\sigma\text{.}\; e(\sigma)$.

[4] We prefer unsigned integers because our quantitative "specifications" (aka *expectations*) must evaluate to non-negative numbers. Otherwise, expectations like $x+y$ are not well-defined, and, as a remedy, we would frequently have to take the absolute value of every program variable. Restricting ourselves to unsigned variables does not decrease expressive power as signed variables can be emulated (cf. [9, Sec. 11.2]).

[5] In order to avoid any technical issues pertaining to measurability.

Table 1. Rules defining the weakest preexpectation transformer.

C	$\mathsf{wp}\ [\![C]\!]\ (g)$
$\texttt{skip}$	g
$x := e$	$g\,[x/e]$
$C_1\,;\,C_2$	$\mathsf{wp}[\![C_1]\!]\,\big(\mathsf{wp}[\![C_2]\!]\,(g)\big)$
$\{\,C_1\,\}\ [\,p\,]\ \{\,C_2\,\}$	$p\cdot\mathsf{wp}[\![C_1]\!]\,(g)+(1-p)\cdot\mathsf{wp}[\![C_2]\!]\,(g)$
$\texttt{if}\ (\,\varphi\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}$	$[\varphi]\cdot\mathsf{wp}[\![C_1]\!]\,(g)+[\neg\varphi]\cdot\mathsf{wp}[\![C_2]\!]\,(g)$
$\texttt{while}\,(\,\varphi\,)\,\{\,C'\,\}$	$\mathsf{lfp}\ h\bullet\ [\neg\varphi]\cdot g+[\varphi]\cdot\mathsf{wp}[\![C']\!]\,(h)$

The least element of $(\mathbb{E}, \preceq)$ is 0 and the greatest element is ∞. We employ the *Iverson bracket* notation to cast Boolean expressions into expectations, i.e.,

$$[\varphi] \quad = \quad \lambda\sigma\bullet \begin{cases} 1 & \text{if } \varphi(\sigma) = \mathsf{true}, \\ 0 & \text{if } \varphi(\sigma) = \mathsf{false}. \end{cases}$$

The *weakest preexpectation transformer* $\mathsf{wp}\colon \mathsf{pGCL} \to (\mathbb{E} \to \mathbb{E})$ is defined in Table 1, where $g\,[x/e]$ denotes the substitution of variable x by expression e, i.e.,

$$g\,[x/e] \triangleq \lambda\sigma\bullet\ g(\sigma\,[x \mapsto e(\sigma)]),\quad \text{where}\quad \sigma\,[x \mapsto e(\sigma)] \triangleq \lambda y\bullet \begin{cases} e(\sigma) & \text{if } y = x, \\ \sigma(y) & \text{otherwise.} \end{cases}$$

We call $\mathsf{wp}[\![C]\!]\,(g)$ the *weakest preexpectation* of program C w.r.t. postexpectation g. The weakest preexpectation $\mathsf{wp}[\![C]\!]\,(g)$ is itself an expectation of type $\mathbb{E}$, which maps each initial state σ to the expected value of g after running C on σ. More formally, if μ_C^σ is the distribution over final states obtained by executing C on initial state σ, then for any postexpectation g [44],

$$\mathsf{wp}[\![C]\!]\,(g)\,(\sigma) \;=\; \sum\nolimits_{\tau\in\Sigma} \mu_C^\sigma(\tau)\cdot g(\tau).$$

For a gentle introduction to weakest preexpectations, see [38, Chap. 2 and 4].

7 BMC and *k*-Induction for Probabilistic Programs

We now instantiate latticed κ-induction and BMC (as developed in Sects. 2 to 5) to enable verification of loops written in pGCL; we discuss practical aspects later in Sects. 7.1 to 7.3 and Sect. 8. For the next two sections, we fix a loop

$$C_{\text{loop}} \quad = \quad \texttt{while}\,(\,\varphi\,)\,\{\,C\,\}\,.$$

For simplicity, we assume that the loop body C is loop-free (every probabilistic program can be rewritten as a single while loop with loop-free body [62]).

Given an expectation $g \in \mathbb{E}$ and a candidate upper bound $f \in \mathbb{E}$ on the expected value of g after executing C_{loop} (i.e. $\mathsf{wp}[\![C_{\text{loop}}]\!]\,(g)$), we will apply latticed verification techniques to check whether f indeed upper-bounds $\mathsf{wp}[\![C_{\text{loop}}]\!]\,(g)$.

To this end, we denote by Φ the *characteristic functional* of C_{loop} and g, i.e.,

$$\Phi\colon \quad \mathbb{E} \;\to\; \mathbb{E}, \qquad h \;\mapsto\; [\neg\varphi]\cdot g + [\varphi]\cdot \mathsf{wp}[\![C]\!]\,(h)\,,$$

whose least fixed point defines $\mathsf{wp}[\![C_{\text{loop}}]\!]\,(g)$ (cf. Table 1). We remark that Φ is a monotonic—and in fact even continuous—operator over the complete lattice $(\mathbb{E},\, \preceq)$ (cf. Sect. 6.2). In this lattice, the meet is a pointwise minimum, i.e.,

$$h \sqcap h' \;=\; h \,\mathsf{min}\, h' \;\triangleq\; \lambda\sigma\text{.}\;\; \min\{\, h(\sigma), h'(\sigma)\,\}\,.$$

By Definition 1, Φ and g then induce the (continuous) κ-induction operator

$$\Psi_f\colon \quad \mathbb{E} \;\to\; \mathbb{E}, \qquad h \;\mapsto\; \Phi\,(h)\;\mathsf{min}\; f.$$

With this setup, we obtain the following proof rule for reasoning about probabilistic loops as an immediate consequence of Theorem 2:

Corollary 3 (k-Induction for pGCL). *For every natural number $k \in \mathbb{N}$,*

$$\Phi\left(\Psi_f^{\lfloor k \rfloor}(f)\right) \;\preceq\; f \qquad \text{implies} \qquad \mathsf{wp}[\![C_{\text{loop}}]\!]\,(g) \;\preceq\; f.$$

Analogously, refuting that f upper-bounds the expected value of g after execution of C_{loop} via bounded model checking is an instance of Corollary 2:

Corollary 4 (Bounded Model Checking for pGCL).

$$\exists\, n \in \mathbb{N}\colon \quad \Phi^n(0) \;\not\preceq\; f \qquad \text{iff} \qquad \mathsf{wp}[\![C_{\text{loop}}]\!]\,(g) \;\not\preceq\; f.$$

Example 2 (Geometric Loop). The pGCL program

$$C_{\text{geo}} \quad = \quad \texttt{while}\,(\,x = 1\,)\,\{\,\{\,x \coloneqq 0\,\}\,[\,0.5\,]\,\{\,c \coloneqq c + 1\,\}\,\}$$

keeps flipping a fair coin x until it flips heads, sets x to 0, and terminates. Whenever it flips tails instead, it increments the counter c and continues. We refer to C_{geo} as the "geometric loop" because after its execution, the counter variable c is distributed according to a geometric distribution.

What is a (preferably small) upper bound on the expected value $\mathsf{wp}[\![C_{\text{geo}}]\!]\,(c)$ of c after execution of C_{geo}? Using 2-induction, we can (automatically) verify that $c + 1$ is indeed an upper bound: Since $\Phi\,(\Psi_{c+1}(c+1)) \preceq c+1$, where Φ denotes the characteristic functional of C_{geo}, Corollary 3 yields $\mathsf{wp}[\![C_{\text{geo}}]\!]\,(c) \preceq c+1$.

However, $c + 1$ *cannot* be proven an upper bound using Park induction as it is *not* inductive. Moreover, it is indeed the *least* upper bound, i.e., any smaller bound is refutable using BMC (cf. Corollary 4). For example, we have $\mathsf{wp}[\![C_{\text{geo}}]\!]\,(c) \not\preceq c + 0.99$, since $\Phi^{\lceil 11 \rceil}\,(0) \not\preceq c + 0.99$. Finally, we remark that some correct upper bounds only become κ-inductive for *transfinite* ordinals κ. For instance, the innocuous-looking bound $2 \cdot c + 1$ is not k-inductive for any natural number k, but it is $(\omega+1)$-inductive, since $\Phi\big(\Psi_{2\cdot c+1}^{\lfloor \omega \rfloor}(2\cdot c+1)\big) \preceq 2\cdot c+1$. $\lhd$

In principle, we can semi-decide whether $\mathsf{wp}[\![C_{\text{loop}}]\!]\,(g) \not\preceq f$ holds or whether f is k-inductive for some k: it suffices to run Algorithms 1 and 2 in parallel. However, for these two algorithms to actually be semi-decision procedures, we cannot admit arbitrary expectations. Rather, we restrict ourselves to a suitable subset Exp of expectations in $\mathbb{E}$ satisfying all of the following requirements:

1. Exp is closed under computing the characteristic functional Φ, i.e.,
$$\forall\, h \in \mathsf{Exp}\colon \quad \Phi\,(h) \text{ is computable and belongs to } \mathsf{Exp}.$$
2. Quantitative entailments between expectations in Exp are decidable, i.e.,
$$\forall\, h, h' \in \mathsf{Exp}\colon \quad \text{it is decidable whether } h \;\preceq\; h'.$$
3. (For k-induction) Exp is closed under computing meets, i.e.,
$$\forall\, h, h' \in \mathsf{Exp}\colon \quad h \;\mathsf{min}\; h' \text{ is computable and belongs to } \mathsf{Exp}.$$

Below, we show that *linear expectations* meet all of the above requirements.

7.1 Linear Expectations

Recall from Fig. 2b that we assume all expressions appearing in pGCL programs to be linear. For our fragment of syntactic expectations, we consider *extended* linear expressions $\tilde{e}$ that (1) are defined over *rationals* instead of natural numbers and (2) admit ∞ as a constant (but not as a *sub*expression). Formally, the set of extended linear expressions is given by the following grammar:

$$\tilde{e} \;::=\; e \mid \infty \qquad\qquad e \;::=\; r \mid x \mid r \cdot e \mid e + e \mid e \dot{-} e \qquad\qquad (r \in \mathbb{Q}_{\geq 0})$$

Similarly, we admit extended linear expressions (without ∞) in linear guards φ.[6] With these adjustments to expressions and guards in mind, the set LinExp of *linear expectations* is defined by the grammar

$$h \;::=\; \tilde{e} \quad\mid\quad [\varphi] \cdot h \quad\mid\quad h + h.$$

We write $h = h'$ if h and h' are *syntactically identical*; and $h \equiv h'$ if they are *semantically equivalent*, i.e., if for all states σ, we have $h(\sigma) = h'(\sigma)$.

Furthermore, the *rescaling* $c \cdot h$ of a linear expectation h by a constant $c \in \mathbb{Q}_{\geq 0}$ is syntactic sugar for rescaling suitable[7] arithmetic subexpressions of h, e.g.,

$$1/2 \cdot ([x = 1] \cdot 4 + 1/3 \cdot x + \infty) \;\equiv\; 1/2 \cdot [x = 1] \cdot 4 + 1/2 \cdot 1/3 \cdot x + \infty \in \mathsf{LinExp}.$$

A formal definition of the rescaling $c \cdot h$ is found in [8, Appx A.5].

[6] We do not admit ∞ in guards for convenience. In principle, all comparisons with ∞ in guards can be removed by a simple preprocessing step.

[7] We do not rescale every subexpression to account for the corner cases $c \cdot \infty = \infty$ and $0 \cdot \infty = 0$.

If we choose a linear expectation h as a postexpectation, then a quick inspection of Table 1 reveals that the weakest preexpectation $\mathsf{wp}[\![C]\!]\,(h)$ of any *loop-free* pGCL program C and h yields a linear expectation again. Hence, linear expectations are closed under applying Φ— Requirement 1 above—because

$$\forall\, g, h \in \mathsf{LinExp}\colon \quad \Phi\,(h) \;=\; \underbrace{\underbrace{[\neg\varphi]\cdot g}_{\in\ \mathsf{LinExp}} + [\varphi]\cdot \underbrace{\mathsf{wp}[\![C]\!]\,(h)}_{\in\ \mathsf{LinExp}}}_{\in\ \mathsf{LinExp}}\,.$$

7.2 Deciding Quantitative Entailments Between Linear Expectations

To prove that linear expectations meet Requirement 2—decidability of quantitative entailments—we effectively reduce the question of whether an entailment $h \preceq h'$ holds to the decidable satisfiability problem for QF_LIRA—quantifier-free mixed linear integer and real arithmetic (cf. [42]).

As a first step, we show that every linear expectation can be represented as a sum of mutually exclusive extended arithmetic expressions—a representation we refer to as the *guarded normal form* (similar to [41, Lem. 1], [9, Lem. A.2]).

Definition 2 (Guarded Normal Form (GNF)). *$h \in \mathsf{LinExp}$ is in GNF if*

$$h \;=\; \sum\nolimits_{i=1}^{n} [\varphi_i]\cdot \tilde{e}_i,$$

where $\tilde{e}_1, \ldots, \tilde{e}_n$ are extended linear expressions, $n \in \mathbb{N}$ is some natural number, and $\varphi_1, \ldots, \varphi_n$ are linear Boolean expressions that partition the set of states, i.e., for each $\sigma \in \Sigma$ there exists exactly one *$i \in \{1, \ldots, n\}$ such that $\varphi_i(\sigma) = \mathsf{true}$.*

Lemma 3. *Every linear expectation $h \in \mathsf{LinExp}$ can effectively be transformed into an equivalent linear expectation $\mathsf{GNF}\,(h) \equiv h$ in guarded normal form.*

The number of summands $|\mathsf{GNF}\,(h)\,|$ in $\mathsf{GNF}\,(h)$ is, in general, exponential in the number of summands in h. In practice, however, this exponential blow-up can often be mitigated by pruning summands with unsatisfiable guards. Throughout the remainder of this paper, we denote the components of $\mathsf{GNF}\,(h)$ and $\mathsf{GNF}\,(h')$, where h and h' are arbitrary linear expectations, as follows:

$$\mathsf{GNF}\,(h) \;=\; \sum\nolimits_{i=1}^{n} [\varphi_i]\cdot \tilde{e}_i \quad \text{and} \quad \mathsf{GNF}\,(h') \;=\; \sum\nolimits_{j=1}^{m} [\psi_j]\cdot \tilde{a}_j.$$

We now present a decision procedure for the *quantitative entailment* over LinExp.

Theorem 7. (Decidability of Quantitative Entailment over LinExp). *For $h, h' \in \mathsf{LinExp}$, it is decidable whether $h \preceq h'$ holds.*

Proof. Let $h, h' \in \mathsf{LinExp}$. By Lemma 3, we have $h \preceq h'$ iff $\mathsf{GNF}\,(h) \preceq \mathsf{GNF}\,(h')$.

Let σ be some state. By definition of the GNF, σ satisfies exactly one guard φ_i and exactly one guard ψ_j. Hence, the inequality $\mathsf{GNF}\,(h)\,(\sigma) \leq \mathsf{GNF}\,(h')\,(\sigma)$

does *not* hold iff $\tilde{e}_i(\sigma) > \tilde{a}_j(\sigma)$ holds for the expressions $\tilde{e}_i$ and $\tilde{a}_j$ guarded by φ_i and ψ_j, respectively. Based on this observation, we construct a $\mathsf{QF\_LIRA}$ formula $\mathsf{cex}_{\preceq}(h, h')$ that is *unsatisfiable* iff there is no counterexample to $h \preceq h'$:

$$\mathsf{cex}_{\preceq}(h, h') \triangleq \bigvee_{i=1}^{n} \bigvee_{j=1,\, \tilde{a}_j \neq \infty}^{m} (\varphi_i \wedge \psi_j \wedge \mathsf{encodeInfty}(\tilde{e}_i) > \tilde{a}_j)\,.$$

Here, we identify every program variable in h or h' with an $\mathbb{N}$-valued SMT variable. Moreover, to account for comparisons with ∞, we rely on the fact that our (extended) arithmetic expressions either evaluate to ∞ for *every* state or *never* evaluate to ∞. To deal with the case $\tilde{e}_i > \infty$, which is always false, we can thus safely exclude cases in which $\tilde{a}_j = \infty$ holds. To deal with the case $\infty > \tilde{a}_j$, we represent ∞ by some unbounded number, i.e., we introduce a fresh, unconstrained $\mathbb{N}$-valued SMT variable infty and set $\mathsf{encodeInfty}(\tilde{e})$ to infty if $\tilde{e} = \infty$; otherwise, $\mathsf{encodeInfty}(\tilde{e}) = \tilde{e}$. Since $\mathsf{QF\_LIRA}$ is decidable (cf. [42]), we conclude that the quantitative entailment problem is decidable. □

Since quantitative entailments are decidable, we can already conclude that, for linear expectations, Algorithm 2 is a semi-decision procedure.

7.3 Computing Minima of Linear Expectations

To ensure that latticed k-induction on pGCL programs (cf. Algorithm 1 and Sect. 7) is a semi-decision procedure when considering linear expectations, we have to consider Requirement 3—the expressability and computability of meets:

Theorem 8. LinExp *is effectively closed under taking minima.*

Proof. For $k \in \mathbb{N}$, let $\mathbf{k} \triangleq \{1, \ldots, k\}$. Then, for two linear expectations h, h', the linear expectation $\mathsf{GNF}(h) \mathrel{\mathsf{min}} \mathsf{GNF}(h') \in \mathsf{LinExp}$ is given by:

$$\sum_{(i,j) \in \mathbf{n} \times \mathbf{m}} \begin{cases} [\varphi_i \wedge \psi_j] \cdot \tilde{a}_j, & \text{if } \tilde{e}_i = \infty, \\ [\varphi_i \wedge \psi_j] \cdot \tilde{e}_i, & \text{if } \tilde{a}_i = \infty, \\ [\varphi_i \wedge \psi_j \wedge \tilde{e}_i \leq \tilde{a}_j] \cdot \tilde{e}_i + [\varphi_i \wedge \psi_j \wedge \tilde{e}_i > \tilde{a}_j] \cdot \tilde{a}_j & \text{otherwise,} \end{cases}$$

where we exploit that, for every state, exactly one guard φ_i and exactly one guard ψ_j is satisfied (cf. Lemma 3). Notice that in the last case we indeed obtain a linear expectation since neither $\tilde{e}$ nor $\tilde{a}$ are equal to ∞. □

8 Implementation

We have implemented a prototype called KIPRO2—k-Induction for PRObabilistic PROgrams—in Python 3.7 using the SMT solver Z3 [54] and the solver-API PySMT [25]. Our tool, its source code, and our experiments are available online.[8]

[8] https://github.com/moves-rwth/kipro2.

KIPRO2 performs in parallel latticed k-induction and BMC to fully automatically verify upper bounds on expected values of pGCL programs as described in Sect. 7. In addition to reasoning about expected values, KIPRO2 supports verifying bounds on *expected runtimes* of pGCL programs, which are characterized as least fixed points à la [40]. Rather than fixing a specific runtime model, we took inspiration from [56] and added a statement $\texttt{tick}(n)$ that does not affect the program state but consumes $n \in \mathbb{N}$ time units.

To discharge quantitative entailments and compute the meet, we use the constructions in Theorems 7 and 8, respectively. As an additional optimization, we do not iteratively apply the k-induction operator Ψ_f directly but use an *incremental encoding*. We briefly sketch our encoding for k-induction (Algorithm 2); the encoding for BMC is similar. In both cases, we employ uninterpreted functions on top of mixed integer and real arithmetic, i.e., QF_UFLIRA.

Recall Example 2, the geometric loop C_{geo}, where we used k-induction to prove $\mathsf{wp}[\![C_{\text{geo}}]\!]\,(c) \preceq c+1$. For every $k \in \mathbb{N}$, $\Phi(\Psi_{c+1}^{\lfloor k \rfloor}(c+1))$ is given by

$$\underbrace{[x=1]\cdot\Big(0.5\cdot\underbrace{\Psi_{c+1}^{\lfloor k \rfloor}(c+1)}_{Q_k}[x/0] \;+\; 0.5\cdot\underbrace{\Psi_{c+1}^{\lfloor k \rfloor}(c+1)}_{Q_k}[c/c+1]\Big) \;+\; [x\neq 1]\cdot c}_{P_k}\,.$$

To obtain an incremental encoding, we introduce an uninterpreted function $P_k\colon \mathbb{N}\times\mathbb{N}\to\mathbb{R}_{\geq 0}$ and a formula $\rho_k(c,x)$ specifying that $P_k(c,x)$ characterizes $\Phi(\Psi_{c+1}^{\lfloor k \rfloor}(c+1))$, i.e., for all $\sigma \in \Sigma$ and $r \in \mathbb{R}_{\geq 0}$ with $\Phi(\Psi_{c+1}^{\lfloor k \rfloor}(c+1))(\sigma) < \infty$,[9]

$$\rho_k(\sigma(c),\sigma(x)) \wedge P_k(\sigma(c),\sigma(x)) = r \text{ is satisfiable} \quad \text{iff} \quad r = \Phi\Big(\Psi_{c+1}^{\lfloor k \rfloor}(c+1)\Big)(\sigma).$$

If $\Phi(\Psi_{c+1}^{\lfloor k \rfloor}(c+1))(\sigma) = \infty$, our construction of $\rho_k(x,c)$ ensures that the above conjunction is satisfiable for arbitrarily large r. Analogously, we introduce an uninterpreted function $Q_k\colon \mathbb{N}\times\mathbb{N}\to\mathbb{R}_{\geq 0}$ that characterizes $\Psi_{c+1}^{\lfloor k \rfloor}(c+1)$.

In particular, may use all uninterpreted functions introduced for smaller or equal values of k—not just the function $P_k(c,x)$ it needs to characterize. This enables an incremental encoding, i.e., $\rho_k(c,x)$ can be computed on top of $\rho_{k-1}(c,x)$ by reusing $P_{k-1}(c,x)$, $Q_k(c,x)$, and the construction in Theorem 8.

Moreover, we can reuse $\rho_k(c,x)$ to avoid computing the (expensive) GNF for deciding certain quantitative entailments (cf. Theorem 7): For example, to check whether $\Phi(\Psi_{c+1}^{\lfloor k \rfloor}(c+1)) \not\preceq h'$ holds, we only need to transform the right-hand side into GNF (cf. Sect. 7.2), i.e., if $\mathsf{GNF}\,(h') = \sum_{j=1}^{m} [\psi_j]\cdot\tilde{a}_j$, then

$$\Phi\Big(\Psi_{c+1}^{\lfloor k \rfloor}(c+1)\Big) \not\preceq g \quad \text{iff} \quad \rho_k \wedge \bigvee_{j=1,\,\tilde{a}_j\neq\infty}^{m} \psi_j \wedge P_k(c,x) > \tilde{a}_j \text{ is satisfiable.}$$

[9] Notice that we do *not* axiomatize in $\rho_k(c,x)$ that $\Phi(\Psi_{c+1}^{\lfloor k \rfloor}(c+1))$ and $P_k(c,x)$ are the same function because we have no access to universal quantifiers. Rather, we specify that both functions coincide for any fixed concrete values assigned to c and x. This weaker notion is *not* robust against formal modifications of the parameters, e.g., through substitution. For example, to assign the correct interpretation to $P_k(c,x)\,[c/c+1]$, we have to construct a (second) formula $\rho_k(c,x)\,[c/c+1]$.

9 Experiments

We evaluate KIPRO2 on two sets of benchmarks. The first set, shown in Table 2, consists of four (infinite-state) probabilistic systems compiled from the literature; each benchmark is evaluated on multiple variants of candidate upper bounds:

(1) `brp` is a `pGCL` variant of the bounded retransmission protocol [19,32]. The goal is to transmit *toSend* many packages via an unreliable channel allowing for at most *maxFail* many retransmissions per package (cf. Example 1). The variable *totalFail* keeps track of the total number of failed attempts to send a package. We verified upper bounds on the expected outcome of *totalFail* (variants 1–4). In doing so, we bound the number of packages to send by 4 (10, 20, 70) while keeping *maxFail unbounded*, i.e., we still verify an infinite-state system. We notice that $k > 1$ is required for proving any of the candidate bounds; for up to $k = 11$, KIPRO2 manages to prove non-trivial bounds within a few seconds. However, unsurprisingly, the complexity increases rapidly with larger k. While KIPRO2 can prove variant 3, it needs to increase k to 23; we observe that the complexity grows rapidly both in terms of the size of formulae and in terms of runtime with increased k. Furthermore, variants 5–7 correspond to (increasing) incorrect candidate bounds ($\mathit{totalFail} + 1$, $\mathit{totalFail} + 1.5$, $\mathit{totalFail} + 3$) that are refuted (or time out) when not imposing any restriction on *toSend*.

(2) `geo` corresponds to the geometric loop from Example 2. We verify that $c+1$ upper-bounds the expected value of c for every initial state (variant 1); we refute the incorrect candidates $c + 0.99$ and $c + 0.999999999999$ (variants 2–3).

(3) `rabin` is a variant of Rabin's mutual exclusion algorithm [46] taken from [34]. We aim to verify that the probability of obtaining a unique winning process is at most $2/3$ for at most 2 (3, 4) participants (variants 1–3) and refute both $1/3$ (variant 4) and $3/5$ (variant 5) for an unbounded number of participants.

(4) `unif_gen` implements the algorithm in [48] for generating a discrete uniform distribution over some interval $\{l, \ldots, l+n-1\}$ using only fair coin flips. We aim to verify that $1/n$ upper-bounds the probability of sampling a particular element from *any* such interval of size at most $n = 2$ (3, 4, 5, 6) (variants 1–5).

Our second set of benchmarks, shown in Table 3, confirms the correctness of (1-inductive) bounds on the expected runtime of `pGCL` programs synthesized by the runtime analyzers ABSYNTH [56] and (later) KoAT [52]; this gives a baseline for evaluating the performance of our implementation. Moreover, it demonstrates the flexibility of our approach as we effortlessly apply the expected runtime calculus [40] instead of the weakest preexpectation calculus for verification.

Setup. We ran Algorithms 1 and 2 in parallel using an AMD Ryzen 5 3600X processor with a shared memory limit of 8GB and a 15-minute timeout. For every benchmark finishing within the time limit, KIPRO2 either finds the smallest k required to prove the candidate bound by k-induction or the smallest unrolling depth k to refute it. If KIPRO2 refutes, the SMT solver provides a concrete initial state witnessing that violation. In Tables 2 and 3, column #formulae gives the maximal number of conjuncts on the solver stack; formulae_t, sat_t, and total_t give the amount of time spent on (1) computing formulae, (2) satisfiability checking, and (3) everything (including preprocessing), respectively. The input consists of a program, a candidate upper bound, and a postexpectation; in Table 3, the latter is fixed to "postruntime" 0 and thus omitted.

Table 2. Empirical results for the first benchmark set (time in seconds).

	postexpectation	variant	result	k	#formulae	formulae_t	sat_t	total_t
`brp`	*totalFail*	1	ind	5	285	0.15	0.01	0.28
		2	ind	11	2812	1.77	0.12	2.03
		3	ind	23	26284	17.68	28.09	45.94
		4	TO	–	–	–	–	–
		5	ref	13	949	0.84	14.39	15.28
		6	TO	–	–	–	–	–
		7	TO	–	–	–	–	–
`geo`	c	1	ind	2	18	0.01	0.00	0.08
		2	ref	11	103	0.04	0.01	0.09
		3	ref	46	1223	0.39	0.04	0.48
`rabin`	$[i = 1]$	1	ind	1	21	0.01	0.00	0.15
		2	ind	5	1796	1.27	0.03	1.44
		3	TO	–	–	–	–	–
		4	ref	4	458	0.31	0.03	0.40
		5	ref	8	10508	8.76	2.85	11.68
`unif_gen`	$[c = i]$	1	ind	2	267	0.27	0.02	0.56
		2	ind	3	1402	1.45	0.10	1.81
		3	ind	3	1402	1.48	0.11	1.86
		4	ind	5	40568	47.31	15.70	63.28
		5	TO	–	–	–	–	–

Evaluation of Benchmark Set 1. Table 2 empirically underlines that probabilistic program verification can benefit from k-induction to the same extent as classical software verification: KIPRO2 *fully automatically* verifies relevant properties of *infinite-state* randomized algorithms and stochastic processes from the literature that require *k to be strictly larger than* 1. That is, proving these properties using (1-)inductive invariants requires either non-trivial invariant synthesis or additional user annotations. This indicates that k-induction mitigates the need for complicated specifications in probabilistic program verification (cf. [40]).

We observe that k-induction tends to succeed if *some* variable is bounded in the candidate upper bound under consideration (cf. `brp`, `rabin`, `unif_gen`). However, k-induction can also succeed without any bounds (cf. `geo`). The time and formulae required for checking k-inductivity increases rapidly for larger k; this is particularly striking for `rabin` and `unif_gen`. When refuting candidate bounds with BMC, we obtain a similar picture. Both the time and formulae required for refutation increase if the candidate bound increases (cf. `brp`, `geo`, `rabin`).

For both k-induction and BMC, we observe a direct correlation between the complexity of the loop, i.e., the number of possible traces through the loop from some fixed initial state after some bounded number of iterations, and the required time and space (number of formulae). Whereas for `geo` and `brp`—which exhibit a rather simple structure—these checks tend to be fast, this is not the case for `rabin` and `unif_gen`, which have more complex loop bodies. For such complex loops, k-induction and BMC quickly become infeasible as k increases.

Table 3. Empirical results for (a subset of) the ERTs [56] (time in *milliseconds*).

	runtime bound candidate	result	k	#formulae	formulae_t	sat_t	total_t
`2drwalk`	$2 \cdot (n + 1 \dot{-} d)$	TO	–	–	–	–	–
`bayesian_network`	$5 \cdot n$	TO	–	–	–	–	–
`ber`	$2 \cdot (n \dot{-} x)$	ind	1	9	7.22	0.44	88.12
`C4B_t303`	$0.5 \cdot (x + 2) + 0.5 \cdot (y + 2)$	ind	3	129	91.38	10.01	216.11
`condand`	$m + n$	ind	1	10	7.10	0.43	76.21
`fcall`	$2 \cdot (n \dot{-} x)$	ind	1	9	6.73	0.41	75.73
`hyper`	$5 \cdot (n \dot{-} x)$	ind	1	11	7.24	0.46	97.52
`linear01`	$0.6 \cdot x$	ind	1	11	7.19	0.49	74.38
`prdwalk`	$1.14286 \cdot (n + 4 \dot{-} x)$	ind	1	17	7.64	0.72	194.44
`prspeed`	$2 \cdot (m \dot{-} y) + 0.6666667 \cdot (n \dot{-} x)$	ind	1	18	7.64	0.81	145.13
`race`	$0.666667 \cdot (t + 9 \dot{-} h)$	ind	1	30	9.21	0.86	695.89
`rdspeed`	$2 \cdot (m \dot{-} y) + 0.666667 \cdot (n \dot{-} x)$	ind	1	19	7.70	0.78	143.45
`rdwalk`	$2 \cdot (n + 1 \dot{-} x)$	ind	1	12	10.22	0.75	85.03
`sprdwalk`	$2 \cdot (n \dot{-} x)$	ind	1	9	7.28	0.42	83.40

Evaluation of Benchmark Set 2. From Table 3, we observe that—in almost every case—verification is instantaneous and requires very few formulae. The programs we verify are equivalent to the programs provided in [56] up to interpreting minus as *monus* and using $\mathbb{N}$-typed (instead of $\mathbb{Z}$) variables. A manual inspection reveals that this matters for `C4B_t303` and `rdwalk`, which is the reason why the runtime bound for `C4B_t303` is 3-inductive rather than 1-inductive.

There are two timeouts (`2drwalk`, `bayesian_network`) due to the GNF construction from Lemma 3, which exhibits a runtime exponential in the number of possible execution branches through the loop body. We conjecture that further preprocessing (by pruning infeasible branches upfront) can mitigate this, rendering `2drwalk` and `bayesian_network` tractable as well. We consider a thorough investigation of suitable preprocessing strategies for GNF construction, which is outside the scope of this paper, a worthwhile direction for future research.

10 Conclusion

We presented κ-induction, a generalization of classical k-induction to arbitrary complete lattices, and—together with a complementary bounded model checking approach—obtained a fully automated technique for verifying infinite-state probabilistic programs. Experiments showed that this technique can prove non-trivial properties in an automated manner that using existing techniques cannot be proven—at least not without synthesizing a stronger inductive invariant. If a given candidate bound is k-inductive for some k, then our prototypical tool will find that k for linear programs and linear expectations. In theory, our tool is also applicable to non-linear programs at the expense of an undecidability quantitative entailment problem. It is left for future work to consider (positive) real-valued program variables for non-linear expectations.

Acknowledgements. B. L. Kaminski thanks Larry Fischer for his linguistic advice.

References

1. Abramsky, Jung: Domain theory. In: Handbook of Logic in Computer Science, vol. 3 (1994)
2. Agrawal, Chatterjee, Novotný: Lexicographic ranking supermartingales. PACMPL **2**(POPL) (2018)
3. Amtoft, Banerjee: A theory of slicing for imperative probabilistic programs. TOPLAS **42**(2) (2020)
4. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for Markov decision processes. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 160–180. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_8
5. Baldan, et al.: Fixpoint theory - upside down. In: FoSSaCS (2021)
6. Barthe, G., Espitau, T., Ferrer Fioriti, L.M., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 43–61. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_3
7. Batz, K., Junges, S., Kaminski, B.L., Katoen, J.-P., Matheja, C., Schröer, P.: PrIC3: property directed reachability for MDPs. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 512–538. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_27
8. Batz, et al.: Latticed k-induction with an application to probabilistic programs (extended version). arXiv (2021)
9. Batz, et al.: Relatively complete verification of probabilistic programs. PACMPL **5**(POPL) (2021)
10. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_42
11. Biere: Bounded model checking. In: Handbook of Satisfiability (2009)
12. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14

13. Biere, A., Clarke, E., Raimi, R., Zhu, Y.: Verifying safety properties of a PowerPC – microprocessor using symbolic model checking without BDDs. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 60–71. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_8
14. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
15. Chadha, Viswanathan: A counterexample-guided abstraction-refinement framework for Markov decision processes. TOCL **12**(1) (2010)
16. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
17. Clarke, et al.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1) (2001)
18. Cousot, Cousot: Constructive versions of Tarski's fixed point theorems. Pacific J. Math. **82**(1) (1979)
19. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: de Alfaro, L., Gilmore, S. (eds.) PAPM-PROBMIV 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44804-7_3
20. Déharbe, D., Moreira, A.M.: Using induction and BDDs to model check invariants. In: Advances in Hardware Design and Verification. IAICT, vol. 105, pp. 203–213. Springer, Boston, MA (1997). https://doi.org/10.1007/978-0-387-35190-2_13
21. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 280–295. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_24
22. Donaldson, Kroening, Rümmer: Automatic analysis of DMA races using model checking and k-induction. Formal Methods Syst. Des. **39**(1) (2011)
23. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_26
24. Feng, Y., Zhang, L., Jansen, D.N., Zhan, N., Xia, B.: Finding polynomial loop invariants for probabilistic programs. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 400–416. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_26
25. Gario, Micheli: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: SMT Workshop (2015)
26. Gehr, T., Misailovic, S., Vechev, M.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 62–83. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_4
27. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
28. Gretz, Katoen: McIver: operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Eval. **73** (2014)
29. Gurfinkel, Ivrii: K-induction without unrolling. In: FMCAD (2017)
30. Han, Katoen, Damman: Counterexample generation in probabilistic model checking. IEEE Trans. Softw. Eng. **35**(2) (2009)

31. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 488–511. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_26
32. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 127–165. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58085-9_75
33. Huang, Z., Wang, Z., Misailovic, S.: PSense: automatic sensitivity analysis for probabilistic programs. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 387–403. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_23
34. Hurd, McIver, Morgan: Probabilistic guarded commands mechanized in HOL. Theor. Comput. Sci. **346**(1) (2005)
35. Jansen, N., Dehnert, C., Kaminski, B.L., Katoen, J.-P., Westhofen, L.: Bounded model checking for probabilistic programs. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 68–85. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_5
36. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_33
37. Jovanović, Dutertre: Property-directed k-induction. In: FMCAD (2016)
38. Kaminski: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University, Germany (2019)
39. Kaminski, Katoen, Matheja: On the hardness of analyzing probabilistic programs. Acta Inform. **56**(3) (2019)
40. Kaminski, et al.: Weakest precondition reasoning for expected runtimes of randomized algorithms. J. ACM **65**(5) (2018)
41. Katoen, J.-P., McIver, A.K., Meinicke, L.A., Morgan, C.C.: Linear-invariant generation for probabilistic programs: automated support for proof-based methods. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 390–406. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_24
42. King, Barrett, Tinelli: Leveraging linear and mixed integer programming for SMT. In: SMT (2014)
43. Knaster: Un théorème sur les functions d'ensembles. Ann. Soc. Pol. Math. **6** (1928)
44. Kozen: A probabilistic PDL. J. Comput. Syst. Sci. **30**(2) (1985)
45. Vediramana Krishnan, H.G., Vizel, Y., Ganesh, V., Gurfinkel, A.: Interpolating strong induction. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 367–385. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_21
46. Kushilevitz, Rabin: Randomized mutual exclusion algorithms revisited. In: PODC (1992)
47. Lassez, Nguyen, Sonenberg: Fixed point theorems and semantics. Inf. Process. Lett. **14**(3) (1982)
48. Lumbroso: Optimal discrete uniform generation from coin flips, and applications. arXiv (2013)
49. McIver, Morgan: Abstraction, refinement and proof for probabilistic systems (2005)
50. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
51. McMillan: An interpolating theorem prover. Theor. Comput. Sci. **345**(1) (2005)
52. Meyer, Hark, Giesl: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: TACAS (2021, to appear)

53. Milner: Communication and concurrency (1989)
54. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
55. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: from refutation to verification. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_2
56. Ngo, Carbonneaux, Hoffmann: Bounded expectations: resource analysis for probabilistic programs. In: PLDI (2018)
57. Park: Fixpoint induction and proofs of program properties. Mach. Intell. **5** (1969)
58. Pous, D.: Complete lattices and up-to techniques. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 351–366. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76637-7_24
59. Pous, Sangiorgi: Enhancements of the bisimulation proof method. In: Advanced Topics in Bisimulation and Coinduction, vol. 52 (2012)
60. Puterman: Markov Decision Processes (1994)
61. Quatmann, T., Katoen, J.-P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 643–661. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_37
62. Rabehaja, Sanders: Refinement algebra with explicit probabilism. In: TASE (2009)
63. Rocha, W., Rocha, H., Ismail, H., Cordeiro, L., Fischer, B.: DepthK: a k-induction verifier based on invariant inference for C programs. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 360–364. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_23
64. Schüle, Schneider: Bounded model checking of infinite state systems. Formal Methods Syst. Des. **30**(1) (2007)
65. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
66. Tarski: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2) (1955)
67. Wang, Hoffmann, Reps: PMAF: an algebraic framework for static analysis of probabilistic programs. In: PLDI (2018)
68. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 366–380. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-93900-9_29